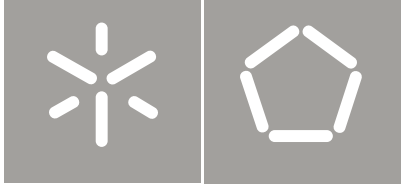
Universidade do Minho
Escola de Engenharia

José Carlos Rufino Amaro

Co-operação de Tabelas de Hash Distribuídas em Clusters Heterogéneos

Setembro de 2008 [edição revista]



Universidade do Minho
Escola de Engenharia

José Carlos Rufino Amaro

Co-operação de Tabelas de Hash Distribuídas em Clusters Heterogéneos

Tese de Doutoramento em Informática
Área de Conhecimento de Engenharia de Computadores

Trabalho efectuado sob a orientação do
Professor Doutor António Manuel da Silva Pina

Setembro de 2007 [edição original] ; Setembro de 2008 [edição revista]

DECLARAÇÃO

Nome: JOSÉ CARLOS RUFINO AMARO

Endereço Electrónico: rufino@ipb.pt

Título da Tese de Doutoramento:

Co-operação de Tabelas de Hash Distribuídas em Clusters Heterogéneos

Orientador:

Professor Doutor António Manuel da Silva Pina

Ano de conclusão: 2007

Ramo de Conhecimento do Doutoramento:

Informática, Área de Conhecimento de Engenharia de Computadores

Universidade do Minho, / /

Assinatura: _____

Ao André, à Célia e aos meus Pais.

Agradecimentos

Ao meu orientador, Prof. António Pina, pela sua orientação científica, grande disponibilidade e encorajamento constante. Soube, desde sempre, respeitar algum voluntarismo prospectivo da minha parte e, quando necessário, recolocar-me no caminho certo.

Aos meus colegas de investigação, Albano, Exposto e Prof. Joaquim Macedo, pelas discussões técnicas frutuosas que me proporcionaram, e pela sua amizade e camaradagem.

Aos meus colegas de profissão mais próximos, Maria João e Rui Pedro, pela sobrecarga que a minha dedicação ao trabalho por vezes lhes causou, e pela sua amizade e apoio.

Ao Instituto Politécnico de Bragança e à Universidade do Minho, por me terem proporcionado sempre as melhores condições para a realização da minha investigação.

Aos meus Pais, pelo encorajamento constante e disponibilidade em ajudar à sua medida.

Ao meu rebento André e à minha mulher Célia, pelo seu apoio e também pelo seu esforço e sacrifício pessoal, com a promessa de lhes tentar compensar as inúmeras privações do convívio familiar motivadas pela realização deste trabalho.

Resumo

As Estruturas de Dados Distribuídas (DDSs) constituem uma abordagem ao Armazenamento Distribuído adequada a aplicações com requisitos de elevada capacidade de armazenamento, escalabilidade e disponibilidade. Ao mesmo tempo, apresentando-se com interfaces simples e familiares, permitem encurtar o ciclo de desenvolvimento de aplicações.

Num ambiente de *cluster*, caracterizado pela co-existência de grande variedade de aplicações e utilizadores com requisitos dinâmicos, e de gerações de equipamento com desempenhos diferenciados (tanto ao nível do poder de cálculo, como das comunicações e do armazenamento), a necessidade de operação conjunta e eficiente de múltiplas DDSs pelas aplicações constituiu um desafio científico e tecnológico que esta tese se propôs superar.

Esta tese apresenta as propostas e os resultados alcançados durante o estudo e desenvolvimento dos modelos e da plataforma de suporte à arquitectura Domus para a *co-operação* de múltiplas Tabelas de Hash Distribuídas (DHTs) em *clusters* partilhados e heterogéneos.

A plataforma contempla várias classes de atributos e operações sobre DHTs permitindo, entre outras possibilidades, a definição administrativa de limites à expansão/contracção, o recurso a diferentes tecnologias de armazenamento e a suspensão/retoma administrativa da operação. Numa outra vertente, para responder aos requisitos impostos pela execução simultânea de múltiplas e diferenciadas aplicações, foram formulados e desenvolvidos mecanismos de balanceamento dinâmico de carga que visam a optimização e rentabilização dos recursos computacionais, comunicacionais e de armazenamento disponíveis no *cluster*.

Na base da abordagem estão modelos que dão uma resposta qualificada aos problemas do *particionamento* e da *localização* de entradas numa DHT. Em relação ao *particionamento*, são propostos modelos de *distribuição* para definição do *número* de entradas de cada nó de uma DHT, suportando variação dinâmica do número total de nós; os modelos asseguram uma distribuição *óptima* do número de entradas, nos quatro cenários que resultam da combinação de Hashing Estático ou Dinâmico, com Distribuição Homogénea ou Heterogénea; com Hashing Dinâmico, a *qualidade da distribuição* é parametrizável e, com Distribuição Homogénea, o esforço de re-distribuição é de ordem $O(1)$ face ao total de nós da DHT.

No que toca à *localização*, definiram-se *algoritmos de encaminhamento acelerado* para localização distribuída em topologias DeBruijn e Chord, que respondem à *descontinuidade* das partições das DHTs geradas pelos modelos de *distribuição*. Para o efeito, explorou-se a possível coexistência de múltiplas tabelas de encaminhamento em cada nó de uma DHT, para tentar aproximar o maior custo da localização sobre grafos completos (em que os vértices são entradas da DHT), ao custo de referência sobre grafos esparsos (em que os vértices são nós da DHT), tendo-se obtido valores de 70% a 90% do custo de referência.

Abstract

Distributed Data Structures (DDSs) as an approach to Distributed Storage are adequate to applications that require high storage capacity, scalability and availability. At the same time, DDSs present simple and familiar interfaces that allow shorter development cycles.

In a cluster environment, shared by multiple users and applications with dynamic requisites, and built on different hardware generations (with different computing, communication and storage power), the need to simultaneously and efficiently operate several DDSs by user applications presents the scientific and technological challenge embraced by this thesis.

This thesis presents the proposals and the results achieved during the study of the models and development of the platform that supports the Domus architecture for the *co-operation* of multiple Distributed Hash Tables (DHTs) in shared and heterogeneous clusters.

The platform supports several classes of attributes and operations on DHTs allowing, among other possibilities, the administrative definition of per DHT node boundaries and storage technologies, or the deactivation/reactivation of DHTs. In addition, Domus DHTs are managed by dynamic load balancing mechanisms which ensure certain service levels (like storage space or access performance) and optimize the utilization of cluster resources.

The architecture builds on specific models for the *partitioning* and *lookup* of a DHT address space. The *partitioning* models ensure optimal distribution of the number of DHT buckets per node, for a dynamic number of DHT nodes, and support the different scenarios that arise from the combination of Static/Dynamic Hashing, with Homogeneous/Heterogeneous Distributions. The quality of the distribution is tuneable under Dynamic Hashing, and repartitioning involves $O(1)$ nodes of the DHT under Homogeneous Distributions.

With regard to the *lookup* problem, a set of *accelerated routing algorithms* were designed for distributed lookup, both for DeBruijn and Chord topologies, in order to deal with the discontinuous DHT partitions generated by our partitioning models. The algorithms explore the availability of multiple routing tables per DHT node with the aim of reducing the lookup cost with full graphs, in which the vertexes are all the DHT buckets, towards the reference lookup cost with sparse graphs, in which the vertexes are all the DHT nodes. The best lookup algorithms ensure a lookup cost of 70% to 90% of the reference cost.

Conteúdo

Capítulo 1

Introdução

A evolução constante na capacidade de processamento dos microprocessadores (de que são testemunho o surgimento recente de CPUs com vários núcleos [ABC+06]), em conjunção com a introdução de tecnologias de rede local cada vez mais rápidas (como Myrinet, Infiniband e 10G-Ethernet [BFP06]), aliada à generalização de ferramentas e standards de programação paralela/distribuída (como MPI e OpenMP [Qui03]), têm contribuído para o reforço da importância do paradigma de computação baseada em *clusters* [ACP95]. Mais recentemente, a computação em *Grid* [FK03] apresenta-se como um salto evolutivo daquele paradigma, tendo como objectivo aproveitar as capacidades conjuntas de múltiplos *clusters*, geograficamente dispersos e pertencentes a diferentes domínios administrativos.

A computação baseada em *clusters* tem sido usada na resolução de problemas com requisitos de elevado poder de cálculo e/ou de manipulação de grandes quantidades de dados. O suporte a grandes volumes de dados inclui o tratamento de questões ligadas quer à viabilização do seu armazenamento, quer à necessidade de garantir um acesso eficiente aos mesmos. Neste contexto, as vantagens do ambiente de execução paralelo/distribuído fornecido pelos *clusters* são bem conhecidas: i) elevada capacidade de armazenamento, pela combinação de recursos de armazenamento de vários nós; ii) elevado poder de processamento dos dados, potenciado pela possibilidade de acesso e processamento paralelos.

Tradicionalmente, o armazenamento paralelo/distribuído em ambiente *cluster* é fornecido por Sistemas de Ficheiros ou Bases de Dados paralelos(as)/distribuídos(as). As Estruturas de Dados Distribuídas (DDSs), entendidas como versões distribuídas de estruturas de dados clássicas (listas, árvores, tabelas de *hash*, etc.), representam uma abordagem alternativa, capaz de fornecer às aplicações clientes propriedades como elevado desempenho, escalabilidade, disponibilidade e durabilidade, para além de elevada capacidade de armazenamento. Ao mesmo tempo, a disponibilização de interfaces simples e familiares pelas DDSs, permite aos programadores encurtar o ciclo de desenvolvimento de aplicações.

No âmbito desta tese estamos essencialmente interessados nas questões ligadas a um tipo específico de DDSs: Tabelas de Hash Distribuídas (DHTs), como abordagem à realização de Dicionários Distribuídos (DDs), assumindo-se o *cluster* como ambiente de operação.

Um *dicionário* é um tipo abstracto de dados que pode ser visto como uma colecção de

registos de esquema <*chave*, *dados*>, indexados pelo campo *chave*, e suportando operações de inserção, consulta e remoção de registos. Para certas classes de aplicações, o "paradigma" dos Dicionários Distribuídos é particularmente adequado. Por exemplo, num indexador Web paralelo/distribuído, estruturas como i) a *cache* de DNS ou ii) as listas de URLs visitados (ou a visitar), podem ser realizadas à custa de Dicionários Distribuídos.

A assunção do *cluster* como ambiente alvo da nossa investigação implicou uma maior aproximação dos nossos modelos a DHTs de uma primeira geração, formuladas pela comunidade de investigação de SDDSs (*Scalable Distributed Data Structures*), do que a um conjunto mais recente de abordagens, de segunda geração, especialmente orientadas para ambientes distribuídos P2P (*Peer-to-Peer*). Todavia, como veremos, estas abordagens ofereceram contribuições importantes no projecto das nossas DHTs orientadas ao *cluster*.

Aplicações como o indexador Web referido personificam também uma classe de *aplicações duradouras*, cujo regime de execução é contínuo ou exige longos períodos de tempo; para a execução de aplicações desse tipo são desejáveis mecanismos que tolerem a co-execução dinâmica de outras aplicações no *cluster* e assegurem, nessas circunstâncias, a melhor rentabilização dos seus recursos, designadamente por parte das DDSs/DHTs que servem as aplicações; adicionalmente, a possibilidade de libertar temporariamente o *cluster* para outras aplicações e de, posteriormente, retomar a execução de uma aplicação duradoura, beneficia de eventuais funcionalidades de suspensão/reactivação embutidas nas DHTs.

1.1 Motivações da Investigação

A rentabilização permanente dos recursos do *cluster* pressupõe a consciência da heterogeneidade i) *física* e ii) *lógica* do seu ambiente de execução. A heterogeneidade física resulta da possível co-existência de nós computacionais e/ou tecnologias de comunicação de gerações/espécies diferentes, podendo ter na sua origem: i1) o crescimento progressivo do *cluster* a partir de uma instalação inicial; i2) a especificidade de certas tarefas a realizar (*e.g.*, *nós de armazenamento* com acesso directo a uma SAN, *nós de processamento gráfico/vectorial* munidos de GPUs, etc.). A heterogeneidade lógica resulta da utilização específica e dinâmica que as aplicações fazem dos recursos do *cluster*, sendo incontornável.

Assim, uma motivação de fundo para a nossa investigação foi a constatação, no seu início, de um suporte ausente/incipiente à heterogeneidade do ambiente de execução, por parte das abordagens de referência de então a Tabelas de Hash Distribuídas [LNS93a, Dev93, HBC97, VBW98, GBHC00]. Na maior parte dessas abordagens, a redistribuição de uma DHT tinha causas endógenas, relacionadas apenas com a optimização da utilização dos recursos de armazenamento: a DHT sofria uma expansão/contracção automática, um nó de cada vez, em resposta ao maior/menor consumo de recursos de armazenamento. Menos frequentemente, a expansão/contracção administrativas podiam também ser suportadas [GBHC00], assim como a migração automática de partes (de grão mais fino) da DHT para balanceamento dinâmico da carga de acesso à DHT [VBW98] (mas sem qualquer tentativa de o fazer tendo em conta outras aplicações executando no *cluster*). Em suma, os modelos previstos eram inapropriados à instanciação em *clusters* partilhados, onde é admissível a partilha dos nós por mais de uma tarefa. O suporte a um regime de exploração

partilhado permite a) aumentar a utilização dos recursos e b) diminuir o tempo de retorno (*turnaround time*); no entanto, carece de mecanismos de gestão de carga mais sofisticados.

Precisamente, num ambiente desse tipo, em que várias aplicações co-executam, operando uma ou mais DHTs (como no indexador Web já referido), a rentabilização dos recursos do *cluster* beneficiará de uma gestão conjunta das várias DHTs aí instanciadas. Nesta perspectiva, por exemplo, as decisões de balanceamento dinâmico relativas a cada DHT não devem ser tomadas isoladamente das outras DHTs o que, a acontecer, pode contribuir para a sub-utilização do *cluster* e/ou a instabilidade dos mecanismos de balanceamento.

Assim, outra motivação do nosso trabalho foram as questões (quase ignoradas nas ditas abordagens de referência) ligadas à *operação conjunta de múltiplas DHTs*, entendida como "instanciação, gestão e exploração de múltiplas DHTs em simultâneo, realizada de forma integrada e cooperativa, pela mesma colecção de serviços distribuídos". Neste contexto, para além do contributo de tal operação conjunta, para a rentabilização permanente dos recursos do *cluster*, surge também, de forma natural, a motivação para suportar a possibilidade de cada DHT exibir propriedades diferentes em função de requisitos aplicacionais específicos (ou seja, a motivação para suportar um certo grau de *heterogeneidade de DHTs*).

1.1.1 Objectivos da Investigação

No seguimento do exposto, definimos como objectivo estratégico do nosso trabalho a definição de "uma arquitectura de co-operação de múltiplas DHTs para ambientes do tipo *cluster*, em regime de exploração *partilhado*", capaz de dar resposta aos seguintes requisitos:

1. requisitos de *eficiência*: exploração eficiente do *cluster* por DHTs cientes da heterogeneidade do ambiente de execução e auto-ajustáveis às suas condições dinâmicas;
2. requisitos de *conveniência*: fornecimento de abstracções adequadas ao suporte integrado de múltiplas DHTs e à sua configuração de acordo com requisitos aplicacionais.

1.2 Contribuições Principais

As principais contribuições técnico-científicas da tese são, abreviadamente, as seguintes:

1. modelos de distribuição *óptima* do contradomínio de funções de *hash* por nós computacionais, suportando i) número variável de nós, ii) Hashing Estático ou Dinâmico, iii) Distribuição Homogénea ou Heterogénea e iv) *qualidade de distribuição* afinável;
2. algoritmos para *encaminhamento acelerado*, adequados a localização distribuída em grafos DeBruijn [dB46] e Chord [SMK+01] completos no domínio dos *hashes*; os algoritmos investigados permitem alcançar um custo médio de localização de até 50% face ao obtido com *encaminhamento convencional* no mesmo tipo de grafos; o mesmo custo médio representa ainda entre 70% a 90% do custo médio de localização com *encaminhamento convencional* sobre grafos completos no domínio dos nós;

3. a arquitectura Domus que, integrando as contribuições anteriores, suporta a instanciação, gestão e exploração de múltiplas DHTs, realizada de forma cooperativa, regulada por dois mecanismos de balanceamento dinâmico complementares (um centrado nas qualidades de serviço das DHTs, outro na optimização dos recursos do *cluster*);

4. um protótipo de código aberto, que realiza parte substancial da arquitectura Domus.

1.2.1 Publicações Principais

A maior parte dos principais resultados obtidos em cada etapa da investigação foram publicados e validados em conferências internacionais. Os artigos publicados, que traduzem a cronologia de algumas das contribuições científicas mais relevantes, são os seguintes:

1. Toward a Dynamically Balanced Cluster-oriented DHT [RPAE04b]
   ↪ definição de um modelo (centralizado) para a distribuição pesada de uma DHT através de um conjunto de nós computacionais, com uma "qualidade da distribuição" parametrizável; inclui a apresentação de resultados de várias simulações ao modelo;

2. A Cluster-oriented Model for Dynamically Balanced DHTs [RPAE04a]
   ↪ descrição de uma variante distribuída do modelo anterior [RPAE04b], na qual o processo de repartição e evolução da DHT é menos dependente de informação global; inclui a apresentação de simulações ao modelo e a sua comparação com o anterior;

3. Domus - An Architecture for Cluster-oriented Distributed Hash Tables [RPAE05]
   ↪ descrição de uma primeira iteração da arquitectura Domus, para a co-operação de DHTs; inclui a descrição dos vários subsistemas e das suas interacções principais;

4. pDomus: a Prototype for Cluster-oriented Distributed Hash Tables [RPAE07b]
   ↪ primeira apresentação do protótipo pDomus da arquitectura Domus; inclui i) a avaliação exaustiva do *desempenho* das tecnologias de armazenamento suportadas e ii) a definição de métricas de auxílio à selecção das tecnologias de armazenamento;

5. Full-Speed Scalability of the pDomus platform for DHTs [RPAE07a]
   ↪ segunda apresentação do pDomus, baseada na avaliação exaustiva da sua *escalabilidade*, com diferentes tecnologias de armazenamento e estratégias de localização.

§ As publicações 1, 3, 4 e 5 encontram-se indexadas no portal *ISI Web of Knowledge*. A publicação 2 encontra-se indexada no portal *IEEE Xplore*.

1.3 Organização da Dissertação

Os artigos associados à dissertação constituem documentos que, embora veiculem algumas das principais ideias preconizadas, são de abrangência e profundidade limitadas, pelo que a sua leitura não substitui a da dissertação. Relativamente a esta, a organização adoptada

tem como propósito uma apresentação de conceitos mais completa, coerente e integradora, remetendo para segundo plano a cronologia dos artigos e dos trabalhos desenvolvidos.

Assim, no capítulo 2 contextualiza-se o trabalho desenvolvido, no quadro de um levantamento de abordagens e tecnologias relacionadas com os temas aprofundados na dissertação. Em paralelo, introduzem-se conceitos importantes à compreensão dos capítulos seguintes. Estes terminam, em geral, num confronto com abordagens referenciadas no capítulo 2.

No capítulo 3 apresentam-se modelos capazes de assegurar distribuição *óptima* do contradomínio de uma função de hash, por um conjunto de nós computacionais. Os modelos cobrem os vários cenários que emergem considerando a combinação de Distribuições Homogéneas ou Heterogéneas, com a utilização de Hashing Estático ou Hashing Dinâmico. A nossa abordagem é também comparada com o Hashing Consistente, a fim de se apreender o real significado da *qualidade da distribuição* (superior) alcançável pelos nossos modelos.

No capítulo 4 descrevem-se mecanismos de posicionamento e localização compatíveis com os mecanismos de distribuição do capítulo anterior. Os mecanismos de localização (distribuída) exploram grafos DeBruijn binários e grafos Chord completos, com algoritmos de *encaminhamento acelerado*; estes, tirando partido de várias fontes de informação topológica disponíveis em cada nó de uma DHT, diminuem o esforço de localização (número de saltos na topologia) face ao uso de *encaminhamento convencional*. Descreve-se também a interacção entre os mecanismos de posicionamento e localização na evolução das DHTs.

No capítulo 5 introduz-se a arquitectura Domus de suporte à *co-operação* (operação conjunta e integrada) de múltiplas DHTs independentes, vistas como um serviço distribuído do *cluster*, orientado ao armazenamento de grandes dicionários. São apresentadas as entidades e relações da arquitectura, incluindo i) a análise em detalhe dos seus componentes e subsistemas e ii) a definição dos atributos e invariantes que regulam as suas relações.

No capítulo 6 descrevem-se, de forma aprofundada, os dois mecanismos complementares de gestão dinâmica de carga, previstos na arquitectura Domus (um mecanismo centrado na optimização de níveis de serviço das DHTs, outro na rentabilização dos recursos do *cluster*). A descrição desses mecanismos gira em torno do papel que desempenham nos momentos fundamentais da evolução das DHTs. Concomitantemente, descrevem-se os atributos e as medidas em que os mecanismos se apoiam para tomar as suas decisões.

O capítulo 7 descreve os aspectos mais relevantes de um protótipo da arquitectura Domus: i) os seus componentes de software e a sua relação com os componentes da arquitectura, ii) os mecanismos de caracterização e monitorização do *cluster*, iii) o acesso às funcionalidades do protótipo por programadores e administradores. O capítulo termina com a apresentação e discussão dos resultados de dois conjuntos de testes de avaliação do protótipo.

No capítulo 8 discutem-se as contribuições da tese e as perspectivas de trabalho futuro.

O apêndice A contém o resumo das principais publicações associadas à dissertação. O apêndice B fornece conceitos básicos de Hashing Não-Distribuído (Estático e Dinâmico), complementando o capítulo 2. Em apoio ao capítulo 4, o apêndice C sintetiza conceitos básicos de Teoria de Grafos, e o apêndice D contém demonstrações formais de propriedades. O apêndice E contém código comentado, que ilustra a utilização do protótipo desenvolvido.

Capítulo 2

Enquadramento

Resumo

Neste capítulo contextualiza-se o trabalho desenvolvido e introduzem-se alguns conceitos importantes à compreensão do restante texto, no quadro de um levantamento das abordagens e tecnologias relacionadas com os principais temas aprofundados na dissertação.

2.1 Prólogo

No que se segue não se pretende sintetizar, de forma exaustiva, toda a investigação externa com ligações ao nosso trabalho, mas tão somente um conjunto referencial de abordagens e tecnologias, algumas das quais marcaram opções efectuadas ao longo do percurso que conduziu à escrita da dissertação. Assim, 1) resumimos o essencial das abordagens convencionais ao armazenamento paralelo/distribuído, 2) introduzimos o conceito de Estrutura de Dados Distribuída (DDS) como alternativa, 3) efectuamos um estudo comparativo das principais DHTs de primeira geração, nascidas à luz do conceito de DDS, 4) referenciamos brevemente, numa perspectiva evolutiva, as principais aproximações de segunda geração e, por fim, 5) fazemos referência a diversas abordagens à resolução dos vários problemas específicos que atacamos na tese. Ao longo do capítulo introduzem-se também conceitos importantes; em complemento, o apêndice B fornece um resumo dos principais conceitos associados ao Hashing Não-Distribuído, nas suas modalidades Estática ou Dinâmica.

2.2 Armazenamento Paralelo/Distribuído Convencional

Tradicionalmente, o armazenamento paralelo/distribuído de grandes volumes de dados em ambiente *cluster* socorre-se de uma das seguintes abordagens: i) Sistemas de Ficheiros Paralelos/Distribuídos ou ii) Bases de Dados Paralelas/Distribuídas. Nesta secção resumem-se alguns dos conceitos principais associados, preparando o terreno para a descrição subsequente das Estruturas de Dados Distribuídas como abordagem alternativa.

2.2.1 Bases de Dados Paralelas/Distribuídas

Um Sistema de Gestão de Bases de Dados (DBMS) garante persistência e consistência dos dados, designadamente propriedades ACID (*Atomicity, Consistency, Isolation, and Durability*), derivadas do uso de transacções [Gra81]. Todavia, garantir estas propriedades incorre em custos elevados, em termos de complexidade e desempenho pelo que, na prática, algumas dessas propriedades são por vezes relaxadas. Por outro lado, é também à custa de maior complexidade de implementação, e do inerente impacto no desempenho, que um DBMS é capaz de oferecer um elevado grau de independência relativamente aos tipos de dados armazenáveis, permitindo desacoplar a visão lógica que os utilizadores têm dos dados, da disposição física dos mesmos nos suportes de armazenamento. Esse desacoplamento permite, por exemplo, grande versatilidade na interacção com a base de dados, através da realização de interrogações limitadas apenas pela expressividade de linguagens como o SQL. Tal versatilidade (assente em complexidade adicional) limita o desempenho e a paralelização do acesso à base de dados, problemas agravados num ambiente distribuído.

Uma Base de Dados Distribuída (DDBS) pode ser definida como "uma colecção de bases de dados logicamente inter-relacionadas, distribuída por uma rede computadores" [OV99]. Neste contexto, um Sistema de Gestão de Bases de Dados Distribuídas (DDBMS) será "o software que suporta a gestão da DDBS e torna a distribuição transparente aos seus clientes". Numa DDBS, cada nó da rede executa localmente o mesmo tipo de DBMS, para gerir a base de dados local, sendo da responsabilidade do DDBMS a coordenação do acesso à base de dados "virtual" representada pela DDBS; por exemplo, uma interrogação pode ser paralelizada/distribuída, sendo da responsabilidade do DDBMS a sua coordenação.

Uma Base de Dados Paralela (PDBS) pode ser vista como uma variante de uma DDBS em que a primazia é dada ao desempenho. Este é optimizado através da paralelização das operações mais pesadas, através de vários nós, como sejam i) o carregamento de dados, de Disco para RAM, ii) a construção de índices, iii) o processamento de interrogações, etc. Se uma DDBS pode surgir motivada apenas pela necessidade de explorar capacidade de armazenamento distribuída, já uma PDBS surge, na essência, por razões de desempenho.

2.2.2 Sistemas de Ficheiros Paralelos/Distribuídos

Em termos de interface, os Sistemas de Ficheiros (SFs) expõem funcionalidades de baixo nível e de reduzida independência dos dados: um sistema de ficheiros é organizado como uma hierarquia de directórios, com ficheiros, sendo estes simples sequências de bytes, de dimensão variável; estes elementos básicos (directórios, ficheiros e bytes) são directamente expostos aos clientes do sistema de ficheiros, sendo da responsabilidade dos clientes a estruturação dos dados aplicacionais com base naqueles elementos. Relativamente à consistência, os SFs obedecem, em geral, a modelos mais relaxados que as Bases de Dados.

Os Sistemas de Ficheiros Distribuídos (DFSs) são também conhecidos por Sistemas de Ficheiros de Rede. Tipicamente, possibilitam o acesso concorrente a partes dos sistemas de ficheiros, de um ou mais sistemas servidores, partilhados por um conjunto de clientes, através de uma rede de computadores. Sistemas como o NFS [SGK+85] oferecem garantias

fracas de consistência e são pouco escaláveis; outros, como o AFS [HKM+88], garantem uma imagem coerente do sistema de ficheiros, e não dependem de serviços centralizados.

Num Sistema de Ficheiros Paralelo (PFS) os blocos dos ficheiros são espalhados (e por vezes replicados) por múltiplos sistemas interligados em rede, sendo esse processo transparente para os clientes do PFS. O objectivo dessa dispersão é melhorar o desempenho no acesso, utilizando estratégias semelhantes às usadas pelos sistemas RAID, mas num ambiente distribuído. Tal como os DFSs descritos anteriormente, a robustez e escalabilidade dos PFSs varia consoante as implementações (*e.g.*, sistemas como o GFS [SEP+97, Pea99] são considerados mais maturos, robustos e escaláveis do que outros, como o PVFS [CLRT00]).

2.3 Estruturas de Dados Distribuídas

As Estruturas de Dados Distribuídas (DDSs), entendidas como versões distribuídas de estruturas de dados clássicas e centralizadas (listas, árvores, tabelas de *hash*, etc.), representam um paradigma diferente e mais recente que as abordagens convencionais anteriores.

Assim [GBHC00], o interface apresentado por uma DDS é mais estruturado e de nível mais elevado que o fornecido pelos sistemas de ficheiros, dado que o grão da operação é o *registo* da estrutura de dados, e não uma *sequência de bytes arbitrária*; por outro lado, o conjunto de operações suportado por uma DDS, exposto pelos métodos da sua API, é fixo e mais limitado, quando comparado com o permitido por uma linguagem de interrogação a bases de dados, como o SQL; espera-se, todavia, um melhor desempenho da DDS, dado que nesta não tem lugar o processamento necessário à interpretação e optimização das interrogações à base de dados. Em suma, uma DDS situa-se num nível de abstracção intermédio, entre um Sistema de Ficheiros Distribuído e uma Base de Dados Distribuída; e, embora, o interface de acesso à DDS não seja tão flexível como o de linguagens como o SQL, é suficientemente rico para a construção de serviços robustos e sofisticados [GBHC00].

Em complemento à caracterização anterior de Gribble et al. [GBHC00], Martin et al. [MNN01] identificam um conjunto relevante de questões fundamentais a resolver no desenho de DDS(s), designadamente, a necessidade de: 1) desenvolver DDSs com interface de programação familiar (semelhante ao das versões convencionais/centralizadas) e que consigam isolar o programador dos problemas típicos de um ambiente distribuído (*e.g.*, falha ou indisponibilidade de nós/serviços); 2) identificar os serviços básicos/nucleares (*e.g.*, protocolos de gestão de grupos, de consistência, etc.), necessários à construção de DDSs, bem como de descortinar os mecanismos necessários para a partilha correcta desses serviços por múltiplas DDSs; 3) desenvolver mecanismos de balanceamento dinâmico capazes de lidar simultaneamente com múltiplas DDSs, de forma a evitar decisões de balanceamento contraditórias que causem instabilidade no ambiente de exploração; 4) suportar uma plataforma de computação dinâmica, na qual é possível acrescentar/retirar (administrativamente) nós à/da DDS; 5) suportar mecanismos de *checkpointing* (para maior fiabilidade) e de desactivação/reactivação (por conveniência); 6) suportar alguma variedade de requisitos aplicacionais (diferentes níveis de persistência, consistência, etc.). Em particular, Martin et al. [MNN01] reforçam a dificuldade que a resolução de todas estas questões poderá ter num quadro de operação integrada e cooperativa com múltiplas DDSs.

2.3.1 Estruturas de Dados Distribuídas e Escaláveis (SDDSs)

Um sistema diz-se *escalável* se a adição de mais recursos ao sistema tem como efeito o aumento linear (aproximadamente) do seu desempenho. Dito de outra forma, se os recursos associados duplicarem, o desempenho do sistema também deve duplicar. À luz desta definição, uma *estrutura de dados escalável* cumprirá os seguintes requisitos [Kar97]: 1) o tempo de acesso aos registos, para operações nucleares (escrita, leitura, remoção) é constante, *i.e.*, independente do número de registos armazenados pela estrutura; 2) a estrutura comporta qualquer quantidade de dados, sem degradação de desempenho; 3) a expansão ou contracção da estrutura ocorre incrementalmente, sem necessidade de reorganização total, e suportando o acesso contínuo (*online*) dos clientes aos registos.

O conceito de SDDSs foi introduzido no contexto da abordagem LH* [LNS93a] ao Hashing Dinâmico Distribuído; abordagens que se lhe seguiram, como a EH* [HBC97] e a DDH [Dev93], são também classificáveis de SDDSs (a secção 2.4 descreve as três abordagens). Uma SDDS é uma Estrutura de Dados Distribuída (DDS) que respeita três propriedades fundamentais [LNS93a]: P1) a expansão da DDS para novos nós deve acontecer suavemente (de forma graciosa) e apenas quando os nós actuais estiverem eficientemente ocupados; P2) o processo de localização dos registos não deve depender de informação centralizada; P3) as primitivas de acesso e gestão da DDS não devem necessitar da actualização simultânea (atómica) de informação de estado em múltiplos nós (servidores e clientes).

A primeira propriedade parte do princípio de que a utilização inicial de demasiados nós pode ser contra-producente para o desempenho da SDDS. Assim, dado que i) o número ideal de nós pode ser desconhecido à partida, e ii) esse número ideal pode até ser dinâmico, então convém que a SDDS disponha de um mecanismo de expansão/contracção graciosa.

O cumprimento do segundo requisito (propriedade P2) é essencial por diversas razões: i) melhora a fiabilidade da DDS, ao evitar "pontos únicos de falha"; ii) melhora o desempenho da DDS, ao evitar "pontos quentes". O processo de localização dos registos da DDS deixa assim de ficar dependente da disponibilidade de um único serviço/nó, bem como da sua capacidade de processamento, necessariamente limitada (e portanto limitante do desempenho global da DDS). No que toca a este requisito (P2), Karlson [Kar97] fornece uma descrição mais precisa. Assim: a) a DDS não deve depender de uma *directoria* centralizada; b) cada cliente da DDS deve manter uma imagem o mais actual possível da distribuição da DDS, devendo essa imagem ser corrigida sempre que há erros de endereçamento; c) quando há um erro de endereçamento, a DDS deve i) efectuar o redireccionamento das mensagens para o destino correcto e ii) garantir a correcção da imagem do cliente.

A terceira propriedade (P3) está relacionada com a necessidade de evitar que clientes e servidores da DDS tenham de se conhecer (na totalidade) uns aos outros. Se não for possível respeitar esse requisito, a escalabilidade da DDS poderá ser seriamente comprometida.

2.4 DHTs de Primeira Geração

Nesta secção revêem-se abordagens seminais que entabularam a nossa investigação: DHTs de uma primeira geração, baseadas em Hashing Dinâmico Distribuído e concebidas para

armazenamento distribuído em ambiente *cluster*; neste contexto, a arquitectura dessas DHTs assume [Abe01] em geral a) um ambiente de execução fiável, b) um número pequeno (face a padrões actuais) de nós, c) garantias de execução determinísticas e, por vezes, d) certos serviços centralizados. Na sua maioria, as abordagens representam uma evolução de antecessoras não-distribuídas, resumidas na secção B.4 do apêndice B, a ter presente.

Convém ainda vincar, antes de prosseguir, diferenças fundamentais entre a primeira e uma segunda geração de DHTs, mais recente, que emergiu no contexto dos sistemas *Peer-to-Peer*. Assim, nas DHTs de primeira geração, a preocupação fundamental é a rentabilização dos recursos de armazenamento dos nós: as DHTs tendem a ser criadas com base num conjunto mínimo de nós e a integração/remoção de nós na/da DHT ocorre *progressivamente*, à medida das necessidades efectivas de armazenamento; este processo, que pode ser *automático* ou *administrativo*, é agilizado através da utilização de Hashing Dinâmico.

As DHTs de segunda geração seguem uma filosofia de evolução diferente, suportando a adição/remoção dinâmica de nós por questões que podem ser independentes do armazenamento; com efeito, as primeiras abordagens desta geração foram introduzidas para resolver, essencialmente, problemas de localização distribuída, com base num paradigma de Hashing Consistente, que é essencialmente Estático (o número de bits da função de hash é elevado, mas basicamente fixo). Em suma, nas DHTs de primeira geração "é a DHT que, dinamicamente, procura/liberta nós"; nas de segunda "são os nós que, dinamicamente, entram/saiem da DHT", o que pode ocorrer com frequência elevada; esta filosofia diferente, conjugada com ambientes de operação ortogonais, distancia as duas gerações.

2.4.1 Parâmetros de Caracterização

A nossa descrição das DHTs de primeira geração recorre a parâmetros de caracterização definidos de seguida. Os mesmos parâmetros serão usados numa análise comparativa final.

2.4.1.1 Estratégia de Evolução

Nas abordagens a analisar, a expansão/contracção automática das DHTs pode ser feita de forma a) *síncrona* e *centralizada* (uma entrada/contentor de cada vez, sob coordenação de um nó especial), ou b) *assíncrona* e *distribuída* (várias entradas/contentores em simultâneo, por iniciativa autónoma dos nós da DHT, e sem necessidade de coordenação centralizada).

2.4.1.2 Paridade Funcional dos Nós

Os nós de uma DHT podem ou não ser *pares* entre si nas funções desempenhadas. Por definição, uma Estratégia de Evolução *centralizada* socorre-se de pelo menos um nó *coordenador*. Note-se, porém, que uma Estratégia de Evolução *distribuída* não implica automaticamente a paridade funcional dos nós (ver o parâmetro Simetria do Acesso). Com nós *díspares*, a escalabilidade e a tolerância a faltas das DHTs serão, em geral, mais limitados.

2.4.1.3 Simetria do Acesso à DHT

Se o acesso a uma DHT tiver de atravessar uma primeira linha de nós/serviços especiais (de *frontend*), o acesso é *assimétrico*; se uma entidade cliente da DHT puder contactar directamente qualquer dos nós/serviços da DHT, o acesso é *simétrico*. Em ambos os cenários, o pedido original da entidade cliente terá de ser redireccionado até ao nó/serviço final, responsável, em última instância, pela realização da operação solicitada pelo cliente.

O recurso a nós/serviços de *frontend* é útil, por exemplo, quando se pretende que entidades externas a um *cluster* possam aceder a uma DHT realizada neste; em tal caso, os nós de *frontend* assumem, tipicamente, funções de balanceamento da carga do acesso, redireccionando os pedidos para qualquer um dos nós que sustentam efectivamente a DHT, ou seja, transformando um acesso assimétrico num acesso simétrico. Quando as entidades clientes estão confinadas ao *cluster*, o acesso simétrico é à partida viável e, em geral, suficiente.

2.4.1.4 (Re)Posicionamento das Entradas

Numa DHT, o *posicionamento* de uma entrada/contentor refere-se à definição do nó hospedeiro correspondente, ou seja, à definição de uma correspondência "entrada $\mapsto$ nó".

As correspondências poderão ser *fixas* ou *mutáveis*; no primeiro caso, a associação de uma entrada a um nó é permanente, durante a vida da DHT; no segundo, a entrada pode ser re-atribuída a outro nó, *e.g.*, em resultado i) de mecanismos de balanceamento dinâmico de carga ou ii) por razões administrativas. Sob Hashing Dinâmico, o número de correspondências "entrada $\mapsto$ nó" é variável, pois o número de entradas da DHT é variável.

2.4.1.5 Localização das Entradas

A *localização* de uma entrada refere-se à reconstituição da correspondência "entrada $\mapsto$ nó". Neste contexto, os mecanismos de *posicionamento* e *localização* terão de ser compatíveis.

Assim, o conjunto global das correspondências "entrada $\mapsto$ nó" define a *informação de posicionamento/localização*[1] de uma DHT, podendo ser registada de forma i) *centralizada* ou ii) *distribuída*. O registo centralizado corresponde à manutenção e exploração de uma tabela global, armazenada num só nó, bem conhecido. O registo distribuído pode seguir diversas modalidades: i) replicação na totalidade, ii) replicação parcial[2] ou iii) particionamento[3]. A distribuição permite balancear a carga de acesso à informação e os recursos necessários ao seu armazenamento. O particionamento é a variante mais escalável (na carga de acesso e na quantidade de informação por nó) mas exige um mecanismo de localização apropriado, que também deverá ser escalável, incluindo o número de mensagens necessárias para descobrir uma correspondência "entrada $\mapsto$ nó"; concretamente, esse mecanismo considera-se escalável se não necessitar de visitar mais de $log_2\mathcal{N}$ nós (sendo $\mathcal{N}$ o número total de nós da DHT), para reconstituir uma correspondência "entrada $\mapsto$ nó".

[1] As formas *informação de posicionamento* e *informação de localização* são usadas de forma indistinta.
[2] O que admite sobreposições, entre diferentes réplicas.
[3] O que implica a divisão em *partições* (porções mutuamente exclusivas).

2.4.1.6 Suporte a Distribuições Heterogéneas

Uma característica dos *clusters* é a relativa homogeneidade do *hardware* e do sistema de exploração dos seus nós. Todavia, a utilização partilhada dos nós por diversas aplicações e/ou utilizadores, provoca flutuações dinâmicas na disponibilidade dos seus recursos. Assim, sob o ponto de vista de uma aplicação distribuída, como é o caso de uma DHT, é desejável poder ajustar dinamicamente a contribuição de cada nó para essa aplicação.

O Suporte a Distribuições Heterogéneas de DHTs assenta, em primeiro lugar, na possibilidade de definir, de forma independente e pesada, para cada nó, o seu *número* de entradas da DHT. Idealmente, essa definição deve poder ser dinâmica, acompanhando de perto a evolução das condições do ambiente de execução. O suporte à (re)definição dinâmica da *posição* das entradas representa um outro nível, acrescido, de ajustamento a essa evolução.

2.4.1.7 Tolerância a Faltas

A administração centralizada, o relativo isolamento e a utilização de mecanismos de alimentação eléctrica redundante fazem dos *clusters* ambientes estáveis que não são, porém, imunes a faltas. Neste contexto, se for necessário garantir uma elevada disponibilidade dos registos armazenados numa DHT, o recurso à replicação dos registos é uma opção. Esta, conjugada com outras (*e.g.*, *checkpointing*), poderá garantir elevada tolerância a faltas.

2.4.1.8 Suporte a Distribuições Não-Uniformes

Idealmente, a função de hash de uma DHT é capaz de dispersar uniformemente os registos inseridos,pelo seu contra-domínio. Se isso não acontecer, a DHT deve dispor de mecanismos capazes de continuar a assegurar uma utilização eficiente dos recursos de armazenamento, em tais circunstâncias. Neste contexto, é particularmente útil a possibilidade dos contentores se subdividirem de forma independente. Assim, o Suporte a Distribuições Não-Uniformes está directamente associado ao tipo de Estratégia de Evolução prosseguida.

2.4.1.9 Suporte à Operação de Múltiplas DHTs

A operação de múltiplas DHTs pode caracterizar-se por diferentes níveis/gradações de integração. A classificação seguinte traduz uma perspectiva pessoal dessa problemática.

Assim, a um nível mais *básico*, cada DHT é suportada por um conjunto de serviços exclusivamente dedicados à DHT; tais serviços não necessitam de conhecer a existência de outras DHTs e seus serviços; para que as aplicações possam aceder, de forma separada, a cada DHT, é necessário o equivalente a um *directório* ou *serviço de directoria* que forneça pontos (serviços) de entrada nas DHTs e identificadores globais únicos para as DHTs.

Num nível *intermédio*, um serviço pode suportar várias DHTs (que partilham os recursos dos nós hospedeiros), mas ainda não existe balanceamento de carga entre os serviços.

A um nível mais *avançado*, os serviços são capazes de cooperar entre si, no âmbito de um

mecanismo de balanceamento global, que procura assegurar níveis de QoS a cada DHT em particular e, ao mesmo tempo, optimizar a utilização dos recursos do *cluster* em geral.

2.4.2 Hashing Linear Distribuído (LH*)

O Hashing Linear Distribuído (LH*) [LNS93b, LNS93a] representa a versão distribuída do Hashing Linear de Litwin (LH) [Lit80] (ver secção B.4.3). Na abordagem LH a tabela de hash regista uma evolução linear: cresce/decresce por adição/remoção de um contentor de cada vez, sendo os contentores alinhados contiguamente no suporte de armazenamento; esse alinhamento permite um acesso directo e dispensa uma *directoria*; essa dispensa não é viável na abordagem LH*, pois é necessário registar correspondências "contentor $\mapsto$ nó".

Numa primeira versão [LNS93b], o LH* assegura o crescimento suave e incremental da DHT, um contentor de cada vez, mas depende de um serviço *coordenador*, designado por *split coordinator*. Numa segunda versão [LNS93a, LNS96], o LH* dispensa essa entidade, mas à custa de surtos de operações de subdivisão de contentores (*cascading splits*), como forma de prevenção da sua sobrecarga, o que configura uma evolução intermitente da DHT.

Para a gestão das correspondências "contentor $\mapsto$ nó" são propostos dois mecanismos: a) o primeiro, compatível com ambas as versões do LH*, é baseado numa tabela total/estática que identifica todos os nós do *cluster* e é replicada em todos os nós da DHT; a tabela é usável por qualquer nó para posicionar/localizar da mesma forma qualquer entrada da DHT[4]; b) o segundo é baseado numa tabela incremental/dinâmica, mantida pelo *split coordinator* e actualizada durante a subdivisão de contentores; neste caso, cada nó escolhe autonomamente o nó com quem vai dividir o contentor e informa o *coordenador* da escolha; apenas o *coordenador* necessita de manter a totalidade das correspondências "contentor $\mapsto$ nó"; os restantes servidores mantêm apenas as relativas aos contentores que criaram. Os dois esquemas de endereçamento permitem aceder a um contentor com um número inferior de mensagens (3), face ao número logarítmico de abordagens como a DDH e EH*.

Cada nó da DHT aloja um só contentor, não existindo suporte a distribuições heterogéneas.

2.4.3 Hashing Dinâmico Distribuído de Devine (DDH)

O Hashing Dinâmico Distribuído de Devine (DDH) [Dev93] é uma versão distribuída do Hashing Dinâmico de Larson [Lar78] (ver secção B.4.1). Basicamente, na aproximação não-distribuída, os contentores (*buckets*) da tabela de hash são endereçados com base numa *trie* de bits; o endereço (lógico) de um contentor é definido pela sequência de bits da raiz da *trie* até uma certa folha. A *trie* permite assim que cada contentor seja endereçado com um número de bits específico, que cresce/decresce quando o contentor se divide/funde. A divisão/fusão de um contentor é uma decisão que depende apenas da sua taxa de ocupação.

Na abordagem DDH, a transposição do esquema anterior para um ambiente distribuído, em que um nó é responsável por um ou mais contentores, permite que cada nó tome decisões autónomas sobre a evolução dos contentores à sua guarda. Por sua vez, esta autonomia

[4]Por exemplo, o nó a atribuir à h'ésima entrada da DHT poderá ser o h'ésimo nó da tabela.

permite que os nós sejam pares em termos funcionais e que a DHT possa evoluir através de múltiplas operações de criação/fusão de contentores, de forma assíncrona e em paralelo.

Exige-se, todavia, que cada nó da DHT conheça a lista (um conjunto ordenado) de todos os nós *potenciais* da DHT (e não apenas a lista dos nós *actuais*). Este requisito é imposto pelo recurso a um esquema determinístico de distribuição de contentores – ver a seguir.

As correspondências "contentor $\mapsto$ nó" são definidas através de uma política Round-Robin (RR), que afecta novos contentores a todos os nós potenciais, de forma rotativa. Aplicada de forma independente por cada servidor, essa política resulta numa distribuição homogénea da DHT pelos seus nós. Uma política RR pesada permitiria realizar uma distribuição heterogénea, mas de tipo estático, uma vez que a função de distribuição é fixa.

Inicialmente, clientes e servidores conhecem apenas o responsável pelo contentor 0 (zero), para quem todos os pedidos de acesso à DHT são direccionados; à medida que os erros de endereçamento vão sendo corrigidos, clientes e servidores constroem uma visão parcial da distribuição da DHT; a informação de posicionamento mantida é suficiente para garantir que qualquer erro de endereçamento pode ser corrigido visitando, no pior caso, os nós responsáveis por $log_2 \# B$ contentores, para um número actual $\# B$ de contentores da DHT.

Devine argumenta ainda que, embora a sua abordagem seja realizável sobre conexões WAN, estas são inadequadas às operações de divisão/fusão de contentores, que podem movimentar grandes quantidades de dados. Este argumento posiciona claramente, no território do ambiente *cluster*, as DHTs usadas para armazenamento distribuído, em contraste com DHTs de segunda geração, usadas para localização distribuída em ambientes P2P.

2.4.4 Hashing Extensível Distribuído (EH*)

O Hashing Extensível Distribuído (EH*) [HBC97] é uma versão distribuída do Hashing Extensível (EH) [FNPS79] (ver secção B.4.2). Na abordagem EH, os contentores são endereçados com base numa tabela, designada por *directoria*, que faz a correspondência entre os endereços lógicos (sequência de bits) dos contentores e os seus endereços físicos (em Disco ou RAM). Todos os endereços lógicos usam, num primeiro nível, o mesmo número de bits, número que define a profundidade (*split-level*) global da tabela. É ainda possível que vários endereços lógicos correspondam ao mesmo endereço físico, de um mesmo contentor, o que significa que apenas uma parte dos bits dos endereços lógicos é relevante; o número desses bits define a profundidade local do correspondente contentor. A profundidade global da *directoria* poderá crescer, à medida que os contentores esgotam a sua capacidade, sendo necessário criar novos contentores, e portanto deitar mão de mais bits para endereçá-los sendo que, por cada bit adicional, o número de entradas da *directoria* será duplicada.

Uma transposição directa do esquema de endereçamento anterior, para um ambiente distribuído, no qual cada nó participante aloja um ou mais contentores, exigiria réplicas da *directoria* em cada nó. Esta abordagem simples teria pelo menos dois inconvenientes: i) necessidade de sincronização das réplicas; ii) desperdício de recursos de armazenamento, representado pelas entradas, em cada *directoria*, que referenciam o mesmo contentor. A abordagem EH* procura precisamente combater estes problemas, através de *CacheTables*.

Assim, cada nó mantém uma *CacheTable*, com um subconjunto das correspondências entre endereços lógicos e físicos (estes correspondem agora à identificação do nó que aloja o contentor). Essa colecção fornece uma visão parcial (e possivelmente desactualizada) da evolução da DHT mas, tal como na abordagem DDH, é suficiente para assegurar o encaminhamento dos pedidos de acesso aos contentores da DHT, para os nós correctos, em não mais de $log_2 \#B$ passos, sendo $\#B$ o número total de contentores da DHT. Mais especificamente [HBC97]: uma *CacheTable* usa uma gama de profundidades globais; quando se tenta associar o endereço lógico do contentor a um nó recorre-se, em primeiro lugar às entradas da tabela com a maior profundidade; havendo erros de endereçamento recorre-se, sucessivamente, a entradas com menor profundidade; a *CacheTable* acaba por ser actualizada com a informação resultante da aprendizagem de novas correspondências, o que configura um mecanismo do tipo *lazy update*, similar ao usado pela abordagem DDH.

A localização de contentores com *CacheTables* fornece a flexibilidade necessária para que cada nó possa tomar, de forma autónoma, decisões sobre a sua evolução (divisão/fusão). Consequentemente, os nós são *pares* em termos funcionais e a DHT suporta múltiplas operações sobre os contentores, despoletáveis assíncronamente e realizáveis em paralelo.

Na descrição original desta abordagem [HBC97] não são especificados os critérios que definem a escolha de um nó computacional, em concreto, para alojar um certo contentor. À partida, podem ser usados esquemas similares aos definidos pela abordagem DDH. De igual forma, o suporte à possível heterogeneidade dos nós também não é discutido, mas os esquemas utilizados pela abordagem DDH parecem ser, mais uma vez, compatíveis.

2.4.5 Hashing Dinâmico Distribuído de Gribble

Gribble [GBHC00] propõe uma abordagem a DHTs especialmente orientadas ao suporte de serviços Internet (*e.g.*, servidores/*caches* WWW, servidores de ficheiros/aplicações, etc.). Os clientes são assim entidades externas ao *cluster*, que acedem à(s) DHT(s) através de nós/serviços de *frontend*; estes redireccionam os pedidos para serviços de armazenamento, designados por *storage bricks*[5], que embora possam co-existir com os de *frontend*, residirão tipicamente em nós separados, de *backend*. O acesso à DHT é, portanto, assimétrico e a configuração típica assenta na disparidade funcional dos nós que suportam a(s) DHT(s).

O reforço da capacidade global de armazenamento é uma operação administrativa, que consiste na criação de novos *storage bricks*, em nós do *cluster* disponibilizados para o efeito. Um *storage brick* é partilhável por várias DHTs e tem uma capacidade limitada. Quando essa capacidade esgota, é necessário movimentar parte dos registos de uma ou mais DHTs para outro *storage brick* e actualizar a *directoria* da(s) DHT(s) de forma apropriada.

O mecanismo de distribuição aproxima-se do usado pelas abordagens DDH e EH*, no sentido de que, tal como nestas, a base é a existência de uma *directoria* que regista as correspondências "entrada $\mapsto$ nó"; todavia, ao contrário do que se passa nessas abordagens, na de Gribble existe a preocupação de manter uma réplica total (e o mais actualizada possível) da *directoria*, em cada nó de *backend*. Os nós de *frontend* também guardam réplicas da *directoria*, mas a sua actualização rege-se por mecanismos do tipo *lazy*.

[5] Capazes de suportarem, de forma individualizada, o armazenamento de registos de DHTs diferentes.

A abordagem prosseguida recorre à replicação[6] dos registos como forma de aumentar i) a fiabilidade e ii) o desempenho dos acessos de leitura (os predominantes, no cenário alvo). A utilização de replicação tem reflexos importantes nos mecanismos de distribuição usados. Mais especificamente, a *directoria* desdobra-se em duas estruturas de dados: a) uma *trie* que define os hashes válidos actuais da DHT; b) uma tabela que define, para cada hash, a lista dos *storage bricks* que armazenam as réplicas dos registos associados ao hash.

Embora de forma estática, é suportada uma forma rudimentar de distribuição heterogénea: o número de *storage bricks* criados em cada nó do *cluster* é igual ao seu número de CPUs.

A abordagem oferece um suporte *intermédio* (rever secção 2.4.1.9) a múltiplas DHTs: são utilizados identificadores específicos para cada DHT e cada *storage brick* é capaz de armazenar registos de DHTs diferentes. Suporta ainda a durabilidade das DHTs, através de operações de *checkpointing*, desencadeadas administrativamente, de forma síncrona.

2.4.6 Análise Comparativa

Abordagens / Parâmetros	**LH***	**DDH**	**EH***	**Gribble**	**Domus**
Estratégia de Evolução	síncrona [LNS93b]; assíncrona[LNS93a];	assíncrona (autónoma)	assíncrona (autónoma)	síncrona (administrativa)	assíncrona (coordenada)
Paridade Funcional dos Nós	não [LNS93b]; sim [LNS93a]	sim	sim	não	flexibilidade funcional[a]
(Simetria do) Acesso à DHT	simétrico	simétrico	simétrico	assimétrico	simétrico
Posicionamento das Entradas	fixo (inicial ordenado)	fixo (inicial rotativo)	fixo (indefinido)	fixo (indefinido)	dinâmico (migração)
Localização das Entradas	distribuída; info. total ou parcial/ replicada	distribuída; info. parcial/ replicada	distribuída; info. parcial/ replicada	distribuída; info. total/ replicada	distribuída; informação particionada
Distribuições Heterogéneas	suporte ausente	suporte ausente	suporte ausente	suporte parcial	suporte avançado
Tolerância a Faltas	suporte ausente	suporte ausente	suporte ausente	replicação de registos	suporte ausente[b]
Distribuições Não-Uniformes	suporte parcial[c]	suportadas	suportadas	suportadas	suporte ausente[d]
Múltiplas DHTs	suporte ausente	suporte ausente	suporte ausente	suporte intermédio	suporte avançado

Tabela 2.1: Comparação de Abordagens ao Hashing Dinâmico Distribuído.

[a] A abordagem explora o facto de os nós poderem assumir diferentes funções, de forma dinâmica.
[b] Mas facilmente conseguido explorando o tipo de *checkpointing* realizado em operações de desactivação.
[c] A subdivisão por uma ordem fixa/linear pode atrasar a subdivisão de contentores em sobrecarga.
[d] Mas facilmente conseguido através de *fragmentação* dos registos – ver secção 6.10.

A tabela 2.1 sintetiza a caracterização das abordagens ao Hashing Dinâmico Distribuído

[6] Não são definidos, todavia, os critérios para o estabelecimento do número de réplicas, nem os que presidem à selecção dos *storage bricks* que vão armazenar as réplicas.

que acabamos de descrever, com base nos parâmetros definidos na secção 2.4.1. A tabela inclui também a caracterização da abordagem Domus, desenvolvida ao longo desta dissertação, permitindo desde logo confrontá-la com as restantes abordagens e antecipar a aproximação utilizada à resolução de certos problemas. Assim, considerando apenas as abordagens de referência anteriores (LH*, DDH, EH* e Gribble), pode-se concluir que:

- a Estratégia de Evolução maioritária é a assíncrona, potencialmente mais escalável;
- a Paridade Funcional dos Nós da DHT é suportada pela maioria das abordagens;
- o Acesso simétrico à DHT, mais escalável, predomina na maioria das abordagens;
- o Posicionamento das Entradas é fixo (ou pouco flexível) em todas as abordagens;
- as abordagens DDH e EH* prosseguem as aproximações mais escaláveis à gestão da Localização de Entradas (mas o custo, em mensagens, da LH*, é menor);
- só a abordagem de Gribble fornece suporte (estático) a distribuições heterogéneas;
- apenas a abordagem de Gribble suporta um nível mínimo de tolerância a faltas;
- o suporte a Distribuições Não-Uniformes é generalizado, mas com limitações na LH*;
- apenas a abordagem de Gribble oferece suporte (*intermédio*) a múltiplas DHTs.

Em resumo, das abordagens analisadas, nenhuma consegue reunir, em simultâneo, todas as propriedades desejáveis, tendo em conta os parâmetros de caracterização escolhidos, emboras as abordagens DDH e EH* pareçam ser as mais satisfatórias, em termos globais.

Finalmente, a comparação anterior faz emergir as motivações essenciais da nossa investigação, relacionadas com a ausência de abordagens capazes de dar resposta satisfatória i) à heterogeneidade do ambiente de execução e ii) à problemática da operação conjunta de DHTs. Precisamente, contrapondo a abordagem Domus com as restantes da tabela, as mais valias principais da nossa residem no suporte *avançado* à resolução desses dois problemas. Essa resolução beneficia também de outras qualidades, como a *flexibilidade funcional* dos nós, a simetria do acesso às DHTs e a opção por uma estratégia *distribuída* e *particionada* para a gestão da localização de entradas. Por outro lado, uma estratégia de evolução centralizada (dependente de um serviço coordenador, como forma de atingir um balanceamento mais preciso) e uma menor tolerância a faltas (em boa parte devido à estratégia de evolução centralizada), representam pontos menos fortes do Domus.

2.5 DHTs de Segunda Geração

A necessidade de esquemas de localização (distribuída) de objectos, adequados às especificidades dos sistemas *Peer-to-Peer* (P2P) [MKL+02, BKK+03, LCP+04, SAB+05, SW05], fez emergir uma segunda geração de DHTs, de que se podem considerar fundadores abordagens como o Chord [SMK+01], o CAN [RFH+01], o Pastry [RD01] e o Tapestry [ZKJ01].

Assim, em ambientes P2P *estruturados*, a localização de objectos socorre-se de uma topologia (um grafo) de nível aplicacional (*overlay network*), sobreposta ao encaminhamento IP convencional, unindo as diversas partes (*partições*) de uma DHT; a DHT é utilizada para guardar a localização de objectos (*i.e.*, usada como mecanismo de indirecção) ou para guardar os próprios objectos (*i.e.*, usada como sistema de armazenamento distribuído).

As DHTs de segunda geração são portanto orientadas a um domínio de aplicabilidade bastante diferente do das suas precursoras de primeira geração, caracterizado por i) número potencialmente muito elevado de nós, ii) elevado dinamismo na composição do sistema (em nós e objectos), iii) ambiente de operação propício a desconexões dos nós / falhas dos canais de comunicação, iv) elevada variabilidade de largura de banda, vi) elevada heterogeneidade dos nós, vii) eventuais requisitos de segurança/privacidade das interacções, etc. Em geral, estas propriedades/requisitos esbatem-se (ou assumem valor dual) nos ambientes do tipo *cluster*/NOW, que contextualizaram a concepção das abordagens de primeira geração, e que representam, precisamente, o tipo de ambiente ao qual se orientou a nossa investigação.

Actualmente, contudo, o conceito de DHT continua intimamente ligado aos sistemas P2P. Usado, ao nível mais fundamental, para localização distribuída, o "paradigma DHT" orientou aplicações em áreas tão diversas como i) sistemas de armazenamento/ficheiros distribuídos (*e.g.*, OceanStore [KBCC00], PAST [DR01a], CFS [DKKM01], Ivy [Mut02]) ii) partilha de ficheiros (*e.g.*, Azureus [azu, FPJ+07], eMule [emu, SBE07]), iii) *web caching* e distribuição de conteúdos e (*e.g.*, Squirrel [IRD02], Codeen [WPP+04], Coral [FFM04]), v) indexação e pesquisa da *Web* (*e.g.*, Herodotus [Bur02], Apoidea [SSLM03], [ZYKG04]).

No âmbito desta dissertação, a importância da segunda geração de DHTs reside, essencialmente, na possibilidade de confronto com e/ou de reutilização das suas aproximações à resolução de questões fulcrais como i) *particionamento*, ii) *posicionamento* e iii) *localização*. Assim, no capítulo 3, teremos oportunidade de confrontar a nossa própria estratégia de particionamento com a estratégia do Hashing Consistente [KLL+97], substrato de muitas das DHTs de segunda geração. Posteriormente, no capítulo 4, acoplamos mecanismos de localização distribuída baseados na abordagem Chord [SMK+01], aos mecanismos de particionamento do capítulo 3, e desenvolvemos mecanismos de posicionamento compatíveis.

Ao contrário da opção seguida para as DHTs de primeira geração, não se apresenta, neste capítulo, um estudo comparativo, auto-contido, das DHTs de segunda geração (consultar, para o efeito, documentos de referência da área [BKK+03, LCP+04, SAB+05]). Em alternativa, as principais soluções preconizadas à resolução das questões fundamentais acima referidas (particionamento e localização) serão apresentadas ao longo das secções seguintes.

2.6 Estratégias de Particionamento

No contexto de uma DHT, *particionamento* refere-se à subdivisão completa do contradomínio H da função de hash, em subconjuntos disjuntos, designados por *partições*. Uma partição é pois um subconjunto de hashes de H (ou, equivalentemente, de entradas da DHT); esses hashes não são necessariamente contíguos, *i.e.*, uma partição não é necessariamente um subintervalo de H. O resultado do particionamento é a atribuição de um certo número de partições, com um certo número de hashes concretos, a cada nó da DHT.

Se a definição do *número* e da *identidade* dos hashes de um nó forem processos independentes, então faz sentido falar em estratégias de *distribuição* e *posicionamento*, respectivamente; uma estratégia de *posicionamento* implica a opção por uma certa forma de *registo* das correspondências "hash $\mapsto$ nó", com influência na estratégia de *localização* a utilizar.

2.6.1 Particionamento em DHTs de Primeira Geração

Nas DHTs de primeira geração, o recurso a Hashing Dinâmico viabiliza uma Estratégia de Evolução que, como antes se referiu (ver secção 2.4.1.1), permite o crescimento/decrescimento incremental do número de entradas/contentores da DHT de forma que, num determinado instante, a) o número de contentores corresponde ao mínimo absolutamente necessário e b) poderão co-existir contentores identificados por hashes com diferente número de bits (como resposta a distribuições não-uniformes de registos – rever secção 2.4.1.8).

O domínio de particionamento (*i.e.*, o conjunto global de hashes/entradas) é portanto dinâmico, acompanhando as necessidades de armazenamento da DHT. Além disso, a maioria das abordagens estudadas na secção 2.4 suportam apenas distribuições homogéneas, assentes na atribuição de uma só entrada/contentor a cada nó (rever secção 2.4.1.6). Em contraponto, as DHTs de segunda geração recorrem tipicamente a Hashing Estático[7], na forma de Hashing Consistente (ver a seguir) e o particionamento do contradomínio da função de hash resulta na atribuição de várias entradas (contíguas ou não) a cada nó da DHT (independentemente da distribuição de registos pelas entradas ser uniforme ou não).

2.6.2 Particionamento em DHTs de Segunda Geração

2.6.2.1 Hashing Consistente Original

A abordagem do Hashing Consistente (HC) [KLL+97] foi inicialmente aplicada em balanceamento de carga para Web Caching cooperativo [KSB+99]. Posteriormente, viria a ser adoptada (e adaptada) pelo Chord [SMK+01], uma abordagem de localização distribuída de objectos, baseada no paradigma DHT, concebida para ambientes Peer-to-Peer (P2P)[8].

Sob Hashing Consistente, o contradomínio H de inteiros, de uma função de *hash* estática[9], é aplicado no intervalo circular $[0, 1)$ de reais, orientado no sentido dos ponteiros do relógio. Depois, a cada nó da DHT, faz-se corresponder um ou mais pontos do intervalo $[0, 1)$, de forma que este fica particionado em sub-intervalos delimitados por aqueles pontos. Um nó da DHT será então responsável pelas partições de $[0, 1)$ que terminam nos seus pontos.

Atribuindo-se apenas uma ponto/partição do círculo $[0, 1)$ a cada nó da DHT, a distribuição terá um desequilíbrio de ordem $O(log\mathcal{N})$ [KLL+97], face a uma distribuição *homogénea*, sendo $\mathcal{N}$ o número total de nós da DHT[10]. Complementarmente, se forem atribuídos $k \times log_2\mathcal{N}$ pontos/partições, a distribuição será tanto mais *homogénea* quanto maior for k.

É importante realçar que a criação de partições em número suficientemente elevado permite

[7] Usando-se funções de hash suficientemente prolíficas, como SHA-1 [sha95], de 160 bits.
[8] Na realidade, boa parte das DHTs para esses ambientes assentam em variantes do Hashing Consistente.
[9] Mas com um número elevado de bits (160 bits, na definição original).
[10] Note-se que o número de pontos de cada intervalo é aleatório.

não só uma distribuição de H relativamente homogénea, como ainda suporta convenientemente distribuições não-uniformes de registos da DHT, situação em que estes tendem a concentrar-se num número reduzido de partições. Se as partições forem pequenas e suficientemente dispersas pelos nós computacionais, então é maior a probabilidade de uma distribuição não-uniforme de registos ser afectada de forma uniforme aos nós da DHT[11].

A atribuição de múltiplos pontos/partições do círculo $[0, 1)$ a cada nó computacional socorre-se de um simples artifício conceptual: é como se, por cada nó computacional, houvesse múltiplos *nós virtuais*, com pseudo-identificadores deriváveis do identificador do nó, aos quais a aplicação da função de hash permite associar outros tantos pontos de $[0, 1)$.

A grande vantagem do Hashing Consistente é o impacto limitado e distribuído da junção/remoção de um nó computacional à DHT: por cada nó virtual v desse nó, é apenas preciso trocar registos com o sucessor de v no círculo (no sentido dos ponteiros do relógio). Basicamente, é esta a propriedade que qualifica como *consistente* a abordagem em causa.

2.6.2.2 Hashing Consistente Pesado

Originalmente, o Hashing Consistente e o Chord são orientados a distribuições homogéneas. Uma extensão óbvia a distribuições heterogéneas é a definição independente do número de nós virtuais por cada nó, proporcionalmente a uma certa capacidade (estática ou dinâmica) do nó. A aplicação dessa abordagem é ilustrada no CFS [DKKM01, Dab01], um Sistema de Ficheiros Distribuído (*Write-once-Read-many*), para ambientes WAN/P2P.

Embora permitam distribuições balanceadas, múltiplos nós virtuais por nó computacional são inadequados se o número de nós computacionais for muito elevado, pelo efeito multiplicativo sobre a quantidade de informação de localização necessária (complexidade espacial acrescida), e sobre o esforço necessário à localização (complexidade temporal acrescida). Neste âmbito, Karger et al. [KR04], introduzem uma abordagem que suporta vários nós virtuais por nó, mas apenas permite a "activação" de um nó virtual de cada vez, escolhido no quadro de uma estratégia de balanceamento dinâmico de carga (ver secção 2.8.2).

Posteriormente, Godfrey et al. [GS05] propõem a abordagem Y_0 (derivada do Chord) para distribuições heterogéneas, assente na concentração dos nós virtuais (ou melhor, das partições) de cada nó numa zona contígua de $[0, 1)$, pressupondo uma distribuição uniforme de carga[12]; dessa forma, a complexidade espacial e temporal da localização continua a ser de ordem $O(log\mathcal{N})$, mesmo com $O(log\mathcal{N})$ nós virtuis por nó (com nós virtuais dispersos, a complexidade seria $O(log^2\mathcal{N})$). Adicionalmente, observam que em distribuições heterogéneas, o esforço de localização é inferior (<50%) ao de distribuições homogéneas.

Schindelhauer et al. [SS05] revisitam o Hashing Consistente para distribuições heterogéneas, propondo um *Método Linear* e um *Método Logarítmico* de cálculo de distâncias no círculo $[0, 1)$ que leva em conta um certo peso/capacidade de um nó computacional, quando se calcula a distância do seu ponto no círculo a um outro ponto qualquer. Ambos os métodos assentam na criação de $O(log\mathcal{N})$ partições de $[0, 1)$, por cada nó da DHT (para

[11] Este raciocínio parte do princípio de que a dimensão dos registos é relativamente uniforme.
[12] Requisito desnecessário quando a partições associadas aos nós virtuais são dispersas em $[0, 1)$.

um total de $\mathcal{N}$ nós) para se atingir uma distribuição equilibrada, sendo que o *Método Logarítmico* oferece melhor *qualidade da distribuição*. O *Método Linear* tem a particularidade de ter sido aplicado, com sucesso, no encaminhamento em redes sem fios *ad-hoc* [SBR04].

2.6.3 Particionamento para SANs

A definição de estratégias de particionamento adequadas representa um problema também recorrente em outras áreas, como acontece no contexto das SANs (*Storage Area Networks*).

Brinkman [BSS00], por exemplo, investigou o particionamento de dados para cenários homogéneos (discos de igual capacidade) e heterogéneos (discos de capacidade arbitrária), apresentando estratégias caracterizadas por: 1) boa *qualidade da distribuição* (distribuição o mais equitativa/proporcional possível); 2) *localização* eficiente (no tempo e no espaço) dos dados; 3) boa *adaptabilidade* à adição/remoção de discos ou blocos de dados[13].

A abordagem de Brinkman [BSS00] partilha semelhanças com o Hashing Consistente (HC) original [KLL$^+$97]: sendo $U = \{1, ..., p\}$ os inteiros sequenciais disponíveis para indexar blocos de dados (com p arbitrariamente grande), o recurso a uma função de hash permite estabelecer a correspondência entre elementos de U e o intervalo de reais $[0, 1]$; num dado instante, apenas um subconjunto dos índices de U corresponderá a blocos efectivamente instanciados; uma *estratégia de assimilação* permite particionar $[0, 1]$ em intervalos contíguos e atribuí-los a discos da SAN de forma a cumprir os requisitos acima enunciados.

As estratégias de assimilação propostas usam algoritmos determinísticos de movimentação de blocos, que permitem reconstruir todas as movimentações efectuadas desde um estágio inicial, de forma a calcular-se a localização actual de um bloco. Desta forma, evita-se o recurso a uma tabela de localização dos blocos que, em ambiente SAN (com múltiplos discos, com muitos blocos cada), poderia ter dimensões enormes. Porém, as estratégias propostas implicam a participação de todos os discos da SAN em qualquer adição/remoção.

Em comparação com a abordagem do Hashing Consistente (HC) original [KLL$^+$97], a abordagem de Brinkman [BSS00] incorre num custo temporal de localização de ordem $O(log(n))$ (no número de blocos) face ao custo de ordem $O(1)$ da abordagem HC (com tabela global); todavia, esta incorre num custo espacial de ordem $O(nlog^2(n))$ (em número de bits), face a um custo inferior de $O(log(n))$, na abordagem de Brinkman [BSS00].

2.6.4 Modelo *Balls-into-Bins*

Brinkman extendeu a abordagem anterior, para um cenário de distribuição de objectos num sistema distribuído [BSS02]. Nessa reformulação, adopta-se e adapta-se o modelo "*balls-into-bins*" (de atribuição aleatória de *balls* a *bins*[14] [ABKU94, RS98]), fazendo corresponder objectos a *balls*, e servidores a *bins* de certa capacidade; admitindo a variação dinâmica do número de objectos, servidores e capacidades destes, o modelo procura

[13] Notar que a adição/remoção de um disco é comparável à adição/remoção de um nó a/de uma DHT, e a adição/remoção de um bloco de dados é comparável à inserção/remoção de um registo em/de uma DHT.

[14] O modelo formaliza um tipo de jogos em que se atiram bolas, aleatoriamente, para dentro de um conjunto de cestos; na sua formulação clássica, os cestos são escolhidos de forma *independente* e *uniforme*.

distribuir objectos por servidores de forma a cumprir os requisitos acima definidos. A distribuição partilha semelhanças com o Hashing Consistente, pois envolve um conceito de distância entre cada objecto (*ball*) e cada servidor (*bin*), para se decidir a que servidor atribuir o objecto.

Czumaj et al. [CRS03] investigaram esquemas que permitem distribuições homogéneas *perfeitas* (ver definição a seguir), assentes na escolha de um de dois servidores seleccionados aleatoriamente (o servidor ao qual se atribui um objecto é o servidor menos sobrecarregado dos dois); essa escolha configura a aplicação do paradigma *power-of-two-choices* [MRS01], como uma forma simples de usar a aleatoriedade para atingir distribuições balanceadas.

2.6.4.1 Distribuições Perfeitas e Óptimas

Sob o modelo "*balls-into-bins*", uma distribuição homogénea *perfeita* é aquela em que, dados m objectos (*balls*) e n servidores (*bins*), a carga de qualquer servidor nunca excede $(m/n)+1$ nem é inferior a (m/n) [CRS03]. Esta definição é extensível a qualquer cenário onde se distribua uma grandeza discreta, de forma homogénea, por um conjunto de nós. Para cenários heterogéneos onde se consegue o melhor ajustamento possível (medido com base numa certa métrica de *qualidade da distribuição*) às diferentes proporções previstas para cada nó, o termo *óptimo* é mais apropriado para classificar a distribuição conseguida.

2.7 Estratégias de Localização

Tipicamente, uma estratégia de *particionamento* tem implícita a utilização de uma certa estratégia de *localização* (embora haja quem, tal como nós, advogue uma separação entre as duas questões [Man04]); esta, como definido na secção 2.4.1.4, diz respeito aos modelos e mecanismos que permitem reconstituir as correspondências "entrada $\mapsto$ nó" de uma DHT.

2.7.1 Localização em DHTs de Primeira Geração

Como ressalta do estudo comparativo da secção 2.4, é já possível encontrar, nalgumas DHTs de primeira geração (designadamente nas abordagens DDH [Dev93] e EH* [HBC97]), mecanismos de localização relativamente evoluídos, de tipo distribuído. Basicamente, esses mecanismos assentam numa visão local e parcial, em *trie*, sobre a evolução global da DHT, mantida por cada nó ou cliente da DHT; diferentes entidades terão visões independentes, com eventuais sobreposições; mensagens de correcção de erros de endereçamento permitem actualizar/corrigir, progressivamente, as *tries* locais, na tentativa de assegurar a sua convergência com uma *trie* virtual global; esta corresponde ao verdadeiro estágio de evolução em que se encontra a DHT, por via da aplicação de Hashing Dinâmico e ocorrência de divisão/fusão de contentores. Graças à visão em *trie*, o esforço de acesso à DHT (no número de nós visitados) é de ordem $O(log\mathcal{N})$, sendo $\mathcal{N}$ o número total de nós/contentores da DHT. Noutras abordagens de primeira geração, as aproximações prosseguidas são menos escaláveis, pelo recurso à replicação total da informação de posicionamento [LNS93b, GBHC00].

2.7.2 Localização em DHTs de Segunda Geração

2.7.2.1 Localização Distribuída (Multi-HOP)

No Hashing Consistente original [KLL+97], os mecanismos de *posicionamento* e de *localização* baseiam-se numa tabela global, que regista o nó (computacional) hospedeiro de cada partição de $[0, 1)$, permitindo acesso directo (1-HOP) aos nós adequados; saber a que nó computacional n se associou um hash h implica determinar 1) a correspondência entre h e um ponto c do círculo $[0, 1)$, 2) a que partição p pertence c, 3) a que nó virtual v foi associada a partição p e, finalmente, 4) a que nó computacional n pertence o nó virtual v.

Porém, uma tabela global[15], sendo suficiente para um conjunto de nós (computacionais/virtuais) relativamente reduzido e estático, é inadequada num ambiente P2P, com milhões de nós e composição dinâmica. E, como já se referiu, foi precisamente a necessidade de mecanismos de localização escaláveis, adequados a esses ambientes, que motivou o desenvolvimento de estratégias de localização distribuída, assentes no paradigma DHT.

A filosofia base dessas estratégias de localização é semelhante: as partições da DHT são unidas num grafo (*overlay network*), de forma que, por cada partição, é registada a localização (nó computacional hospedeiro) de um conjunto de outras partições, correspondente à sua *vizinhança* (topológica); dado um hash h, um nó é responsável pela partição que contém h ou então sabe determinar qual o nó (da sua vizinhança) mais *próximo* de h, para onde encaminhará qualquer pedido de localização de h; o método de definição da vizinhança e a medida de proximidade entre nós/hashes, são específicos de cada abordagem.

Cada abordagem procura atingir, à sua maneira, o melhor compromisso entre dois factores antagónicos: a necessidade de rotas curtas *versus* a necessidade de vizinhanças pequenas, no grafo subjacente, *i.e.*, recorrendo a conceitos básicos de Teoria de Grafos (ver apêndice C), está em causa a minimização simultânea do *diâmetro* do grafo e do *grau* dos vértices.

Assim, nas DHTs de segunda geração, o diâmetro e o grau do grafo de localização começam por ser ambos de ordem logarítmica, face ao número total de vértices (nós virtuais ou computacionais) do grafo; é o caso das abordagens Chord [SMK+01] (descendente do Hashing Consistente original [KLL+97]), Pastry [RD01] e Tapestry [ZKJ01] (descendentes do trabalho de Plaxton et al. [PRR97]), e Kademlia [MM02] (baseada numa métrica XOR de distância); a abordagem CAN [RFH+01] representa uma excepção, assegurando diâmetro de ordem $O(d\mathcal{N}^{1/d})$ e grau de ordem $O(d)$, para um espaço a d dimensões[16].

Posteriormente, as abordagens Viceroy [MNR02], Koorde [KK03] e D2B [FG03] conseguem graus de ordem $O(1)$, mantendo diâmetro de ordem $O(log\mathcal{N})$. O Viceroy adopta e adapta a topologia *butterfly*, usada em sistemas paralelos [Sie79], utilizando-a de forma conjugada com o Chord, para obter grau 7. O Koorde e o D2B exploram ambos grafos DeBruijn [dB46], sendo ambos capazes de virtualizar os vértices necessários a um grafo DeBruijn completo, de grau 2; em particular, o Koorde combina grafos DeBruijn com grafos Chord.

A preocupação em minimizar mais o diâmetro encontrou eco na descoberta de

[15] Não necessariamente centralizada: num cenário de Web Caching admitem-se réplicas dessincronizadas.
[16] Embora, quando $d = log\mathcal{N}$, o diâmetro e o grau do grafo passem a ser de ordem $O(log\mathcal{N})$.

$O(log\mathcal{N}/loglog\mathcal{N})$ como limite mínimo teórico, para um grau de ordem $O(log\mathcal{N})$ [Xu03]. Essa descoberta foi corroborada pela abordagem Ulysses [KMXY03], baseada, tal como a Viceroy, na topologia *butterfly*; todavia, o Ulysses suporta um número variável de vértices e evita situações de congestão (em que certos vértices/arestas do grafo são particularmente sobrecarregados em relação aos restantes). Importa ainda referir que, aumentando o grau para $O(log\mathcal{N})$, o Koorde também suporta diâmetro $O(log\mathcal{N}/loglog\mathcal{N})$.

Das abordagens anteriormente referenciadas, são-nos particularmente relevantes os grafos Chord e DeBruijn, como representantes de duas estirpes diferentes (de diâmetro comum, igual a $O(log\mathcal{N})$, mas grau $O(log\mathcal{N})$ nos grafos Chord, e $O(1)$ nos DeBruijn), capazes de assegurar localização distribuída de forma compatível com a operação dos nossos mecanismos de particionamento e posicionamento. Nesse contexto, o capítulo 4 é dedicado à exploração desses grafos, relegando-se para esse âmbito uma descrição sua mais detalhada.

2.7.2.2 Localização Directa (1-HOP)

Contrariando a linha de investigação seguida até então, Gupta et al. [GLR03] re-analisam a viabilidade e a(s) consequência(s) da manutenção de informação total sobre a composição de um sistema P2P com $\mathcal{N} \lessapprox 10^6$ nós. A motivação são as latências elevadas (mesmo com diâmetro de ordem $O(log\mathcal{N})$), devido ao desfasamento entre os grafos de localização (topologias de nível aplicacional) e as redes físicas[17]. A abordagem prosseguida passa pela sobreposição de uma estrutura hierárquica (*dissemination tree*) sobre um anel Chord. Com base em valores observados do dinamismo do sistema Gnutella [gnu], concluem então da viabilidade de encaminhamento 1-HOP: para valores de $\mathcal{N}$ entre 10^5 e 10^6, os nós responsáveis pela propagação de alterações à composição do sistema necessitam entre 35 Kbps a 350 Kbps de largura de banda garantida. Posteriormente, para sistemas com $\mathcal{N} \gg 10^6$ nós, Gupta et al. [GLR04] propõem um esquema 2-HOP, com encaminhamento em dois saltos e informação de posicionamento de ordem $O(\mathcal{N}^{1/2})$ por cada nó da DHT.

Na senda do trabalho de Gupta, Rodrigues [RB04], demonstra que o recurso a localização distribuída/*Multi-Hop* apenas compensa para sistemas que apresentam, em simultâneo, 1) número muito elevado de nós (pelo menos da ordem das dezenas de milhões), e 2) elevado dinamismo na sua composição; para cenários diferentes, advogam-se esquemas 1-HOP.

Posteriormente, Monnerat et al. [MA06] propõem a abordagem D1HT para uma DHT com localização 1-HOP, capaz de 1) diminuir consideravelmente (em pelo menos uma ordem de magnitude) os requisitos de largura de banda da abordagem de Gupta et al. [GLR04] (3 Kbps por nó, para 10^6 nós) e 2) suportar elevado dinamismo na composição do sistema.

Mais recentemente, Brown et al. [BBK07] apresentam a abordagem Tork, adequada a sistemas P2P heterogéneos, com elevada variabilidade de largura de banda: nós com elevada largura de banda (necessária à manutenção de informação total actualizada sobre a composição do sistema), utilizam localização 1-HOP; nós com pouca largura recorrem a localização *Multi-Hop*, de desempenho (nº de saltos) ajustável à heterogeneidade dos nós.

[17] Esse desfasamento motivou desde cedo, nas DHTs de segunda geração, a preocupação em minimizar a latência durante a construção da topologia, como acontece no Tapestry [ZKJ01] e no Pastry [RD01].

No contexto do nosso trabalho, designadamente na arquitectura Domus (descrita a partir do capítulo 5) a localização directa (1-HOP) surge como *método de localização* possível, no quadro de uma *estratégia de localização* que permite a combinação de vários métodos, incluindo a localização distribuída (*Multi-Hop*) "clássica" e recurso a *caches de localização*.

2.8 Balanceamento Dinâmico de Carga

As estratégias de particionamento e localização anteriormente referenciadas estão associadas, implícita ou explicitamente, a certas estratégias de balanceamento de carga[18]. Nesta secção referimos algumas das mais representativas. Comum a todas é o facto de o balanceamento se centrar na DHT (procurando assegurar certos níveis de serviço aos seus clientes) e menos na optimização global dos recursos do ambiente de exploração. Aplicando a classificação genérica de Li et al [LL04], o balanceamento centrado na DHT é de *nível aplicacional* (*Application-level*), ao passo que o segundo é de *nível sistema* (*System-level*); no mesmo contexto, a terminologia de Zaki et al [ZLP96] permite classificar o balanceamento centrado na DHT como *orientado-à-aplicação* (*application-driven*) ou *à medida*.

2.8.1 Balanceamento Dinâmico em DHTs/DDSs de Primeira Geração

Nas DHTs de primeira geração [LNS93a, Dev93, HBC97, GBHC00] procura-se uniformizar o consumo que a DHT faz dos recursos de armazenamento do ambiente de exploração (tipicamente *cluster*/NOW). Nessas DHTs a razão primordial para a contracção/expansão do número de entradas ou contentores (*buckets*) é a folga/esgotamento da capacidade de armazenamento associada às entradas (de capacidade pré-definida e igual para todas). O facto de as entradas/contentores poderem evoluir (subdividindo-se ou fundindo-se) de forma independente umas das outras soluciona, de forma natural, dois problemas distintos: i) distribuição não-uniforme do *número* de registos (rever secção 2.4.1.8), que pode concentrar muitos registos em poucos contentores; ii) distribuição não-uniforme da *dimensão* de registos, pela qual poucos registos podem esgotar a capacidade de um contentor.

Ao balanceamento da carga de armazenamento, a abordagem de Gribble et al. [GBHC00] acrescenta ainda suporte a um primeiro nível de balanceamento da carga de acesso, baseado na replicação dos registos da DHT, que permite dispersar a carga de leitura. Porém, o nível de replicação é estático (fixo, durante a vida da DHT) e os registos não são migráveis para nós mais folgados (como resposta a sobrecargas de acesso). Neste contexto, a abordagem SNOWBALL [VBW98] ao armazenamento de um Dicionário Distribuído para serviços WWW é paradigmática, pois complementa o balanceamento da carga de armazenamento com o da carga de acesso; para o efeito, a abordagem prevê: i) mecanismos de monitorização e migração da carga que operam em várias granularidades (registos, conjunto de registos, etc.) e de forma incremental (migração imediata, incremental, etc.); ii)

[18] A problemática do balanceamento dinâmico de carga tem sido amplamente investigada pela comunidade da Computação Paralela e Distribuída [SHK95, ZLP96, XL97]. No contexto desta tese, interessam-nos apenas as questões ligadas à aplicabilidade de técnicas genéricas (*e.g.*, *load-stealing*, *load-shedding*, suporte a distribuições heterogéneas, monitorização de carga e de recursos) ao contexto específico das DHTs.

suporte à migração de registos em simultâneo com o seu acesso (o que é mais facilmente exequível para registos (quase-)apenas de leitura, como é o caso de documentos da Web).

2.8.2 Balanceamento Dinâmico em DHTs de Segunda Geração

No sistema CFS [DKKM01, Dab01] (já referido na secção 2.6.2.2), um nó sobrecarregado (em termos de armazenamento), pode aliviar carga (*load-shedding*) removendo (ou melhor, atribuindo a outro nó) alguns dos seus nós virtuais; todavia, uma aplicação descuidada desta abordagem simples poderá originar situações de *thrashing*, em que a carga atribuída a outro nó sobrecarrega-o de imediato e tem de ser de novo transferida para um outro nó.

Rao et al. [RLS+03] investigaram esquemas de balanceamento dinâmico de carga, também no contexto do Chord, com definição independente do número de nós virtuais por nó (computacional). Assim, começam por salientar uma vantagem importante da admissão de múltiplos nós virtuais por nó: a transferência de nós virtuais entre nós permite transferir facilmente carga entre nós não adjacentes no círculo $[0, 1)$[19]. Tendo por objectivo o balanceamento dinâmico de um só tipo de recurso, propõem então três esquemas de balanceamento: 1) *one-to-one load-stealing* (cada nó subcarregado selecciona aleatoriamente um nó sobrecarregado, a quem pede um nó virtual); 2) *one-to-many load-shedding* (cada nó sobrecarregado selecciona um nó subcarregado, a quem cede um nó virtual; a selecção assenta na inspecção de *directorias* distribuídas, com a carga de nós leves); 3) *many-to-many* (esquema centralizado que, de uma só vez, considera todos os nós da DHT, e redistribui a carga dos sobrecarregados pelos subcarregados). Os esquemas anteriores exibem (pela sua ordem de apresentação), autonomia decrescente, requisitos de informação crescentes e uma eficácia crescente (de 85% a 95% do valor óptimo[20]).

Godfrey et al. [GLS+04] extendem a abordagem anterior, de Rao et al. [RLS+03], com suporte a um sistema altamente dinâmico, com i) inserção/remoção contínua de registos, ii) entrada/saída contínua de nós, iii) distribuição não-uniforme dos registos no círculo $[0, 1)$, iv) distribuição não-uniforme da dimensão dos registos. Em particular, usa-se o algoritmo *many-to-many* para balanceamento global periódico, com base em informação de carga mantida em uma ou mais *directorias* distribuídas, reduzindo essencialmente o problema do "balanceamento distribuído" ao de um "balanceamento centralizado" por cada directoria; adicionalmente, usa-se o algoritmo *one-to-many load-shedding* para balanceamentos locais de emergência. Um resultado interessante é o de que, com nós computacionais heterogéneos, o balanceamento da carga exige menos nós virtuais do que com nós homogéneos.

Como referido previamente (na secção 2.6.2.2), Karger et al. [KR04] conceberam uma abordagem que suporta vários nós virtuais por nó computacional mas apenas permite a "activação" de um nó virtual de cada vez. Essa abordagem suporta dois esquemas de balanceamento dinâmico com diferentes objectivos: 1) balanceamento do *espaço de endereçamento* da DHT (*i.e.*, particionamento homogéneo), em que cada nó computacional assume obrigatoriamente um dos seus $O(log\mathcal{N})$ nós virtuais; 2) balanceamento dos registos da DHT, em que um nó computacional pode assumir *qualquer* nó virtual do sistema (*e.g.*,

[19] Com um só nó virtual por nó computacional, a transferência de carga ocorreria entre nós adjacentes.
[20] Ver a secção 2.6.4.1 para uma definição de distribuição *perfeita* ou *óptima*.

aquele onde a concentração de registos é maior), com base numa estratégia de *load-stealing*.

Byers et al. [BCM03] exploram o paradigma *power-of-two-choices* [MRS01] no contexto da abordagem Chord, demonstrando que é possível alcançar melhor balanceamento de carga do que com a utilização (convencional) de vários nós virtuais por nó computacional; a abordagem proposta baseia-se na utilização de $d \geq 2$ funções de hash que, para cada registo, permitem determinar d nós hospedeiros possíveis; o registo será então armazenado no nó com menos registos e os $d-1$ nós restantes guardam *indirecções* para ele; desta forma evitam-se d localizações (que poderiam até ocorrer em paralelo) por cada acesso a um registo, uma vez que pode ser usada qualquer uma das d funções de hash para iniciar a busca (embora seja de $(d-1)/d$ a probabilidade de essa busca comportar o passo adicional representado por uma indirecção); a abordagem pressupõe que os registos têm dimensão semelhante, mas admite frequências de acesso variáveis, o que clama por alguma forma de balanceamento dinâmico; neste contexto, a exploração conveniente do mecanismo de indirecções viabiliza a utilização de técnicas de *load-stealing* e *load-shedding*, bem como de replicação (esta como forma de dispersar carga e assegurar alguma tolerância a faltas).

Aberer et al. [ADH03, ADH05] desenvolvem mecanismos que, no contexto da abordagem P-Grid [Abe01, AHPS02] a um Dicionário Distribuído, permitem balancear, simultaneamente, 1) carga de armazenamento e 2) carga de replicação. Na abordagem P-Grid, o Dicionário Distribuído configura-se numa *trie* binária virtual, onde cada nó é responsável por uma sequência de bits, de dimensão variável. Adicionalmente, não há qualquer relação (*e.g.*, de distância topológica, como no Hashing Consistente e derivados) entre o identificador do nó e a sequência de bits de que ele é responsável, facilitando a re-atribuição dessa sequência a outro nó, sem necessidade de *indirecções* (que introduzem sempre ineficiências). Em conjunto, estas propriedades permitem a adaptação dinâmica da *trie* virtual a distribuições não-uniformes, bem como uma estratégia de replicação das *partições* que, de forma diferente de outras, determina o número de réplicas dinâmica e permanentemente.

2.8.3 Balanceamento Dinâmico Ciente dos Recursos

Para além de estratégias de balanceamento centradas na(s) DHT(s) (de *nível aplicacional* ou *orientado-à-aplicação*), interessam-nos também, no contexto desta dissertação, a sua combinação com estratégias de *nível sistema* e *cientes-dos-recursos* (*resource-aware*); estas, assentes num conhecimento relativamente preciso dos recursos disponíveis no ambiente de execução (e da taxa de utilização), viabilizam um balanceamento de carga mais efectivo.

Uma abordagem compatível com este tipo de requisitos é fornecida pelo modelo DRUM (*Dynamic Resource Utilization Model*) [Fai05, TFF05], em cooperação com as plataformas NWS [WSH99] e Zoltran [DBH+02]. Em particular, esta combinação tolera a co-execução de outras aplicações no *cluster*, ao mesmo tempo que balanceia dinamicamente as que executam sob o seu controlo; estas correspondem, essencialmente, a tarefas de cálculo/computação científica. O DRUM assenta numa visão hierárquica, em árvore, do ambiente de execução, na qual as folhas são *nós de computação* e os outros níveis são ocupados por *nós de rede* (encaminhadores, comutadores, repetidores, etc.); essa visão suporta a definição de uma métrica *linear* de "potência", de granularidade variável (*i.e.*, aplicável a

nós computacionais ou a (sub)árvores); a "potência" de um nó (ou de uma (sub)árvore) é uma soma pesada de certas capacidades estáticas (medidas por *benchmarks* especializados, ou *benchmarks* genéricos da suite LINPACK [lin]) e taxas de utilização dinâmicas (monitorizadas por agentes próprios ou pelo sistema NWS); essa soma (entre 0 e 1) representa a percentagem da carga (*workload*) global a atribuir ao nó (ou (sub)árvore). De facto, o DRUM não efectua, por si só, balanceamento dinâmico; antes participa nesse processo através da produção de métricas sintéticas que, alimentando plataformas *standard* de particionamento de carga (como o Zoltran), permitem o particionamento (dinâmico) de um problema de forma coerente com a heterogeneidade (dinâmica) dos nós computacionais.

A definição dinâmica de capacidades computacionais em *clusters* heterogéneos e partilhados, com base num modelo *linear*, pode também ser encontrada na abordagem de Sinha et al. [SP01] ao balanceamento dinâmico de aplicações AMR (*Adaptive Mesh Refinement*). Como no DRUM, o foco incide no balanceamento de tarefas de cálculo científico (embora num contexto mais específico) e conta-se com os bons ofícios do sistema NWS [WSH99] de monitorização distribuída de recursos; todavia, a abordagem de Sinha et al. leva também em conta a disponibilidade de memória principal, para além de CPU e largura de banda.

2.9 Dissociação Endereçamento-Armazenamento

Genericamente, podemos encontrar na arquitectura de qualquer DHT componentes ou subsistemas associados à resolução de questões de Endereçamento e de Armazenamento. No caso do Endereçamento, está em causa a prossecução de determinadas estratégias de *particionamento* do espaço de endereçamento da DHT e de *localização* das partições resultantes, de que são exemplos as referenciadas nas secções anteriores. No caso do Armazenamento, estão em causa as questões mais directamente ligadas à salvaguarda de dados/registos na DHT (estruturas de dados adequadas, nível de persistência, etc.).

Um aspecto arquitectural relevante no contexto desta dissertação é o suporte à *dissociação* entre o Endereçamento e o Armazenamento de uma mesma entrada. Numa situação de *associação*, o nó da DHT em que desemboca a localização de uma entrada (*nó de endereçamento*) é também o responsável pelo armazenamento de todos os registos da entrada (*nó de armazenamento*); numa situação de dissociação, o uso de *indirecções* permite referenciar um *nó de armazenamento* diferente, a partir do *nó de endereçamento* da entrada.

Nas DHTs de primeira geração (orientadas ao armazenamento distribuído) a situação mais frequente é a associação Endereçamento-Armazenamento. A abordagem de Gribble et al. [GBHC00] exibe, porém, uma certa forma de dissociação, demonstrada pela existência de *nós de frontend*, que redireccionam pedidos de acesso à DHT para *nós de armazenamento*.

Nas DHTs de segunda geração (orientadas a ambientes P2P), a dissociação Endereçamento-Armazenamento é suportada naturalmente, através de *indirecções* [CNY03]; de facto, tais DHTs surgiram, inicialmente, como mecanismo de localização distribuída, relegando-se para um plano mais secundário a co-assunção de funções de armazenamento pelos nós da DHT. Por exemplo, em sistemas P2P de partilha de ficheiros é vulgar existir um "núcleo duro" de nós de endereçamento (*superpeers*), que participam activamente na localização

distribuída e não assumem funções de armazenamento; por outro lado, um nó/cliente regular, que partilha ficheiros (referenciados pelos nós de endereçamento), pode assumir, de forma voluntária, funções de endereçamento (transformando-se assim num *superpeer*).

A dissociação Endereçamento-Armazenamento é possível porque envolve o tratamento de questões mais ou menos independentes. Essa independência é por vezes extensível a um nível mais microscópico da arquitectura de uma DHT. Por exemplo, Manku [Man04] demonstra, na sua arquitectura Dipsea para uma DHT modular, que *particionamento* e *localização* (sub-problemas do Endereçamento), podem ser atacados de forma independente.

2.10 Paradigmas de Operação de DHTs

Sob o ponto de vista das suas aplicações clientes, a operação de uma DHT pode seguir duas abordagens completamente distintas: 1) a DHT apresenta-se como mais um *serviço distribuído*, com o qual podem interactuar, simultaneamente, múltiplas aplicações clientes; 2) a DHT é embebida no seio da aplicação paralela/distribuída que dela usufrui, aplicação que é a única responsável pela *operação* (instanciação, exploração e gestão) da DHT.

A primeira abordagem, correspondente à "visão de uma DHT como um *serviço*", preconiza uma separação estrita entre o código das aplicações e o da DHT; a realização de uma DHT é feita à custa de uma colecção de serviços que se apresenta como uma "caixa preta"; às aplicações é oferecido um interface relativamente limitado (confinado às operações usuais sobre dicionários), mas a partilha de uma DHT por várias aplicações e a co-operação de várias DHTs são facilitadas. Esta filosofia, que está na base das DHTs de primeira geração (e que encontra na abordagem de Gribble et al. [GBHC00] o seu expoente máximo – rever secção 2.4.5), é também a prosseguida pela nossa arquitectura Domus (ver capítulo 7).

Karp et al. [KRRS04] caracterizam a segunda abordagem, correspondente ao que os autores designam de "visão de uma DHT como uma *biblioteca*". Assim, a co-localização de código aplicacional juntamente com o código da DHT permite a uma aplicação aceder ao estado local da DHT ou receber *upcalls* dela (porque ambas fazem parte do mesmo processo, ou recorrendo a RPCs locais). Esta aproximação tem vantagens importantes, sendo a principal o facto de permitir a realização de operações mais complexas / *à-medida* em cada nó da DHT, para lá de simples inserções, remoções e consultas de registos (*e.g.*, *range-queries* ou outro tipo de processamento de um (sub)conjunto de registos locais da DHT). Neste contexto, Karp et al. distinguem dois tipos de operações: a) *per-hop operations* – as operações são realizadas por todos os nós ao longo de um certo percurso / rota na topologia da DHT; b) *end-point operations* – as operações são apenas executadas no nó terminal de uma rota. Por outro lado, a abordagem em causa apresenta certas desvantagens: a) requer um conhecimento mais aprofundado do código da DHT; b) necessita que o código de todas as operações complexas a realizar esteja presente na aplicação, *a priori*, pois pode não ser possível inserir código em tempo de execução; c) limita a utilidade de uma DHT à aplicação hospedeira, dificultando a partilha de DHTs por várias aplicações. A exploração de DHTs como bibliotecas encontra bastantes exemplos de aplicação no contexto das DHTs de segunda geração [KBCC00, DR01b, SAZ+02, CDK+03, HCH+05].

A caracterização anterior da segunda abordagem surge no âmbito de uma descrição da plataforma OpenHash; esta, com raízes no Pastry [RD01] (uma das abordagens seminais a DHTs de segunda geração), disponibiliza o acesso a uma DHT genérica, de interface simples, oferecida como um serviço público e aberto, com o objectivo de agilizar o desenvolvimento de aplicações baseadas no "paradigma DHT"[21]. Karp et al. propõem então um mecanismo de redireccionamento (ReDiR) que permite dotar a plataforma OpenHash (que oferece uma "DHT como serviço") da possibilidade de invocar código residente em nós externos, no contexto de operações *end-point* (ver acima), basicamente definindo uma ponte entre os dois paradigmas de operação de DHTs ("como biblioteca" e "como serviço").

Mais recentemente, no contexto do desenvolvimento de um sistema de posicionamento geográfico, Chawathe et al. [CRR+05] investigaram também a possibilidade de construir, sobre DHTs genéricas, aplicações com requisitos mais sofisticados que as operações básicas sobre dicionários. Para o efeito, recorreram à plataforma OpenDHT [RGK+05] (a re-encarnação mais recente da plataforma OpenHash anteriormente referida), sobre a qual acomodaram uma *trie* distribuída, apropriada a *range-queries*. Se, por um lado, se confirmou a viabilidade da re-utilização de uma plataforma padrão de DHTs no desenvolvimento da ferramenta pretendida, por outro o desempenho alcançado foi sub-óptimo face ao que seria possível de conseguir com uma DHT *à-medida* (*i.e.*, embebida na própria ferramenta). A abordagem prosseguida por Chawathe et al. assenta numa visão por camadas, da arquitectura da ferramenta de localização geográfica desenvolvida. No nível mais baixo dessa arquitectura, encontra-se a plataforma OpenDHT, à qual se fez o *outsourcing* da *instanciação* e *gestão* de uma DHT, cabendo à ferramenta de localização a sua *exploração*.

2.11 Operação (Conjunta) de Múltiplas DHTs

A "operação conjunta de múltiplas DHTs" (ou, mais simplesmente, a "co-operação[22] de DHTs") pode-se realizar com diferentes objectivos e de forma mais ou menos integrada. No nosso caso, o objectivo final é sempre o suporte ao armazenamento distribuído e a integração pretendida corresponde ao nível *avançado*, definido na secção 2.4.1.9. Noutras abordagens, os objectivos perseguidos e os níveis de integração poderão ser diferentes.

A co-exploração de múltiplas DDSs/DHTs é suportada explicitamente, pela primeira vez, na abordagem de Gribble [GBHC00] (rever secção 2.4.5); nessa abordagem, o conceito de *namespaces* (contextos de identificação) permite individualizar o acesso a cada DHT e as DHTs partilham os recursos de serviços dedicados ao armazenamento (*storage bricks*).

Posteriormente, Martin et al. [MNN01] identificam, de forma sistemática, uma série de questões fulcrais ligadas ao desenho de DDSs[23] (rever secção 2.3), entre as quais a necessidade de suporte adequado à operação simultânea de múltiplas instâncias; neste contexto, preconiza-se i) a possibilidade de cada DDS possuir atributos configuráveis, de forma a responder aos requisitos das aplicações clientes e ii) a necessidade de que eventuais meca-

[21] Ou seja, o enfoque não é colocado no desempenho final da abordagem (pois os clientes acedem ao serviço com base em conexões WAN), mas na possibilidade de prototipagem rápida de aplicações leves.

[22] *Co-operação* tem portanto a semântica de "operação em conjunto" e não de "operação em colaboração".

[23] Refira-se, a propósito, que parte dos requisitos já eram cumpridos na abordagem de Gribble [GBHC00].

nismos de balanceamento dinâmico operem de forma concertada sobre as várias DDSs.

Com DHTs de segunda geração, a necessidade/conveniência de co-operação de DHTs surge numa diversidade maior de situações (no quadro da operação em ambientes P2P), para lá do cenário mais convencional da primeira geração (limitado a armazenamento distribuído).

Por exemplo, o conceito de *namespaces* usado na abordagem de Gribble et al. [GBHC00] volta a aparecer, desta feita na plataforma OpenHash [KRRS04], mais especificamente no contexto do mecanismo ReDiR (rever secção 2.10). Neste caso, o objectivo é suportar várias DHTs virtuais à custa da DHT real (e genérica) fornecida de base; como parte do mecanismo ReDiR, as aplicações podem registar nós externos à DHT real, univocamente identificados num certo domínio de nomeação comum (*namespace*); esses nós externos passam a fazer parte de uma DHT virtual, juntamente com os nós internos que os referenciam.

Na abordagem SkipNet [HJS+03] (uma generalização distribuída de Skip-Lists [Pug90]) prevê-se a possibilidade de um mesmo nó participar em várias DHTs que, todavia, não são completamente independentes umas das outras, podendo-se sobrepor ou aninhar; as DHTs organizam-se numa hierarquia de nomeação e cada nó que participa em várias DHTs ocupa, simultaneamente, lugares diferentes nessa hierarquia; o objectivo da abordagem é o de conseguir confinar i) o posicionamento, ii) a localização e iii) o balanceamento de carga de certos dados/registos a certos domínios administrativos (pertencentes à referida hierarquia), sem que isso implique perda de conectividade com outros domínios; para o efeito, a nomeação das entidades é feita com base em sequências de caracteres que reflectem a hierarquia administrativa/organizacional; essa nomeação implica, por exemplo, que o identificador de um objecto/registo inclua o identificador da sua DHT e nó hospedeiro.

Nesta tese, todavia, prossegue-se uma estratégia de suporte à co-operação de DHTs que, em alternativa à *virtualização* de várias DHTs sobre uma DHT de base, assume a co-existência de DHTs independentes, realizáveis sobre um substrato comum de serviços. Precisamente, as abordagens referenciadas de seguida são já enquadráveis nessa filosofia.

Assim, no âmbito da utilização de DHTs de segunda geração para encaminhamento em redes sem fios (domínio aplicacional recente), as abordagens de Heer et al. [HGRW06] e Cheng et al. [COJ+06] prevêem a co-existência de várias DHTs (uma por cada célula/zona da rede) e, sobre estas, a realização de operações de *fusão* (combinação de várias DHTs numa só) ou de *partição* (subdivisão de uma DHT em duas ou mais DHTs). As abordagens de Heer et al. e Cheng et al. recorrem, respectivamente, a topologias Chord e CAN; mais recentemente, Cheng et al. estendem a sua abordagem de forma a suportar DHTs baseadas em diferentes topologias e com propriedades específicas (*e.g.*, funções de *hash* e dimensão (número de bits) do *hash*) [CJO+07]; a possibilidade de *bridging* para a interacção entre nós de DHTs vizinhas, sem necessidade de fusão, é também investigada por Cheng et al.

As abordagens P2P encontraram também terreno fértil em ambientes *Grid*, contribuindo para a descentralização de certos serviços. Talia et al. [TTZ06], por exemplo, propõem uma arquitectura para a *descoberta de recursos* em ambiente *Grid*, baseada na operação de múltiplas DHTs[24]. Mais especificamente, a arquitectura assume a categorização dos

[24] No pressuposto de que, para além de interrogações baseadas em correspondência exacta (*exact-match-queries*), as DHTs usadas também suportam interrogações que cobrem gamas de valores (*range-queries*).

recursos da *Grid* em *classes* (*e.g.*, recursos computacionais, de armazenamento, de rede, de software, de dados, de instrumentação, etc.), sendo cada classe composta por vários *atributos* (*e.g.*, para a classe de recursos computacionais, atributos possíveis são "OS name", "CPU Speed", "Free memory", etc.); prevêem-se então um conjunto de *planos virtuais*, um para cada classe de recursos e, para cada plano/classe, é construída uma DHT por cada atributo estático[25]; desta forma, cada nó da *Grid* participa em múltiplas DHTs (em número dado pela soma do número de atributos, para todas as classes); uma DHT genérica, interligando todos os nós da *Grid*, é também usada para a difusão global de mensagens.

2.12 Ambientes de Exploração e de Desenvolvimento

No que toca ao ambiente de exploração (e tendo como pano de fundo o mundo UNIX), a computação em ambientes *cluster* de tipo Beowulf[26] [beo] representou o nosso ambiente alvo de investigação. Actualmente, existem diversas plataformas de qualidade, que simplificam bastante a instalação, manutenção e utilização desse tipo de *clusters* [Slo04]. Considerando apenas as baseadas exclusivamente em código aberto, o ROCKS [roc] e o OSCAR [osc] representam duas abordagens de referência[27], mas diferentes na sua filosofia. Assim, o ROCKS é uma distribuição Linux completa[28], suportando diversos i) modelos de execução (*partilha*, *trabalho em lotes*, *Grid*), ii) tecnologias de comunicação de alto desempenho (Myrinet, Infiniband, etc.), iii) bibliotecas de passagem de mensagens (PVM, MPICH, etc.), assim como iv) facilidades avançadas de administração e gestão do *cluster* (monitorização de recursos, instalação automática de nós, etc). Enquanto que o ROCKS é uma solução de tipo "chave-na-mão", o OSCAR prossegue uma filosofia diferente, consistindo num conjunto de pacotes de software, de funcionalidade semelhante à oferecida pelo ambiente ROCKS, mas agnósticos relativamente à distribuição (de Linux) hospedeira.

No desenvolvimento do protótipo da nossa arquitectura Domus, o recurso ao ambiente ROCKS e à linguagem Python [pyt] revelou-se crucial, permitindo acelerar o processo de desenvolvimento e teste. Embora se possa advogar que a linguagem Python, sendo interpretada, não é adequada para computação com maiores requisitos de desempenho, a verdade é que existe uma considerável dinâmica de utilização nesse contexto [Lan04, sci, Hin07], incluindo esforços no sentido de aproximar a eficiência de código Python à de linguagens de mais baixo nível [psy, Rig04, Lus07]. O futuro da linguagem Python no desenvolvimento de aplicações paralelas/distribuídas parece ser assim bastante promissor.

2.13 Epílogo

Este capítulo passou em revista os principais temas abordados nesta dissertação. Ao longo dos próximos capítulos, os tópicos abordados serão revisitados, à medida que apresentarmos as nossas próprias contribuições e as confrontarmos com outras da mesma área.

[25] Considera-se que, para atributos dinâmicos, o custo da actualização das suas DHTs seria proibitivo.

[26] Essencialmente *clusters* que, na sua arquitectura mais simples, são construídos com base em *hardware* vulgar (*commodity-hardware*), executam sistemas de exploração e *software* (tipicamente) de código aberto e recorrem a bibliotecas de passagem de mensagens para a prossecução de computação paralela/distribuída.

[27] Ver `http://www.cro-ngi.hr/index.php?id=1400&L=1` para um lista que inclui outras possibilidades.

[28] Baseada no CentOS [cen], uma versão comunitária do Red Hat Enterprise Linux [rhe] para ambientes de produção, onde a primazia é a estabilidade de operação em detrimento das funcionalidades mais recentes.

Capítulo 3

Modelos de Distribuição

Resumo

Neste capítulo apresentamos modelos capazes de assegurar distribuições *óptimas*[1] do contradomínio de uma função de hash, por um conjunto de nós computacionais. Os modelos cobrem os vários cenários que emergem considerando a combinação de Distribuições Homogéneas ou Heterogéneas, com a utilização de Hashing Estático ou Hashing Dinâmico. A nossa abordagem é também confrontada com o Hashing Consistente, a fim de se apreender o real significado da *qualidade da distribuição* (superior) alcançável pelos nossos modelos.

3.1 Prólogo

Os modelos apresentados neste capítulo dão uma resposta ao problema do particionamento, na definição do *número* de partições por nó, e do *número* de entradas por partição; a definição da *identidade* das entradas de cada partição, segundo uma certa estratégia de *posicionamento*, é tratada apenas no capítulo 4, juntamente com a questão da *localização*.

3.2 Conceitos Básicos

3.2.1 Entradas de um Nó

Seja $f : K \mapsto H$ uma *função de hash* que produz *hashes* $h \in H$ de chaves $k \in K$, e seja N o conjunto dos nós computacionais que, num dado instante, suportam a DHT associada.

Dada a equivalência entre os *hashes* de H e as entradas da DHT associada, convencionamos denotar por $H(n)$ o subconjunto das entradas da DHT atribuídas ao nó $n \in N$ do *cluster*.

Um subconjunto $H(n)$ corresponde a uma *partição* de H, ou seja: 1) a cada nó n será

[1] Ou mesmo *perfeitas* (estando em causa o conceito de distribuição *óptima* ou *perfeita* da secção 2.6.4.1).

atribuído um subconjunto $H(n)$ diferente do atribuído aos outros nós e 2) nenhuma entrada de H ficará por atribuir, restrições que podem ser formalizadas pelas seguintes expressões:

$$\bigcap_{n \in N} H(n) = \oslash \tag{3.1}$$

$$\bigcup_{n \in N} H(n) = H \tag{3.2}$$

As expressões anteriores traduzem o *particionamento* de H em $\#N$ subconjuntos disjuntos.

3.2.2 Quota (Real) de um Nó

Sendo $\mathcal{H}$ o total de entradas da DHT, $1/\mathcal{H}$ corresponde à quota (real) da DHT, associada a cada entrada, donde a quota (real) associada a um nó n, detentor de $\mathcal{H}(n)$ entradas, é

$$\mathcal{Q}^r(n) = \frac{\mathcal{H}(n)}{\mathcal{H}}, \forall n \in N \tag{3.3}$$

, para um total de $\mathcal{H} = \#H$ entradas disponíveis, que irá obedecer à seguinte expressão:

$$\mathcal{H} = \sum_{n \in N} \mathcal{H}(n) \tag{3.4}$$

3.2.3 Qualidade da Distribuição

A definição de uma *distribuição* de entradas através de um conjunto de nós comporta, em primeira instância, a definição do *número* de entradas a atribuir a cada nó. Nos nossos modelos, esse número é suficiente para alimentar uma métrica de *qualidade da distribuição*.

Ora, a *qualidade* de uma certa distribuição pode ser veiculada por qualquer métrica estatística que indique, directa ou indirectamente, o grau de aproximação entre a *quota real* de cada nó e uma sua *quota ideal*, que convencionamos designar por $Q^i(n)$ e obedece a:

$$\sum_{n \in N} \mathcal{Q}^i(n) = 1 \tag{3.5}$$

A *quota ideal* de um nó corresponde a uma certa fracção do número de entradas da DHT que o nó é suposto gerir. Como as entradas da DHT são, por definição, indivisíveis, então a fracção efectivamente gerida (a *quota real*), poderá ser diferente da fracção pretendida (a *quota ideal*). Um modelo de distribuição que tenha como objectivo maximizar, permanentemente, a qualidade da distribuição, terá de minimizar, permanentemente, e em simultâneo para todos os nós, a discrepância entre as suas quotas ideais e as reais.

3.3 Modelo M1: Dist. Homogénea com Hashing Estático

Considere-se uma DHT operada sob Hashing Estático (ou seja, com $\mathcal{H}$ constante) mas em que se admite a variação do número de nós que suportam a DHT (ou seja, $\mathcal{N}$ pode variar).

3.3.1 Quota Ideal de um Nó

Assumindo uma participação homogénea de nós computacionais no suporte a uma DHT, então todos têm direito à mesma fracção da DHT, donde a *quota ideal* comum a todos é:

$$\mathcal{Q}^i(n) = \overline{\mathcal{Q}}(n) = \frac{1}{\mathcal{N}}, \forall n \in N \tag{3.6}$$

Esta quota não é estática, variando em função do número efectivo de nós da DHT, $\mathcal{N}$.

3.3.2 Métricas de Qualidade

Neste contexto, a *qualidade da distribuição* de uma DHT pode ser, numa primeira aproximação, aferida pela *Soma dos Desvios Absolutos* (SDA) das quotas reais face às ideais:

$$SDA[\mathcal{Q}(n)] = \sum_{n \in N} |\Delta[\mathcal{Q}(n)]| \tag{3.7}$$

com

$$\Delta[\mathcal{Q}(n)] = \mathcal{Q}^r(n) - \overline{\mathcal{Q}}(n), \forall n \in N \tag{3.8}$$

A *Soma dos Desvios Absolutos* (SDA)[2] representa uma medida intuitiva de dispersão, mas a sua manipulação matemática pode ser complicada, pela utilização de valores absolutos. Neste sentido, o *Desvio Quadrático Médio*, ou até mesmo o *Desvio Padrão*, constituem métricas de utilização mais disseminada, dado que evitam os referidos problemas [GC97].

O *Desvio Padrão Absoluto*, como métrica alternativa a $SDA[\mathcal{Q}(n)]$, é neste caso dado por:

$$\sigma[\mathcal{Q}(n)] = \sqrt{\frac{\sum_{n \in N}[\mathcal{Q}^r(n) - \overline{\mathcal{Q}}(n)]^2}{\mathcal{N}}} = \sqrt{\sum_{n \in N}[\Delta[\mathcal{Q}(n)]^2 \times \overline{\mathcal{Q}}(n)]} \tag{3.9}$$

Frequentemente, recorre-se também ao *Desvio Padrão Relativo*[3], obtido pela divisão do *Desvio Padrão Absoluto* pela *Média*. Esta medida tem a vantagem de permitir comparar a dispersão de séries de natureza eventualmente diferente, face às suas *Médias* individuais. Tendo em conta que $\overline{\mathcal{Q}}(n)$ assume, no caso presente, o papel de *Média*, então o *Desvio Padrão Relativo* correspondente ao *Desvio Padrão Absoluto* definido anteriormente será:

$$\overline{\sigma}[\mathcal{Q}(n)] = \frac{\sigma[\mathcal{Q}(n)]}{\overline{\mathcal{Q}}(n)} \tag{3.10}$$

[2] Ou o *Desvio Absoluto Médio* (DAM), dado pela divisão de SDA pela dimensão da amostra.
[3] Mais rigorosamente designado de *Coeficiente de Variação/de Variabilidade de Pearson* [GC97].

3.3.3 Função Objectivo

Informalmente, o objectivo do modelo M1 é "maximizar a qualidade da distribuição, mantendo $\mathcal{H}$ constante e permitindo a variação de $\mathcal{N}$". A qualidade da distribuição será tanto maior quanto menor for o valor de $SDA[\mathcal{Q}(n)]$, $\sigma[\mathcal{Q}(n)]$ ou $\overline{\sigma}[\mathcal{Q}(n)]$. Estas métricas são, por definição, correlacionadas. Todavia, tendo em conta os méritos do uso do *Desvio Padrão Relativo*, fixa-se a "minimização de $\overline{\sigma}[\mathcal{Q}(n)]$" para função objectivo do modelo M1.

3.3.4 Procedimento de (Re)Distribuição

No modelo M1 de distribuição homogénea, o *número* de entradas da DHT atribuído a cada nó $n \in N$ é então definido de forma a minimizar $\overline{\sigma}[\mathcal{Q}(n)]$. Essa definição ocorre, a primeira vez, durante a criação da DHT, com base num valor inicial de $\mathcal{N}$, mas tem de ser refeita sempre que o valor $\mathcal{N}$ se altera, ou seja, sempre que a DHT ganhar ou perder um ou mais nós. Assim, i) sempre que a DHT ganhar um nó, este terá de receber entradas, cedidas de um ou mais dos outros nós; ii) complementarmente, sempre que a DHT perder um nó, as entradas deste nó terão de ser redistribuídas, por um ou mais dos outros nós.

3.3.4.1 Definição do Número Específico de Entradas por Nó

Genericamente, o Procedimento de (Re)Distribuição do modelo M1 reparte, o mais *equitativamente* possível, as entradas da DHT (em número $\mathcal{H}$), pelos seus nós (em número $\mathcal{N}$[4]), respeitando a indivisibilidade das entradas. Assim, numa primeira fase, atribuem-se $\mathcal{H}_{div}(n) = \mathcal{H}\ div\ \mathcal{N}$ entradas a cada nó (em que *div* representa a divisão inteira), do que resulta a atribuição de um total de entradas $\mathcal{H}_{div} = \mathcal{H}_{div}(n) \times \mathcal{N}$; depois, atribuem-se as $\mathcal{H}_{mod} = \mathcal{H} - \mathcal{H}_{div}$ entradas remanescentes, uma a uma, a outros tantos nós. Este acerto pode explorar uma certa ordem (lexicográfica, baseada na identificação dos nós, ou temporal, baseada no instante da sua junção à DHT) para determinar os nós beneficiados. No final do processo, o conjunto de nós N será divisível em dois subconjuntos: um, em que cada nó tem $\mathcal{H}_{div}$ entradas; outro, em que cada nó tem $\mathcal{H}_{div} + 1$ entradas; ou seja: será de uma unidade a diferença máxima entre o número entradas de qualquer par de nós, o que faz da distribuição alcançada uma *distribuição perfeita*, pela definição da secção 2.6.4.1.

3.3.4.2 Transferências de Entradas entre Nós

O procedimento acima descrito apenas define o *número* final (total) de entradas por cada nó, de forma a maximizar a qualidade da distribuição. De facto, o procedimento não inclui a definição 1) da *identidade* dos nós que têm de ceder/receber entradas, nem 2) do *número* individual de entradas que cada nó deve ceder/receber. No primeiro caso, já se deu a entender (rever secção 3.1) que a identidade das entradas a movimentar é irrelevante. No segundo caso, a definição em causa pode socorrer-se de uma *tabela de distribuição* (TD), de esquema $< n, \mathcal{H}(n) >$, que regista o número de entradas $\mathcal{H}(n)$ de cada nó $n \in N$.

Assim, a comparação das versões da TD anterior e posterior à aplicação do Procedimento de (Re)Distribuição, permite identificar quais os nós que devem ceder/receber entradas e

[4] $\mathcal{N}$ corresponde ao valor final do número de nós, após a adição ou remoção de um ou mais nós.

em que quantidade. Depois, resta definir os actores de cada transferência a realizar, ou seja, associar cada nó doador a um ou mais nós beneficiários; essa associação pode ser i) definida de forma centralizada por um nó coordenador, ou ii) inferida de forma autónoma e determinística por qualquer nó da DHT (o que exige que cada um deles disponha das duas versões da TD e aplique um critério comum (predefinido) de ordenação dessas tabelas).

Idealmente, as movimentações de entradas deveriam realizar-se em paralelo, apenas com a intervenção dos nós afectados, e tolerando acessos à DHT por parte de aplicações clientes.

As metodologias que acabamos de definir para a transferência de entradas entre nós, no caso do modelo M1, são também aplicáveis aos restantes modelos apresentados neste capítulo, pelo que nos iremos abster de retomar o tema na apresentação desses modelos.

3.3.5 Qualidade da Distribuição

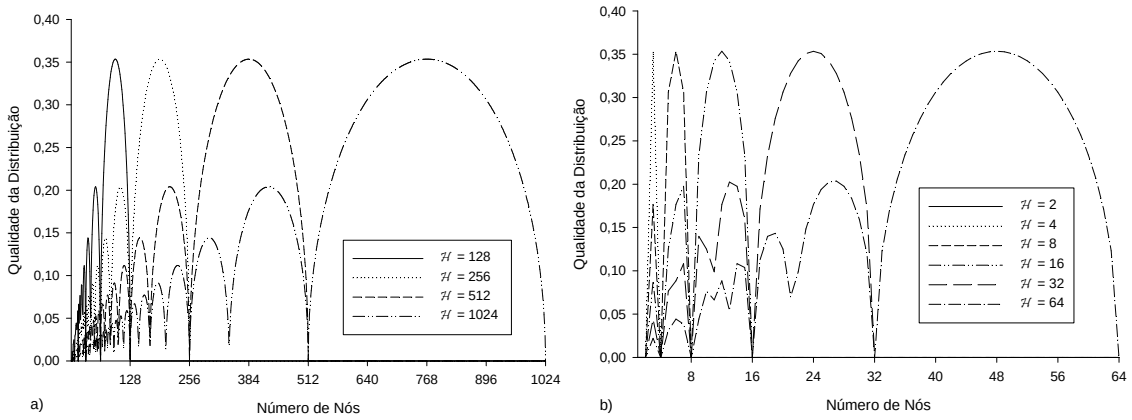


Figura 3.1: $\overline{\sigma}[\mathcal{Q}(n)]$ para $\mathcal{N} \leq \mathcal{H}$ e a) $\mathcal{H} \in \{128, 256, ..., 1024\}$, b) $\mathcal{H} \in \{2, 4, ..., 64\}$.

As figuras 3.1.a) e 3.1.b) apresentam resultados da simulação do modelo M1, expressos na métrica $\overline{\sigma}[\mathcal{Q}(n)]$, para valores de $\mathcal{H} \in \{2, 4, 8, ..., 1024\}$ e de $\mathcal{N} \leq \mathcal{H}$. Os resultados foram separados em duas figuras para melhor visualização dos valores obtidos para $\mathcal{H} \leq 64$.

Para cada valor de $\mathcal{H}$, parte-se de uma situação com apenas um nó (ou seja, todas as entradas da DHT estão concentradas num só nó), e vão-se acrescentando[5] nós, um-a-um, até que $\mathcal{N} = \mathcal{H}$ (*i.e.*, cada nó comporta uma só entrada). À medida que se acrescentam nós, é aplicado o procedimento da secção 3.3.4.1, para minimizar $\overline{\sigma}[\mathcal{Q}(n)]$. Assim, as figuras apresentam, para cada valor de $\mathcal{H}$ e de $\mathcal{N}$, o valor correspondente da métrica $\overline{\sigma}[\mathcal{Q}(n)]$.

Da observação das figuras resulta a identificação de i) uma gama de variação, e de ii) um padrão de evolução, comuns para os vários valores de $\mathcal{H}$. De facto, $\overline{\sigma}[\mathcal{Q}(n)]$ oscila entre valores mínimos que podem chegar a zero, e máximos que crescem linearmente até ao máximo absoluto de ≈ 0.35 (ou, equivalentemente, de 35%); os mínimos são obtidos quando $\mathcal{H}$ é divisível por $\mathcal{N}$ o que, sendo $\mathcal{H}$ uma potência de 2, acontece apenas quando $\mathcal{N}$ é também uma potência de 2; o máximo absoluto é obtido aproximadamente a meio do intervalo entre dois mínimos consecutivos. Este padrão repete-se para valores $\mathcal{H} > 1024$.

[5]Apresentam-se só resultados para a adição de nós. A remoção exibe uma evolução inversa/simétrica.

Outra conclusão relevante, que se extrai da observação das figuras, pela comparação dos resultados obtidos para um mesmo valor de $\mathcal{N}$ e vários valores de $\mathcal{H}$, é a de que quanto maior for $\mathcal{H}$, menor é $\overline{\sigma}[\mathcal{Q}(n)]$, ou seja, melhor é a qualidade da distribuição. Esta observação encontra justificação na seguinte argumentação, de carácter intuitivo: para o mesmo número de nós, um maior número total de entradas permite ajustar, de forma mais precisa, as quotas reais às ideais. Desta maneira, a fixação de um valor de $\mathcal{H}$ revela-se contra-producente, quer sob o ponto de vista da limitação natural que impõe ao grau de distribuição da DHT (pois $\mathcal{N} \leq \mathcal{H}$ é um invariante intrínseco de qualquer DHT), quer pelo facto de, para valores de $\mathcal{N}$ próximos de $\mathcal{H}$, a qualidade da distribuição tender a piorar[6].

3.4 Modelo M2: Dist. Homogénea com Hashing Dinâmico

A secção anterior concluiu pela apresentação de argumentação que, indirectamente, parece favorecer a opção por abordagens baseadas em Hashing Dinâmico, em vez de Hashing Estático. Com efeito, se $\mathcal{H}$ não for fixo, então poderemos tentar variá-lo, em função de $\mathcal{N}$, de forma a assegurar permanentemente bons níveis da qualidade da distribuição. Por exemplo, no contexto da figura 3.1.a), com $\mathcal{H} = 512$, a qualidade da distribuição deteriora-se substancialmente quando $\mathcal{N}$ ultrapassa 256; uma hipótese de resolver o problema seria então a *duplicação* do número de entradas $\mathcal{H}$, de 512 para 1024; de facto, a curva para $\mathcal{H} = 1024$, é bem mais favorável no intervalo $256 < \mathcal{N} < 512$ do que a curva para $\mathcal{H} = 512$. Note-se que a *quadriplicação* (etc.) de $\mathcal{H}$ permitiria obter ainda melhores resultados ...

O raciocínio anterior, que advoga o aumento do número de entradas da DHT (por *duplicação*, *quadriplicação*, etc.), configura uma abordagem de Hashing Dinâmico em que $\mathcal{H}$ cresce *por antecipação*, na tentativa de manter elevados níveis da qualidade da distribuição (em vez de uma abordagem de Hashing Dinâmico mais simples, como seria a baseada em *duplicações* sucessivas de $\mathcal{H}$, que ocorrem *por necessidade*, *i.e.*, quando $\mathcal{N}$ ultrapassar $\mathcal{H}$).

Duplicando-se o número de entradas apenas quando $\mathcal{N}$ ultrapassar $\mathcal{H}$ (em vez de $\mathcal{H}/2$, ou $\mathcal{H}/4$, etc.) e aplicando-se o mesmo Procedimento de (Re)Distribuição do modelo M1, obtêm-se os resultados apresentados pela figura 3.2, também expressos na métrica $\overline{\sigma}[\mathcal{Q}(n)]$.

Como se pode observar, a figura 3.2 tem uma relação directa com as figuras 3.1.a) e 3.1.b): a figura 3.2 obtém-se considerando, para cada uma das 10 curvas das figuras 3.1.a) e 3.1.b), apenas o pulso de pico mais elevado, sendo que, para um certo valor de $\mathcal{H}$, esse pulso ocorre para $\mathcal{H}/2 < \mathcal{N} \leq \mathcal{H}$. Por exemplo, i) para $512 < \mathcal{N} \leq 1024$, os valores da figura 3.2 correspondem aos valores da curva $\mathcal{H} = 1024$ da figura 3.1.a); ii) para $256 < \mathcal{N} \leq 512$, os valores da figura 3.2 correspondem aos valores da curva $\mathcal{H} = 512$ da figura 3.1.a); iii) etc.

Os valores de $\overline{\sigma}[\mathcal{Q}(n)]$ da figura 3.2 continuam pois a oscilar entre o mesmo mínimo (de zero) e máximo (de 0.35) registados para o nosso modelo com $\mathcal{H}$ constante, o que indica claramente que a simples opção por uma estratégia de Hashing Dinâmico não é suficiente para melhorar a qualidade da distribuição, sendo necessários mecanismos complementares. Neste contexto, é também preciso garantir que a um nó são atribuídas, permanentemente, entradas suficientes para assegurar a qualidade da distribuição desejada (ver a seguir).

[6]Excepto para valores de $\mathcal{N}$ vizinhos de potências de 2, bem como de outros mínimos locais esporádicos.

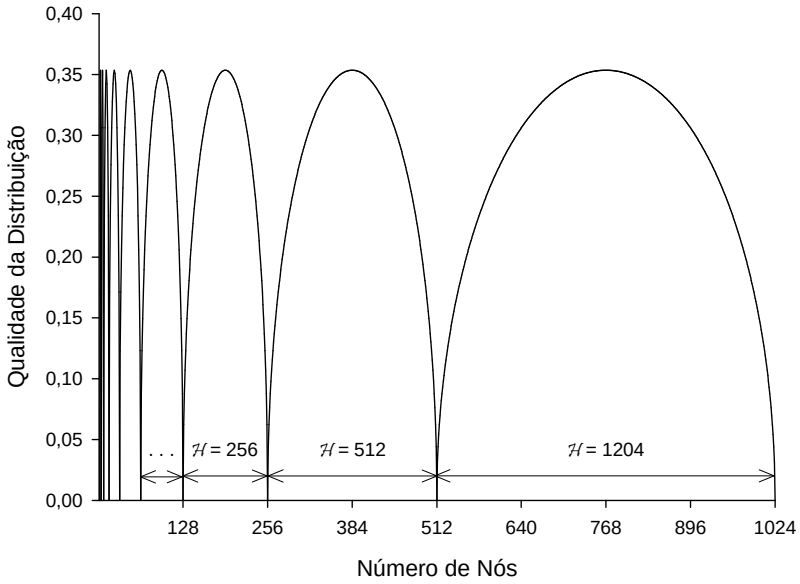


Figura 3.2: $\overline{\sigma}[\mathcal{Q}(n)]$ para $\mathcal{N} \leq \mathcal{H}$ e $\mathcal{H} \in \{2, 4, 8, ..., 1024\}$.

A fixação de um número mínimo de entradas da DHT, que deve ser assegurado a qualquer um dos seus nós, permite a prossecução de uma abordagem de Hashing Dinâmico em que $\mathcal{H}$ cresce *por antecipação*, com o objectivo de manter $\overline{\sigma}[\mathcal{Q}(n)]$ entre limites parametrizáveis. Esta abordagem configura o modelo M2 de Distribuição Homogénea, descrito de seguida.

3.4.1 Número Mínimo de Entradas por Nó

Seja $\mathcal{H}_{min}(n)$ um parâmetro do modelo M2, que define o "número mínimo admissível de entradas da DHT, a garantir para qualquer nó". Assim, para um número total de nós $\mathcal{N}$, deve-se ter $\mathcal{H} \geq \mathcal{N} \times \mathcal{H}_{min}(n)$. Como $\mathcal{H}$ tem que ser uma potência de 2 (por definição), então $\mathcal{H}$ deve ser a potência de 2 mais próxima de $\mathcal{N} \times \mathcal{H}_{min}(n)$, por excesso, ou seja:

$$\mathcal{H} = 2^{\mathcal{L}} \text{ com } \mathcal{L} = ceil(log_2[\mathcal{N} \times \mathcal{H}_{min}(n)]) \tag{3.11}$$

, em que $ceil(x)$ é x se x for inteiro, ou é o inteiro mais próximo de x por excesso. Por exemplo: i) com $\mathcal{N} = 3$ e $\mathcal{H}_{min}(n) = 1$ temos que $\mathcal{H} = 2^{ceil(log_2(3))} = 2^{ceil(1.58)} = 2^2 = 4$; ii) já com $\mathcal{N} = 3$ e $\mathcal{H}_{min}(n) = 2$ temos que $\mathcal{H} = 2^{ceil(log_2(6))} = 2^{ceil(2.58)} = 2^3 = 8$; iii) etc.

3.4.2 Procedimento de (Re)Distribuição

Sempre que se adicionar um nó à DHT[7], calcula-se $\mathcal{H}$ através da fórmula 3.11. Se o novo valor de $\mathcal{H}$ coincidir com o valor anterior, aplica-se o Procedimento de (Re)Distribuição do modelo M1, para minimizar $\overline{\sigma}[\mathcal{Q}(n)]$. Caso contrário, sendo o novo $\mathcal{H}$ o dobro do

[7]Considera-se apenas o crescimento da DHT, sendo que o decrescimento prossegue uma lógica simétrica.

anterior[8], duplica-se o número de entradas da DHT e só depois se aplica o Procedimento de (Re)Distribuição. Esta duplicação implica a subdivisão de todas as entradas em duas, passando a ser necessário mais um bit para definir a identidade das entradas da DHT, o que configura uma abordagem de Hashing Dinâmico (embora com diferenças relevantes sobre abordagens de referência, com as quais fazemos o devido confronto na secção 3.9).

3.4.3 Qualidade da Distribuição

A figura 3.3 ilustra o efeito de diferentes valores $\mathcal{H}_{min}(n) \in \{1, 2, 4, 8\}$ sobre a métrica $\overline{\sigma}[\mathcal{Q}(n)]$, à medida que se acrescentam novos nós à DHT, até ao máximo de $\mathcal{N} = 1024$.

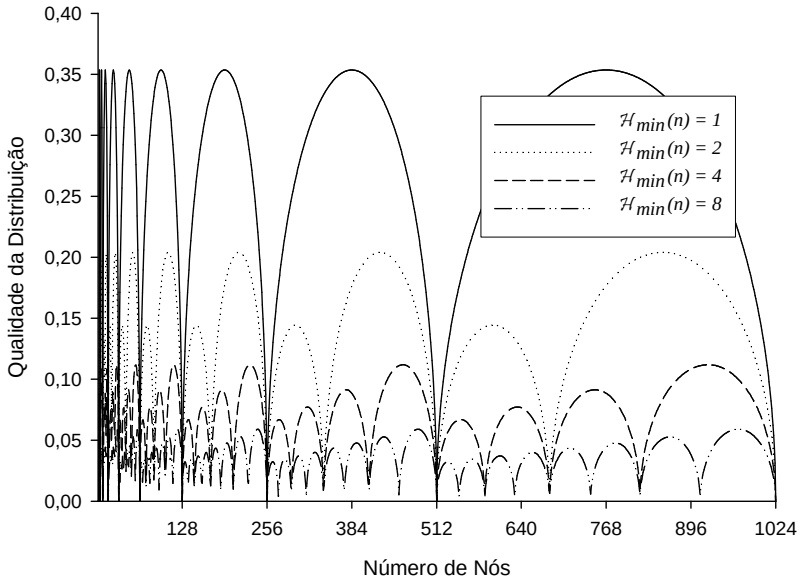


Figura 3.3: $\overline{\sigma}[\mathcal{Q}(n)]$ para $\mathcal{N} \leq \mathcal{H}$ e $\mathcal{H}_{min}(n) \in \{1, 2, 4, 8\}$.

Para $\mathcal{H}_{min}(n) = 1$, a curva resultante coincide com a da figura 3.2, como era esperado. Para $\mathcal{H}_{min}(n) > 1$ verifica-se que a métrica $\overline{\sigma}[\mathcal{Q}(n)]$ vai reduzindo sensivelmente para metade (em termos médios), à medida que $\mathcal{H}_{min}(n)$ duplica, através dos valores $\{2, 4, 8\}$. Na prática, cada vez que $\mathcal{H}_{min}(n)$ cresce, antecipa-se o instante em que $\mathcal{H}$ duplica, o que contribui para minimizar mais o valor de $\overline{\sigma}[\mathcal{Q}(n)]$, para um mesmo número de nós, $\mathcal{N}$.

3.4.4 Número Máximo de Entradas por Nó

Se $\mathcal{H}_{min}(n)$ for potência de 2, da conjunção dessa restrição com a fórmula 3.11 resulta a definição de $\mathcal{H}_{max}(n) = 2 \times \mathcal{H}_{min}(n)$ como número máximo de entradas de qualquer nó.

[8]Note-se que $\mathcal{H}$ só cresce por duplicações sucessivas (bem como só decresce por subdivisões sucessivas).

3.4.5 Evolução de $\mathcal{H}$ e do Número Específico de Entradas por Nó

Tendo em conta a restrição $\mathcal{H}_{min}(n) \leq \mathcal{H}(n) \leq \mathcal{H}_{max}(n)$, e o Procedimento de (Re)Distribuição acima definido, a evolução de $\mathcal{H}$ e de $\mathcal{H}(n)$, a partir de $\mathcal{N} = 1$, segue um padrão:

- com $\mathcal{N} = 1$, $\mathcal{H}(n) = \mathcal{H}$ e a fórmula 3.11 dita $\mathcal{H} = \mathcal{H}_{min}(n)$ (que é potência de 2);
- na transição de $\mathcal{N} = 1$ para $\mathcal{N} = 2$, a mesma fórmula dita a duplicação de $\mathcal{H}$, para $2 \times \mathcal{H}_{min}(n)$ (única forma de garantir pelo menos $\mathcal{H}_{min}(n)$ entradas por cada um dos 2 nós); assim, imediatamente antes da atribuição de entradas ao segundo nó, o primeiro detém, momentaneamente, $\mathcal{H}_{max}(n) = 2 \times \mathcal{H}_{min}(n)$ entradas; após a atribuição, o número de entradas regressa ao valor mínimo de $\mathcal{H}_{min}(n)$, para ambos;
- na passagem de $\mathcal{N} = 2$ para $\mathcal{N} = 3$, a fórmula 3.11 dita de novo a duplicação de $\mathcal{H}$ (pois é a única forma de garantir pelo menos $\mathcal{H}_{min}(n)$ entradas por cada um dos 3 nós); donde, imediatamente antes da atribuição de entradas ao terceiro nó, os dois nós anteriores ficam, momentaneamente, com $\mathcal{H}_{max}(n) = 2 \times \mathcal{H}_{min}(n)$ entradas, cada; após essa atribuição, haverá nós com menos de $\mathcal{H}_{max}(n)$ entradas, pois houve nós antigos que tiveram de ceder entradas ao novo[9]; complementarmente, haverá nós com mais de $\mathcal{H}_{min}(n)$ entradas, pois o número total de nós, $\mathcal{N}$, ainda não é suficiente para garantir a repartição perfeita entre eles, do total de entradas, $\mathcal{H}$;
- na passagem de $\mathcal{N} = 3$ para $\mathcal{N} = 4$, o número global de entradas, $\mathcal{H}$, ainda é suficiente para garantir $\mathcal{H}_{min}(n)$ entradas por nó; como $\mathcal{N} = 4$ é uma potência de 2, então $\mathcal{N}$ divide $\mathcal{H}$ (divisão inteira perfeita), donde resulta $\mathcal{H}(n) = \mathcal{H}_{min}(n), \forall n \in N$;
- o padrão evolutivo anterior repete-se, entre dois valores consecutivos de $\mathcal{H}$.

3.4.6 Evolução do Número Médio de Entradas por Nó

As figuras 3.4.a) e 3.4.b) ilustram o padrão evolutivo do "número médio de entradas por nó", $\overline{\mathcal{H}}(n) = \mathcal{H}/\mathcal{N}$, para $\mathcal{H}_{min}(n) = 8$. A figura 3.4.b) amplia a figura 3.4.a), para $\mathcal{N} \leq 32$.

Verifica-se assim que: i) $\mathcal{H}_{min}(n) = 8 \leq \overline{\mathcal{H}}(n) < \mathcal{H}_{max}(n) = 16\ \forall \mathcal{N} \in \{1, 2, ..., 1024\}$; ii) o valor máximo da média $\overline{\mathcal{H}}(n)$ converge para $\mathcal{H}_{max}(n)$ à medida que $\mathcal{N}$ cresce. A convergência do máximo de $\overline{\mathcal{H}}(n)$ para $\mathcal{H}_{max}(n)$ deduz-se facilmente: sendo $\mathcal{N}$ uma potência de 2, então $\overline{\mathcal{H}}(n) = \mathcal{H}_{min}(n)$; imediatamente antes da adição de mais um nó, $\mathcal{H}$ duplica e $\overline{\mathcal{H}}(n) = \mathcal{H}_{max}(n)$; existirão assim $\mathcal{H} = \mathcal{N} \times \mathcal{H}_{max}(n)$ entradas, a repartir por $\mathcal{N} + 1$ nós, sendo que o novo $\overline{\mathcal{H}}(n)$ será $[\mathcal{N} \times \mathcal{H}_{max}(n)]/[\mathcal{N}+1]$, correspondente ao valor máximo neste novo estágio da evolução da DHT; ora, à medida que $\mathcal{N}$ aumenta, é evidente que a razão $[\mathcal{N} \times \mathcal{H}_{max}(n)]/[\mathcal{N} + 1]$ converge para $\mathcal{H}_{max}(n)$, como se pretendia demonstrar.

[9] Note-se que, para um número pequeno de nós, todos terão que contribuir com entradas para um novo nó; com um número maior, apenas um subconjunto terá que contribuir; posteriormente, veremos que esse subconjunto é de tamanho limitado, o que contribui para a escalabilidade do modelo M2.

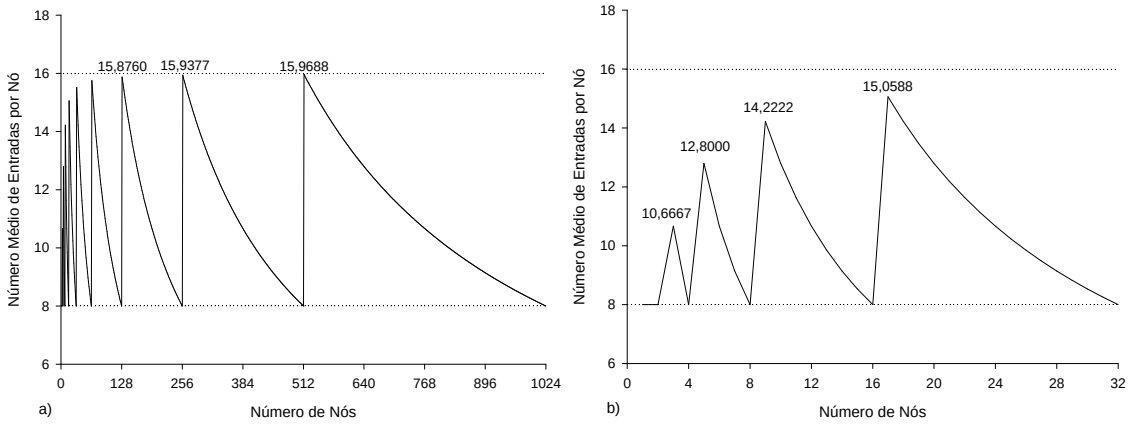


Figura 3.4: $\overline{\mathcal{H}}(n)$ para $\mathcal{H}_{min}(n) = 8$ e a) $\mathcal{N} \leq 1024$, b) $\mathcal{N} \leq 32$.

3.4.7 Invariantes do Modelo M2

As restrições e o padrão evolutivo anteriores são sintetizáveis nos seguintes *invariantes*:

$\mathcal{I}_1$: o número específico de entradas de qualquer nó, $\mathcal{H}(n)$, varia entre os limites mínimo $\mathcal{H}_{min}(n)$ e máximo $\mathcal{H}_{max}(n)$, comuns a todos os nós, com $\mathcal{H}_{max}(n) = 2 \times \mathcal{H}_{min}(n)$:

$$\mathcal{H}_{min}(n) \leq \mathcal{H}(n) \leq \mathcal{H}_{max}(n) = 2 \times \mathcal{H}_{min}(n), \forall n \in N \tag{3.12}$$

$\mathcal{I}_2$: em qualquer instante, o total de entradas da DHT, $\mathcal{H}$, obedece à fórmula 3.11;

$\mathcal{I}_3$: $\mathcal{H}_{min}(n)$ é potência de 2 (donde $\mathcal{H}_{max}(n) = 2 \times \mathcal{H}_{min}(n)$ é também potência de 2);

$\mathcal{I}_4$: sempre que $\mathcal{N}$ for potência de 2, então deve-se ter $\mathcal{H}(n) = \mathcal{H}_{min}(n), \forall n \in N$.

Os invariantes $\mathcal{I}_1$ e $\mathcal{I}_3$ aplicam-se também ao número médio de entradas por nó, $\overline{\mathcal{H}}(n)$.

3.4.8 Breve Análise de Escalabilidade

Nesta secção finalizamos a introdução do modelo M2, analisando a evolução de grandezas que fornecem indicações sobre a escalabilidade do modelo durante a adição de nós à DHT[10].

Número de Nós Doadores: Comecemos por analisar a evolução do "número de nós doadores", $\mathcal{N}^-(n^+)$, correspondente ao número de nós que, face à adição de um nó n^+ à DHT, terão que ceder a n^+ pelo menos uma entrada. As figuras 3.5.a) e 3.5.b), mostram a evolução de $\mathcal{N}^-(n^+)$ para $\mathcal{H}_{min} = 8$. A figura 3.5.b) amplia a figura 3.5.a) para $\mathcal{N} \leq 32$.

A evolução de $\mathcal{N}^-(n^+)$ atravessa três *estágios* diferentes, mais evidentes na figura 3.5.b):

[10]Os resultados obtidos são também aplicáveis, por simetria, à remoção de nós.

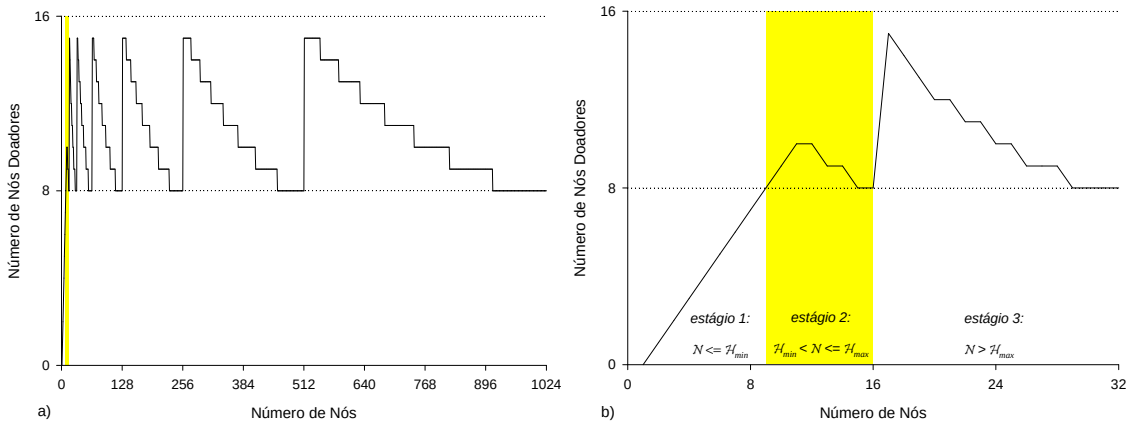


Figura 3.5: $\mathcal{N}^-(n^+)$ para $\mathcal{H}_{min}(n) = 8$ e a) $\mathcal{N} \leq 1024$, b) $\mathcal{N} \leq 32$.

1. estágio 1 ($1 \leq \mathcal{N} \leq \mathcal{H}_{min}(n)$) em que $\mathcal{N}^-(n^+) = \mathcal{N} - 1$: havendo menos nós que o número mínimo de entradas por nó então, quando se adiciona o $\mathcal{N}$'ésimo nó, todos os $\mathcal{N} - 1$ nós anteriores contribuirão com uma ou mais entradas para o novo nó;

2. estágio 2 ($\mathcal{H}_{min}(n) < \mathcal{N} \leq \mathcal{H}_{max}(n)$) em que $\mathcal{N}^-(n^+) \gtrapprox \mathcal{H}_{min}(n)$: apenas um subconjunto de aproximadamente $\mathcal{H}_{min}(n)$ nós cederá entradas a cada novo nó;

3. estágio 3 ($\mathcal{N} > \mathcal{H}_{max}(n)$) em que $\mathcal{H}_{min}(n) \leq \mathcal{N}^-(n^+) < \mathcal{H}_{max}(n)$: após $\mathcal{N}$ atingir $\mathcal{H}_{max}(n)$ (que é potência de 2), $\mathcal{N}^-(n^+)$ diminui de um máximo, próximo de $\mathcal{H}_{max}(n)$, para o mínimo $\mathcal{H}_{min}(n)$, quando $\mathcal{N}$ atinge a próxima potência de 2; este padrão evolutivo de $\mathcal{N}^-(n^+)$ repete-se entre potências de 2 consecutivas para $\mathcal{N}$.

Resumindo, $\mathcal{N}^-(n^+)$ escala linearmente com $\mathcal{N}$ no estágio 1, mantém-se mais ou menos constante durante o estágio 2 e, no estágio 3, permanece delimitado entre $\mathcal{H}_{min}(n)$ e $\mathcal{H}_{max}(n)$[11]. Assim, à excepção do estágio 1, os valores de $\mathcal{N}^-(n^+)$ são de ordem $O(1)$ relativamente a $\mathcal{N}$, o que denota uma excelente escalabilidade (traduzida pelo facto de, após o estágio 1, e à medida que $\mathcal{N}$ aumenta, a razão $\mathcal{N}^-(n^+)/\mathcal{N}$ diminuir exponencialmente).

Número de Entradas Transferidas: Para além do número $\mathcal{N}^-(n^+)$ de nós chamados a intervir sempre que se altera o conjunto de nós da DHT, é também importante estimar a carga adicional em que incorrem. O "número total de entradas transferidas dos nós doadores para um novo nó", $\mathcal{H}^-[\mathcal{N}^-(n^+)]$, fornece uma indicação indirecta dessa carga[12].

A figura 3.6 mostra a evolução de $\mathcal{H}^-[\mathcal{N}^-(n^+)]$, de novo para $\mathcal{H}_{min}(n) = 8$, mas agora apenas para $\mathcal{N} \leq 32$. Ainda na mesma figura, representa-se também i) a evolução de $\mathcal{N}^-(n^+)$, tal como apresentada previamente na figura 3.5.b), bem como ii) a evolução do número médio de entradas por nó, $\overline{\mathcal{H}}(n)$, representada previamente pela figura 3.4.b).

[11] A utilização de outros valores $\mathcal{H}_{min}(n)$ preserva, naturalmente, o mesmo tipo de escalabilidade.

[12] Note-se que a re-associação de entradas da DHT a outro nó poderá ter implicações para além da mera transferência de registos. Por exemplo, dependendo da estratégia (centralizada ou distribuída) usada para a *localização* das entradas, poderá ser necessário actualizar informação de localização noutros nós para além dos directamente envolvidos na transferência. Esta problemática será tratada no próximo capítulo.

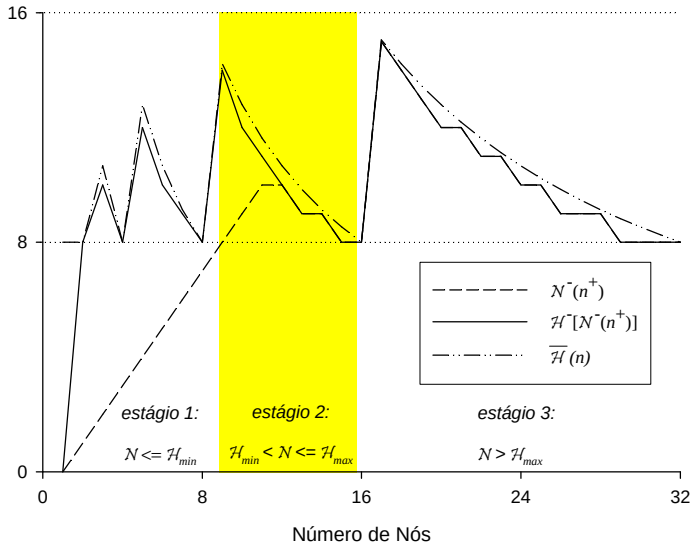


Figura 3.6: $\mathcal{H}^-[\mathcal{N}^-(n^+)]$, $\overline{\mathcal{H}}(n)$ e $\mathcal{N}^-(n^+)$, para $\mathcal{N} \leq 32$ e $\mathcal{H}_{min}(n) = 8$.

A evolução de $\mathcal{H}^-[\mathcal{N}^-(n^+)]$ é categorizável nos mesmos estágios da evolução de $\mathcal{N}^-(n^+)$: até meados do estágio 2, tem-se $\mathcal{H}^-[\mathcal{N}^-(n^+)] > \mathcal{N}^-(n^+)$, donde cada nó doador cede, em termos médios, mais de uma entrada; o valor médio da cedência, que denotamos por $\overline{\mathcal{H}}^-[\mathcal{N}^-(n^+)]$, é de ≈ 2.75 entradas, por cada nó doador; a partir da segunda metade do estágio 2, tem-se $\mathcal{H}^-[\mathcal{N}^-(n^+)] = \mathcal{N}^-(n^+)$, *i.e.*, já há nós suficientes para que todo e qualquer nó doador ceda apenas uma entrada. A relação entre $\mathcal{H}^-[\mathcal{N}^-(n^+)]$ e $\mathcal{N}^-(n^+)$ permite deduzir para $\mathcal{H}^-[\mathcal{N}^-(n^+)]$ o mesmo tipo de escalabilidade deduzida para $\mathcal{N}^-(n^+)$.

Por fim, é de registar a proximidade dos valores de $\overline{\mathcal{H}}(n)$ e $\mathcal{H}^-[\mathcal{N}^-(n^+)]$, coerente com os valores reduzidos da métrica de qualidade $\overline{\sigma}[\mathcal{Q}(n)]$. Assim, o número de entradas atribuído a um novo nó, $\mathcal{H}^-[\mathcal{N}^-(n^+)]$, anda próximo do número médio de entradas dos outros nós, $\overline{\mathcal{H}}(n)$, embora tenda a ser ligeiramente inferior, devido ao facto de o algoritmo usado na simulação do Procedimento de (Re)Distribuição favorecer os nós mais antigos, no acerto final do número de entradas por nó (uma das possibilidades previstas na secção 3.3.4.1).

3.5 Modelo M3: Dist. Heterogénea com Hashing Estático

O modelo M3 de Distribuição Heterogénea com Hashing Estático, é o sucessor directo do modelo M1 de Distribuição Homogénea. O modelo M3 assenta num conceito de *nó virtual*.

3.5.1 Quota Ideal de um Nó

No contexto do modelo M3 de Distribuição Heterogénea, a participação de um nó numa DHT mede-se 1) em termos *reais*, por unidades de *grão fino*, correspondentes às *entradas* da DHT e 2) em termos *ideais*, por unidades de *grão grosso*, correspondentes a *nós virtuais*[13].

[13]Existem razões históricas para a escolha desta designação, que serão esclarecidas na secção 3.8.

No primeiro caso, o facto de um nó n ser responsável por $\mathcal{H}(n)$ entradas da DHT equivale a dizer que o nó detém uma quota real $\mathcal{Q}^r(n) = \mathcal{H}(n)/\mathcal{H}$ (rever expressão 3.3) da DHT. No segundo caso, a associação de $\mathcal{V}(n)$ nós virtuais a um nó n traduz-se numa quota ideal

$$\mathcal{Q}^i(n) = \frac{\mathcal{V}(n)}{\mathcal{V}}, \forall n \in N \tag{3.13}$$

, com

$$\mathcal{V} = \sum_{n \in N} \mathcal{V}(n) \tag{3.14}$$

3.5.2 Conceito de Nó Virtual

Em conjunto, as grandezas $\mathcal{V}(n)$ reflectem o mérito dos vários nós de N para a realização da DHT: quanto maior for o número de nós virtuais de um nó, $\mathcal{V}(n)$, maior deverá ser o número de entradas da DHT atribuído ao nó, $\mathcal{H}(n)$. Dito de outra forma: os valores $\mathcal{V}(n)$ reflectem a eventual *heterogeneidade* dos nós da DHT, à luz de determinados critérios de caracterização dos nós, considerados relevantes sob o ponto de vista da realização da DHT. No contexto do modelo M3, essa heterogeneidade assume-se *dinâmica*, ou seja, parte-se do princípio de que a variação das grandezas $\mathcal{V}(n)$ é arbitrária e independente para cada nó. Consequentemente, as quotas ideais estão também sujeitas a variações dinâmicas, pois a modificação de um valor $\mathcal{V}(n)$ afecta todas as quotas $\mathcal{Q}^i(n)$ (note-se que a adição/remoção de nós provoca a redefinição de $\mathcal{V}$ e logo das quotas $\mathcal{Q}^i(n)$, como acontecia no modelo M1).

3.5.2.1 Definição do Número de Nós Virtuais de um Nó Computacional

A definição de valores concretos para $\mathcal{V}(n)$ (e, por consequência, para $\mathcal{V}$) pode obedecer a lógicas diversas: 1) por exemplo, esses valores podem reflectir tão somente a eventual heterogeneidade física dos nós do *cluster*, sendo as diferentes capacidades de base (em termos de recursos de hardware) dos nós que ditam diferentes níveis de participação (estáticos) numa DHT; 2) ou então, podem levar em conta (cumulativamente ou não) os níveis de utilização (dinâmicos) dos recursos; 3) outra hipótese, é definir $\mathcal{V}$ de forma a satisfazer-se um requisito de capacidade de armazenamento global da DHT (o que pressupõe associar uma certa capacidade a cada nó virtual), ou de paralelismo potencial no acesso à DHT (sendo $\mathcal{V} \leq \mathcal{N}$, o paralelismo máximo ocorre quando $\mathcal{N} = \mathcal{V}$); na prática, qualquer lógica que dite a necessidade de uma distribuição heterogénea da DHT, e que seja acompanhada de critérios para a definição dos diferentes níveis de participação dos nós, é compatível com o conceito de nó virtual; posteriormente, nos capítulos 5 e 6 esta problemática é revisitada.

3.5.3 Métrica de Qualidade

Com base na definição anterior de *nó virtual*, uma primeira métrica de qualidade do modelo M3 seria a *Soma dos Desvios Absolutos* (SDA), tal como definida pela fórmula 3.7,

tendo em conta que a *quota real* de cada nó continua a fornecida pela fórmula 3.3, mas a definição da quota ideal é agora dada pela fórmula 3.13. Todavia, a expressão resultante para $SDA[\mathcal{Q}(n)]$ não se enquadra no conceito clássico de *Soma dos Desvios Absolutos* [GC97], em que se mede a distância de cada um dos valores de uma única série Z face a um único valor de referência z', ou seja, $SDA[Z, z'] = \sum_{z \in Z} |z - z'|$. Com efeito, no contexto da fórmula 3.7, existem agora múltiplos valores de referência, $\mathcal{Q}^i(n)$, um para cada nó n, em vez de um só, como acontecia com uma distribuição homogénea. Contudo, essa assimetria não impede a definição de grandezas equivalentes ao *Desvio Padrão Absoluto* e ao *Desvio Padrão Relativo*, deitando-se mão, para o efeito, do conceito de *Variância Amostral Ponderada* [GC97]. De facto, no cenário heterogéneo, faz sentido pesar, para cada nó n, a aproximação de $\mathcal{Q}^r(n)$ à *Média* específica $\mathcal{Q}^i(n)$. As fórmulas 3.15 e 3.16 fornecem pois um *Desvio Padrão Absoluto Pesado* e um *Desvio Padrão Relativo Pesado*:

$$\sigma[\mathcal{Q}(n)] = \sqrt{\sum_{n \in N} [[\mathcal{Q}^r(n) - \mathcal{Q}^i(n)]^2 \times \mathcal{Q}^i(n)]} = \sqrt{\sum_{n \in N} [\Delta[\mathcal{Q}(n)]^2 \times \mathcal{Q}^i(n)]} \tag{3.15}$$

$$\overline{\sigma}[\mathcal{Q}(n)] = \frac{\sigma[\mathcal{Q}(n)]}{\sum_{n \in N} [\mathcal{Q}^i(n)]^2} \tag{3.16}$$

com

$$\Delta[\mathcal{Q}(n)] = \mathcal{Q}^r(n) - \mathcal{Q}^i(n), \forall n \in N \tag{3.17}$$

Assim, na fórmula de $\sigma[\mathcal{Q}(n)]$, cada factor $\Delta[\mathcal{Q}(n)^2$ é pesado por $\mathcal{Q}^i(n)$, em que $\mathcal{Q}^i(n)$ é a *quota ideal* de n, sendo que, por definição, $\sum_{n \in N} \mathcal{Q}^i(n) = 1$. Adicionalmente, na fórmula de $\overline{\sigma}[\mathcal{Q}(n)]$, divide-se $\sigma[\mathcal{Q}(n)]$ por uma *Média Pesada* ideal, dada por $\sum_{n \in N} [\mathcal{Q}^i(n)]^2$.

É de realçar que as fórmulas 3.15 e 3.16 são também aplicáveis ao modelo M1. De facto, estas fórmulas representam a generalização das fórmulas 3.9 e 3.10 a um cenário heterogéneo. A demonstração desta asserção é simples. Assim, sendo um cenário homogéneo caracterizado por[14] $\mathcal{Q}^i(n) = \overline{\mathcal{Q}}(n) = 1/\mathcal{N}$ (rever fórmula 3.6), então a fórmula 3.15 origina

$$\sigma[\mathcal{Q}(n)] = \sqrt{\sum_{n \in N} [[\mathcal{Q}^r(n) - \overline{\mathcal{Q}}(n)]^2 \times \overline{\mathcal{Q}}(n)]} = \sqrt{\frac{\sum_{n \in N} [\mathcal{Q}^r(n) - \overline{\mathcal{Q}}(n)]^2}{\mathcal{N}}} \tag{3.18}$$

, que corresponde à fórmula 3.9. Adicionalmente, a fórmula 3.16 pode ser reescrita como

$$\overline{\sigma}[\mathcal{Q}(n)] = \frac{\sigma[\mathcal{Q}(n)]}{\sum_{n \in N} [\mathcal{Q}^i(n)]^2} = \frac{\sigma[\mathcal{Q}(n)]}{\sum_{n \in N} [\frac{1}{\mathcal{N}}]^2} = \frac{\sigma[\mathcal{Q}(n)]}{\mathcal{N} \times [\frac{1}{\mathcal{N}}]^2} = \frac{\sigma[\mathcal{Q}(n)]}{\frac{1}{\mathcal{N}}} = \frac{\sigma[\mathcal{Q}(n)]}{\overline{\mathcal{Q}}(n)} \tag{3.19}$$

, o que demonstra a correspondência efectiva da fórmula 3.16 com a fórmula 3.10.

[14] Quando todos os nós têm o mesmo número de nós virtuais, $\mathcal{Q}^i(n) = \mathcal{V}(n)/\mathcal{V} = \mathcal{V}(n)/[\mathcal{V}(n) \times \mathcal{N}] = 1/\mathcal{N}$.

3.5.4 Função Objectivo

Prosseguindo uma lógica semelhante à usada pelo modelo M1, a função objectivo continua a ser a "minimização de $\overline{\sigma}[\mathcal{Q}(n)]$", sendo que agora $\overline{\sigma}[\mathcal{Q}(n)]$ é calculável pela fórmula 3.16.

3.5.5 Procedimento de (Re)Distribuição

No modelo M3 de Distribuição Heterogénea, o Procedimento de (Re)Distribução distribui as $\mathcal{H}$ entradas da DHT pelos seus N nós, *proporcionalmente* ao número de nós virtuais destes. O procedimento tira partido de uma *tabela de distribuição* (TD) de esquema $< n, \mathcal{H}(n), \mathcal{V}(n) >$, semelhante à preconizada pelo modelo M1 (rever secção 3.3.4.2), mas que agora também regista o número de nós virtuais $\mathcal{V}(n)$ por cada nó $n \in N$.

Assim, numa primeira fase do procedimento, (re)define-se $\mathcal{H}(n) = \mathcal{H}_{div}(n) = [\mathcal{H}\ div\ \mathcal{V}] \times \mathcal{V}(n) = int\ [\mathcal{Q}^i(n) \times \mathcal{H}]$, para todos os $n \in N$ (com *int* denotando a parte inteira de um real). Desta iteração resulta i) a distribuição de um número total de entradas $\mathcal{H}_{div} = \sum_{n \in N} \mathcal{H}_{div}(n)$ e ii) um certo número total de entradas, $\mathcal{H}_{mod} = \mathcal{H} - \mathcal{H}_{div}$, por atribuir.

Numa segunda fase, consideram-se os nós ordenados por $\Delta[\mathcal{Q}(n)] = \mathcal{Q}^r(n) - \mathcal{Q}^i(n)$, a diferença entre a quota real e a ideal (rever fórmula 3.17). Se $\Delta[\mathcal{Q}(n)] > 0$, os nós em causa apresentam um *excesso* de entradas, face ao número ideal de entradas a que têm direito; se $\Delta[\mathcal{Q}(n)] = 0$, os nós n detêm o número de entradas exactamente previsto pela sua quota ideal; se $\Delta[\mathcal{Q}(n)] < 0$, os nós n exibem um *débito* de entradas. As $\mathcal{H}_{mod}$ entradas remanescentes são então distribuídas pelos nós em *débito*, proporcionalmente ao valor absoluto de $\Delta[\mathcal{Q}(n)]$; cada nó receberá um número de entradas adicionais, $\mathcal{H}_{mod}(n)$.

Numa terceira fase, a execução do algoritmo 3.1 garante a minimização efectiva de $\overline{\sigma}[\mathcal{Q}(n)]$[15]:

Algoritmo 3.1: Modelo M3: Algoritmo de Miminização de $\overline{\sigma}[\mathcal{Q}(n)]$.

1. ordenar os nós da TD pelo valor $\Delta[\mathcal{Q}(n)]$
2. atribuir uma entrada do nó com maior $\Delta[\mathcal{Q}(n)]$, ao nó com menor $\Delta[\mathcal{Q}(n)]$
3. calcular o valor de $\overline{\sigma}[\mathcal{Q}(n)]$ e compará-lo com o valor anterior ao passo 1:
   (a) se o novo valor de $\overline{\sigma}[\mathcal{Q}(n)]$ é menor, prosseguir no passo 1;
   (b) caso contrário, assumir a distribuição (TD) anterior ao passo 2 e terminar;

Este algoritmo comporta um número reduzido de iterações, dado que o ponto de partida é uma distribuição que, na maior parte das vezes, é *óptima* ou já muito próxima de o ser.

Por fim, uma vez definida a nova distribuição da DHT, a fase da Transferência de Entradas entre Nós, que conduz à efectivação da distribuição, prosseguiria uma metodologia semelhante à inicialmente apresentada no contexto do modelo M1 (rever secção 3.3.4.2).

[15] Na realidade, este algoritmo seria suficiente para produzir a nova versão da *tabela de distribuição*, garantindo a minimização de $\overline{\sigma}[\mathcal{Q}(n)]$, sem necessidade de se realizar a primeira e segunda fases em separado; todavia, a realização prévia dessas fases permite que o número de iterações da terceira fase seja residual.

3.5.6 Qualidade da Distribuição

O facto de cada nó ter direito a várias entradas (uma, pelo menos) da DHT, por cada nó virtual, contribuirá, por si só, para melhorar a *qualidade da distribuição*, face a um cenário homogéneo regulado pelo modelo M1, em que cada nó reclama apenas uma entrada[16] (vide resultados apresentados na figura 3.2, e na figura 3.3 para $\mathcal{H}_{min}(n) = 1$). Com efeito, já verificamos que quanto maior for o número total de entradas, $\mathcal{H}$, para um mesmo número $\mathcal{N}$ de nós, melhor tende a ser a qualidade da distribuição (rever figuras 3.1.a) e 3.1.b)).

Ora, a utilização de nós virtuais contribui precisamente para o aumento do número global de entradas ($\mathcal{H}$) necessárias para um mesmo número global de nós ($\mathcal{N}$): i) por definição de nó virtual, tem-se necessariamente $\mathcal{V} \geq \mathcal{N}$ (*i.e.*, há pelo menos um nó virtual por nó); ii) por definição de nó virtual, cada nó virtual confere o direito a pelo menos uma entrada, donde $\mathcal{H} \geq \mathcal{V} \geq \mathcal{N}$; iii) logo, o número de entradas necessárias para distribuir a DHT por $\mathcal{V}$ nós virtuais, dado por $\mathcal{H}(\mathcal{V}) = 2^{ceil(log_2(\mathcal{V}))}$, é pelo menos igual ou maior que o número de entradas necessárias para distribuir a DHT por $\mathcal{N}$ nós, dado por $\mathcal{H}(\mathcal{N}) = 2^{ceil(log_2(\mathcal{N}))}$.

3.6 Modelo M4: Dist. Heterogénea com Hashing Dinâmico

Nesta secção faz-se a extensão do modelo M2 de Distribuição Homogénea com Hashing Dinâmico, para o modelo equivalente M4 de Distribuição Heterogénea. Adicionalmente, o modelo M4 apoia-se no conceito de *nó virtual* introduzido na descrição do modelo M3.

3.6.1 Número Mínimo de Entradas por Nó Virtual

No modelo M4, a manutenção da *qualidade da distribuição* (neste caso, heterogénea) entre determinados limites assenta num parâmetro $\mathcal{H}_{min}(v)$, que define "o número mínimo admissível de entradas da DHT, a garantir a qualquer nó computacional, por cada um dos seus nós virtuais". Desta forma, o número (dinâmico) de entradas da DHT, $\mathcal{H}$, é dado por

$$\mathcal{H} = 2^{\mathcal{L}} \text{ com } \mathcal{L} = ceil(log_2[\mathcal{V} \times \mathcal{H}_{min}(v)]) \tag{3.20}$$

, tendo em conta que $\mathcal{H}$ deve ser a potência de 2 mais próxima de $\mathcal{V} \times \mathcal{H}_{min}(v)$, por excesso.

Note-se que v é apenas um artefacto notacional do modelo M4, permitindo descrevê-lo de forma similar ao modelo M2; o conceito de "identificador de nó virtual" não faz sentido à luz do conceito de nó virtual usado até agora ("nó virtual como número de entradas").

3.6.2 Invariantes do Modelo M4

O próximo passo é estabelecer que $\mathcal{H}_{min}(v)$ é também potência de 2, após o que se pode definir o seguinte conjunto de invariantes, semelhantes aos definidos para o modelo M2:

[16] Embora depois possa receber mais, por via dos acertos necessários.

$\mathcal{I}_1$: o número médio de entradas por nó virtual, $\overline{\mathcal{H}}(v)$, varia entre $\mathcal{H}_{min}(v)$ e $\mathcal{H}_{max}(v)$:

$$\mathcal{H}_{min}(v) \leq \overline{\mathcal{H}}(v) \leq \mathcal{H}_{max}(v) = 2 \times \mathcal{H}_{min}(v) \tag{3.21}$$

$\mathcal{I}_2$: em qualquer instante, o total de entradas da DHT, $\mathcal{H}$, obedece à fórmula 3.20;

$\mathcal{I}_3$: $\mathcal{H}_{min}(v)$ é potência de 2 (donde $\mathcal{H}_{max}(v) = 2 \times \mathcal{H}_{min}(v)$ é também potência de 2);

$\mathcal{I}_4$: sempre que $\mathcal{V}$ for potência de 2, então deve-se ter $\overline{\mathcal{H}}(v) = \mathcal{H}_{min}(v)$.

3.6.3 Evolução de $\mathcal{H}$ e do Número Médio de Entradas por Nó Virtual

A evolução do número global de entradas da DHT, $\mathcal{H}$, é governada pelos invariantes anteriores: sempre que $\mathcal{V}$ se altera (seja por modificação do número de nós virtuais de um ou mais nós da DHT, seja por adição/remoção de nós à/da DHT) recalcula-se $\mathcal{H}$ pela fórmula 3.20; se $\mathcal{H}$ se mantiver, é aplicado de imediato o Procedimento de (Re)Distribuição do modelo M3, adequado a distribuições heterogéneas (rever secção 3.5.5); caso contrário, duplica-se/divide-se $\mathcal{H}$ e só depois se executa o referido Procedimento de (Re)Distribuição.

O número médio de entradas por nó virtual, $\overline{\mathcal{H}}(v)$, é dado por $\mathcal{H}/\mathcal{V}$. A evolução de $\overline{\mathcal{H}}(v)$, segue um padrão semelhante à evolução de $\overline{\mathcal{H}}(n)$ sob o modelo M2 (rever secção 3.4.6).

3.6.4 Qualidade da Distribuição

A métrica de *qualidade da distribuição* do modelo M4 é igual à do M3 (rever fórmula 3.16).

A figura 3.7 exibe, no contexto do modelo M4, o efeito de diferentes valores $\mathcal{H}_{min}(v) \in \{1, 2, 4, 8\}$ sobre $\overline{\sigma}[\mathcal{Q}(n)]$. Para cada valor $\mathcal{H}_{min}(v)$, a métrica $\overline{\sigma}[\mathcal{Q}(n)]$ é calculada à medida que a DHT ganha nós computacionais, até que o total de nós virtuais, $\mathcal{V}$, ultrapasse 1024; o número de nós virtuais de cada nó, $\mathcal{V}(n)$, é aleatório, extraído do intervalo $\{1, 2, ..., 8\}$.

Embora a figura resulte de uma distribuição heterogénea específica, verificou-se experimentalmente que outras distribuições, geradas em resultado de outras sequências aleatórias de $\mathcal{V}(n)$, exibem o mesmo comportamento oscilatório para $\overline{\sigma}[\mathcal{Q}(n)]$, enquadrado por limites semelhantes, com a diferença de que os valores de pico para $\overline{\sigma}[\mathcal{Q}(n)]$ são atingidos para diferentes números de nós, $\mathcal{N}$, o que encontra explicação no facto de o número global de nós virtuais, $\mathcal{V}$, progredir de forma diferente, para diferentes sequências aleatórias de $\mathcal{V}(n)$.

O comportamento da métrica $\overline{\sigma}[\mathcal{Q}(n)]$ sob M4 segue um padrão similar ao observado no cenário homogéneo de M2 (rever figura 3.3). Assim, a duplicação dos valores de $\mathcal{H}_{min}(v)$ provoca a divisão, para aproximadamente metade, dos valores de $\overline{\sigma}[\mathcal{Q}(n)]$, sendo que a duplicação dos valores de $\mathcal{H}_{min}(n)$, no contexto da figura 3.3, tinha a mesma consequência.

Adicionalmente, os valores que delimitam $\overline{\sigma}[\mathcal{Q}(n)]$ são coerentes com os observados para o cenário homogéneo de M2, na figura 3.3. Com efeito, tendo-se registado, no cenário heterogéneo simulado, um número médio de nós virtuais por nó de $\overline{\mathcal{V}}(n) \approx 4.5$, então esperava-se que a qualidade da distribuição fosse melhor que a fornecida no cenário homogéneo com $\mathcal{H}_{min}(n) = 4$. Ora, como se pode comprovar observando a figura 3.8, os valores de $\overline{\sigma}[\mathcal{Q}(n)]$

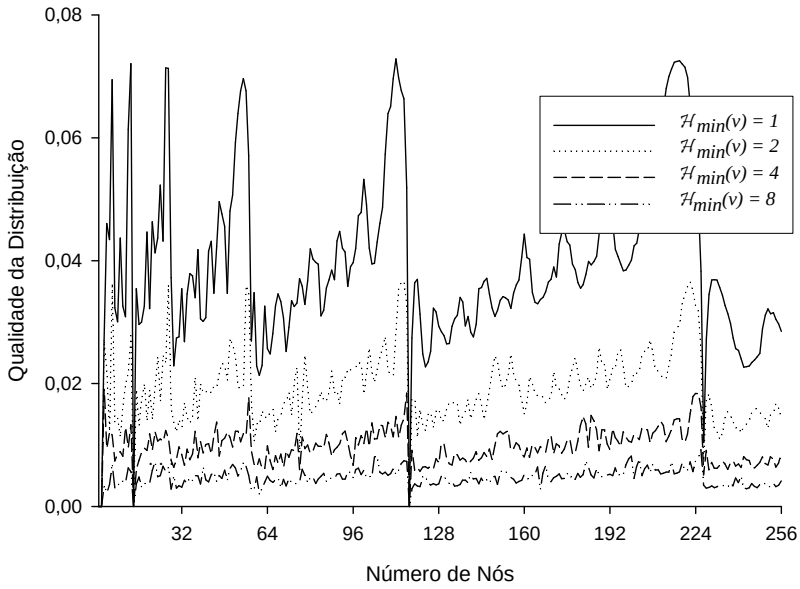


Figura 3.7: $\overline{\sigma}[\mathcal{Q}(n)]$ com $\mathcal{N} \leq 256$, $\mathcal{V} \gtrapprox 1024$, $\mathcal{V}(n) = random(1,8)$ e $\mathcal{H}_{min}(v) \in \{1,2,4,8\}$.

da figura 3.7 apresentam-se em geral inferiores aos da figura 3.3 para $\mathcal{H}_{min}(n) = 4$.

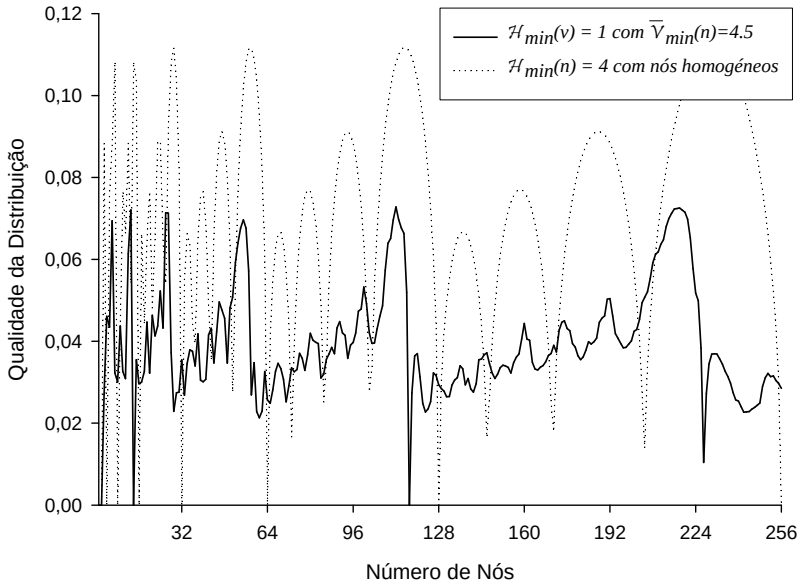


Figura 3.8: $\overline{\sigma}[\mathcal{Q}(n)]$ para $\mathcal{H}_{min}(n) = 4$ (figura 3.3) versus $\mathcal{H}_{min}(v) = 1$ (figura 3.7).

3.7 Modelo M4': Modelo Alternativo ao Modelo M4

Nesta secção apresenta-se o modelo M4', como modelo de Distribuição Heterogénea com Hashing Dinâmico, alternativo ao modelo M4. Basicamente, o modelo M4' é uma reformu-

lação do M4, baseada no refinamento do conceito de nó virtual introduzido no modelo M3. Esse refinamento representa uma re-aproximação conceptual aos modelos de Distribuição Homogénea (M1 e M2). De facto, nesses modelos, cada nó persegue uma mesma quota (ideal) da DHT e, como veremos, o mesmo se passa no modelo M4', para cada nó virtual.

Basicamente, o modelo M4' serve para demonstrar a flexibilidade do conceito de nó virtual. Na prática, o modelo M4 é preferível: ambos geram o mesmo número de entradas por nó, mas o Procedimento de (Re)Distribuição de M4 é mais simples/eficiente que o de M4'.

3.7.1 Nó Virtual como Conjunto de Entradas

Nos modelos M3 e M4 de Distribuição Heterogénea, cada nó n da DHT tem direito a um certo número total de entradas, $\mathcal{H}(n)$, por virtude de exibir um certo número total de nós virtuais, $\mathcal{V}(n)$. Na prática, é como se, *por cada um* dos seus $\mathcal{V}(n)$ nós virtuais, um nó n tivesse direito a um certo *número de entradas* que, em termos médios, será $\mathcal{H}(n)/\mathcal{V}(n)$; esta média local será semelhante à média global $\mathcal{H}/\mathcal{V}$, uma vez que o Procedimento de (Re)Distribuição atribui entradas aos nós *proporcionalmente* ao seu número de nós virtuais. Em suma, os modelos M3 e M4 assentam na visão do "nó virtual como número de entradas".

Uma visão refinada, derivável da anterior, é a do "nó virtual como (sub)conjunto de entradas". Em termos globais, esta visão corresponde a considerar que o conjunto global das entradas da DHT, H, é particionado em $\mathcal{V}$ subconjuntos (um por cada nó virtual), com um número médio de entradas de $\mathcal{H}/\mathcal{V}$; em termos locais, equivale a considerar que o conjunto local das $H(n)$ entradas atribuídas a um nó n é particionado em $\mathcal{V}(n)$ subconjuntos, com um número médio de $\mathcal{H}(n)/\mathcal{V}(n)$ entradas. Os *subconjuntos* de entradas são *nós virtuais* que, à luz do seu novo conceito, necessitam de alguma forma de identificação[17]; neste contexto, passa também a ser lícito fazer referência às "entradas de um nó virtual".

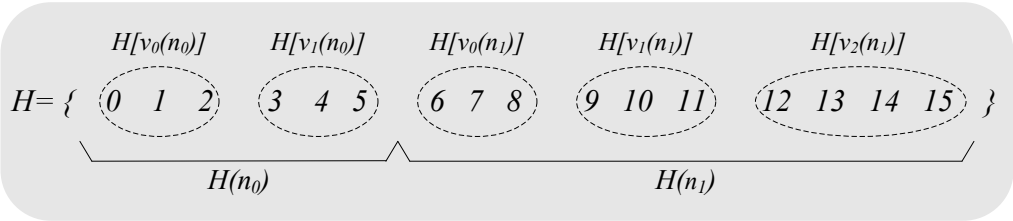


Figura 3.9: Particionamento de H em Nós Virtuais.

A figura 3.9 ilustra a aplicação do novo conceito de nó virtual e apoia a introdução de notação adequada. Assim, começamos por convencionar o uso da notação $v_i(n)$ (com $i = 0, 1, ..., \mathcal{V}(n) - 1$) para identificar individualmente os nós virtuais de um nó computacional n, com $\mathcal{V}(n)$ nós virtuais; neste contexto, o conjunto dos nós virtuais de n é $V(n) = \{v_0(n), v_1(n), ..., v_{\mathcal{V}(n)-1}(n)\}$. O conjunto global dos nós virtuais, V, é então dado por

$$\bigcup_{n \in N} V(n) = V \tag{3.22}$$

[17] Ao contrário do conceito anterior, que a dispensava – rever secção 3.6.1.

, relação que complementa a definida anteriormente pela expressão 3.14: $\mathcal{V} = \sum_{n \in N} \mathcal{V}(n)$.

Prosseguindo, denotamos por $H[v(n)]$ o *conjunto de entradas* associadas ao nó virtual v do nó n, e por $\mathcal{H}[v(n)]$ o *número de entradas* correspondente. Neste contexto, e com base na notação já introduzida, podemos exprimir de imediato uma série de relações básicas:

$\mathcal{R}_1$: a união das entradas dos nós virtuais de um nó, corresponde às entradas do nó

$$\bigcup_{v(n) \in V(n)} H[v(n)] = H[V(n)] = H(n), \forall n \in N \tag{3.23}$$

$\mathcal{R}_2$: a soma do cardinal dos nós virtuais de um nó, é igual ao número de entradas do nó

$$\sum_{v(n) \in V(n)} \mathcal{H}[v(n)] = \mathcal{H}[V(n)] = \mathcal{H}(n), \forall n \in N \tag{3.24}$$

$\mathcal{R}_3$: a intersecção das entradas de todos os nós virtuais de um nó é o conjunto vazio

$$\bigcap_{v(n) \in V(n)} H[v(n)] = \oslash, \forall n \in N \tag{3.25}$$

$\mathcal{R}_4$: a união das entradas dos nós virtuais de todos os nós é igual às entradas da DHT

$$\bigcup_{v(n) \in V(n), n \in N} H[v(n)] = \bigcup_{n \in N} H[V(n)] = \bigcup_{n \in N} H(n) = H \tag{3.26}$$

$\mathcal{R}_5$: a soma do cardinal dos nós virtuais de todos os nós é o total de entradas da DHT

$$\sum_{v(n) \in V(n), n \in N} \mathcal{H}[v(n)] = \sum_{n \in N} \mathcal{H}[V(n)] = \sum_{n \in N} \mathcal{H}(n) = \mathcal{H} \tag{3.27}$$

$\mathcal{R}_6$: a intersecção das entradas dos nós virtuais de todos os nós é o conjunto vazio

$$\bigcap_{v(n) \in V(n), n \in N} H[v(n)] = \oslash \tag{3.28}$$

Por exemplo, na figura 3.9, temos $H(n_0) = H[v_0(n_0)] \cup H[v_1(n_0)] = \{0, 1, 2\} \cup \{3, 4, 5\} = \{0, ..., 5\}$ e, correspondentemente, $\mathcal{H}(n_0) = \mathcal{H}[v_0(n_0)] + \mathcal{H}[v_1(n_0)] = 3 + 3 = 6$. Adicionalmente, $H = H(n_0) \cup H(n_1) = \{0, ..., 5\} \cup \{6, ..., 15\}$ com $\mathcal{H} = \mathcal{H}(n_0) + \mathcal{H}(n_0) = 6 + 10 = 16$.

Convém ainda referir que, no contexto da figura 3.9, as entradas da DHT que constituem cada nó virtual são inteiros sequenciais por uma questão de simplificação da representação. De facto, não há qualquer exigência de contiguidade dessas entradas[18], podendo um nó virtual v ser constituído por quaisquer entradas da DHT, desde que em número $\mathcal{H}(v)$, e respeitando todas as relações e restrições definidas pelas fórmulas anteriores (3.24 a 3.26).

As relações $\mathcal{R}_1$ e $\mathcal{R}_3$ garantem o *particionamento* de $H(n)$ em $\mathcal{V}(n)$ subconjuntos. Equivalentemente, $\mathcal{R}_4$ e $\mathcal{R}_6$ asseguram o *particionamento* de H em $\mathcal{V}$ subconjuntos.

[18] Embora, como veremos no capítulo 5, a contiguidade possa surgir na distribuição inicial de uma DHT, em resultado de um algoritmo que permite a qualquer nó inicial deduzir, de forma autónoma, a distribuição.

3.7.2 Quota Real e Ideal de um Nó Virtual

Em termos globais, a quota real e ideal da DHT, para qualquer nó virtual $v \in V$, será:

$$\mathcal{Q}^r(v) = \frac{\mathcal{H}(v)}{\mathcal{H}} \tag{3.29}$$

$$\mathcal{Q}^i(v) = \overline{\mathcal{Q}}(v) = \frac{1}{\mathcal{V}} \tag{3.30}$$

Assim, num determinado instante, um nó virtual $v \in V$ comporta a fracção $\mathcal{Q}^r(v)$ das entradas de H, quando deveria, idealmente, comportar a fracção $\mathcal{Q}^i(v)$, igual para todos. A definição de $\mathcal{Q}^i(v)$ é conceptualmente semelhante à de $\mathcal{Q}^i(n)$ para os modelos M1 e M2.

3.7.3 Quota Real e Ideal de um Nó Computacional

As quotas de um nó podem ser expressas em função das quotas dos seus nós virtuais, da forma que se segue. Assim, para a quota ideal de um nó, dada pela fórmula 3.13, tem-se

$$\mathcal{Q}^i(n) = \frac{\mathcal{V}(n)}{\mathcal{V}} = \sum_{v(n)\in V(n)} \frac{1}{\mathcal{V}} = \sum_{v(n)\in V(n)} \mathcal{Q}^i[v(n)] \tag{3.31}$$

Análogamente, para a quota real de um nó computacional, dada pela fórmula 3.3, tem-se

$$\mathcal{Q}^r(n) = \frac{\mathcal{H}(n)}{\mathcal{H}} = \sum_{v(n)\in V(n)} \frac{\mathcal{H}[v(n)]}{\mathcal{H}} = \sum_{v(n)\in V(n)} \mathcal{Q}^r[v(n)] \tag{3.32}$$

, tendo em conta a fórmula 3.24 que estabelece que $\mathcal{H}(n) = \mathcal{H}[V(n)] = \sum_{v(n)\in V(n)} \mathcal{H}[v(n)]$.

3.7.4 Métrica de Qualidade

O grau de aproximação entre quotas reais e ideais, dos nós virtuais, pode-se medir através de métricas semelhantes às desenvolvidas anteriormente, para os modelos M1 e M2. Assim, o facto de $\mathcal{Q}^i(v)$ ser igual para todos os nós virtuais (pois $\mathcal{Q}^i(v)$ coincide com a *Média* $\overline{\mathcal{Q}}(v)$) viabiliza a seguinte definição clássica de *Soma dos Desvios Absolutos* (SDA):

$$SDA[\mathcal{Q}(v)] = \sum_{v\in V} |\Delta[\mathcal{Q}(v)]| \tag{3.33}$$

com

$$\Delta[\mathcal{Q}(v)] = \mathcal{Q}^r(v) - \overline{\mathcal{Q}}(v), \forall v \in V \tag{3.34}$$

Neste contexto, definimos o *Desvio Padrão Absoluto*, correspondente a $SDA[\mathcal{Q}(v)]$, como

$$\sigma[\mathcal{Q}(v)] = \sqrt{\frac{\sum_{v \in V}[\mathcal{Q}^r(v) - \overline{\mathcal{Q}}(v)]^2}{\mathcal{V}}} = \sqrt{\frac{\sum_{v \in V} \Delta[\mathcal{Q}(v)]^2}{\mathcal{V}}} \tag{3.35}$$

e definimos o *Desvio Padrão Relativo*, correspondente ao *Desvio Padrão Absoluto*, como

$$\overline{\sigma}[\mathcal{Q}(v)] = \frac{\sigma[\mathcal{Q}(v)]}{\overline{\mathcal{Q}}(v)} \tag{3.36}$$

3.7.5 Função Objectivo

De forma coerente com as definições anteriores, o objectivo do modelo M4' é a "minimização de $\overline{\sigma}[\mathcal{Q}(v)]$". Esse objectivo é conciliável com o do modelo M4 ("minimização de $\overline{\sigma}[\mathcal{Q}(n)]$").

3.7.6 Procedimento de (Re)Distribuição

O Procedimento de (Re)Distribuição do modelo M4' é um misto do usado pelos modelos M4 e M1; as afinidades com o procedimento de M4 devem-se ao uso de nós virtuais; as afinidades com o de M1 devem-se à distribuição equitativa de entradas pelos nós virtuais.

Assim, sempre que $\mathcal{V}$ se alterar (por modificação do número de nós virtuais de um ou mais nós, ou por adição/remoção de nós à/da DHT) recalcula-se $\mathcal{H}$ pela fórmula 3.20; mantendo-se $\mathcal{H}$ constante, executa-se de imediato o resto do procedimento, semelhante ao do modelo M1; caso contrário, essa execução é precedida da duplicação/subdivisão de $\mathcal{H}$.

O resto do procedimento, semelhante ao do modelo M1, procura repartir o mais *equitativamente* possível as $\mathcal{H}$ entradas da DHT pelos $\mathcal{V}$ nós virtuais, conservando a indivisibilidade das entradas. Assim, numa primeira fase, atribuem-se $\mathcal{H}_{div}(v) = \mathcal{H}\ div\ \mathcal{V}$ entradas a cada nó virtual, sendo que $\mathcal{H}_{div}(v) = \mathcal{H}_{min}(v)$, igualdade que é uma consequência dos invariantes do modelo M4. Desta iteração resulta i) a distribuição de um número total de entradas $\mathcal{H}_{div} = \sum_{v \in V} \mathcal{H}_{div}(v)$, ii) um número total de entradas $\mathcal{H}_{mod} = \mathcal{H} - \mathcal{H}_{div}$ por atribuir e iii) a atribuição a cada nó de um total provisório de $\mathcal{H}_{div}(n) = \mathcal{H}_{div}(v) \times \mathcal{V}(n)$ entradas.

Numa segunda fase, as $\mathcal{H}_{mod}$ entradas são atribuídas, uma a uma, a outros tantos nós virtuais, (de forma que V acaba por ser divisível em dois subconjuntos, um em que cada nó virtual tem $\mathcal{H}_{div}(v)$ entradas, e outro em que cada nó virtual tem $\mathcal{H}_{div}(v)+1$ entradas).

Este procedimento é suficiente para minimizar a métrica $\overline{\sigma}[\mathcal{Q}(v)]$ do modelo M4', mas não garante a minimização da métrica $\overline{\sigma}[\mathcal{Q}(n)]$ do modelo M4; uma forma de o conseguir será escolher criteriosamente quais os nós virtuais beneficiários das $\mathcal{H}_{mod}$ entradas, em vez de atribuir essas entradas de forma indiscriminada (ou seja, considerando os nós virtuais anónimos, desligados de nós computacionais, situação em que qualquer nó virtual pode ser beneficiário); os beneficiários devem ser nós virtuais (quaisquer) dos nós computacionais com maior débito de entradas, pelo critério usado na segunda fase do Procedimento de (Re)Distribuição do modelo M3 (rever secção 3.5.5); aplicando esse critério e o algoritmo de ajuste associado, garante-se a minimização de $\overline{\sigma}[\mathcal{Q}(n)]$, além da minimização de $\overline{\sigma}[\mathcal{Q}(v)]$.

3.7.7 Comparação das Métricas de Qualidade dos Modelos M4 e M4'

Uma vez que o modelo M4' pretende ser alternativa ao modelo M4, importa verificar se as funções objectivo dos dois modelos são compatíveis, ou seja, se a qualidade da distribuição produzida por M4' em resultado da "minimização de $\overline{\sigma}[\mathcal{Q}(v)]$" está correlacionada com a qualidade da distribuição produzida por M4 em resultado da "minimização de $\overline{\sigma}[\mathcal{Q}(n)]$".

Embora a minimização de $\overline{\sigma}[\mathcal{Q}(v)]$ não garanta, por si só, a minimização de $\overline{\sigma}[\mathcal{Q}(n)]$, a sua evolução segue o mesmo padrão. A figura 3.10 ilustra um caso particular dessa evolução.

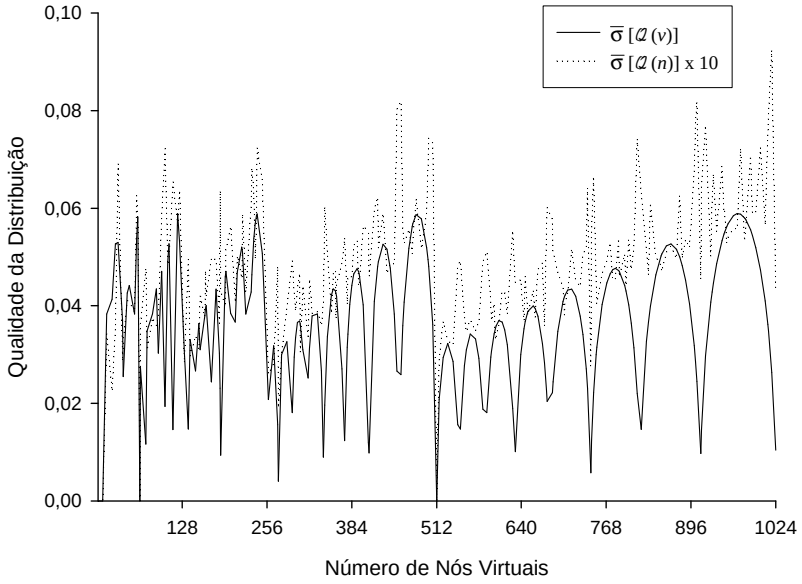


Figura 3.10: $\overline{\sigma}[\mathcal{Q}(v)]$ versus $\overline{\sigma}[\mathcal{Q}(n)]$, para $\mathcal{H}_{min}(v) = 8$ e $\mathcal{V} \leq 1024$.

Assim, a figura reproduz a evolução de $\overline{\sigma}[\mathcal{Q}(n)]$ para $H_{min}(v) = 8$ da figura 3.7, permitindo compará-la com a evolução de $\overline{\sigma}[\mathcal{Q}(v)]$ calculada nas mesmas condições. Para facilitar a comparação, os valores de $\overline{\sigma}[\mathcal{Q}(n)]$ foram ampliados 10 vezes em relação aos da figura 3.7. As métricas são apresentadas em função do número total de nós virtuais ($\mathcal{V}$) da DHT, medido após a adição de cada nó. A tendência global de evolução das métricas é semelhante, mas os seus valores pontuais apresentam tendências relativamente simétricas.

A figura 3.11 fornece uma visão alternativa da evolução dos valores da figura 3.10: para cada $\mathcal{V}$, a figura 3.11 apresenta a média (acumulada) de todos os valores das métricas anteriores a $\mathcal{V}$ (inclusive). Graficamente, é evidente a correlação entre as médias acumuladas $\overline{\overline{\sigma}}[\mathcal{Q}(n)]$ e $\overline{\overline{\sigma}}[\mathcal{Q}(n)]$ (analiticamente, essa correlação é de ≈ 0.83). Indirectamente, a figura 3.11 comprova a existência de uma correlação na evolução das métricas $\overline{\sigma}[\mathcal{Q}(v)]$ e $\overline{\sigma}[\mathcal{Q}(n)]$.

3.8 Comparação com Hashing Consistente

Tendo em vista a compreensão da relevância da qualidade da distribuição assegurada pelos nossos modelos, é necessária a sua comparação com outras abordagens ao particionamento, tendo-se elegido a abordagem do Hashing Consistente (HC) [KLL+97] para esse efeito.

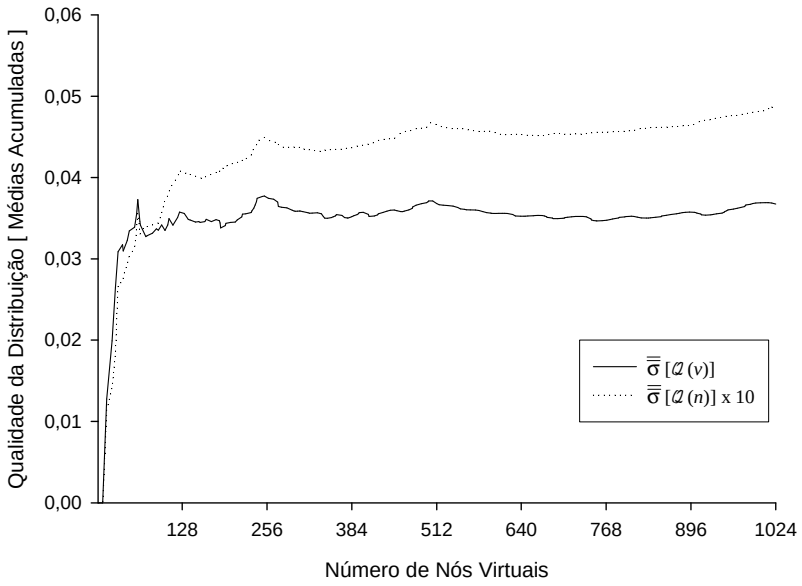


Figura 3.11: $\overline{\overline{\sigma}}[\mathcal{Q}(v)]$ versus $\overline{\overline{\sigma}}[\mathcal{Q}(n)]$, para $\mathcal{H}_{min}(v) = 8$ e $\mathcal{V} \leq 1024$.

Para além do facto de ser uma abordagem de referência da literatura, o Hashing Consistente é também representativo de uma classe em que se explora a aleatoriedade na tentativa de obter uma distribuição balanceada da DHT, o que reforça o interesse da sua comparação com os nossos modelos, os quais, de certa forma, se situam num extremo oposto, pelo facto de prosseguirem métodos essencialmente determinísticos para se atingir o mesmo objectivo. Adicionalmente, a investigação realizada no âmbito da tese contemplou também a temática da localização distribuída, tendo-se desenvolvido algoritmos de *localização acelerada* para grafos Chord (ver capítulo 5), o que representa uma motivação acrescida para a análise do Hashing Consistente, antecessor directo da abordagem Chord.

3.8.1 Mecanismo Base

Antes da comparação propriamente dita, recuperamos e complementamos a descrição do HC fornecida na secção 2.6.2.1, recorrendo a um dialecto mais próximo do usado na descrição dos nossos modelos. Em particular, e sem perda de generalidade, baseamos a descrição que se segue num contradomínio circular $H = \{0, 1, ..., 2^{\mathcal{L}_f} - 1)$, em vez do círculo $[0, 1)$.

Sob HC, o contradomínio H da função de *hash* estática f, de $\mathcal{L}_f$ bits, é particionado num certo número *bem definido* de subconjuntos/partições contínuos(as), com um número *aleatório* de entradas cada. Depois, a cada nó que suporta a DHT, é atribuído um certo número *bem definido* dessas partições: para um total de $\mathcal{N}$ nós da DHT, a abordagem HC advoga que cada nó deva receber pelo menos $k \times log_2\mathcal{N}$ partições, para que os nós suportem uma quota semelhante da DHT, ou seja, a atribuição de múltiplas partições da DHT a cada nó é feita na tentativa de assegurar uma *distribuição homogénea* da DHT. De facto, atribuindo-se *várias* partições a cada nó, a probabilidade de obter uma distribuição equitativa da DHT, entre os vários nós, é maior do que a correspondente à atribuição

de *uma só* partição a cada nó. Adicionalmente, a abordagem HC permite controlar a qualidade da distribuição através do parâmetro k: o aumento de k traduz-se no aumento do número (médio) de partições da DHT por nó, o que aumenta a qualidade da distribuição.

Mais especificamente, o particionamento é realizado considerando que, para cada nó n_j, são deduzíveis $k \times log_2\mathcal{N}$ pseudo-identificadores $v_i(n_j)$, designados por *nós virtuais*; a aplicação de uma função de hash f a cada nó virtual $v_i(n_j)$ gera um hash $f(v_i(n_j))$; a colecção de todos os hashes $f(v_i(n_j))$ (de todos os nós virtuais, de todos os nós computacionais), subdivide H num número equivalente de intervalos contíguos, delimitados por esses hashes.

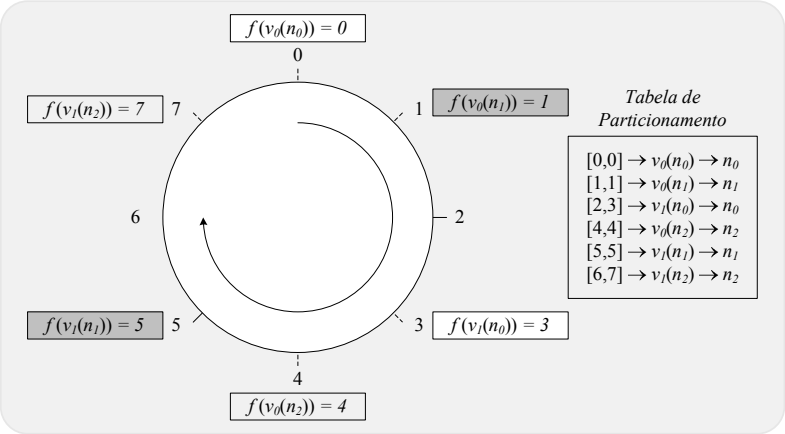


Figura 3.12: Visão do Hashing Consistente com Nós Virtuais.

A figura 3.12 ilustra a aplicação do particionamento do HC ao contradomínio H=$\{0, 1, ..., 7)$, orientado no sentido dos ponteiros do relógio, para um cenário em que i) a DHT é suportada por $\mathcal{N} = 3$ nós computacionais, correspondentes a $N = \{n_0, n_1, n_2\}$, e ii) $k = 1$ (pelo que são necessários $k \times log_2(\mathcal{N}) = log_2(3) \approx 2$ nós virtuais por cada nó computacional).

Na figura, cada hash $f(v_i(n_j))$ corresponde a um ponto do círculo H, e define um sector (intervalo): operando-se no sentido dos ponteiros do relógio, o sector inicia-se (exclusive) no hash $f(v'_i(n'_j))$ imediatamente anterior, e termina (inclusive) no próprio hash $f(v_i(n_j))$. Assim, o nó virtual responsável por um hash h é simplesmente o nó virtual v cujo valor $f(v)$ é o mais próximo de h (em distância euclidiana), no sentido dos ponteiros do relógio.

Saber a que nó computacional se associou um hash h implica determinar 1) a que intervalo pertence o hash h, 2) a que nó virtual pertence o intervalo e, finalmente, 3) a que nó computacional pertence o nó virtual (assumindo-se que o identificador do nó computacional pode ser deduzido do identificador do nó virtual, convenciona-se denotar essa dedução por $v_i(n_j) \rightarrow n_j$, em que n_j representa o nó computacional hospedeiro do nó virtual $v_i(n_j)$). A figura inclui uma tabela que sintetiza o *posicionamento* resultante do *particionamento*.

O conceito de *nó virtual*, aplicado no contexto do *particionamento* de DHTs, surge pela primeira vez na abordagem HC e, embora seja entendido, em primeira instância, como um pseudo-identificador, a sua associação unívoca a uma partição de H permite que também possa ser visto, de forma equivalente, como partição/subconjunto de entradas; essa visão representa uma aproximação ao conceito de nó virtual dos nossos modelos (aproximação

mais forte no caso do modelo M4'), viabilizando a sua comparação; note-se, todavia, que sob HC o número de entradas por nó virtual é *aleatório*, ao passo que, nos nossos modelos, o Procedimento de (Re)Distribuição procura homogeneizar esse número para todos os nós virtuais (sendo que, no caso específico do modelo M4, esse número é entabulado entre limites bem definidos – $\mathcal{H}_{min}(v)$ e $\mathcal{H}_{max}(v)$ –, derivados a partir de parâmetros do modelo).

3.8.2 Qualidade da Distribuição

A qualidade de uma distribuição originada sob Hashing Consistente pode ser medida recorrendo às métricas que definimos neste capítulo, designadamente no âmbito dos modelos M3 e M4 de distribuição heterogénea. Com efeito, esses modelos assentam num conceito de nó virtual compatível com o preconizado pelo Hashing Consistente. Adicionalmente, a métrica de qualidade utilizada por esses modelos (fórmula 3.16) é também aplicável a uma distribuição homogénea, o tipo de distribuição almejada pelo Hashing Consistente.

Para medir a qualidade de uma distribuição originada sob HC, recorrendo à fórmula 3.16, é necessário definir de forma adequada as quotas ideais e reais de cada nó da DHT. Como se pretende uma distribuição homogénea, então as quotas ideais são comuns a todos os nós, sendo dadas por $\mathcal{Q}^i(n) = 1/\mathcal{N}, \forall n \in N$. As quotas reais (dadas genericamente pela fórmula 3.3), poderão diferir entre nós, em resultado da distribuição aleatória de entradas.

Adicionalmente, e de forma a permitir uma comparação mais justa com os nossos modelos baseados em Hashing Dinâmico, o número total de entradas da DHT é recalculado para cada valor de $\mathcal{N}$. Basicamente, o cálculo procura fazer respeitar o invariante base do Hashing Consistente, segundo o qual cada nó deve receber pelo menos $\mathcal{V}_{min}(n) = k \times log_2\mathcal{N}$ nós virtuais; sendo necessária pelo menos uma entrada da DHT por cada nó virtual, então cada nó deverá receber pelo menos $\mathcal{H}_{min}(n) = \mathcal{V}_{min}(n)$ entradas, donde a DHT deverá ter pelo menos $\mathcal{H}_{min} = \mathcal{N} \times \mathcal{H}_{min}(n)$ entradas; por fim, o número efectivo de entradas da DHT, $\mathcal{H}$, obtém-se arredondando $\mathcal{H}_{min}$ à potência de 2 mais próxima, c.f. a fórmula 3.37:

$$\mathcal{H} = 2^{\mathcal{L}} \text{ com } \mathcal{L} = ceil[log_2(\mathcal{H}_{min})] \text{ e } \mathcal{H}_{min} = \mathcal{N} \times k \times log_2\mathcal{N} \tag{3.37}$$

Os valores de $\mathcal{H}$ assim calculados ficarão muito aquém do valor fixo previsto pelo HC com base em Hashing Estático, correspondente a 2^{160} (devido à utilização da função de hash SHA-1, de 160 bits). Ora, sendo verdade que um menor valor de $\mathcal{H}$ prejudicará a qualidade da distribuição do HC, também é verdade que tal limitação afectará os nossos modelos.

A figura 3.13 apresenta $\overline{\sigma}[\mathcal{Q}(n)]$, calculado pela fórmula 3.16, como medida da qualidade de distribuições geradas por Hashing Consistente, para vários valores de k e de $\mathcal{N}$. Assim, para cada k e para cada $\mathcal{N}$, calculou-se $\mathcal{H}$ pela fórmula 3.37 e repetiu-se o seguinte procedimento 10 vezes: 1) particionou-se H em $\mathcal{N} \times \mathcal{V}_{min}(n)$ nós virtuais; 2) atribuíram-se $\mathcal{V}_{min}(n)$ nós virtuais a cada nó; 3) calculou-se a métrica $\overline{\sigma}[\mathcal{Q}(n)]$. Cada valor $\overline{\sigma}[\mathcal{Q}(n)]$ apresentado no gráfico corresponde à média dos valores $\overline{\sigma}[\mathcal{Q}(n)]$ do passo 3). A figura evidencia ainda a melhoria da qualidade da distribuição que resulta do aumento de k[19].

[19]Recorde-se que, quanto *menor* for o valor de $\overline{\sigma}[\mathcal{Q}(n)]$, *melhor* é a qualidade da distribuição.

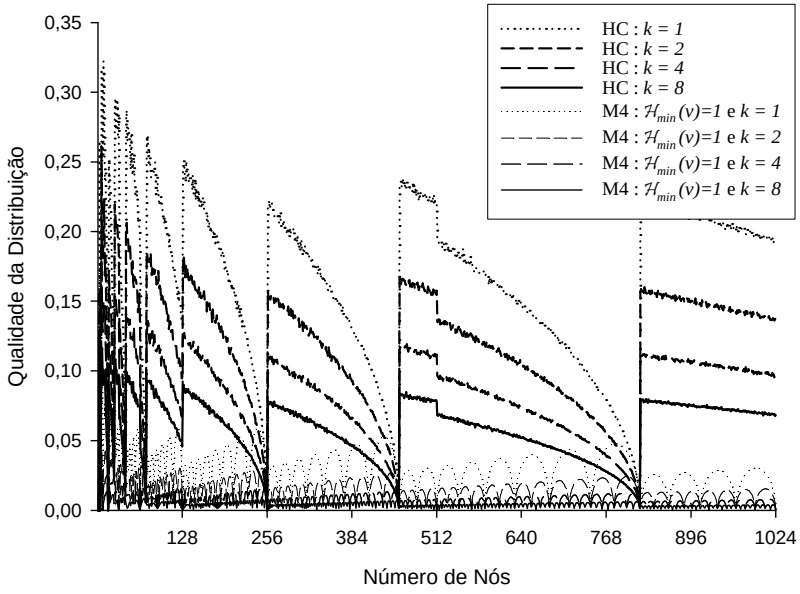


Figura 3.13: $\overline{\sigma}[\mathcal{Q}(n)]$ para Hashing Consistente e M4, com $k \in \{1, 2, 4, 8\}$ e $1 \leq \mathcal{N} \leq 1024$.

3.8.3 Comparação com o Modelo M4

Comparamos agora a qualidade da distribuição fornecida pelo Hashing Consistente, com a fornecida pelo modelo M4. Para que a comparação seja justa, as condições de avaliação de M4 têm de ser aproximadas, o mais possível, às da avaliação do Hashing Consistente. Assim, 1) definimos o número de nós virtuais por nó como $\mathcal{V}(n) = k \times log_2\mathcal{N}, \forall n \in N$ e 2) fixamos o número mínimo de entradas por nó virtual como $\mathcal{H}_{min}(v) = 1, \forall v \in V$[20]. Depois, para cada k e para cada $\mathcal{N}$, calcula-se $\mathcal{H}$ pela fórmula 3.37 (o que garante que o número (médio) de entradas por nó – $\overline{\mathcal{H}}(n)$ – será igual sob HC e M4) mas depois aplica-se o Procedimento de (Re)Distribuição do modelo M4, igual ao do M3 (rever secção 3.5.5).

A qualidade da distribuição, resultante da aplicação do modelo M4 nas condições anteriores, é também apresentada na figura 3.13, permitindo o seu confronto com a do Hashing Consistente. Como se pode observar, a qualidade oferecida pelo modelo M4 é substancialmente melhor, para os mesmos valores de k e de $\mathcal{N}$. É ainda visível o diferente tipo de impacto que a duplicação de k tem na qualidade da distribuição, para ambos os modelos. Assim, no Hashing Consistente, a duplicação de k traduz-se em decréscimos sucessivos de $\approx 30\%$, ao passo que, no modelo M4, essa duplicação se traduz em decréscimos de $\approx 50\%$.

3.8.4 Comparação com o Modelo M2

Uma outra forma de comparar o Hashing Consistente com os nossos modelos é verificar a hipótese de ser possível alcançar uma qualidade igual ou superior à do Hashing Consistente, mas com um número inferior de entradas por nó. Esta questão é particularmente relevante,

[20] Ao passo que, no contexto das simulações que deram origem à figura 3.7, 1) definimos $\mathcal{V}(n)$ aleatoriamente, como $\mathcal{V}(n) = random(1, 8)$ e 2) fizemos variar $\mathcal{H}_{min}(v)$ no intervalo $\{1, 2, 4, 8\}$.

porquanto um maior número de entradas por nó acarreta uma maior sobrecarga associada à gestão do seu armazenamento e do seu endereçamento (ver capítulo 5). Neste contexto, realizamos uma comparação com o nosso modelo M2, exclusivamente vocacionado para distribuição homogénea. Neste modelo, recorde-se, o conceito de nó virtual está ainda ausente, pelo que o número (médio) de entradas por nó tende a ser menor, face à utilização de nós virtuais. O modelo M2 suporta, todavia, a definição de um número mínimo de entradas por nó, dado pelo parâmetro $\mathcal{H}_{min}(n)$, o qual pode ser usado para aumentar a qualidade da distribuição, à custa, porém, de um aumento do número global de entradas.

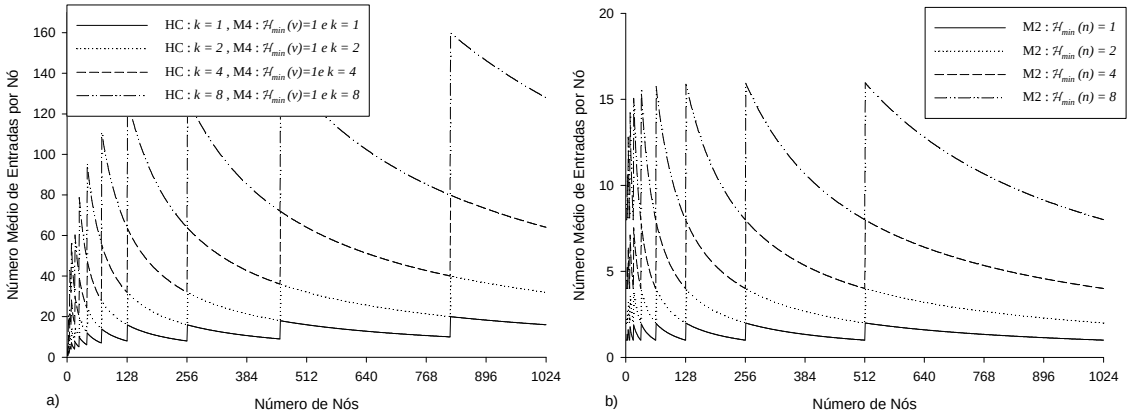


Figura 3.14: $\overline{\mathcal{H}}(n)$ para a) HC com $k \in \{1, 2, 4, 8\}$ e b) M2 com $\mathcal{H}_{min}(n) \in \{1, 2, 4, 8\}$.

Assim, começamos por apresentar, na figura 3.14, a evolução do número médio de entradas por nó – $\overline{\mathcal{H}}(n)$ –, para a) o Hashing Consistente (e M4[21]) e para b) o modelo M2. Como se pode observar, M2 exibe valores consideravelmente inferiores aos do Hashing Consistente. Todavia, resta saber se a qualidade da distribuição assegurada pelo modelo M2 é suficientemente competitiva (como a de M4) com a do Hashing Consistente. Neste contexto, a comparação gráfica dos valores para M2 (previamente apresentados na figura 3.3), com os valores para o Hashing Consistente (apresentados na figura 3.13), não é suficientemente elucidativa, dado que as curvas dos gráficos respectivos se sobrepõem. Alternativamente, a comparação pode ser feita de forma analítica, através de uma métrica que combine ambos os factores i) *número de entradas por nó* e ii) *qualidade da distribuição*, permitindo então definir qual a abordagem mais atractiva (se Hashing Consistente, se M2). Uma métrica possível para a comparação de dois modelos m' e m'', para um dado número de nós $\mathcal{N}$, é:

$$\mathcal{R}(m', m'', \mathcal{N}) = \frac{\overline{\sigma}[\mathcal{Q}(m', \mathcal{N})]}{\overline{\sigma}[\mathcal{Q}(m'', \mathcal{N})]} \times \frac{\overline{\mathcal{H}}(m', \mathcal{N})}{\overline{\mathcal{H}}(m'', \mathcal{N})} \tag{3.38}$$

$\mathcal{R}(m', m'', \mathcal{N})$ pesa o quociente da qualidade alcançada, multiplicando-o pelo quociente do número de entradas necessárias (note-se que se aplicássemos esta métrica à comparação de HC com M4, teríamos simplesmente[21] $\mathcal{R}(m', m'', \mathcal{N}) = \frac{\overline{\sigma}[\mathcal{Q}(m', \mathcal{N})]}{\overline{\sigma}[\mathcal{Q}(m'', \mathcal{N})]}$; neste caso, todavia, a comparação de HC com M4 (na qualidade da distribuição), pode ser feita de forma qualitativa, através da figura 3.13, não sendo necessário enveredar pela via analítica).

[21] Uma vez que, nas condições em que M4 e HC se simularam, $\overline{\mathcal{H}}(n)$ é igual para os dois modelos.

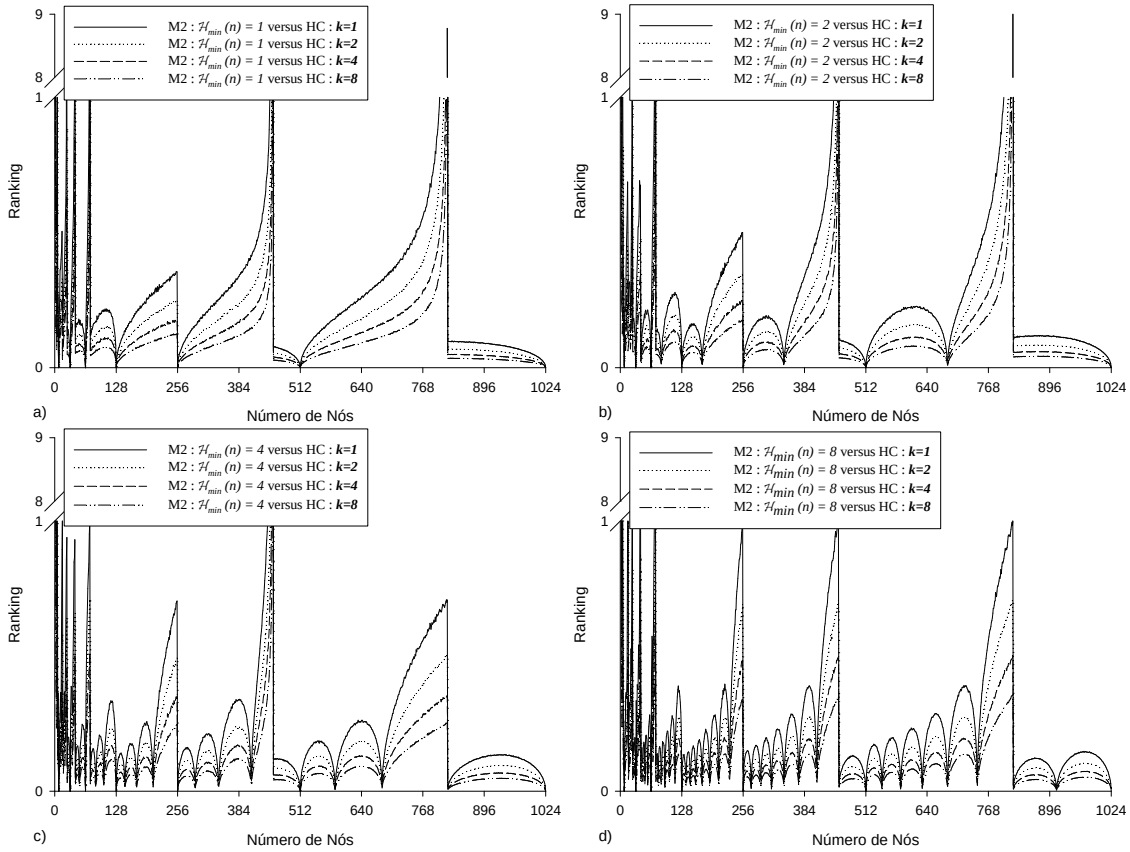


Figura 3.15: $\mathcal{R}$ para $1 \leq \mathcal{N} \leq 1024$, $k \in \{1, 2, 4, 8\}$ e $\mathcal{H}_{min}(n) =$ a) 1, b) 2, c) 4 e d) 8.

A figura 3.15 mostra o resultado da aplicação da métrica $\mathcal{R}(m', m'', \mathcal{N})$ a m' =M2 (com $\mathcal{H}_{min}(n) \in \{1, 2, 4, 8\}$) e m''=HC (com $k \in \{1, 2, 4, 8\}$), para $1 \leq \mathcal{N} \leq 1024$. Os resultados foram divididos em quatro gráficos; cada gráfico diz respeito aos resultados da comparação de M2 com um certo valor de $\mathcal{H}_{min}(n)$, relativamente ao HC com todos os valores de k.

Os gráficos revelam que, salvo algumas excepções, a métrica $\mathcal{R}(m', m'', \mathcal{N})$ mantém-se abaixo da unidade, sinal de que M2 oferece, em geral, o melhor compromisso entre *número de entradas por nó* e *qualidade da distribuição*. Adicionalmente, à medida que $\mathcal{H}_{min}(n)$ cresce, o valor médio acumulado (não apresentado) da métrica $\mathcal{R}(m', m'', \mathcal{N})$ permanece mais ou menos constante, dado que o aumento progressivo do *número de entradas por nó* (exibido na figura 3.14) é anulado pela melhoria da *qualidade da distribuição* (observável na figura 3.3). Este resultado significa que, numa óptica de minimização do *número de entradas por nó* e maximização simultânea da *qualidade da distribuição*, é suficiente definir $\mathcal{H}_{min}(n) = 1$ para o modelo M2. Análogamente, para o modelo M4, os resultados da figura 3.13 sugerem que $\mathcal{H}_{min}(v) = 1$ é já suficiente para um compromisso da mesma categoria.

3.9 Relação com Outras Abordagens

Nas abordagens clássicas ao Hashing Dinâmico Distribuído (HDD), a expansão ou contracção do número de entradas da DHT é motivada pela necessidade de subdividir (operação *split*) ou fundir (operação *merge*) entradas. Nesse contexto, as entradas têm uma certa capacidade de armazenamento associada, designando-se de contentores; a subdivisão de um contentor implica a atribuição de parte do conteúdo a um novo contentor, podendo ser necessário integrar um novo nó na DHT, para gerir o novo contentor; a fusão de dois contentores permite dispensar um nó da guarda de um deles, podendo esse nó sair da DHT se não for responsável por mais contentores. Assim, é a gestão da capacidade dos contentores que está na base da evolução dinâmica do número global de entradas da DHT e, indirectamente, do seu número de nós: em função da taxa de ocupação dos contentores da DHT, podem ser necessários novos nós, ou podem ser dispensados nós antigos da DHT.

Os modelos M2 e M4 descritos neste capítulo configuram também a utilização de uma estratégia de Hashing Dinâmico Distribuído. Todavia, não assumem causas específicas para a adição/remoção de nós à/da DHT (ou nós virtuais, no caso de M4). Assim, esses eventos podem ocorrer i) pelas mesmas razões que as subjacentes às abordagens clássicas de HDD (essencialmente ligadas à gestão de recursos de armazenamento), ou por outros motivos, tais como ii) razões de carácter administrativo (*e.g.*, necessidade de retirar um nó do *cluster*, da DHT, para associá-lo a outra tarefa), iii) gestão do paralelismo potencial no acesso à DHT (tanto maior quanto maior for o número de nós da DHT), etc. É então em função do número de nós[22] que se define o número global de entradas da DHT, sendo a principal preocupação dos nossos modelos a manutenção do equilíbrio das distribuições.

Outro aspecto importante que distingue os nossos modelos de distribuição de DHTs, das abordagens clássicas de HDD, é a forma como se processa a evolução estrutural da DHT. Assim, a estratégia prosseguida pelos modelos M2 e M4 implica que, em certos estágios da evolução da DHT, *todas* as entradas tenham que ser subdivididas/fundidas, a fim de manter a qualidade da distribuição dentro de certos limites[23]. Ora, se as subdivisões/fusões forem realizadas no quadro de uma única operação colectiva, por todos os nós da DHT, tal exigirá sincronização global o que pode constituir um entrave à escalabilidade da DHT[24].

Nas estratégias clássicas de HDD, cada entrada pode-se dividir/fundir de forma independente. Porém, tal não garante, por si só, a escalabilidade da abordagem. Por exemplo, numa primeira variante do LH* (secção 2.4.2), um *split coordinator* coordena o processo de divisão/fusão e, numa segunda versão, o preço pago pela sua ausência é uma maior variância na carga dos contentores (e logo dos seus nós, algo que pretendemos evitar com os nossos modelos). Outro exemplo ainda é o fornecido pela abordagem de Gribble [GBHC00] (secção 2.4.5) que, apesar de permitir a evolução independente (mas despoletada administrativamente) das entradas da DHT segundo uma configuração em *trie* (como no EH* ou no DDH), peca pela manutenção de informação total de localização das entradas, que tem

[22] E, eventualmente, de invariantes complementares, específicos de cada modelo.

[23] Dados, indirectamente, pelos parâmetros $\mathcal{H}_{min}(n)$ e $\mathcal{H}_{min}(v)$, dos modelos M2 e M4, respectivamente.

[24] Teoricamente, no quadro dos nossos modelos M2 ou M4, um nó poderia adiar a subdivisão efectiva das suas entradas, até que fosse necessário cedê-las a outro(s) nó(s); todavia, a co-existência de dois ou mais níveis de subdivisão, teria de ser suportada convenientemente pelos mecanismos do capítulo 4.

de ser replicada pelos vários nós sempre que há alterações; ora, como veremos no próximo capítulo, é possível acoplar aos nossos modelos certos esquemas de localização distribuída, que permitem a cada nó da DHT (uma vez instruídos em colectivo para tal) a actualização autónoma da sua informação (parcial) de localização, sem troca adicional de mensagens.

Acresce ainda que, embora o requisito da subdivisão/fusão global possa, como já referimos, prejudicar a escalabilidade dos nossos modelos, exige, por outro lado, menos informação de estado: é apenas necessário manter um nível de subdivisão (*splitlevel*) global, em vez de um nível de subdivisão por cada entrada da DHT. Além disso, à medida que o número de entradas global da DHT cresce, são cada vez menos frequentes os eventos globais de subdivisão, dado que no próximo evento terão de intervir o dobro dos nós (ou nós virtuais).

Abordagens de Hashing Consistente Pesado, em que o número de nós virtuais por nó computacional se relaciona com certas qualidades estáticas/dinâmicas dos nós, foram previamente referenciadas na secção 2.6.2.2 [DKKM01]. No nosso caso, a semântica do conceito de nó virtual é mais rica, como deixam entender as várias possibilidades enunciadas na secção 3.5.2.1, e aprofundadas na secção 5.5. Adicionalmente, a sobrecarga espacial (em informação de posicionamento) e temporal (em esforço de localização) acrescida, derivada dos múltiplos nós virtuais por nó computacional, podendo ser elevada numa DHT para ambiente P2P (rever secção 2.6.2.2), será diminuta numa DHT realizada num *cluster*. Por exemplo, no contexto dos nossos modelos M3 ou M4, uma *tabela de distribuição* para 1024 nós consumirá apenas $1024 * (4 + 4 + 4) = 12$ Kbytes de espaço (em máquinas de 32 bits).

A filosofia base do Procedimento de (Re)Distribuição do nosso modelo M4' é semelhante à da *estratégia de assimilação* para discos heterogéneos da abordagem de Brinkman [BSS00], proposta no quadro do Particionamento para SANs (rever secção 2.6.3). De facto procura-se, em ambos os casos, transformar o problema da obtenção de uma distribuição heterogénea de boa qualidade, no da obtenção de uma distribuição homogénea de boa qualidade. Esse mesmo estratagema é também aplicado por Brinkman [BSS02] à distribuição de objectos num sistema distribuído, com base no modelo *balls-into-bins* (rever secção 2.6.4), sendo referidas vantagens da replicação de uma estrutura *capacity distribution*, equivalente às nossas *tabelas de distribuição* (cuja possível replicação tem vantagens interessantes, *e.g.*, agilizando a transferência de entradas entre nós da DHT, como referido na secção 3.3.4.2). Precisamente, é do contexto das estratégias de particionamento assentes no modelo *balls-into-bins*, concretamente da abordagem de Czumaj [CRS03], que importamos a definição de distribuição (homogénea) *perfeita* (ver secção 2.6.4.1), aplicável aos modelos M1 e M2.

3.10 Epílogo

A questão da definição da *identidade* das entradas associadas a cada nó aborda-se no próximo capítulo, no âmbito da definição de estratégias de posicionamento e localização. O modelo M4 de Distribuição Heterogénea com Hashing Dinâmico é o modelo de distribuição adoptado pela arquitectura Domus, descrita a partir do capítulo 5; no contexto dessa arquitectura, a semântica do nó virtual é enriquecida, pelo desacoplamento das funções de endereçamento e de armazenamento da DHT. Posteriormente, no capítulo 6, procede-se à definição rigorosa dos mecanismos que ditam i) a definição inicial do número de nós virtuais e que ii) regulam a evolução dinâmica desse número (questões, por ora, ignoradas).

Capítulo 4

Posicionamento e Localização

Resumo

Neste capítulo descrevem-se mecanismos de posicionamento e localização compatíveis com os mecanismos de distribuição do capítulo anterior. Os mecanismos de localização (distribuída) exploram grafos DeBruijn binários e grafos Chord completos, com algoritmos de *encaminhamento acelerado*; estes, tirando partido de várias fontes de informação topológica disponíveis em cada nó de uma DHT, diminuem o esforço de localização (número de saltos na topologia) face ao uso de *encaminhamento convencional*. Descreve-se também a interacção entre os mecanismos de posicionamento e localização na evolução das DHTs.

4.1 Prólogo

Os modelos de distribuição do capítulo 3 concentram-se na definição do *número* de entradas de cada nó de uma DHT, de forma a assegurar uma distribuição que respeite, o mais possível, a quota desejada (ideal) de cada nó. Este capítulo incide sobre questões ligadas à gestão da *identidade* das entradas atribuídas aos nós, incluindo: 1) a definição das correspondências "entrada $\mapsto$ nó" na criação de uma DHT (*posicionamento inicial*), 2) a reconstituição dessas correspondências durante a operação da DHT (*localização distribuída*) e 3) o suporte à eventual redefinição de correspondências (*re-posicionamento*).

4.2 Conceitos Básicos

4.2.1 Identidade de uma Entrada

Seja $f : K \mapsto H$ uma *função de hash* de $\mathcal{L}$ bits e $H = \{0, 1, ..., 2^{\mathcal{L}}-1\}$ o conjunto dos *hashes*; a *tabela de hash* associada tem $\mathcal{H} = \#H = 2^{\mathcal{L}}$ entradas, de índices $h = 0, 1, ..., 2^{\mathcal{L}} - 1$; a *identidade* de cada entrada da tabela de *hash* é dada pelo índice correspondente à entrada; por sua vez, o índice corresponde a um *hash*; neste contexto, os conceitos de *entrada*, *índice* ou *hash* confundem-se, sendo usados, ao longo da dissertação, com o mesmo significado.

4.2.2 Posicionamento e Localização de Entradas

Seja N o conjunto dos nós que suportam uma DHT. Então, denotamos por $n(h)$ o nó responsável pela entrada h, com $n(h) \in N$ e $h \in H$. Recordando o definido na secção 2.4.1.4: o *posicionamento* de uma entrada h corresponde à definição de $n(h)$, ou seja, ao estabelecimento de uma correspondência $h \mapsto n(h)$; a *localização* de uma entrada h corresponde à descoberta da correspondência $h \mapsto n(h)$; o colectivo de todas as correspondências desse tipo define a *informação de posicionamento / informação de localização* da DHT.

4.2.3 Propriedades Relevantes dos Grafos de Localização

Na sua essência, a localização distribuída em DHTs assenta na utilização de grafos direccionados (*digrafos*). Em tal contexto, as características mais relevantes dos grafos são:

1. *diâmetro* ($d_{\max}$): distância máxima entre qualquer par de vértices[1], a minimizar;
2. *distância média* ($\overline{d}$): distância média entre qualquer par de vértices[2]; note-se que, na perspectiva dos clientes de uma DHT, a minimização da distância média no grafo subjacente será mais importante do que a minimização da distância máxima[3];
3. *grau* ($\mathcal{K}$): em grafos $\mathcal{K}$-*regulares*, todos os vértices têm $\mathcal{K}$ vizinhos; $\mathcal{K}$ influencia directamente a dimensão da *tabela de encaminhamento* (ver secção 4.4.4), a minimizar;
4. *algoritmo base de navegação*: garante o percurso mais curto entre dois vértices; em cada vértice do percurso, selecciona o sucessor apropriado, de entre $\mathcal{K}$ possíveis.

O apêndice C complementa esta caracterização, fornecendo conceitos básicos de Teoria de Grafos. Adicionalmente, utilizaremos o termo "grafo" de forma equivalente a "digrafo".

4.3 Filosofia do (Re-)Posicionamento

4.3.1 Posicionamento Inicial de Entradas

Na criação de uma DHT, a aplicação dos modelos de distribuição do capítulo anterior resulta na atribuição de um certo *número* de entradas, $\mathcal{H}(n)$, a cada nó n inicial da DHT. Esse número é registado numa *tabela de distribuição* (TD), de esquema $< n, \mathcal{H}(n) >$ ou $< n, \mathcal{H}(n), \mathcal{V}(n) >$, c.f. esteja em causa uma distribuição homogénea (modelos M1 e M2) ou heterogénea (modelos M3, M4 e M4'). Segue-se que, com base na tabela TD, ordenada segundo um determinado critério pré-definido (*e.g.*, ordem lexicográfica dos identificadores

[1] A distância $d(x, y)$ entre x e y é o número de arestas ou saltos, do caminho mais curto entre x e y.

[2] Dada pela fórmula C.3 do apêndice C.

[3] De facto, é até possível reduzir o diâmetro de um grafo e, ao mesmo tempo, aumentar a distância média [Xu03], compromisso pouco interessante para a qualidade da experiência do acesso à DHT.

n dos nós da DHT), a definição do posicionamento inicial pode ser feita de forma trivial, assente numa atribuição i) *contígua*, ii) *rotativa* (*Round Robin*), iii) *pseudo-aleatória*, etc. Por exemplo, tendo-se inicialmente $\mathcal{H} = 8$, $H = \{0, 1, ..., 7\}$, $N = \{n_0, n_1, n_2\}$ e a TD ordenada dada por $\{(n_0, 3), (n_2, 2), (n_1, 3)\}$ então i) de um posicionamento inicial *contíguo* resultaria $H(n_0) = \{0, 1, 2\}$, $H(n_2) = \{3, 4\}$ e $H(n_1) = \{5, 6, 7\}$ e ii) de um posicionamento inicial *rotativo* resultaria $H(n_0) = \{0, 3, 6\}$, $H(n_2) = \{1, 4\}$ e $H(n_1) = \{5, 7\}$.

Face a um posicionamento inicial *contíguo*, um posicionamento inicial *rotativo* tem a vantagem de assegurar uma maior dispersão (a maior possível) de H por N; dessa forma, uma eventual distribuição não-uniforme de registos da DHT terá maior probabilidade de afectar uniformemente os nós da DHT[4]. Por outro lado, um posicionamento *contíguo* permite construir grafos de localização com menos vértices[5], mas a verdade é que a contiguidade desse posicionamento não é garantida durante a vida das nossas DHTs (ver secção 4.3.2).

O facto do posicionamento inicial ser determinístico permite que, na posse da TD inicial[6], os nós da DHT deduzam e implementem, de forma autónoma, a distribuição inicial da DHT. Cumulativamente, cada um desses nós poderá construir, autonomamente, as *tabelas de encaminhamento* que representam a sua participação num grafo de localização distribuída. Note-se porém que, se a distribuição inicial da DHT for definitiva (ou seja, se a DHT for *estática*, com correspondências "entrada $\mapsto$ nó" fixas), a TD é suficiente para deduzir a localização de qualquer entrada da DHT, permitindo acesso *directo* (1-HOP) à DHT. Nesse caso, mecanismos de localização distribuída seriam, obviamente, dispensáveis.

4.3.2 Re-Posicionamento de Entradas

O re-posicionamento de uma ou mais entradas (ou seja, a sua atribuição a um outro nó, traduzida na redefinição da correspondência "entrada $\mapsto$ nó" respectiva) é uma consequência da modificação do número de entradas de um ou mais nós; por seu turno, essa modificação pode resultar da adição/remoção de nós computacionais à/da DHT ou, com distribuições heterogéneas, da alteração do número de nós virtuais dos nós computacionais actuais.

Em qualquer caso, os modelos do capítulo anterior determinam, com base nos Procedimentos de (Re)Distribuição, os protagonistas (nós computacionais ou virtuais) das transferências, assim como o *número* de entradas a transferir entre eles. Quanto à selecção das entradas concretas a transferir (ou seja, quanto à definição da sua *identidade*), os modelos são omissos, mas a verdade é que é suficiente uma escolha *aleatória* das entradas a transferir, de entre as entradas disponíveis na(s) fonte(s) das transferências. De facto, mesmo que a escolha determinasse a cedência apenas de entradas *contíguas* na(s) fonte(s), isso não seria suficiente para garantir a continuidade de todas as entradas do(s) destino(s), dado que estes, ao longo da vida da DHT, acabarão por receber entradas de fontes diferentes.

Desta forma, mesmo que se optasse por um posicionamento inicial *contíguo*, bastariam alguns eventos de re-distribuição e consequente re-posicionamento, para quebrar a conti-

[4]Este mesmo argumento é usado pelo Hashing Consistente / Chord para atribuir várias partições (que, embora contínuas, são pequenas e dispersas), em vez de uma só, a cada nó da DHT – rever secção 2.6.2.1.

[5]Em que os vértices seriam nós da DHT e não os seus *hashes* – ver secção 4.4.2

[6]Embora o conhecimento das versões posteriores possa também ser vantajoso – rever secção 3.3.4.2.

guidade das *partições* de H detidas pelos nós da DHT e fazer com que a sua composição pareça aleatória. Esta característica separa claramente os nossos modelos de particionamento dos baseados em Hashing Consistente (que garante partições *contíguas*) e influencia decisivamente a forma como se constrói o grafo de localização distribuída (ver secção 4.4.2).

Note-se ainda que, à semelhança de um posicionamento inicial *rotativo*, um posicionamento que tende a ser *aleatório* continua a ser adequado a distribuições não-uniformes de registos.

4.4 Especificidades da Localização Distribuída

4.4.1 Adequabilidade a Ambientes *Cluster*

Em ambiente *cluster*, o número de nós disponíveis para instanciar uma DHT é inferior, em várias ordens de grandeza, ao número de nós que, tipicamente, participam num sistema P2P (dezenas/centenas contra milhares/milhões). Desde logo, essa diferença de escala sugere que a *informação de posicionamento* de uma DHT instanciada em *cluster* deverá ser suficientemente pequena (da ordem dos Kbytes ou Mbytes) para que a sua totalidade seja comportável por qualquer nó. Este argumento favorece a utilização de localização baseada num registo centralizado da *informação de localização* (ou até mesmo com replicação total – rever secção 2.4.1.5). Outro argumento que vem ao encontro do anterior é o facto de, em ambiente *cluster*, o conjunto de nós que suportam uma DHT ter composição mais estável.

Todavia, se admitirmos 1) a possibilidade de se instanciarem múltiplas DHTs no *cluster*, em simultâneo e 2) de estas estarem sujeitas a um mecanismo de balanceamento dinâmico[7], então a aplicação de mecanismos de localização distribuída faz sentido. No primeiro caso, interessa balancear a carga de armazenamento da maior quantidade de *informação de localização*, das várias DHTs. No segundo caso, a redistribuição de uma ou mais DHTs[8] acarreta a necessidade de actualizar a sua *informação de localização*; essa actualização pode ser mais ou menos abrangente, dependendo da percentagem redistribuída; com a *informação de localização* distribuída, o esforço de actualização é distribuído por vários nós, e o número de nós envolvidos na actualização será função da dimensão da redistribuição.

4.4.2 Necessidade de Grafos Completos em H

Quando o particionamento de H origina partições *contíguas* (intervalos), é suficiente construir um grafo G^N (ou G^V) para localização distribuída, tomando como vértices os nós computacionais (ou virtuais) da DHT. Por exemplo, no Chord [SMK+01], cada nó tem correspondência com um *hash* em $H = \{0, 1, ..., 2^{\mathcal{L}} - 1)$ (com função de *hash* de $\mathcal{L}$ bits), pelo que apenas um certo número de *hashes*, inferior a $\mathcal{H} = 2^{\mathcal{L}}$, é usado para construir o grafo; este grafo é pois "*esparso* no domínio dos *hashes*" e "*completo* no domínio dos nós".

Ora, como referido na secção 4.3, a aplicação dos nossos modelos de distribuição e po-

[7] Pressupostos que, entre outros, estão na base da arquitectura Domus, discutida no capítulo 5.

[8] A redistribuição traduz-se na transferência de entradas entre um ou mais nós, podendo envolver apenas os nós actuais da DHT, ou nós que abandonam/ingressam a/na DHT.

sicionamento resulta em partições *descontíguas* (com *hashes* dispersos). Essa particularidade impõe a construção de grafos G^H, completos no domínio dos *hashes*; ou seja, sendo $H = \{0, 1, ..., 2^{\mathcal{L}}-1\}$, um grafo G^H terá $\mathcal{H} = 2^{\mathcal{L}}$ vértices, correspondentes a todos os *hashes* de H. O grafo G^H será particionado, de forma implícita, através dos nós da DHT: cada nó será responsável pelos vértices correspondentes aos *hashes* da(s) sua(s) partição(ões).

A necessidade de um grafo G^H em vez de G^N (ou G^V) parece ser uma menos valia importante dos nossos modelos, dado que G^H assenta num maior número (o máximo possível) de vértices. Veremos, no entanto, que é possível desenvolver algoritmos de *encaminhamento acelerado*, que permitem que a localização distribuída em G^H tenha um custo semelhante (e até inferior) ao custo em G^N. Além disso, comparando o custo da localização em G^H com o custo num grafo G^V (em que os vértices são nós virtuais, como acontece no Chord[9]), constata-se que as diferenças no esforço de localização podem ser marginais (ver a seguir).

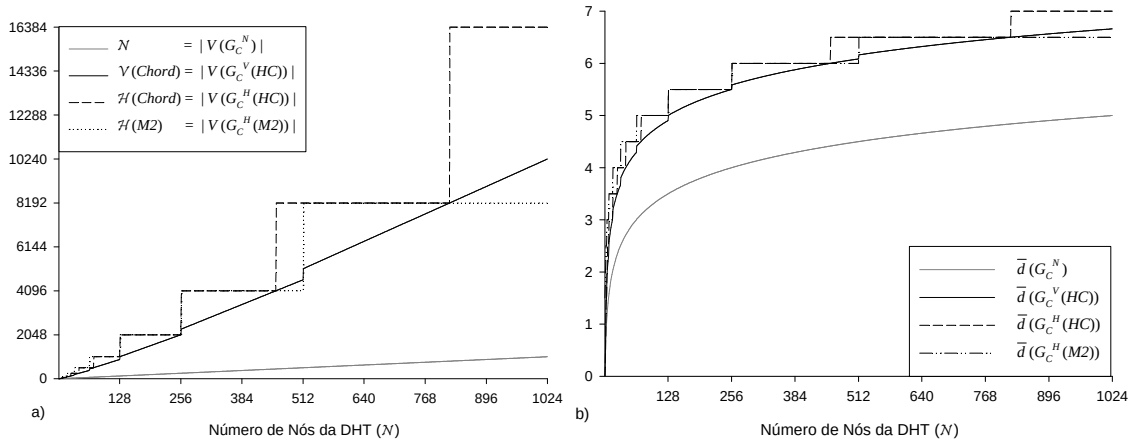


Figura 4.1: a) Número de Vértices e b) Distância Média, p/ várias classes de grafos Chord.

A figura 4.1.a) representa o número de vértices de grafos Chord de diferentes classes, em função do número de nós $\mathcal{N}$ de uma DHT: 1) para a classe G^N, os vértices correspondem a nós computacionais; 2) para as classes $G^V(HC)$ e $G^H(HC)$, os vértices correspondem aos nós virtuais e às entradas, respectivamente, de uma DHT em que o total de nós virtuais, $\mathcal{V}(HC)$, e de entradas, $\mathcal{H}(HC)$, é gerado de acordo com o procedimento descrito na avaliação do Hashing Consistente (HC) da secção 3.8 (com parâmetro $k = 1$); 3) para a classe $G^H(M2)$, os vértices correspondem às entradas de uma DHT em que $\mathcal{H}$ é gerado pela fórmula 3.11 do modelo M2 de distribuição homogénea (com parâmetro $\mathcal{H}_{min}(n) = 8$[10]).

A relativa proximidade do número de vértices dos grafos $G^V(HC)$ e $G^H(HC)$, deixa adivinhar que a navegação nesses grafos terá um custo semelhante. Esse custo é dado

[9]Sendo baseado em Hashing Consistente, cada nó virtual é associado a uma partição contínua. A comparação com o Chord é importante, pois é graças à utilização de múltiplos nós virtuais por cada nó computacional que o Chord assegura distribuições balanceadas, sejam homogéneas, sejam heterogéneas.

[10]A escolha de $k = 1$ e $\mathcal{H}_{min}(n) = 8$ assegura que o valor de $\mathcal{H}$ é semelhante sob HC e M2, o que acaba por favorecer o HC na comparação; de facto, valores $\mathcal{H}_{min}(n)$ inferiores são suficientes para garantir a M2 uma *qualidade da distribuição* superior à do HC (como se demonstrou na avaliação da secção 3.8); como valores inferiores de $\mathcal{H}_{min}(n)$ se traduzem em valores inferiores de $\mathcal{H}(M2)$ então, sob o modelo M2, o grafo G^H correspondente poderia ter, se pretendido, menos vértices que os representados na figura 4.1.a).

pela *distância média*[11], $\overline{d}$, representada na figura 4.1.b). A evolução logarítmica de $\overline{d}$ torna menos vincadas as diferenças entre os grafos, em especial entre $G^V(HC)$ e $G^H(HC)$.

4.4.3 Conceito de Encaminhamento Acelerado

Realizada segundo o *algoritmo base de navegação* do grafo G^H, a localização distribuída (ou, equivalentemente, o mecanismo de encaminhamento que a suporta) é *convencional*, no sentido de que progride entrada-a-entrada e pode resultar na visita repetida de um mesmo nó da DHT. Alternativamente, a análise conjunta das várias tabelas de encaminhamento detidas por cada nó da DHT suporta um *encaminhamento acelerado*; neste, tenta-se progredir nó-a-nó e evitar visitas repetidas do mesmo nó, ao mesmo tempo que se procuram encontrar e explorar eventuais *atalhos topológicos*; o saldo final será uma redução do esforço de localização em G^H, face ao obtido com *encaminhamento convencional*.

O *encaminhamento acelerado* procura assim compensar o facto de, na sequência dos nossos modelos de particionamento (que originam partições *descontínuas* de H), a localização distribuída assentar num grafo G^H, em vez de num grafo G^N. Em termos formais, o objectivo é o de aproximar a distância média em G^H, à distância média em G^N. Precisamente, a figura 4.1.b) contextualiza este raciocínio, para a utilização de grafos Chord.

O *encaminhamento acelerado* deverá pois reduzir a distância média $\overline{d}(G^H)$ pelo menos até ao valor $\overline{d}(G^N)$ e, se possível, abaixo dele, fazendo uma utilização inteligente do conjunto de *informação de localização* dispersa pela várias tabelas de encaminhamento de cada nó.

4.4.4 Tabelas de Encaminhamento

A navegabilidade no grafo G^H (da qual depende, em última instância, o mecanismo de localização distribuída) requer a manutenção, por cada *hash* h de H, de uma *tabela de encaminhamento*, $TE(h)$. Genericamente, se qualquer *hash* $h \in H$ tiver $\mathcal{K}$ sucessores no grafo, então $TE(h)$ guardará 1) a identidade dos $\mathcal{K}$ sucessores ($suc(h,k)$, com $k = 0, 1, ..., \mathcal{K}-1$) e 2) para cada sucessor, a identificação[12] do seu nó hospedeiro ($n(suc(h,k))$). Um nó n responsável por uma partição $H(n)$, alojará $\mathcal{H}(n) = \#H(n)$ tabelas de encaminhamento.

4.4.5 Árvores de Encaminhamento

Para realizar *encaminhamento acelerado* de forma eficiente, todas as tabelas de encaminhamento de cada nó de uma DHT são concentradas numa estrutura de dados local. Genericamente, designamos essa estrutura por *árvore de encaminhamento*. Tendo em conta o seu propósito, uma árvore de encaminhamento deve satisfazer um conjunto de propriedades relevantes: 1) suportar a adição/remoção eficiente de tabelas de encaminhamento, 2) assegurar acesso eficiente a tabelas individuais (no máximo, acesso de ordem logarítmica, face ao número total de tabelas), 3) permitir a travessia eficiente da totalidade das tabelas.

[11] Para um grafo Chord genérico, G, com um certo número $|V(G)|$ de vértices, a distância máxima é $d_{\max}(G) \approx log_2|V(G)|$ e a distância média é $\overline{d}(G) \approx d_{\max}(G)/2$ [LKRG03] (ver também secção 4.6.1).

[12] O tipo de identificação dependerá do mecanismo escolhido para a Troca de Mensagens entre nós.

A escolha do tipo de árvore poderá variar em função do tipo de grafo e das necessidades específicas dos respectivos algoritmos de encaminhamento. Posteriormente, veremos que *tries* compactas [Mor68] são apropriadas para encaminhamento acelerado com grafos DeBruijn, ao passo que AVLs [AVL62] ou Red-Black Trees [GS78] o são para grafos Chord.

4.4.6 Grafos para Localização Distribuída

Tendo em vista a utilização de mecanismos de localização distribuída, compatíveis com os nossos modelos de particionamento, o estudo de várias abordagens[13] determinou a adopção de grafos DeBruijn[14] e grafos Chord, por se encaixarem bem nas especificidades daqueles modelos, designadamente na necessidade de grafos completos em H; tais grafos gozam, todavia, de qualidades complementares, a levar em conta no nosso cenário de aplicação, em que H varia: com grafos Chord, a distância média é menor que com grafos DeBruijn; por outro lado, estes requerem tabelas de encaminhamento de dimensão inferior e constante.

4.5 Localização Distribuída com Grafos DeBruijn Binários

Os grafos DeBruijn [dB46] são dos melhores exemplos conhecidos de grafos que minimizam o *diâmetro*, para um número de vértices e *grau* fixos. Essa propriedade torna-os especialmente atractivos no desenho de Redes de Computadores ou Multicomputadores onde, ao mesmo tempo que se pretende aumentar o número de nós interligados, é desejável conter o grau de cada nó e o diâmetro da rede [II81, F.T91]. Transpostas para a localização distribuída em DHTs, essas qualidades revelam também aí as suas vantagens [LKRG03].

4.5.1 Grafos DeBruijn para um Alfabeto Genérico

Seja A um alfabeto com $\#A$ símbolos. As sequências de $\mathcal{L} = log_{\#A}|V(G_B)|$ símbolos de A definem os $V(G_B)$ vértices de um grafo DeBruijn, G_B, de diâmetro $\mathcal{L}$ e grau $\#A$. Os arcos de G_B representam a transformação da sequência de símbolos de um vértice, na sequência de outro, através de operações de *deslocamento* (*shift*). Genericamente, dados os símbolos s e p, e a sequência de símbolos $x_1x_2 \ldots x_{\mathcal{L}}$, todos do mesmo alfabeto, então:

- s é um *sufixo* unitário[15] do *deslocamento à esquerda* $x_1x_2 \ldots x_{\mathcal{L}} \rightarrow x_2 \ldots x_{\mathcal{L}}s$;
- p é um *prefixo* unitário do *deslocamento à direita* $x_1x_2 \ldots x_{\mathcal{L}} \rightarrow px_1 \ldots x_{\mathcal{L}-1}$.

Os *sucessores* de um vértice obtêm-se através de um *deslocamento à esquerda*; reciprocamente, os *predecessores* são obtidos com um *deslocamento à direita*. Dado um vértice $v = v_1v_2 \ldots v_{\mathcal{L}}$, o seu sucessor de sufixo s, denotado por $suc(v, s)$, é dado pela sequência

$$suc(v, s) = v_2v_3 \ldots v_{\mathcal{L}-1}s \tag{4.1}$$

[13] As mais representativas, na altura em que ocorreu a investigação ligada a este capítulo da dissertação.
[14] Se com estes se assumir a utilização de um alfabeto binário, como foi nossa opção – ver secção 4.5.2.
[15] Ou seja, com apenas um símbolo do alfabeto A.

Neste contexto, o conjunto dos sucessores de v, denotado por $Suc(v)$, pode-se definir como

$$Suc(v) = \{suc(v, s) : s \in A\} \tag{4.2}$$

Qualquer sucessor pode ainda ser obtido recorrendo à seguinte expressão, na base 10:

$$suc(v, s) = (\#A \times v + s) \ mod \ |V(G_B)| \ com \ s = 0, 1, ..., \#A - 1 \tag{4.3}$$

A definição de *predecessor* – $pred(v, p)$ – e de *conjunto de predecessores* – $Pred(v)$ – é dual.

A navegação entre vértices segue um algoritmo simples, que aplica um máximo de $\mathcal{L}$ operações de deslocamento de símbolos, sempre na mesma direcção. Genericamente, o caminho mais curto entre os vértices $x = x_1x_2 \ldots x_{\mathcal{L}}$ e $y = y_1y_2 \ldots y_{\mathcal{L}}$, construído à base de *sucessores*, corresponde à transformação progressiva de x em y, através da injecção em x de um máximo de $\mathcal{L}$ sufixos unitários (um por cada aresta/salto do trajecto no grafo):

$$x_1 \ldots x_{\mathcal{L}} \rightarrow x_2 \ldots x_{\mathcal{L}}y_1 \rightarrow x_3 \ldots x_{\mathcal{L}}y_1y_2 \rightarrow \ldots \rightarrow y_1 \ldots y_{\mathcal{L}}$$

O método de definição de sucessores (e, dualmente, de predecessores) qualifica um grafo G_B como *conexo* e $\mathcal{K}-regular$, com $\mathcal{K} = \#A$ (*i.e.*, *grau de partida=grau de chegada=*$\#A$, para todos os vértices); todavia, o grafo G_B não é *simples*, uma vez que nele ocorrem *laços*.

4.5.1.1 Distâncias entre Vértices

Segue-se que a *distância* $d(x, y)$ entre dois vértices x e y corresponde "ao número de deslocamentos à esquerda (unitários) que é preciso aplicar a x para o transformar em y". Como o número máximo desses deslocamentos é $\mathcal{L}$, o *diâmetro* de um grafo G_B é $\mathcal{L}$:

$$d_{\max}(G_B) = \mathcal{L} \tag{4.4}$$

Para a distância média, não existe uma fórmula genérica, mas conhecem-se limites [BLS93]:

$$\overline{d}_{\min}(G_B) \leq \overline{d}(G_B) \leq \overline{d}_{\max}(G_B) \tag{4.5}$$

com

$$\overline{d}_{\min}(G_B) = \mathcal{L} - \frac{\#A}{(\#A-1)^2} + \frac{\#A}{\#A-1} \times \frac{\mathcal{L}}{|V(G_B)|-1} \tag{4.6}$$

e

$$\overline{d}_{\max}(G_B) = \mathcal{L} - \frac{1}{\#A-1} + \frac{\mathcal{L}}{|V(G_B)|-1} \tag{4.7}$$

4.5.2 Grafos DeBruijn para o Alfabeto Binário

A secção anterior sintetiza o essencial de grafos DeBruijn para qualquer alfabeto A de $\#A$ símbolos. Todavia, estamos apenas interessados em grafos DeBruijn "binários", baseados no alfabeto $A = \{0, 1\}$, nos quais os vértices correspondem a todas as sequências de um certo número de bits, ou seja, "grafos DeBruijn completos em H" . Nesses grafos, que denotamos por $G_B^H(\mathcal{L})$, existem $2^{\mathcal{L}}$ vértices/*hashes*, o diâmetro é $\mathcal{L}$ e o grau é $\mathcal{K} = \#A{=}2$.

Especializando a expressão 4.3, para os sucessores de um vértice/*hash*, na base 10, resulta:

$$suc(h, s) = (2 \times h + s) \ mod \ 2^{\mathcal{L}} \text{ com } s = 0, 1 \tag{4.8}$$

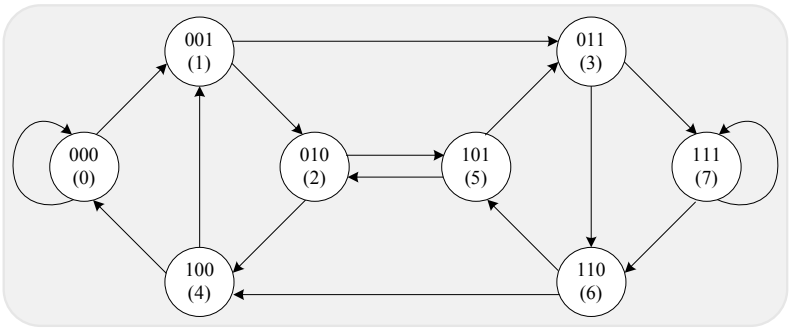


Figura 4.2: Representação do grafo $G_B^H(3)$.

A figura 4.2 representa $G_B^H(\mathcal{L} = 3)$, que tem $2^3 = 8$ vértices/*hashes* e diâmetro 3; os vértices representam-se por círculos e os arcos por setas; em cada vértice é inscrito o respectivo *hash* (entre parêntesis é também apresentado o valor respectivo na base 10).

4.5.3 Distâncias entre Vértices/Hashes

Informalmente, a distância $d(x, y)$ entre os vértices x e y é calculável assim: encontrar o maior valor $l \leq \mathcal{L}$, tal que os l bits menos significativos de x coincidem com os l bits mais significativos de y; $d(x, y)$ é o número de bits que não coincidem, dado por $\mathcal{L} - l$; $d(x, y)$ representa então o número de bits a injectar em x, pela direita, para o transformar em y.

Computacionalmente, $d(x, y)$ (com x e y na base 10), é calculável pelo algoritmo 4.2, em que **&** denota uma conjunção bit-a-bit, e $\gg$ denota um deslocamento à direita de um bit.

Por exemplo, para o grafo $G_B^H(3)$, tem-se $d(4, 6) = 3$ e $d(5, 6) = 2$, como pode ser comprovado pela observação da figura 4.2. As distâncias euclidianas correspondentes, dadas respectivamente por $6 - 4 = 2$ e $6 - 5 = 1$ são, neste caso (e geralmente) diferentes.

Sendo $\#A = 2$ e $|V(G_B^H(\mathcal{L}))| = 2^{\mathcal{L}}$, o minorante e o majorante para a distância média são:

$$\overline{d}_{\min}[G_B^H(\mathcal{L})] = \mathcal{L} - 2 + \frac{2 \times \mathcal{L}}{2^{\mathcal{L}} - 1} \tag{4.9}$$

Algoritmo 4.2: Cálculo da distância entre dois vértices/*hashes* num grafo $G_B^H(\mathcal{L})$.

Entrada: dois vértices x e y (na base 10)
Saída: distância d entre os vértices x e y

$d \leftarrow 0$
enquanto $d < \mathcal{L}$ **fazer**
    $x' \leftarrow x \ \ \& \ \ (2^{\mathcal{L}-d} - 1)$
    $y' \leftarrow y \gg d$
    se $x' = y'$ **então** break **fim se**
    $d \leftarrow d + 1$
fim enquanto
retornar d

e

$$\overline{d}_{\max}[G_B^H(\mathcal{L})] = \mathcal{L} - 1 + \frac{\mathcal{L}}{2^{\mathcal{L}} - 1} \tag{4.10}$$

Estas expressões representam especializações das expressões 4.6 e 4.7 para $A = \{0, 1\}$.

4.5.4 Trie de Encaminhamento

A nossa abordagem à localização distribuída com recurso a grafos $G_B^H(\mathcal{L})$ pressupõe que, em cada nó de uma DHT, todas as *tabelas de encaminhamento* locais sejam reunidas numa *trie*[16], que designamos por *trie de encaminhamento*: basicamente, para cada entrada local da DHT, identificada por uma certa sequência de bits, existirá uma folha nessa *trie*, com a tabela de encaminhamento da entrada; na *trie*, o percurso descendente, da raíz a uma folha, corresponde a percorrer a sequência de bits específica da folha o que, por opção[17], é feito dos bits menos significativos (à direita) para os mais significativos (à esquerda). Com *hashes* (entradas) de $\mathcal{L}$ bits, a profundidade máxima da *trie* de encaminhamento será $\mathcal{L}$.

Para minimizar o número de bifurcações e níveis, a *trie* de encaminhamento é *compacta* [Mor68]: apenas se criam bifurcações se for necessário distinguir duas sequências diferentes (enquanto que, numa *trie* normal, cada folha exigiria um número de bifurcações igual ao número de bits da folha). Uma *trie* compacta tem a vantagem de necessitar de menos recursos de armazenamento do que uma *trie* normal, o que é importante pelo facto de cada nó de uma DHT poder alojar várias tabelas de encaminhamento[18]; além disso, sendo o número de níveis tendencialmente menor, a navegação na *trie* tende a ser mais eficiente.

A figura 4.3 ilustra uma *trie* de encaminhamento compacta, que aloja três tabelas de encaminhamento, para uma DHT em que, correntemente, os *hashes* são de $\mathcal{L} = 5$ bits; o símbolo X ("*don't care*") significa que o valor do bit subjacente é, no contexto actual da

[16] Rever a secção B.4.1.1 para recuperar o conceito de *trie* e a sua primeira aplicação no contexto da tese.
[17] As implicações (vantagens) desta opção serão visíveis na discussão do algoritmo EA-$\mathcal{L}$ (secção 4.5.5.3).
[18] A importância dessa economia é ainda maior se o nó puder participar em múltiplas DHTs ...

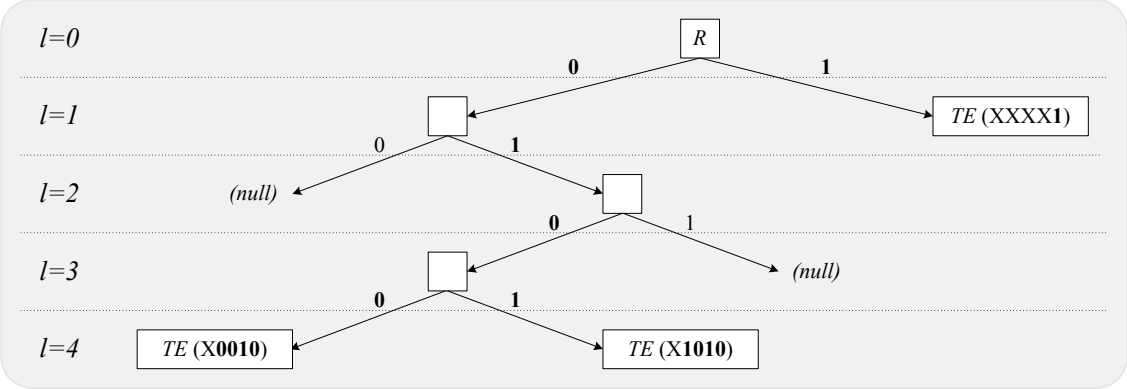


Figura 4.3: Uma *trie de encaminhamento*, para $\mathcal{L} = 5$.

folha, irrelevante; o último nível da *trie* é $l = 4$ (menor que $\mathcal{L} = 5$, graças à "compactação").

A opção por uma estrutura do tipo *trie*, como dicionário local das tabelas de encaminhamento de um nó, deve-se essencialmente ao facto de a mesma permitir realizar encaminhamento acelerado, na variante EA-$\mathcal{L}$ (ver secção 4.5.5.3), de forma bastante eficiente.

4.5.5 Algoritmos de Encaminhamento

Apresentam-se agora vários *algoritmos de encaminhamento* que tiram partido da *trie de encaminhamento*. Neste contexto, denotamos por t (*target*) a entrada (de uma certa DHT) cuja localização se pretende determinar, por c (*current*) a entrada escolhida como destino do anterior salto de encaminhamento e por n (*next*) a entrada escolhida como destino do próximo salto de encaminhamento. Os algoritmos são descritos em linguagem informal.

4.5.5.1 Encaminhamento Convencional (EC)

Algoritmo 4.3: Algoritmo EC (Encaminhamento Convencional).

1. procurar a entrada c na *trie* do nó actualmente visitado
2. calcular as sucessoras de c (fórmula 4.8) e a distância delas a t (algoritmo 4.2)
3. definir n como a sucessora de c que minimiza a distância a t
4. determinar o nó hospedeiro de n na tabela de encaminhamento de c
5. **se** $n = t$ **então** terminar a localização (o hospedeiro de n é o hospedeiro de t)
6. **senão** reencaminhar o pedido de localização de t para o hospedeiro de n **fim se**

O encaminhamento *convencional* (algoritmo 4.3), analisa apenas uma tabela de encami-

nhamento. Neste contexto, a *trie* de encaminhamento ainda é útil, porque concentra todas as tabelas de encaminhamento de um nó, permitindo o acesso expedito a cada uma delas.

4.5.5.2 Encaminhamento Melhorado (EM)

Algoritmo 4.4: Algoritmo EM (Encaminhamento Melhorado).

1. procurar a entrada t na *trie* do nó actualmente visitado
2. **se** t existir **então** terminar a localização (o nó actual é o hospedeiro de t)
3. **senão** executar o Algoritmo EC **fim se**

No encaminhamento *melhorado* (algoritmo 4.4), verifica-se primeiro se a entrada t já faz parte da *trie* local, sendo seguro que c (que corresponde ao n definido pela decisão de encaminhamento anterior) fará parte; se t não pertencer à *trie*, o próximo salto, n, ainda é calculado apenas com base na tabela de encaminhamento de c, através do algoritmo EC.

A pesquisa local de t tenta evitar que o nó hospedeiro de t seja visitado duas vezes: uma, pelo facto de alojar uma entrada intermédia da cadeia de encaminhamento convencional, e outra pelo facto de alojar a entrada final (t) dessa cadeia. O algoritmo EM oferece assim um encaminhamento *melhorado*, face ao encaminhamento *convencional*, mas que ainda não é *acelerado*, por basear as suas decisões de encaminhamento apenas na tabela de encaminhamento de c. Com encaminhamento acelerado, veremos que é possível assegurar que, qualquer nó da DHT (mesmo os que não alojam t) é visitado, no máximo, uma vez.

4.5.5.3 Encaminhamento Acelerado, variante $\mathcal{L}$ (EA-$\mathcal{L}$)

Com encaminhamento *acelerado*, a *trie* é pesquisada, em busca da folha b (*best*) que, de todas as folhas da *trie*, minimiza a distância a t. Na variante $\mathcal{L}$ (algoritmo 4.5), a procura realiza-se evitando a inspecção exaustiva de todas as folhas. Para tal, tira-se partido da organização estrutural escolhida para a *trie*, designadamente do facto do trajecto da raíz às folhas se realizar dos bits menos significativos das folhas, para os mais significativos.

Assim, o algoritmo EA-$\mathcal{L}$ desce a *trie* um máximo de $\mathcal{L}-1$ vezes; por cada descida, tenta encontrar uma folha b à distância m de t (o que corresponde a tentar encontrar uma folha b para a qual os $l = \mathcal{L} - m$ bits menos significativos coincidem com os l bits mais significativos de t[19]); havendo várias dessas folhas, é elegível uma qualquer. Na primeira descida, $m = 1$[20]; nas seguintes, m vai aumentando, uma unidade, até ao máximo de $\mathcal{L}-1$.

Ao contrários dos algoritmos anteriores, em EA-$\mathcal{L}$ não faz sentido falar de uma entrada actual (c) da cadeia de encaminhamento. De facto, a localização distribuída prossegue

[19] Esta lógica foi introduzida previamente na secção 4.5.3.

[20] E não $m = 0$, como expectável à primeira vista; de facto, o algoritmo consegue resolver uma localização sem ser necessário visitar o nó que aloja t, sendo esse o único nó onde faria sentido $m = 0$. Note-se que, pela mesma ordem de razões se explica o facto de se aceder à *trie*, no máximo, $\mathcal{L}-1$ vezes, e não $\mathcal{L}$ vezes.

Algoritmo 4.5: Algoritmo EA-$\mathcal{L}$ (Encaminhamento Acelerado, variante $\mathcal{L}$).

1. $m \leftarrow 1$
2. **enquanto** $m \leq \mathcal{L} - 1$ **fazer**
   - procurar uma entrada b na *trie* do nó actual, à distância m de t
   - **se** b existir **então** break **fim se**
   - $m \leftarrow m + 1$

   fim enquanto
3. calcular as sucessoras de b (fórmula 4.8) e a distância delas a t (algoritmo 4.2)
4. definir n como a sucessora de b que minimiza a distância a t
5. determinar o hospedeiro de n na tabela de encaminhamento de b
6. **se** $n = t$ **então** terminar a localização (o hospedeiro de n é o hospedeiro de t)
7. **senão** reencaminhar o pedido de localização de t para o hospedeiro de n **fim se**

agora de nó-em-nó: quando se reencaminha um pedido de localização para o nó hospedeiro de n, não é com a intenção de, nesse nó, consultar apenas a tabela de encaminhamento de n, mas sim de realizar encaminhamento acelerado com todas as tabelas residentes no nó.

4.5.5.4 Encaminhamento Acelerado, variante *all* (EA-*all*)

Algoritmo 4.6: Algoritmo EA-*all* (Encaminhamento Acelerado, variante *all*).

1. $m \leftarrow \mathcal{L}$
2. **para** cada folha f na *trie* do nó actual **fazer**
   - **se** $d(f,t) < m$ **então** $b \leftarrow f$ e $m \leftarrow d(f,t)$ **fim se**

   fim para
3. a 7.: igual aos passos 3. a 7. do Algoritmo 4.5

Na variante *all* do *encaminhamento acelerado* (algoritmo 4.6), todas as folhas da *trie* são analisadas, em busca da folha b que minimiza a distância a t (note-se que, havendo várias folhas à mesma distância mínima de t, é considerada a primeira encontrada). O papel que se atribui ao algoritmo EA-*all* é apenas o de evidenciar a competitividade do algoritmo EA-$\mathcal{L}$: como veremos, este consegue reduzir as cadeias de encaminhamento com a mesma

eficácia do algoritmo EA-*all*, mas sem a necessidade de uma pesquisa exaustiva da *trie*[21].

O desempenho do algoritmo EA-*all* pode variar consideravelmente em função de certas opções de realização. Por exemplo, uma sub-variante EA-*all-list*, que parte do princípio de que as folhas da *trie* são unidas numa lista ligada, permite uma pesquisa mais rápida, em comparação com uma sub-variante EA-*all-traverse*, que efectua uma travessia clássica (*depth-first*, por exemplo) da *trie*, dado que a travessia encontra outros nós (raiz e nós intermédios) para além das folhas; por outro lado, a sub-variante EA-*all-traverse* tem a vantagem de se adaptar automaticamente a mudanças estruturais na *trie* (provocadas pelo ganho ou perda de folhas); nessa situação, a sub-variante EA-*all-list* é confrontada com a necessidade de operações adicionais, destinadas a manter a conectividade da lista ligada. Por esta ordem de razões, nas simulações efectuadas a opção recaiu pela simulação da variante EA-*all-traverse*, variante doravante implícita na designação mais geral EA-*all*.

Por fim, importa ainda discutir o impacto das diferentes estratégias usadas pelos algoritmos EA-$\mathcal{L}$ e EA-*all*, quando confrontados com várias folhas b, à mesma distância mínima de t; no primeiro caso, recorde-se, foi assumida implicitamente uma selecção aleatória; no segundo caso, já referimos a escolha da primeira folha encontrada nessas condições. Em resumo, aplicados sobre a mesma *trie*, os algoritmos podem resultar na escolha de folhas b diferentes. Ora, se por um lado essas folhas se encontram à mesma distância topológica (número de saltos entre entradas) de t, por outro lado essa distância pode ser diferente no número de nós do *cluster* que é necessário visitar até localizar t (dado que o caminho mais curto entre cada folha b e o alvo t pode atravessar nós diferentes e em número diferente).

4.6 Localização Distribuída com Grafos Chord Completos

Os grafos Chord [SMK+01] têm raízes no Hashing Consistente [KLL+97, KSB+99], que já confrontamos com os nossos modelos de distribuição[22]. No que se segue introduzimos, de forma breve, o essencial dos grafos Chord na sua formulação original ("grafos Chord esparsos em H"), seguida da descrição da nossa variante ("grafos Chord completos em H").

4.6.1 Grafos Chord Esparsos em H

Os grafos Chord viabilizam a aplicação do paradigma do Hashing Consistente (HC) num ambiente descentralizado, ao proporcionar localização distribuída de *hashes*/ partições, em alternativa à tabela global/centralizada do Hashing Consistente original.

Tomando como ponto de partida a figura 3.12, baseada na utilização de nós virtuais, a figura 4.4 faz a sua extensão à visão preconizada pelo Chord; note-se que a utilização de nós virtuais não representa perda de generalidade: com apenas nós computacionais, os mecanismos de localização distribuída são semelhantes, com a diferença de que os grafos Chord são mais esparsos em H e as distâncias médias nesses grafos serão menores.

[21] Ver secção 4.9.4.2.
[22] Rever secção 3.8.

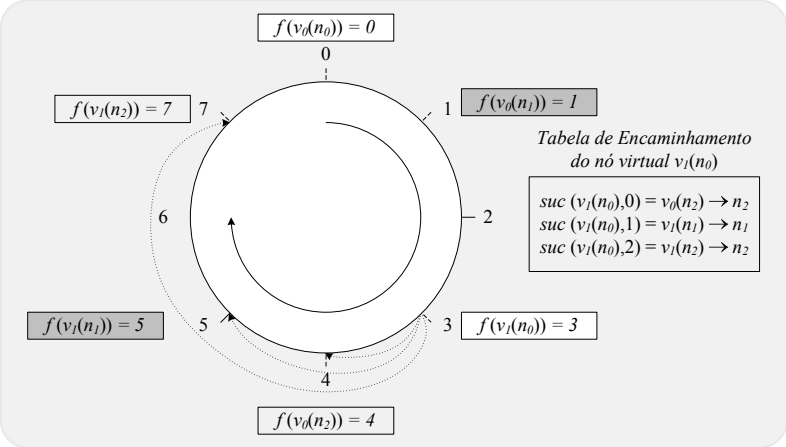


Figura 4.4: Visão do Hashing Consistente no Chord (com nós virtuais).

Por cada nó virtual v, existe uma *tabela de encaminhamento* com $\mathcal{L} = log_2\mathcal{H}$ entradas, de índices $l = 0, 1, ..., \mathcal{L}-1$; a entrada de índice l contém $suc(v, l)$, o identificador do l'ésimo nó virtual *sucessor* de v, num grafo que liga todos os nós virtuais da DHT; descobrir $suc(v, l)$ equivale a descobrir o nó virtual responsável pelo l'ésimo *hash* sucessor de $f(v)$; esse *hash* é $suc(f(v), l) = [f(v) + 2^l] \ mod \ \mathcal{H}$; na figura, é representada a *tabela de encaminhamento* de $v_1(n_0)$; as setas pontilhadas conduzem aos *hashes* $suc[f(v_1(n_0)), l]$; os nós virtuais (e, por dedução, os nós computacionais) hospedeiros desses *hashes* constam da tabela de $v_1(n_0)$).

Para se descobrir o nó computacional responsável por um *hash* h, a partir de um nó virtual v, basta: i) calcular o maior l tal que $suc(f(v), l) \leq h$; ii) reencaminhar o pedido de localização para o nó virtual responsável por $suc(f(v), l)$, ou seja, para o nó virtual $suc(v, l)$ (dado pela l'ésima entrada da tabela de encaminhamento de v); iii) repetir os passos i) e ii). Este algoritmo garante que a localização de qualquer *hash* h visita não mais de $log_2\mathcal{V}$ nós virtuais da DHT. Note-se que o reencaminhamento de pedidos de localização entre nós virtuais envolve a passagem de mensagens entre os seus nós computacionais.

Um grafo G_C^V, para uma função de *hash* de $\mathcal{L} = log_2\mathcal{H}$ bits, é um grafo *conexo*, $\mathcal{K}-regular$ (de grau $\mathcal{K} = \mathcal{L}$, pois todos os vértices têm $\mathcal{L}$ sucessores) e de diâmetro $d_{\max} \approx log_2\#V$, sendo também possível provar [SMK+01, LKRG03] que a distância média é $\overline{d} \approx d_{\max}/2$.

4.6.2 Grafos Chord Completos em H

A nossa abordagem à utilização de grafos Chord adopta como vértices todas as entradas/ *hashes* H de uma DHT. O grafo resultante, $G_C^H(\mathcal{L})$, é um "grafo Chord completo em H" (no que se segue, a forma abreviada "grafo Chord completo" é usada com igual significado).

Em $G_C^H(\mathcal{L})$, o *conjunto dos sucessores* e o l'ésimo *sucessor* de um *hash* h são dados por

$$Suc(h) = \{suc(h, l) : l = 0, 1, ..., \mathcal{L} - 1\} \tag{4.11}$$

$$suc(h,l) = (h + 2^l) \ mod \ 2^{\mathcal{L}} \ , \text{ com } l = 0, 1, ..., \mathcal{L} - 1 \tag{4.12}$$

A definição de *predecessor* – $pred(h,l)$ – e de *conjunto de predecessores* – $Pred(h)$ – é dual.

Para se realizar um grafo $G_C^H(\mathcal{L})$ é então necessário manter, por cada *hash*/entrada da DHT, uma *tabela de encaminhamento*, que regista o nó (virtual ou computacional) responsável por cada um dos $\mathcal{L}$ *hashes* sucessores. Para se descobrir o nó responsável por um *hash* h, com base na tabela de encaminhamento de um *hash* $h' \neq h$, é necessário repetir o seguinte algoritmo (convencional) por cada *hash* visitado: i) calcular o maior l tal que $suc(h', l) \leq h$; ii) reencaminhar o pedido de localização para o nó hospedeiro de $suc(h', l)$ (indicado pela l'ésima entrada da tabela de encaminhamento de h'). O algoritmo garante que a localização de qualquer *hash* h visita não mais de $log_2\mathcal{H} = \mathcal{L}$ entradas da DHT[23].

A figura 4.5 é uma representação do grafo $G_C^H(3)$. A representação afasta-se da configuração habitual em círculo/anel, de forma a permitir uma comparação com o grafo DeBruijn $G_B^H(3)$, sendo evidente a malha de arestas mais densa de $G_C^H(3)$, dado o seu maior *grau*.

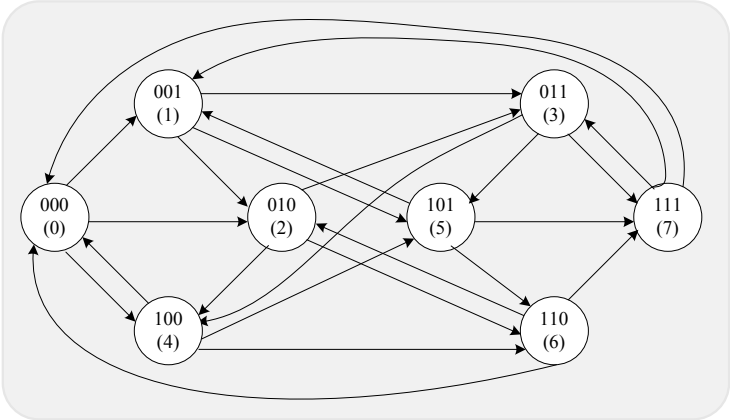


Figura 4.5: Representação do grafo $G_C^H(3)$.

4.6.2.1 Distâncias entre Vértices/Hashes

Num grafo $G_C^H(\mathcal{L})$, a distância $d(x,y)$ entre os vértices x e y corresponde a uma *distância exponencial*, ou seja, o número mínimo de "saltos exponenciais" necessários para chegar de x a y. Esse número é facilmente calculável a partir da *distância euclidiana* entre x e y:

$$d_{\text{euc}}(x,y) = \begin{cases} y - x & \textbf{se} \quad x \leq y \\ 2^{\mathcal{L}} - (x - y) & \textbf{se} \quad y < x \end{cases} \tag{4.13}$$

[23]E como existem, geralmente, várias entradas por nó, a localização de uma entrada pode visitar o mesmo nó repetidas vezes, algo que os algoritmos de *encaminhamento acelerado* pretendem, precisamente, evitar.

A distância $d(x, y)$ corresponde simplesmente ao número de bits 1 no valor da distância $d_{\text{euc}}(x, y)$. Dados x e y na base 10, o algoritmo 4.7 pode ser usado para calcular $d(x, y)$[24]:

Algoritmo 4.7: Cálculo da distância entre dois vértices num grafo $G_C^H(\mathcal{L})$.

Entrada: dois vértices x e y (na base 10)
Saída: distância d entre os vértices x e y

$d \leftarrow 0$
$d_{\text{euc}} \leftarrow d_{\text{euc}}(x, y)$
$l \leftarrow \mathcal{L} - 1$
fazer
    $d \leftarrow d + (d_{\text{euc}} \text{ div } 2^l)$
    $d_{\text{euc}} \leftarrow d_{\text{euc}} \bmod 2^l$
    $l \leftarrow l - 1$
enquanto $d_{\text{euc}} > 0$
retornar d

Basicamente, o algoritmo procura factorizar $d_{\text{euc}}(x, y)$ numa soma de factores de amplitude exponencial 2^l, com $l \in \{0, 1, ..., \mathcal{L} - 1\}$); o número desses factores corresponde a $d(x, y)$.

Para as distâncias máxima e média, as seguintes expressões sistematizam o definido antes:

$$d_{\max}(G_C^H(\mathcal{L})) = \mathcal{L} \tag{4.14}$$

$$\overline{d}(G_C^H(\mathcal{L})) = \frac{\mathcal{L}}{2} \tag{4.15}$$

4.6.2.2 Relações entre Vértices

Para evitar ambiguidades na designação das relações entre os vértices, define-se ainda que:

- um vértice x diz-se *anterior* a um vértice y se "x ocorre *antes* de y no círculo H";
- um vértice x diz-se *posterior* a um vértice y se "x ocorre *após* y no círculo H";
- um vértice x diz-se *predecessor* de um vértice y se $d(x, y) = 1$;
- um vértice x diz-se *sucessor* de um vértice y se $d(y, x) = 1$.

No seguimento destas definições, o *conjunto dos anteriores*, e cada *anterior* em si, são:

$$Ant(v) = \{ant(v, a) : a = 1, 2, ..., 2^{\mathcal{L}} - 1\} \tag{4.16}$$

[24] Em `http://infolab.stanford.edu/~manku/bitcount/bitcount.html` analisam-se vários algoritmos de contagem de bits 1; o mais eficiente é quase 30 vezes mais rápido que o menos eficiente (semelhante ao da figura 4.13); nas simulações que deram origem à secção 4.9 utilizamos o algoritmo mais eficiente.

$$ant(v,a) = (v-a)\ mod\ 2^{\mathcal{L}}\ ,\ \text{com}\ a = 1,2,...,2^{\mathcal{L}}-1 \tag{4.17}$$

A definição de *posterior* – $post(v,p)$ – e de *conjunto de posteriores* – $Post(v)$ – é dual. Estas definições são úteis na fundamentação do Encaminhamento Euclidiano (secção 4.6.5).

4.6.3 AVL/RBTree de Encaminhamento

Na localização distribuída com grafos $G_C^H(\mathcal{L})$, a *árvore de encaminhamento* mantida por cada nó computacional n da DHT é uma árvore binária balanceada, designada genericamente por *avl de encaminhamento*. Cada nó (raiz, intermédio ou folhas) da *avl* preserva uma tabela de encaminhamento, específica de um *hash* da partição $H(n)$ associada ao nó n[25]. Na *avl*, as tabelas são indexadas e ordenadas pelo *hash* a que dizem respeito.

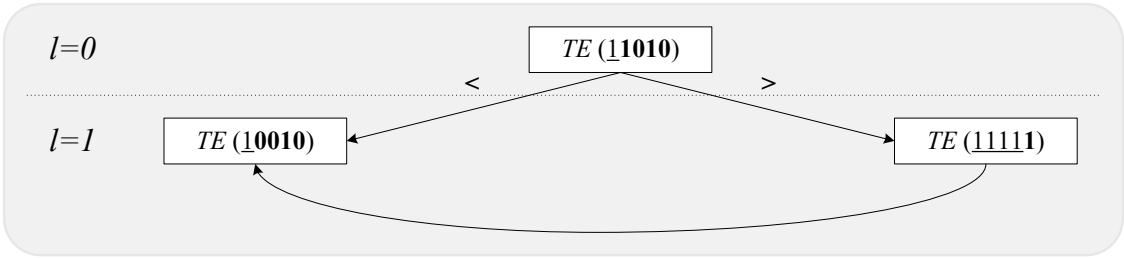


Figura 4.6: Uma *avl de encaminhamento*, para $\mathcal{L} = 5$.

A figura 4.6 representa uma *avl de encaminhamento* com três tabelas de encaminhamento; essas tabelas correspondem às mesmas entradas da DHT, que as tabelas da *trie* da figura 4.3, se assumirmos que os bits então denotados a X têm o valor indicado a sublinhado, na figura 4.6; o último nível da *avl* é $l = 1 \leq \mathcal{L} = 5$, ao passo que, na *trie*, o útimo nível era $l = 4$, o que transmite a ideia de uma maior profundidade média das *tries* face às *avls*.

Na figura 4.6 é ainda possível observar uma ligação da tabela mais à direita (correspondente ao maior *hash*/entrada atribuído ao nó computacional), para a tabela mais à esquerda (correspondente ao menor *hash*); em rigor, essa ligação não é um requisito numa AVL ou estrutura afim mas, a ser suportada, torna mais eficiente os algoritmos de encaminhamento acelerado, nos quais a "circularidade" do espaço dos *hashes* deve ser levada em conta; ainda no contexto desses algoritmos, uma funcionalidade igualmente importante é o suporte a pesquisas por proximidade, que permitem obter o nó da *avl* mais próximo (por defeito, ou por excesso) de um certo nó inexistente na *avl*; na realidade, ambas as funcionalidades são oferecidas numa biblioteca de Red-Black Trees de código aberto [Ive03], que acabou por ser utilizada na avaliação da localização distribuída com grafos $G_C^H(\mathcal{L})$ (ver secção 4.9); no essencial, as Red-Black Trees são também árvores balanceadas, mas mais eficiente que as AVLs, para o padrão específico de acessos gerados nas nossas simulações [Pfa04]. Assim, no que se segue, a designação *rbtree (de encaminhamento)* será usada preferencialmente.

[25] Operando-se com nós virtuais, $H(n)$ seria a união das partições associadas a cada nó virtual de n.

4.6.4 Algoritmos de Encaminhamento

Na senda do que se fez para grafos DeBruijn binários, apresentamos de seguida vários algoritmos de encaminhamento para grafos Chord completos, capazes de explorar de forma adequada uma *rbtree de encaminhamento*. Neste contexto, seguimos as mesmas convenções anteriormente definidas, para designar as entradas da DHT envolvidas numa localização distribuída: i) t (*target*) denota uma entrada/*hash* cuja localização se pretende, ii) c (*current*) denota a entrada escolhida como destino do anterior salto de encaminhamento e iii) n (*next*) denota a entrada escolhida como destino do próximo salto de encaminhamento.

4.6.4.1 Encaminhamento Convencional (EC)

Algoritmo 4.8: Algoritmo EC (Encaminhamento Convencional).

1. procurar a entrada c na *rbtree* do nó actualmente visitado
2. calcular as sucessoras de c (fórmula 4.12) e a distância delas a t (algoritmo 4.7)
3. definir n como a sucessora de c que minimiza a distância a t
4. determinar o nó hospedeiro de n na tabela de encaminhamento de c
5. **se** $n = t$ **então** terminar a localização (o hospedeiro de n é o hospedeiro de t)
6. **senão** reencaminhar o pedido de localização de t para o hospedeiro de n **fim se**

O algoritmo 4.8 é idêntico ao algoritmo 4.3 para grafos DeBruijn binários, com excepção do passo 2., onde a fórmula 4.12 e o algoritmo 4.7 são específicos dos grafos Chord completos.

4.6.4.2 Encaminhamento Melhorado (EM)

Algoritmo 4.9: Algoritmo EM (Encaminhamento Melhorado).

1. procurar a entrada t na *rbtree* do nó actualmente visitado
2. **se** t existir **então** terminar a localização (o nó actual é o hospedeiro de t)
3. **senão** executar o Algoritmo EC **fim se**

O algoritmo 4.9 é similar ao algoritmo 4.4 para grafos DeBruijn binários, com excepção do passo 3., onde se invoca o algoritmo 4.8 para grafos Chord completos. O sentido de um encaminhamento *melhorado* foi já discutido, no âmbito da descrição do algoritmo 4.4.

Algoritmo 4.10: Algoritmo EA-*all* (Encaminhamento Acelerado, variante *all*).

1. $m \leftarrow \mathcal{L}$
2. **para** cada nó h na *rbtree* do nó actual **fazer**
   - **se** $d(h,t) < m$ **então** $b \leftarrow h$ e $m \leftarrow d(h,t)$ **fim se**

   fim para
3. calcular as sucessoras de b (fórmula 4.12) e a distância delas a t (algoritmo 4.7)
4. definir n como a sucessora de b que minimiza a distância a t
5. determinar o hospedeiro de n na tabela de encaminhamento de b
6. **se** $n = t$ **então** terminar a localização (o hospedeiro de n é o hospedeiro de t)
7. **senão** reencaminhar o pedido de localização de t para o hospedeiro de n **fim se**

4.6.4.3 Encaminhamento Acelerado, variante *all* (EA-*all*)

O algoritmo 4.10 é semelhante ao algoritmo 4.6 para grafos DeBruijn binários, com excepção do passo 2., que descreve agora a visita a todos os nós h de uma *rbtree* (em vez da visita a todas as folhas f de uma *trie*), em busca do nó b que minimiza a distância a t.

Tal como já foi referido para o algoritmo 4.6, há pelo menos dois tipos de estratégias, EA-*all-list* e EA-*all-traverse*, para visitar todas as tabelas locais, cada qual representando um compromisso diferente entre desempenho e adaptabilidade ao dinamismo estrutural da árvore de encaminhamento. Retomando a argumentação então utilizada (em favor da adaptabilidade) os resultados das simulações a apresentar posteriormente reflectem a opção pela variante EA-*all-traverse*, implícita doravante na designação mais geral EA-*all*.

4.6.4.4 Encaminhamento Acelerado, variantes Euclidianas (EA-E-1, EA-E-$\mathcal{L}$)

Os algoritmos EA-E-$\mathcal{L}$ e EA-E-1 desempenham um papel semelhante ao do algoritmo EA-$\mathcal{L}$ para grafos DeBruijn binários, ou seja, procuram realizar encaminhamento acelerado sem recorrer à análise exaustiva de todas as tabelas de encaminhamento locais. Para o efeito, os algoritmos EA-E-$\mathcal{L}$ e EA-E-1 (que comportam esforços diferentes) baseiam as suas decisões nos resultados da procura de um sub-conjunto de entradas, em cada *rbtree*.

Assim, o algoritmo EA-E-$\mathcal{L}$ (algoritmo 4.11) envolve $\mathcal{L}$ pesquisas na *rbtree*, c.f. o ciclo do passo 2.; para cada pesquisa $l = 0, 1, ..., \mathcal{L} - 1$, busca-se a entrada p mais próxima (*anterior* ou igual a) de $pred(t, l)$, em distância euclidiana; no final do ciclo, a entrada b, com a menor distância exponencial a t, é escolhida em resultado da análise das tabelas de encaminhamento de c e das entradas encontradas em cada uma das $\mathcal{L}$ pesquisas efectuadas.

Algoritmo 4.11: Algoritmo EA-E-$\mathcal{L}$ (Enc. Acelerado, variante Euclidiana-$\mathcal{L}$).

1. $b \leftarrow c$
2. **para** $l \leftarrow 0, 1, ..., \mathcal{L} - 1$ **fazer**
   - $p \leftarrow rbtree\_search\_MinEuclidianDistance(pred(t, l))$
   - **se** $d(p, t) < d(b, t)$ **então** $b \leftarrow p$ **fim se**

   fim para
3. a 7.: igual aos passos 3. a 7. do Algoritmo EA-*all*

Algoritmo 4.12: Algoritmo EA-E-1 (Enc. Acelerado, variante Euclidiana-1).

1. $p \leftarrow rbtree\_search\_MinEuclidianDistance(pred(t, 0))$
2. **se** $d(p, t) < d(c, t)$ **então** $b \leftarrow p$ **senão** $b \leftarrow c$ **fim se**
3. a 7.: igual aos passos 3. a 7. do Algoritmo EA-*all*

O algoritmo EA-E-1 (algoritmo 4.12) procura, na *rbtree*, a entrada p mais próxima (*anterior* ou igual a) de $pred(t, 0)$, em distância euclidiana; depois, de entre p e c, elege a entrada b com a menor distância exponencial a t (ver passos 1. e 2. do algoritmo).

Na prática os algoritmo EA-E-1 e EA-E-$\mathcal{L}$ podem ser vistos como casos particulares de uma classe de algoritmos EA-E-$(l+1)$, com $l = 0, 1, ..., \mathcal{L} - 1$; para cada valor de l, o algoritmo resultante efectua $l + 1$ pesquisas na *rbtree*, em busca das entradas p mais próximas dos predecessores $pred(t, 0)$, $pred(t, 1)$, ..., $pred(t, l + 1)$; nesta perspectiva, os algoritmos EA-E-2, EA-E-3,..., EA-E-$(\mathcal{L} - 1)$ (cujo desempenho não investigamos), representam outras possibilidades, com esforços crescentes para a tomada da decisão de encaminhamento, mas também com resultados (na minimização da distância a t) tendencialmente melhores[26].

Anteriormente (secção 4.6.3), referiu-se a conveniência de uma implementação de *avls*/ *rbtrees* capaz de suportar, convenientemente, i) a "circularidade" do espaço dos *hashes* e ii) pesquisas por proximidade. É precisamente no contexto dos algoritmos de Encaminhamento Euclidiano, mais especificamente na busca das entradas mais próximas de $pred(t, l)$, que se revela a utilidade dessas funcionalidades; na realidade, o resultado ideal dessa busca seria encontrar os próprios pontos $pred(t, l)$, dado que estes estão à distância exponencial mínima (de um salto apenas) do alvo t; todavia, na ausência de um ou mais desses pontos ideais, a tabela de encaminhamento dos pontos anteriores mais próximos, pode ainda ser explorada com proveito, para tomar uma decisão de encaminhamento melhor informada; na secção seguinte elaboramos sobre a lógica subjacente ao Encaminhamento Euclidiano.

[26] Num contexto de balanceamento dinâmico de carga, a possibilidade de afinar o esforço e os resultados produzidos configuram uma mais valia dos algoritmos euclidianos.

4.6.5 Distância Euclidiana *versus* Distância Exponencial

A lógica subjacente ao Encaminhamento Euclidiano derivou do estudo da relação entre a *distância euclidiana* e a *distância exponencial* nos grafos Chord completos. No que se segue, apresentam-se os resultados principais da investigação realizada em torno do tema.

Basicamente, a *distância euclidiana*, $d_{\text{euc}}(x, y)$, entre os *hashes* x e y, mede o número de *hashes* que separam x de y, no "círculo Chord", no sentido dos ponteiros do relógio; por sua vez, a *distância exponencial*, $d(x, y)$, entre x e y, mede o número mínimo de saltos exponenciais, de amplitude 2^l (com $l = 0, 1, ..., \mathcal{L} - 1$), que separam x de y; como já se referiu, $d_{\text{euc}}(x, y)$ pode-se factorizar numa soma de factores do tipo 2^l (sem repetições[27]).

Para verificar se a minimização da distância euclidiana entre dois vértices/*hashes* contribui para minimizar a distância exponencial, é útil comparar a monotonia de dois tipos de séries:

- $S =< d(Ant(h), h) >$: distân. exponenciais, dos *anteriores* de qualquer *hash* h, a h;
- $S_{\text{euc}} =< d_{\text{euc}}(Ant(h), h) >$: dist. euclidianas, dos *anteriores* de qualquer *hash* h, a h.

A tabela 4.1 apresenta as séries S e S_{euc}, para $\mathcal{L} = 1, 2, 3, 4, 5$; o alinhamento vertical da tabela permite emparelhar facilmente, para o mesmo *hash anterior* de h, os valores correspondentes, nas séries S e S_{euc}, mesmo para valores diferentes de $\mathcal{L}$; em cada série, o símbolo $\vdots$ separa um grupo com 2^l números, do próximo, com 2^{l+1} (sendo $0 \leq l \leq \mathcal{L} - 1$).

$\mathcal{L} = 1$	S=	<1>
	S_{euc}=	<1>
$\mathcal{L} = 2$	S=	<1 $\vdots$ 1 2>
	S_{euc}=	<1 $\vdots$ 2 3>
$\mathcal{L} = 3$	S=	<1 $\vdots$ 1 2 $\vdots$ 1 2 2 3>
	S_{euc}=	<1 $\vdots$ 2 3 $\vdots$ 4 5 6 7>
$\mathcal{L} = 4$	S=	<1 $\vdots$ 1 2 $\vdots$ 1 2 2 3 $\vdots$ 1 2 2 **3** **2** 3 3 4>
	S_{euc}=	<1 $\vdots$ 2 3 $\vdots$ 4 5 6 7 $\vdots$ 8 9 10 **11** **12** 13 14 15>
$\mathcal{L} = 5$	S=	<1 $\vdots$ 1 2 $\vdots$ 1 2 2 3 $\vdots$ 1 2 2 3 2 3 $\vdots$ 1 2 2 3 2 3 3 4 2 3 3 4 3 4 4 5>
	S_{euc}=	<1 $\vdots$ 2 3 $\vdots$ 4 5 6 7 $\vdots$ 8 9 10 11 12 13 14 15 $\vdots$ 16 ... >

Tabela 4.1: Séries $S_{\text{euc}} =< d_{\text{euc}}(Ant(h), h) >$ e $S =< d(Ant(h), h) >$ para $\mathcal{L} = 1, 2, 3, 4, 5$.

Contrariamente às séries S_{euc} (que são, por definição, progressões aritméticas (estritamente) crescentes, de razão 1), as séries S não têm monotonia (ora crescem, ora decrescem). Quer isto dizer que "minimizar a distância euclidiana" não implica necessariamente "minimizar a distância exponencial". Atente-se, por exemplo, no caso dos pontos $ant(h, 11)$ e $ant(h, 12)$, cujas distâncias euclidianas e exponenciais a h, são representadas a negrito,

[27] O que, basicamente, recorda a conversão de um número na base 10 para a base 2.

na secção $\mathcal{L} = 4$ da tabela: ora, por um lado tem-se $d_{\text{euc}}(ant(h, 11), h) = 11 < 12 = d_{\text{euc}}(ant(h, 12), h)$ mas, por outro, tem-se $d(ant(h, 11), h) = 3 > 2 = d(ant(h, 12), h)$.

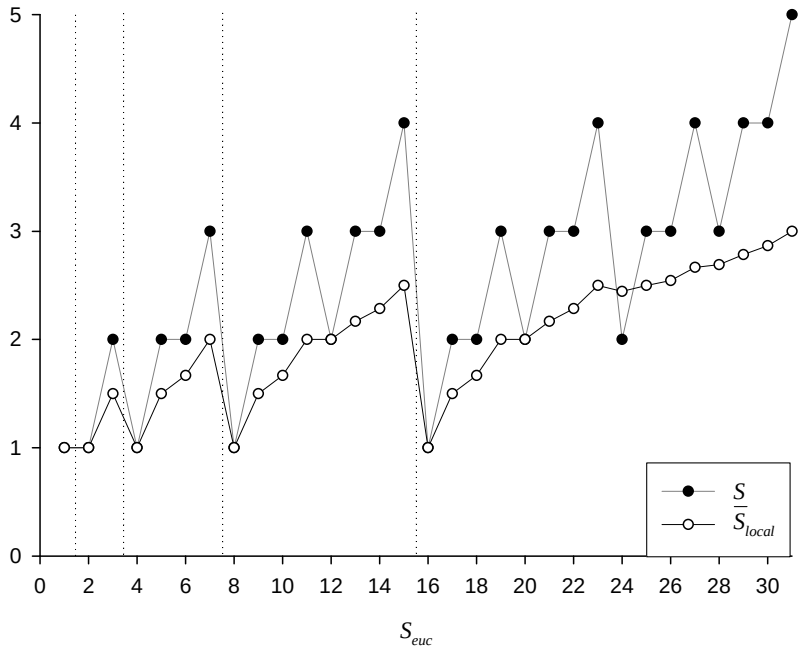


Figura 4.7: Séries S e $\overline{S}_{local}$ em função de S_{euc}.

A figura 4.7 complementa a tabela 4.1, representando a série S em função da série S_{euc}, para o caso particular $\mathcal{L} = 5$; o traço vertical, picotado, divide a figura nas mesmas zonas (com os mesmos 2^l números em cada uma) que as geradas pela notação $\vdots$ na tabela 4.1.

Observa-se assim que: i) quando a distância euclidiana, S_{euc} (eixo horizontal) é potência de 2, a distância exponencial, S, exibe mínimos locais de 1; tal significa que o ponto *anterior* a h, que está à distância euclidiana em causa, é também um ponto *predecessor*; ii) entre um mínimo local e o próximo, a distância euclidiana continua a aumentar linearmente e a tendência é também para o incremento da distância exponencial, mas de forma irregular; neste contexto, ganha relevância a grandeza $\overline{S}_{local}$, que representa a média acumulada de S, entre dois mínimos locais consecutivos; da observação da evolução de $\overline{S}_{local}$, conclui-se que, quanto mais próximos de um mínimo local (na direcção de h[28]), maior será a probabilidade de se minimizar a distância exponencial; precisamente, é esta a lógica que justifica a tentativa, por parte dos algoritmos de Encaminhamento Euclidiano, de procurarem, nas *rbtrees*, os pontos *predecessores* e, na ausência destes, os seus *anteriores* mais próximos.

Para grafos DeBruijn binários, o mesmo tipo de análise não demonstrou a existência de uma relação entre a monotonia da distância euclidiana e a monotonia da distância entre vértices, justificando a ausência de Encaminhamento Euclidiano para esse tipo de grafos.

[28] Ou seja, da direita para a esquerda, na figura.

4.7 Suporte à Duplicação/Subdivisão de Entradas

A aplicação dos modelos de distribuição do capítulo anterior (concretamente de M2 e M4, que operam sob Hashing Dinâmico) conduz, em determinados momentos da evolução de uma DHT, à *duplicação* do número total de entradas, $\mathcal{H}$, ou, equivalentemente, à *subdivisão* de cada entrada em duas. Essa duplicação/subdivisão representa a transição entre um estágio anterior da evolução da DHT, em que eram relevantes $\mathcal{L}$ bits de entre os produzidos pela função de *hash*, para um novo estágio, em que são relevantes $\mathcal{L}+1$ bits.

A duplicação de $\mathcal{H}$ tem implicações diversas, denotando um ponto de sincronização global, a ultrapassar de forma eficiente. No que diz respeito ao suporte à localização distribuída, é necessário derivar o grafo $G_B^H(\mathcal{L}+1)$ ou $G_C^H(\mathcal{L}+1)$, a partir de $G_B^H(\mathcal{L})$ ou $G_C^H(\mathcal{L})$, c.f. o tipo de grafo adoptado; na prática, essa derivação passa pela duplicação do número de tabelas de encaminhamento, seguida da sua actualização; idealmente, despoletada a duplicação, i) cada nó da DHT deveria ser capaz de actualizar as novas tabelas de encaminhamento de forma autónoma, sem necessidade de trocar mensagens para o efeito; adicionalmente, ii) é desejável que a simples duplicação de $\mathcal{H}$ não acarrete a transferência de entradas entre nós da DHT[29]; de seguida descrevem-se os mecanismos que permitem atingir estes objectivos.

4.7.1 Relações Genealógicas

Sempre que se duplica $\mathcal{H}$, cada *hash*/entrada de $\mathcal{L}$ bits é subdividida em duas entradas de $\mathcal{L}+1$ bits. À primeira entrada convencionamos designá-la por *ascendente* e às segundas por *descendentes*. Dos $\mathcal{L}+1$ bits usados para identificar cada descendente, $\mathcal{L}$ bits são herdados da entrada ascendente, como é usual em esquemas de Hashing Dinâmico. O $(\mathcal{L}+1)$'ésimo bit será *prefixo* ou *sufixo* dos $\mathcal{L}$ bits anteriores, c.f. o tipo de grafo adoptado.

Assim, para grafos DeBruijn binários, o bit adicional actuará como *prefixo* e, para grafos Chord completos, o bit adicional actuará como *sufixo*. Estas opções correspondem a assumir que i) com grafos $G_B^H(\mathcal{L})$, são relevantes os $\mathcal{L}$ bits menos significativos do *hash* (logo um bit adicional será prefixo dos anteriores) e ii) com grafos $G_C^H(\mathcal{L})$, são relevantes os $\mathcal{L}$ bits mais significativos do *hash* (donde um bit adicional será sufixo dos bits anteriores).

As opções anteriores, para o papel do bit adicional, permitem i) a actualização autónoma das tabelas de encaminhamento e ii) a repartição dos registos da entrada ascendente, pelas descendentes, sem troca de mensagens; a primeira propriedade será demonstrada na secção 4.7.3; a segunda decorre naturalmente do facto de, na subdivisão, os registos de uma entrada ascendente se repartirem pelas suas descendentes[30], sendo que, imediatamente a seguir à subdivisão, as descendentes são alojadas no mesmo nó hospedeiro da ascendente.

Seguidamente, formalizamos convenientemente as relações genealógicas. Antes porém,

[29] Ou seja, a duplicação em si não deve ser uma causa dessa transferência. Na realidade, o nexo de causalidade é inverso: tipicamente, a duplicação de $\mathcal{H}$ é necessária quando já não há entradas suficientes para alimentar novos nós da DHT, ou nós cuja capacidade aumentou; nesses cenários, após a duplicação de $\mathcal{H}$, haverá necessariamente transferência (migração) de entradas, para responder precisamente aos requisitos que originaram a duplicação de $\mathcal{H}$; o impacto genérico das transferências é discutido na secção 4.8.

[30] Já que a sequência de bits de uma entrada descendente extende em um bit a sequência da ascendente.

introduz-se alguma notação. Assim, sendo h um certo *hash*/entrada na base 10, tem-se

- $h_{|\mathcal{L}|}$: notação para h no nível $\mathcal{L}$;
- $h_{\langle\mathcal{L}\rangle}$: notação para a sequência de $\mathcal{L}$ bits que representa h na base 2.

4.7.1.1 Relações Genealógicas com Grafos $G_B^H(\mathcal{L})$

Dada a entrada $h_{\langle\mathcal{L}\rangle} = h_{\mathcal{L}-1}h_{\mathcal{L}-2}\ldots h_0$, a descendente de nível $\mathcal{L}+1$ e prefixo $p \in \{0,1\}$ é

$$desc(h_{\langle\mathcal{L}\rangle}, p) = ph_{\langle\mathcal{L}\rangle} = ph_{\mathcal{L}-1}h_{\mathcal{L}-2}\ldots h_0 \tag{4.18}$$

O conjunto das descendentes de $h_{\langle\mathcal{L}\rangle}$, denotado por $Desc(h_{\langle\mathcal{L}\rangle})$, pode então definir-se como

$$Desc(h_{\langle\mathcal{L}\rangle}) = \{desc(h_{\langle\mathcal{L}\rangle}, p) : p = 0, 1\} \tag{4.19}$$

Trabalhando na base 10 e no nível $\mathcal{L}$ então para cada $h_{|\mathcal{L}|} \in \{0, 1, ..., 2^{\mathcal{L}} - 1\}$ tem-se

$$desc(h_{|\mathcal{L}|}, p) = h_{|\mathcal{L}|} + (p \times 2^{\mathcal{L}}) = h + (p \times 2^{\mathcal{L}}) \tag{4.20}$$

De forma dual, dada uma entrada $h_{\langle\mathcal{L}+1\rangle} = h_{\mathcal{L}}h_{\mathcal{L}-1}\ldots h_0$, a sua ascendente de nível $\mathcal{L}$ é

$$asc(h_{\langle\mathcal{L}+1\rangle}) = h_{\langle\mathcal{L}\rangle} = h_{\mathcal{L}-1}h_{\mathcal{L}-2}\ldots h_0 \tag{4.21}$$

Na base 10, tem-se $h_{|\mathcal{L}+1|} \in \{0, 1, ..., 2^{\mathcal{L}+1} - 1\}$, bem como $h_{\langle\mathcal{L}+1\rangle} = h_{\mathcal{L}}h_{\langle\mathcal{L}\rangle} = ph_{\langle\mathcal{L}\rangle}$, donde

$$asc(h_{|\mathcal{L}+1|}) = h_{|\mathcal{L}+1|} - (p \times 2^{\mathcal{L}}) \tag{4.22}$$

4.7.1.2 Relações Genealógicas com Grafos $G_C^H(\mathcal{L})$

Dada a entrada $h_{\langle\mathcal{L}\rangle} = h_0h_1\ldots h_{\mathcal{L}-1}$, a *descendente* de nível $\mathcal{L}+1$ e sufixo $s \in \{0,1\}$ é

$$desc(h_{\langle\mathcal{L}\rangle}, s) = h_{\langle\mathcal{L}\rangle}s = h_0h_1\ldots h_{\mathcal{L}-1}s \tag{4.23}$$

O conjunto das descendentes de $h_{\langle\mathcal{L}\rangle}$, denotado por $Desc(h_{\langle\mathcal{L}\rangle})$, define-se agora como

$$Desc(h_{\langle\mathcal{L}\rangle}) = \{desc(h_{\langle\mathcal{L}\rangle}, s) : s = 0, 1\} \tag{4.24}$$

Trabalhando na base 10, então para cada entrada $h_{|\mathcal{L}|} \in \{0, 1, ..., 2^{\mathcal{L}} - 1\}$ tem-se agora

$$desc(h_{|\mathcal{L}|}, s) = 2 \times h_{|\mathcal{L}|} + s = 2 \times h + s \tag{4.25}$$

Para uma entrada $h_{\langle \mathcal{L}+1 \rangle} = h_0 \ldots h_{\mathcal{L}-1} h_{\mathcal{L}}$, a sua *ascendente* de nível $\mathcal{L}$ é agora dada por

$$asc(h_{\langle \mathcal{L}+1 \rangle}) = h_{\langle \mathcal{L} \rangle} = h_0 \ldots h_{\mathcal{L}-2} h_{\mathcal{L}-1} \tag{4.26}$$

Na base 10, tem-se $h_{|\mathcal{L}+1|} \in \{0, 1, ..., 2^{\mathcal{L}+1} - 1\}$, bem como $h_{\langle \mathcal{L}+1 \rangle} = h_{\langle \mathcal{L} \rangle} h_{\mathcal{L}} = h_{\langle \mathcal{L} \rangle} p$, donde

$$asc(h_{|\mathcal{L}+1|}) = (h_{|\mathcal{L}+1|} - s) \; div \; 2 \tag{4.27}$$

4.7.2 Representação em Trie da Evolução da DHT

Uma abstracção do tipo *trie* é particularmente adequada à representação da evolução por estágios das nossas DHTs[31]. Assim, quando o grafo subjacente à DHT é $G_B^H(\mathcal{L})$ ou $G_C^H(\mathcal{L})$, denotamos por $T_B^H(\mathcal{L})$ ou $T_C^H(\mathcal{L})$, respectivamente, a *trie* que representa todas as subdivisões de entradas da DHT, ocorridas até ao nível de subdivisão (ou profundidade) $\mathcal{L}$. Neste contexto, as figuras 4.8.a) e 4.8.b) representam $T_B^H(\mathcal{L})$ e $T_C^H(\mathcal{L})$ para $\mathcal{L} = 1, 2, 3$.

Cada *trie* estrutura-se em níveis (delimitados por linhas horizontais), correspondentes aos vários estágios que a DHT subjacente atravessou; os $2^{\mathcal{L}}$ nós/*hashes* de uma *trie*, em cada nível $\mathcal{L}$, representam as $2^{\mathcal{L}}$ entradas da DHT, no seu estágio de evolução $\mathcal{L}$; cada nó/*hash* de uma *trie*, é identificado pela sequência de bits correspondente ao trajecto da raiz (R), até ao nó (entre parêntesis, é representado o valor da sequência, na base 10); para *tries* $T_B^H(\mathcal{L})$, esse trajecto corresponde a construir o identificador do nó/entrada, do bit menos significativo (lsb) para o mais significativo (msb), ocorrendo o inverso para *tries* $T_C^H(\mathcal{L})$.

As linhas a tracejado denotam relações de descendência entre uma entrada de um certo nível, e um par de entradas do nível seguinte. O bit mais/menos significativo de cada entrada, a negrito, representa assim um dos prefixos/sufixos possíveis (de valor **0** ou **1**) que, aplicado a uma entrada ascendente, permite obter a entrada descendente em causa.

Precisamente, o papel diferente (de *prefixo* ou *sufixo*) atribuído ao bit adicional, na passagem de um nível ao próximo, revela-se na diferente sequência dos *hashes* que se pode observar, da esquerda para a direita, para um mesmo nível $\mathcal{L} \geq 2$, nas *tries* $T_B^H(\mathcal{L})$ e $T_C^H(\mathcal{L})$; por exemplo, para $T_B^H(3)$, a sequência é $0, 4, 2, 6, 1, 5, 3, 7$, ao passo que, para $T_C^H(3)$, a sequência é $0, 1, 2, 3, 4, 5, 6, 7$; estas sequências são determinadas pelas fórmulas 4.20 e 4.25, para o cálculo das entradas descendentes, introduzidas na secção anterior.

4.7.3 Dedução das Tabelas de Encaminhamento

Nesta secção elaboramos sobre a dedução das tabelas de encaminhamento dos grafos $G_B^H(\mathcal{L}+1)$ e $G_C^H(\mathcal{L}+1)$, a partir das tabelas de encaminhamento de $G_B^H(\mathcal{L})$ e $G_C^H(\mathcal{L})$[32].

[31] Estas *tries* são puramente *virtuais*, ao contrário das *tries de encaminhamento* discutidas anteriormente.
[32] Concluída a dedução das tabelas/grafos do nível $\mathcal{L}+1$ as/os do nível anterior serão descartadas(os).

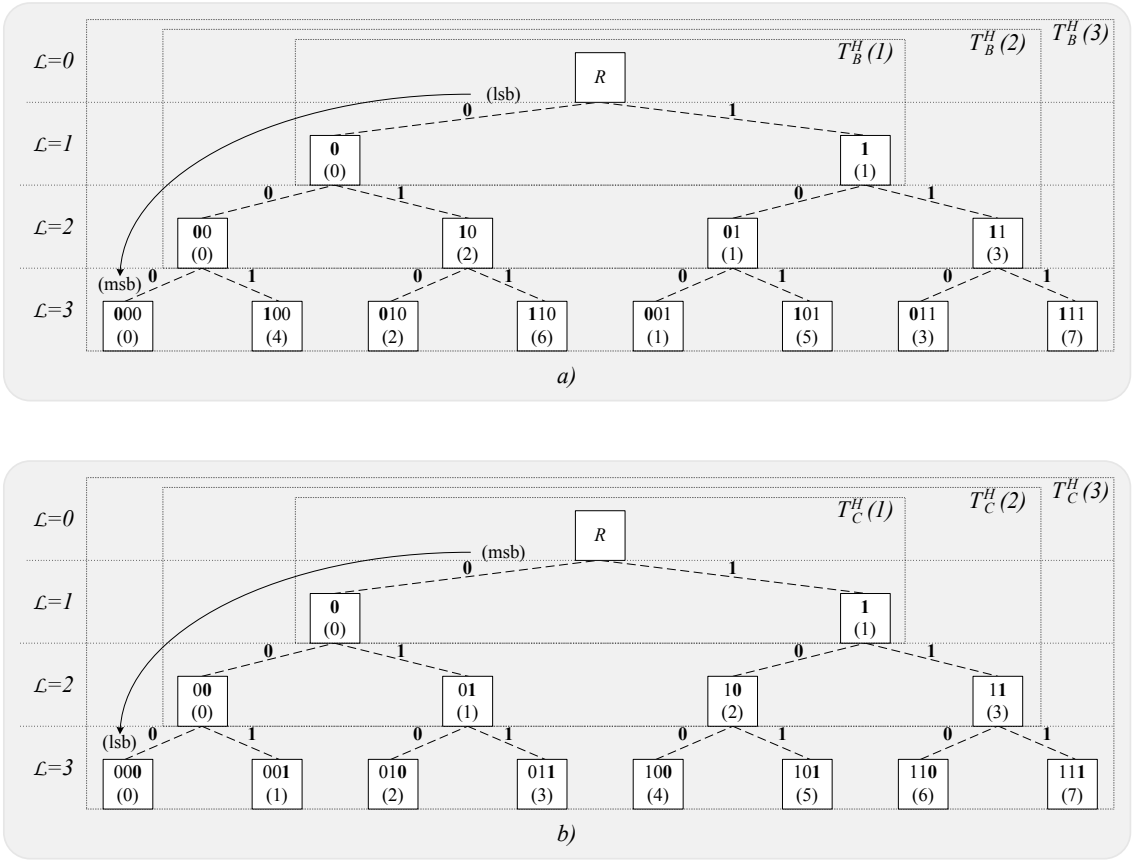


Figura 4.8: Representação de a) $T_B^H(\mathcal{L})$ e b) $T_C^H(\mathcal{L})$, para $\mathcal{L} = 1, 2, 3$.

Na prática, o problema a resolver é o da definição das tabelas das entradas *descendentes* quando $\mathcal{H}$ duplica. Ora, como veremos de seguida, é possível deduzir as tabelas das entradas *descendentes* (entradas que são vértices no grafo $G_B^H(\mathcal{L}+1)$ ou $G_C^H(\mathcal{L}+1)$) apenas com base na tabela da entrada *ascendente* (que é vértice no grafo $G_B^H(\mathcal{L})$ ou $G_C^H(\mathcal{L})$).

4.7.3.1 Dedução das Tabelas de Encaminhamento de $G_B^H(\mathcal{L}+1)$

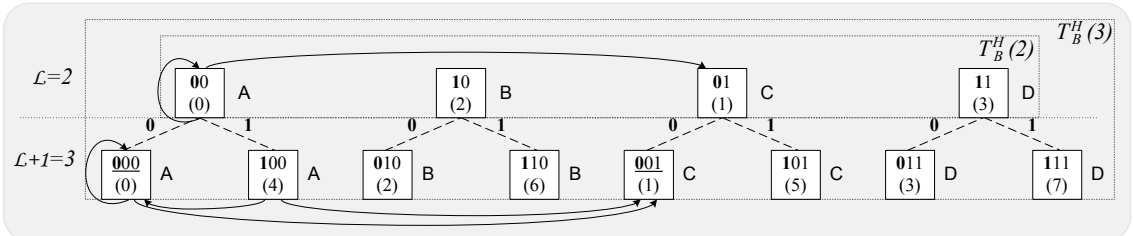


Figura 4.9: Transição $T_B^H(2) \rightarrow T_B^H(3)$: relação entre $G_B^H(2)$ e $G_B^H(3)$.

A figura 4.9 serve para ilustrar a transição do estágio $\mathcal{L} = 2$ para o estágio $\mathcal{L}+1 = 3$, numa

DHT em que a localização distribuída assenta sobre grafos DeBruijn binários. Na figura, são representadas as entradas/*hashes* da DHT existentes nesses dois estágios, incluindo as suas relações "genealógicas": as entradas do nível 2 correspondem às folhas da *trie* $T_B^H(2)$ da figura 4.8.a), e as entradas de nível 3 correspondem às folhas da *trie* $T_B^H(3)$.

Para além da indicação dos nós computacionais hospedeiros (A, B, C ou D) das entradas, a principal "novidade" da figura 4.9 é a representação de arcos dos grafos $G_B^H(2)$ e $G_B^H(3)$; mais especificamente, são representados os arcos de $G_B^H(2)$ que ligam a entrada 00, de nível $\mathcal{L} = 2$, aos seus sucessores nesse nível (esses sucessores são a própria entrada, 00, e a entrada 01); representam-se ainda os arcos do grafo $G_B^H(3)$, que ligam os descendentes de 00, no nível $\mathcal{L} + 1 = 3$, aos seus sucessores nesse nível (os descendentes de 00 são 000 e 100; os sucessores de 000 são $\underline{000}$ e $\underline{001}$; os sucessores de 100 são também $\underline{000}$ e $\underline{001}$).

Apoiando-nos na figura, vejamos agora como se pode deduzir a tabela de encaminhamento da entrada 100 de nível 3; basicamente, pretende-se saber quais são os nós hospedeiros das sucessoras de 100 que, formalmente, se denotam por $suc(100, 0) = \underline{000}$ e $suc(100, 1) = \underline{001}$. Assim, e em primeiro lugar, leva-se em conta que as sucessoras residem no mesmo nó que a sua ascendente[33], ou seja, $n(\underline{000}) = n(asc(\underline{000})) = n(00)$ e $n(\underline{001}) = n(asc(\underline{001})) = n(01)$; em segundo lugar, para se determinar a identidade (A, B, C ou D) de $n(00)$ e de $n(01)$, pesquisa-se afinal a tabela de encaminhamento da ascendente de 100, que é $asc(000) = 00$; com efeito, na tabela de 00, encontram-se os pares (00, A) e (01, C) (dado que 00 e 01 são sucessoras de 00) o que permite desde logo concluir que $n(00)$=A e $n(01)$=C; finalmente, a tabela de encaminhamento de 100 pode ser preenchida, com os pares (000, A) e (001,C).

A descrição anterior, que se reporta a uma situação particular, é desde logo exemplificativa da nossa asserção inicial, de que "é possível deduzir as tabelas das entradas *descendentes* apenas com base na tabela da entrada *ascendente*". Na realidade, essa possibilidade decorre de uma propriedade fundamental que governa a relação entre os grafos $G_B^H(\mathcal{L})$ e $G_B^H(\mathcal{L}+1)$: "dada uma entrada h de nível $\mathcal{L}$, as sucessoras das descendentes de h são um subconjunto das descendentes das sucessoras de h" o que, formalmente, corresponde a[34]

$$Suc(Desc(h)) \subseteq Desc(Suc(h)) \tag{4.28}$$

O método prosseguido na dedução das tabelas das entradas *descendentes* limita-se então a explorar a propriedade 4.28. Em termos genéricos, dada uma entrada h de nível $\mathcal{L}$, as tabelas das descendentes $Desc(h)$ deduzem-se assim: como $Suc(Desc(h)) \subseteq Desc(Suc(h))$, então para determinar os nós responsáveis por $Suc(Desc(h))$ basta determinar os nós responsáveis por $Desc(Suc(h))$; ora, os nós responsáveis por $Desc(Suc(h))$ são os mesmos que os responsáveis por $Suc(h)$; por outro lado, os nós responsáveis por $Suc(h)$ constam precisamente da tabela de h; logo, apenas com base na tabela de h, é possível determinar os nós responsáveis por $Suc(Desc(h))$, de acordo com o pretendido à partida.

Na secção D.1 do apêndice D fornece-se uma demonstração formal da propriedade 4.28.

[33] Pois na transição entre os estágios $\mathcal{L}$ e $\mathcal{L}+1$, as descendentes são criadas no mesmo nó da ascendente.
[34] $Suc(X)$ (ou $Desc(X)$), sendo X conjunto, denota a união de $Suc(x)$ (ou $Desc(x)$) para todos os $x \in X$.

4.7.3.2 Dedução das Tabelas de Encaminhamento de $G_C^H(\mathcal{L}+1)$

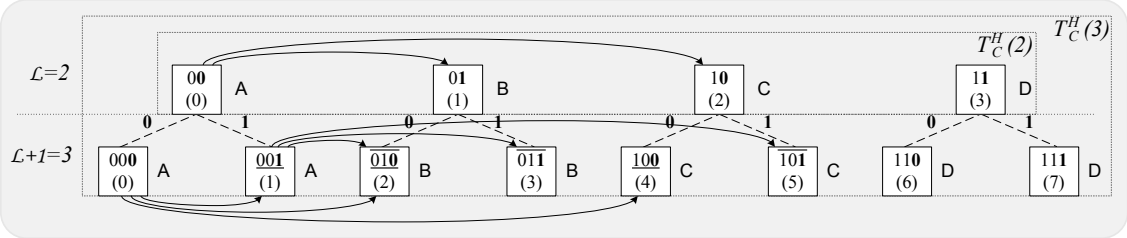


Figura 4.10: Transição $T_C^H(2) \rightarrow T_C^H(3)$: relação entre $G_C^H(2)$ e $G_C^H(3)$.

A figura 4.10 serve de suporte à discussão da transição do estágio $\mathcal{L} = 2$ para o estágio $\mathcal{L}+1 = 3$, numa DHT em que a localização distribuída assenta sobre grafos Chord completos.

A notação é semelhante à utilizada na figura 4.9. Assim, as entradas do nível 2 correspondem às folhas da *trie* $T_C^H(2)$ da figura 4.8.b), e as entradas de nível 3 correspondem às folhas de $T_C^H(3)$; os nós hospedeiros (A, B, C ou D) das entradas são também representados, assim como alguns arcos dos grafos $G_C^H(2)$ e $G_C^H(3)$; em particular, representam-se os arcos de $G_C^H(2)$ que ligam a entrada 00, de nível $\mathcal{L} = 2$, aos seus sucessores (as entradas 01 e 10); representam-se também os arcos de $G_C^H(3)$, que ligam os descendentes de 00, no nível $\mathcal{L}+1 = 3$, aos seus sucessores nesse nível (os descendentes de 00 são 000 e 001; os sucessores de 000 são $\underline{001}$, $\underline{010}$ e $\underline{100}$; os sucessores de 001 são $\overline{010}$, $\overline{011}$ e $\overline{101}$).

Vejamos então como deduzir a tabela de encaminhamento da entrada 000 de nível 3, o que implica determinar os nós das sucessoras de 000; formalmente, essas sucessoras são $suc(000, 0) = \underline{001}$, $suc(000, 1) = \underline{010}$ e $suc(000, 2) = \underline{100}$. Em primeiro lugar, leva-se em conta que as sucessoras residem no mesmo nó que a sua ascendente, ou seja, $n(\underline{001}) = n(asc(\underline{001})) = n(00)$, $n(\underline{010}) = n(asc(\underline{010})) = n(01)$ e $n(\underline{100}) = n(asc(\underline{100})) = n(10)$; em segundo lugar, para se determinar a identidade (A, B, C ou D) de $n(00)$, $n(01)$ e $n(10)$, inspecciona-se a tabela de encaminhamento da ascendente de 000, que é $asc(000) = 00$; de facto, a tabela de 00 contém os pares (01, B) e (10, C) (pois 01 e 10 são sucessoras de 00) o que permite concluir que $n(01)$=B e $n(10)$=C; quanto a $n(00)$, é igual a $n(000)$ e $n(001)$, pois $asc(000) = asc(001) = 00$, donde é imediato que $n(00)$=A; a tabela de encaminhamento de 000 é então preenchida com os pares (001, A), (010, B) e (100,C).

A situação descrita anteriormente exemplifica a "dedução das tabelas das entradas *descendentes* apenas com base na tabela da entrada *ascendente*". A possibilidade de dedução decorre de uma propriedade da relação entre os grafos $G_C^H(\mathcal{L})$ e $G_C^H(\mathcal{L}+1)$ mas que, neste caso, é ligeiramente diferente da enunciada para grafos DeBruijn: "dada uma entrada h de nível $\mathcal{L}$, as sucessoras das descendentes de h fazem parte do conjunto formado pelas descendentes das sucessoras de h juntamente com as próprias descendentes de h", ou seja

$$Suc(Desc(h)) \subseteq [Desc(Suc(h)) \cup Desc(h)] \tag{4.29}$$

Genericamente, dada uma entrada h de nível $\mathcal{L}$, a dedução das tabelas das descendentes $Desc(h)$, com recurso à propriedade 4.29, desenrola-se da seguinte forma: como

$Suc(Desc(h)) \subseteq [Desc(Suc(h)) \cup Desc(h)]$, então para determinar os nós responsáveis por $Suc(Desc(h))$ é suficiente determinar os nós responsáveis por $Desc(Suc(h))$ e $Desc(h)$; ora, os nós responsáveis por $Desc(Suc(h))$ coincidem com os responsáveis por $Suc(h)$ e, por seu turno, os nós responsáveis por $Suc(h)$ constam da tabela de h; quanto aos nós responsáveis por $Desc(h)$, resumem-se a um único nó, $n(h)$, conhecido à partida; donde, a tabela de h é suficiente para determinar os nós responsáveis por $Suc(Desc(h))$.

Na secção D.2 do apêndice D fornece-se uma demonstração formal da propriedade 4.29.

4.8 Impacto do Re-Posicionamento de Entradas

O re-posicionamento de uma entrada (rever causas prováveis na secção 4.3.2) acarreta a necessidade de rectificar a correspondência "entrada $\mapsto$ nó" em uma ou mais tabelas de encaminhamento, a fim de garantir a correcção do grafo de localização distribuída da DHT. Essa actualização tem custos (em mensagens) diferentes, c.f. o tipo de grafo adoptado.

Assim, com grafos DeBruijn binários, o custo *máximo* de actualização é de ordem $O(log\mathcal{H})$: por cada entrada da DHT, existem 2 *predecessoras*, cuja tabela de encaminhamento é necessário corrigir; para o efeito, é necessário localizar cada predecessora, o que incorre num número máximo de $\mathcal{L} = log_2\mathcal{H}$ mensagens para cada predecessora e de $2 \times \mathcal{L}$ no total.

Com grafos Chord completos, o custo *máximo* de actualização é de ordem $O(log^2\mathcal{H})$: agora, por cada entrada da DHT, existem $\mathcal{L} = log_2\mathcal{H}$ entradas *predecessoras*; para corrigir a tabela de encaminhamento de cada predecessora são necessárias, no máximo, $\mathcal{L}$ mensagens; segue-se que a correcção de todas as tabelas comporta, no máximo, um total de $\mathcal{L}^2$ mensagens.

O custo *máximo* corresponde ao pior caso sob Encaminhamento Convencional (EC). O recurso a algoritmos de Encaminhamento Acelerado (EA) permite reduzir substancialmente esse custo, conforme demonstram os resultados das simulações discutidas na secção 4.9.

Assim, com grafos DeBruijn binários, i) o custo *médio* de localização com algoritmos EA é entre 50% a 30% inferior ao custo *médio* com o algoritmo EC (ver secção 4.9.4.2) e ii) o custo *médio* com o algoritmo EC é pouco inferior ao custo *máximo* (ver secção 4.9.4.1); logo, o custo *médio* de actualização com algoritmos EA será entre 50% a 70% de $\mathcal{L}$.

Com grafos Chord completos, i) o custo *médio* de localização com algoritmos EA poderá ser até 55% a 40% inferior ao custo *médio* com o algoritmo EC (ver secção 4.9.4.2) e ii) o custo *médio* com o algoritmo EC é já 50% inferior ao custo *máximo* (ver secção 4.9.4.1); logo, o custo *médio* de localização com algoritmos EA assumirá valores de 22,5% a 30% de $\mathcal{L}$, e o custo *médio* de actualização andará em $(22,5\% \times \mathcal{L})^2$ a $(30\% \times \mathcal{L})^2$ mensagens.

4.9 Avaliação da Localização Distribuída

Nesta secção apresentam-se e discutem-se os resultados de simulações dos algoritmos de encaminhamento apresentados nas secções 4.5.5 e 4.6.4, no âmbito da aplicação de grafos DeBruijn binários e grafos Chord completos, à localização distribuída nas nossas DHTs.

As simulações foram conduzidas no quadro do modelo M2 (rever secção 3.4), que define o *número* de entradas de cada nó de uma DHT num cenário homogéneo. Nesse contexto, assumiu-se $\mathcal{H}_{min}(n) = 8$ (o maior valor de referência usado até agora, para o parâmetro de M2 que estabelece "o número *mínimo* de entradas da DHT, em cada um dos seus nós").

Para a definição da *identidade* das entradas, geraram-se distribuições *aleatórias*, o tipo mais provável, em resultado dos nossos mecanismos de posicionamento (rever secção 4.3).

4.9.1 Metodologia

As simulações desenrolaram-se em duas fases distintas. Numa primeira fase, foram produzidas várias distribuições aleatórias (cada uma delas comportando $\mathcal{H}$ correspondências "entrada $\mapsto$ nó") e, para cada tipo de grafo, foram derivadas as tabelas de encaminhamento correspondentes a cada distribuição, originando-se outras tantas topologias diferentes[35].

Concretamente, sendo $\mathcal{N}$ o número de nós da DHT então, para cada $\mathcal{N} = 1, 2, ..., 1024$: 1) aplicou-se o modelo M2 (com $\mathcal{H}_{min}(n) = 8$) para definir o número total de entradas da DHT, $\mathcal{H}$, e o número de entradas de cada nó, $\mathcal{H}(n)$; 2) por 10 vezes, definiram-se $\mathcal{H}$ correspondências aleatórias "entrada $\mapsto$ nó", o que originou 10 distribuições aleatórias; 3) para cada uma dessas distribuições, e para cada tipo de grafo, definiram-se as tabelas de encaminhamento, processo de que resultaram 20 topologias diferentes (10 para cada tipo de grafo). No total, geraram-se $1024 \times 20 = 20480$ topologias, que foram armazenadas em suporte persistente[36] (ficheiros) para posterior reutilização, na fase seguinte da simulação.

Numa segunda fase, avaliaram-se os algoritmos de encaminhamento, nas topologias geradas na primeira fase (descrita anteriormente). Assim, para cada tipo de grafo, e para cada um dos seus algoritmos, varreu-se a gama $\mathcal{N} = 1, 2, ..., 1024$, testando o algoritmo sobre as 10 topologias disponíveis para cada $\mathcal{N}$; o teste de cada algoritmo sobre uma determinada topologia consistiu em reconstituir as $\mathcal{H}$ correspondências aleatórias "entrada $\mapsto$ nó" subjacentes à topologia; essa reconstituição pode ser feita a partir de qualquer um dos $\mathcal{H}$ vértices do grafo, tomando-o como ponto de partida de $\mathcal{H}$ *cadeias de localização* diferentes, cada uma delas terminando em um dos $\mathcal{H}$ vértices do grafo; donde, havendo $\mathcal{H}$ vértices de partida e, para cada um deles, $\mathcal{H}$ vértices de chegada, há um total de $\mathcal{H}^2$ cadeias de localização que é necessário testar, por cada uma das 10 topologias. Em resumo, para grafos DeBruijn (4 algoritmos) testaram-se 1.57073E+12 cadeias de localização e, para grafos Chord (5 algoritmos) testaram-se 1.96341E+12, num total de 3.53414E+12 cadeias; com testes tão exaustivos pretendeu-se, por um lado, i) validar a metodologia, obtendo resultados experimentais previstos pelos modelos teóricos e, por outro, ii) obter valores mais precisos (face aos produzidos por amostras reduzidas) para as métricas que se escolheu medir; o valor final de cada métrica produzida por cada algoritmo, para cada valor de $\mathcal{N}$, reflecte a média aritmética dos 10 valores produzidos pelas 10 topologias testadas.

[35] Uma topologia é definida pelo conjunto das tabelas de encaminhamento dos vértices de um grafo.

[36] Consumindo 8.4 Gbytes e 34 Gbytes de espaço em disco, para topologias DeBruijn e Chord, respectivamente, sendo que o diferencial de valores para ambas se deve ao diferente número de sucessores que necessitam, por cada vértice do grafo, ou seja, 2 para grafos DeBruijn e $\mathcal{L} = log_2\mathcal{H}$ para grafos Chord.

4.9.2 Tecnologia

As simulações foram conduzidas num *cluster* ROCKS [roc] 4.0.0, de 8 máquinas homogéneas (com um CPU Pentium 4 a 3GHz, 1 GB de SDRAM e chipset i865), permitindo a geração e o teste de diferentes topologias, em paralelo. A geração de topologias foi codificada em Python. O teste aos algoritmos de encaminhamento codificou-se em C, por razões de desempenho. A contabilização de tempos realizou-se com base na instrução `rdtsc` (*read time stamp counter*) disponibilizada pela linguagem *assembly* da arquitectura Intel i86.

Para grafos DeBruijn, as *tries de encaminhamento* foram realizadas com base num módulo de Tries Compactas, codificado em C, e expressamente desenvolvido para o efeito[37]. Para grafos Chord, as *rbtrees de encaminhamento* foram instanciadas com base numa biblioteca de Red-Black-Trees [Ive03], também em C, de domínio público, e que exibe funcionalidades particularmente úteis sob o ponto de vista dos nossos algoritmos de encaminhamento sobre grafos Chord, designadamente a pesquisa de registos por proximidade (*e.g.*, devolução do registo mais próximo, na impossibilidade de encontrar o pretendido).

A realização de testes exaustivos, cobrindo a totalidade das cadeias de encaminhamento possíveis para cada topologia, exigiu vários dias de processamento[38], já com código optimizado, incluindo i) utilização das opções `-O3` e `-mcpu=pentium4` do compilador `gcc`, ii) tipificação como `static inline` para as rotinas mais invocadas, iii) outras optimizações[39].

4.9.3 Métricas

Em cada teste realizado pela metodologia anterior recolheram-se as seguintes métricas:

1. $\overline{d}$ ou $\overline{d}_{\text{chain}}$ – distância média entre qualquer par de entradas[40], dada pela fórmula C.3 para grafos DeBruijn[41] e pela fórmula 4.15 para grafos Chord; esta distância inclui uma distância média *interna* (número de saltos internos/intra-nó, em que a entrada sucessora reside no mesmo nó da antecessora) e uma distância média *externa* (número de saltos externos/entre-nós, em que a entrada sucessora reside num nó diferente), ou seja, $\overline{d}_{\text{chain}} = \overline{d}_{\text{int}} + \overline{d}_{\text{ext}}$; a recolha de $\overline{d}_{\text{chain}}$ fez-se com o objectivo de validar a simulação e estabelecer uma base de comparação para a métrica $\overline{d}_{\text{ext}}$ (ver a seguir);
2. $\overline{d}_{\text{ext}}$ – distância média *externa* entre qualquer par de entradas (como definido acima); a métrica $\overline{d}_{\text{ext}}$, recolhida separadamente de $\overline{d}_{\text{chain}}$, é a métrica que revela, em primeira instância, o efeito acelerador da localização, dos vários algoritmos de localização não-convencional; um dos efeitos prático desses algoritmos deve ser o de evitar a visita repetida do mesmo nó, durante uma cadeia de localização (seja por saltos internos, seja por saltos externos); assim, espera-se que pelo menos $\overline{d}_{\text{ext}}$ converja para $\overline{d}_{\text{chain}}$;

[37] Mas suficientemente genérico, suportando outro tipo de registos, além de tabelas de encaminhamento.

[38] Incluindo depuração, confirmação de resultados e retoma de testes após interrupção por falhas de corrente (só possível pela incorporação de funcionalidades de *checkpointing* no código da simulação).

[39] Por exemplo, o cálculo de distâncias entre vértices de um grafo Chord, com uma rotina para contagem de bits 1 num número inteiro, disponível em `http://infolab.stanford.edu/~manku/bitcount/bitcount.html`, permitiu ganhos de desempenho dramáticos nos algoritmos EA-E-1, EA-E-$\mathcal{L}$ e EA-*all*.

[40] Ou, equivalentemente, "comprimento médio de qualquer cadeia de encaminhamento".

[41] Uma vez que, para estes grafos, não existe uma fórmula não-exaustiva, mas sim minorantes e majorantes, dados pelas fórmulas 4.9 e 4.10, que permitem validar os valores dados pela fórmula C.3.

3. $\overline{CPU}_{\text{hop}}$ – tempo médio de CPU, por salto/decisão de encaminhamento (em μs); cada decisão de encaminhamento comporta um custo, em tempo de CPU; em geral, espera-se que os algoritmos capazes de minimizar $\overline{d}_{\text{chain}}$ mais efectivamente, sejam também mais demorados a tomar cada decisão; o algoritmo ideal será aquele que consegue, em simultâneo, tomar decisões rápidas e que conduzam a cadeias pequenas;

4. $\overline{RAM}_{\text{node}}$ – consumo médio de RAM (em bytes), em tabelas de encaminhamento, por nó; é dependente apenas do tipo de grafo seleccionado (DeBruijn ou Chord).

A partir destas métricas são deriváveis outras, úteis para a comparação de algoritmos:

1. $\overline{CPU}_{\text{chain}}$ – tempo médio de CPU, por cadeia de localização (em μs); depende, por um lado, do número médio de decisões de encaminhamento por cadeia (dado pela métrica $\overline{d}_{\text{chain}}$) e, por outro, do tempo médio que cada decisão de encaminhamento comporta (dado pela métrica $\overline{CPU}_{\text{hop}}$), de forma que $\overline{CPU}_{\text{chain}} = \overline{d}_{\text{chain}} \times \overline{CPU}_{\text{hop}}$;

2. $\overline{NET}_{\text{chain}}$ – tempo médio de Comunicação, por cadeia de localização (em μs); refere-se ao tempo (médio) consumido na troca de mensagens entre diferentes nós da DHT, durante uma cadeia de localização; neste contexto, $\overline{NET}_{\text{chain}} = \overline{d}_{\text{ext}} \times \overline{NET}_{\text{hop}}$;

   - $\overline{NET}_{\text{hop}}$ representa o tempo (médio) em que incorre a troca de *uma* mensagem de encaminhamento entre dois nós; trata-se de um parâmetro (e não uma medida) que procura fazer reflectir (de forma aproximada), na nossa avaliação, o impacto de diferentes tecnologias de rede; neste contexto, admitiu-se $\overline{NET}_{\text{hop}} = 1, 10, 100, 1000\ \mu s$, em que i) 1 μs se refere a uma tecnologia (por enquanto, hipotética) extremamente rápida, na presença da qual acaba por ser praticamente irrelevante o facto de um salto de encaminhamento ser interno ou externo; ii) 10 μs se refere a tecnologias como Myrinet ou Infiniband; iii) 100 μs se refere a tecnologias vulgarizadas, como Fast/Gigabit-Ethernet e iv) 1000 μs se refere a tecnologias do domínio WAN (valor útil para inferir o mérito dos nossos algoritmos em ambientes normalmente associados a DHTs do tipo P2P);

3. $\overline{TIME}_{\text{chain}}$ – tempo médio total, por cadeia de localização (em μs); tendo em conta as métricas definidas anteriormente, $\overline{TIME}_{\text{chain}} \approx \overline{CPU}_{\text{chain}} + \overline{NET}_{\text{chain}}$; implícita nesta formulação está a assunção de que tempo consumido em saltos internos é contabilizado em $\overline{CPU}_{\text{chain}}$, o que corresponde à realidade das simulações realizadas;

4. $\mathcal{R}(a)$ – classificação (*ranking*) do algoritmo a, no contexto de um conjunto A de algoritmos (relativos ao mesmo tipo de grafo, ou a tipos diferentes); genericamente,

$$\mathcal{R}(a) = \omega_{TIME} \times \frac{\overline{TIME}_{\text{chain}}(a)}{\max[\overline{TIME}_{\text{chain}}(A)]} + \omega_{RAM} \times \frac{\overline{RAM}_{\text{node}}(a)}{\max[\overline{RAM}_{\text{node}}(A)]}, \text{ com } \mathcal{R}(a) \in [0,1] \tag{4.30}$$

   , em que ω_{TIME} e ω_{RAM} definem, respectivamente, pesos associados aos factores tempo e RAM, sendo $\omega_{TIME} + \omega_{RAM} = 1$; adicionalmente, $\max[\overline{TIME}_{\text{chain}}(A)]$ e

$\max[\overline{RAM}_{\text{node}}(A)]$ denotam os máximos absolutos de $\overline{TIME}_{\text{chain}}(a)$ e $\overline{RAM}_{\text{node}}(a)$, no contexto do conjunto A de algoritmos; com base na classificação definida pela fórmula 4.30, o algoritmo $a \in A$ mais competitivo será então o que minimizar $\mathcal{R}(a)$.

4.9.4 Distância Total e Distância Externa

4.9.4.1 Encaminhamento Convencional

Procurando estabelecer uma base de comparação para os resultados obtidos com os outros algoritmos de encaminhamento, começamos por analisar, em separado, os resultados específicos do Encaminhamento Convencional (algoritmos EC), exibidos na figura 4.11.

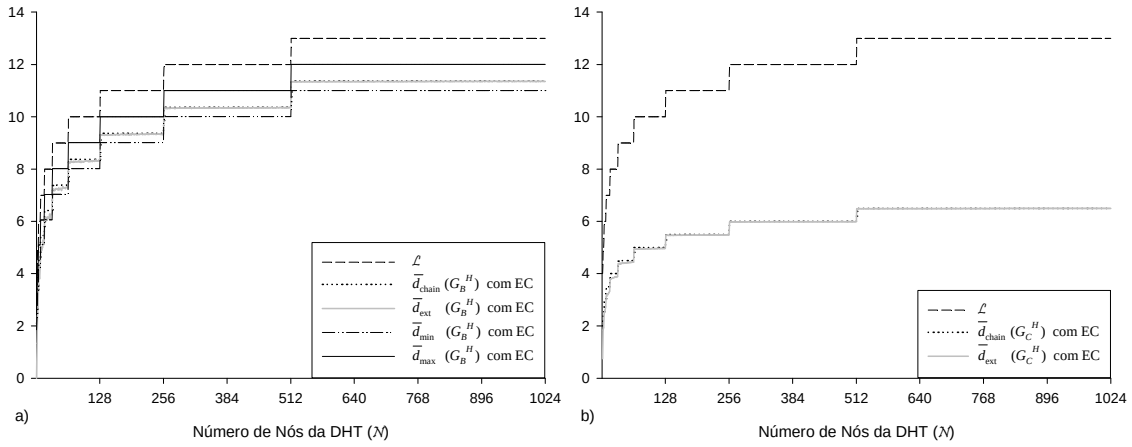


Figura 4.11: Algoritmo EC: $\overline{d}_{\text{chain}}$ e grandezas afins, para grafos a) DeBruijn e b) Chord.

Assim, para ambos os grafos, destaca-se a evolução em escada de $\mathcal{L} = log_2\mathcal{H}$, resultante da aplicação do modelo M2, no qual o número total de entradas da DHT, $\mathcal{H}$, duplica progressivamente, à medida que aumenta o número de nós participantes na DHT, $\mathcal{N}$; a variação em escada de $\mathcal{L}$ acaba por se reflectir no mesmo tipo de variação das outras grandezas, as quais, directa ou indirectamente, dependem de $\mathcal{L}$ (por exemplo, $\mathcal{L}$ representa o diâmetro dos grafos $G_B^H(\mathcal{L})$ e $G_C^H(\mathcal{L})$, ou seja, é a maior distância individual $d(x, y)$ entre quaisquer dois vértices/entradas x e y, pelo que a distância média $\overline{d}_{\text{chain}}$ será necessariamente menor que $\mathcal{L}$); a representação da evolução de $\mathcal{L}$ é útil, no sentido de que fornece um primeiro termo de comparação para a interpretação da evolução das outras grandezas medidas.

A mesma figura apresenta a evolução da distância média, $\overline{d}_{\text{chain}}$, e da distância média *externa*, $\overline{d}_{\text{ext}}$. Para grafos DeBruijn, a distância $\overline{d}_{\text{chain}}$ representa-se enquadrada pelos limites teóricos $\overline{d}_{\text{min}}$ e $\overline{d}_{\text{max}}$, dados pelas fórmulas 4.9 e 4.10; para grafos Chord, a distância $\overline{d}_{\text{chain}}$ corresponde simplesmente a $\mathcal{L}/2$, como estabelecido anteriormente pela fórmula 4.15.

A comparação das figuras 4.11.a) e 4.11.b) permite desde logo constatar uma vantagem da abordagem Chord face à DeBruijn: para o mesmo número global de entradas (dado por $\mathcal{H} = 2^{\mathcal{L}}$), a primeira assegura valores de $\overline{d}_{\text{chain}}$ substancialmente inferiores aos da segunda (em 30% a 45%), uma consequência natural do facto de, num grafo DeBruijn,

cada entrada/vértice ter um número fixo (2) de sucessores, ao passo que, num grafo Chord, esse número ($\mathcal{L}$) é dinâmico, acompanhando a evolução da DHT (dito de outra forma, com grafos DeBruijn, a amplitude da "cobertura topológica" de cada vértice é, em termos absolutos, fixa, e o seu valor relativo reduz-se progressivamente, quando a DHT cresce[42]).

Para ambos os grafos, $\overline{d}_{\text{ext}} \leq \overline{d}_{\text{chain}}$, como esperado; porém, a diferença entre as métricas diminui com o aumento do número de nós da DHT, $\mathcal{N}$, ou seja, numa cadeia de encaminhamento, o número de saltos externos aumenta quando $\mathcal{N}$ aumenta; tal deve-se à evolução imposta pelo modelo M2, para o número de entradas por nó, $\mathcal{H}(n)$; essa evolução faz com que cada nó possua uma fracção/quota real $\mathcal{Q}^r(n) = \mathcal{H}(n)/\mathcal{H}$ do número global de entradas, $\mathcal{H}$, sempre muito próxima de uma fracção/quota ideal $\mathcal{Q}^i(n) = 1/\mathcal{N}$; ora, uma vez que $1/\mathcal{N}$ diminui (exponencialmente) à medida que $\mathcal{N}$ aumenta (linearmente), então cada nó deterá uma proporção de entradas cada vez menor, face ao número global de entradas da DHT; segue-se que, como o número de tabelas de encaminhamento de cada nó é igual ao seu número de entradas, então a amplitude do conhecimento topológico que cada nó possui sobre a DHT, diminui de cada vez que $\mathcal{N}$ aumenta; o saldo é o aumento progressivo do número de saltos externos, ao longo de uma cadeia de encaminhamento, pois o conhecimento topológico necessário à sua minimização é cada vez menos representativo; este fenómeno ocorre também com os algoritmos de encaminhamento não-convencional.

4.9.4.2 Encaminhamento Não-Convencional

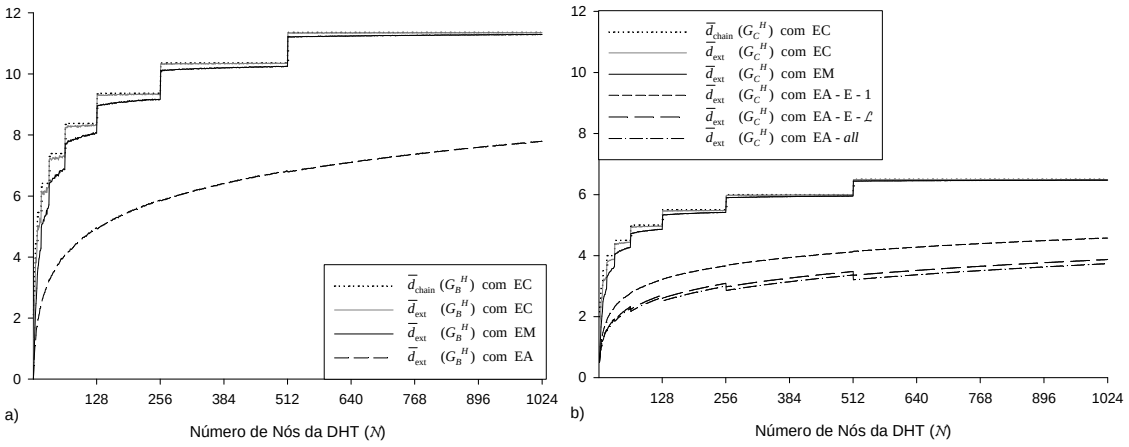


Figura 4.12: Evolução da distância $\overline{d}_{\text{ext}}$, para grafos a) DeBruijn e b) Chord.

A figura 4.12 mostra a evolução de $\overline{d}_{\text{ext}}$ para todos os algoritmos. Juntamente com $\overline{d}_{\text{chain}}$ (que é independente do algoritmo de encaminhamento), os valores de $\overline{d}_{\text{ext}}$, para o Encaminhamento Convencional, servem de referencial de comparação aos outros algoritmos. Na figura 4.12.a), a representação de $\overline{d}_{\text{ext}}$ para EA-$\mathcal{L}$ e EA-*all* serve-se da linha comum EA, dado que as distâncias específicas dos algoritmos são muito próximas; apesar de muito aproximados, os valores em causa não são, em rigor, coincidentes, nem se esperava que o fossem, pelo facto (discutido na secção 4.5.5.4) de não haver garantia de que o trajecto

[42] Grafos DeBruijn k-ários (uma hipótese não explorada) resolveriam este problema – ver secção 4.10.

topológico (entre entradas) e físico (entre nós), seguido pelos algoritmos, seja o mesmo.

Assim, para ambos os grafos, o Encaminhamento Melhorado (EM) produz uma redução marginal (em geral $< 5\%$) na distância externa, face ao Encaminhamento Convencional (EC). Com algoritmos de Encaminhamento Acelerado (EA), a redução é mais significativa: para grafos DeBruijn, os algoritmos EA (representados pela linha comum EA), reduzem a distância externa de 50% a 30%; para grafos Chord, o algoritmo EA-E-1 permite uma redução de 40% a 30% e as variantes EA-E-$\mathcal{L}$ e EA-*all* geram uma redução de 55% a 40%.

Verifica-se pois que, em termos percentuais, os algoritmos não-convencionais produzem melhorias semelhantes, face ao Encaminhamento Convencional, em ambos os grafos. A comparação entre grafos, nos resultados absolutos obtidos confirma, todavia, a supremacia da abordagem Chord; com efeito, mesmo com Encaminhamento Convencional, a abordagem Chord é, quase sempre, mais competitiva que o Encaminhamento Acelerado da abordagem DeBruijn (comprovável pela comparação visual das figuras 4.12.a) e 4.12.b)); grosso-modo, os algoritmos da abordagem Chord asseguram distâncias externas entre 35% a 50% inferiores às distâncias dos algoritmos da mesma categoria, da abordagem DeBruijn.

É também de registar que a competitividade do Encaminhamento Melhorado e do Acelerado, face ao Convencional, tende a diminuir quando $\mathcal{N}$ aumenta[43], pela mesma ordem de razões que, no quadro do Encaminhamento Convencional, motivam a aproximação da distância $\overline{d}_{\text{ext}}$ à distância $\overline{d}_{\text{chain}}$ (rever a discussão associada, no fim da secção 4.9.4.1).

4.9.4.3 Comparação com Grafos Completos em N

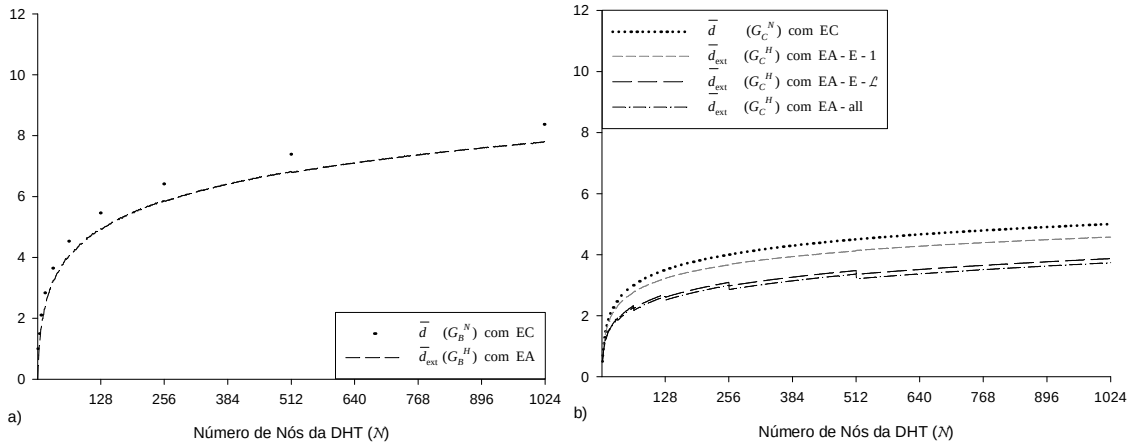


Figura 4.13: Distâncias a) $\overline{d}(G_B^N)$ *versus* $\overline{d}_{\text{ext}}(G_B^H)$ e b) $\overline{d}(G_C^N)$ *versus* $\overline{d}_{\text{ext}}(G_C^H)$.

A figura 4.13 permite confirmar a eficácia dos algoritmos de Encaminhamento Acelerado na aproximação do esforço de localização em grafos G^H, ao esforço de localização em grafos G^N com Encaminhamento Convencional. Essa aproximação, recorde-se (rever secção 4.4.3), foi estabelecida como objectivo do Encaminhamento Acelerado. Na realidade, esse encaminhamento permite obter distâncias inferiores às distâncias médias $\overline{d}(G_B^N)$ e $\overline{d}(G_C^N)$.

[43] Como denunciam os valores percentuais fornecidos, por ordem decrescente, para as reduções de $\overline{d}_{\text{ext}}$.

Assim, para grafos DeBruijn, os algoritmos de Encaminhamento Acelerado (EA) em G^H asseguram uma distância (externa) $\overline{d}_{\text{ext}}(G_B^H)$ da ordem dos 90% da distância média $\overline{d}(G_B^N)$ em G^N. Note-se que, à semelhança de G_B^H, G_B^N é um grafo DeBruijn binário, no qual o número de vértices é potência de 2; logo, G_B^N existe apenas quando $\mathcal{N} = \#N$ é potência de 2 e, por esse motivo, $\overline{d}(G_B^N)$ é representado para um conjunto restrito de valores de $\mathcal{N}$.

Para grafos Chord, os ganhos variam com o tipo de algoritmo de Encaminhamento Acelerado, chegando a ser superiores aos obtidos com grafos DeBruijn. Assim, com o algoritmo EA-E-1, a distância (externa) $\overline{d}_{\text{ext}}(G_C^H)$ é da ordem dos 92% da distância média $\overline{d}(G_C^N)$, o que ainda representa um ganho marginal. Já o algoritmo EA-*all* assegura uma distância (externa) $\overline{d}_{\text{ext}}(G_C^H)$ da ordem dos 73% da distância média $\overline{d}(G_C^N)$, o que representa um ganho substancial (o ganho obtido pelo algoritmo EA-E-$\mathcal{L}$ será ligeiramente inferior).

4.9.5 Tempo de CPU por Salto e Consumo de RAM por Nó

As figuras 4.14.a) e 4.14.b) exibem a evolução da métrica $\overline{CPU}_{\text{hop}}$ (tempo médio de CPU, por decisão de encaminhamento); a figura 4.15.a) mostra a evolução da métrica $\overline{RAM}_{\text{node}}$ (consumo médio de RAM, em tabelas de encaminhamento, por nó); finalmente, a figura 4.15.b) exibe a evolução de grandezas do modelo M2, relevantes na discussão que se segue.

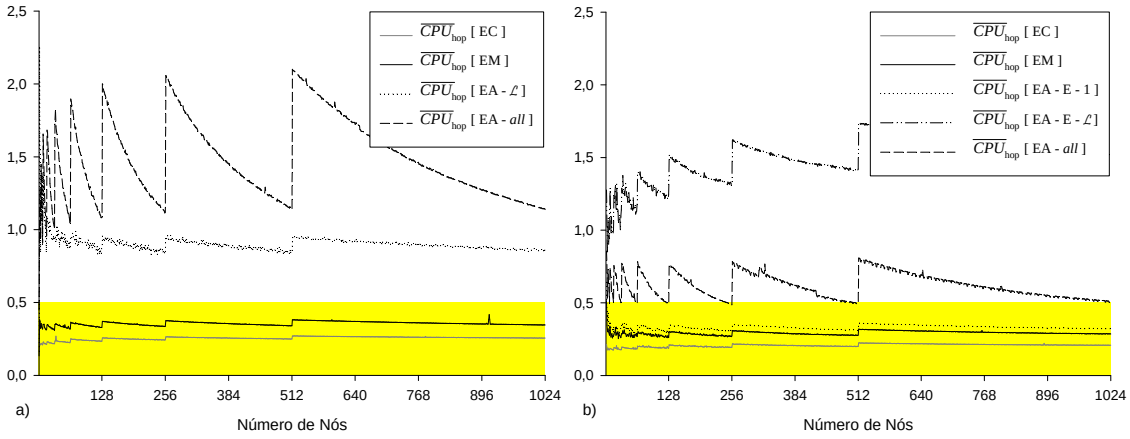


Figura 4.14: Evolução do tempo $\overline{CPU}_{\text{hop}}$ (em μs), para grafos a) DeBruijn e b) Chord.

As métricas evoluem por estágios, de acordo com o modelo M2; neste, i) o número global de entradas, $\mathcal{H}$, duplica entre estágios sucessivos e ii) o número médio de entradas por nó, $\overline{\mathcal{H}}(n)$, decai em cada estágio, de $\mathcal{H}_{max}(n) = 2.\mathcal{H}_{min}(n)$ até $\mathcal{H}_{min}(n)$ – ver figura 4.15.b).

Assim, recordando que a cada entrada da DHT corresponde uma tabela de encaminhamento então, no início de cada estágio, o número (médio) de tabelas de encaminhamento por cada *trie* ou *rbtree de encaminhamento*, atinge o máximo, bem como a profundidade média dessas árvores; isso traduz-se i) num maior tempo de CPU consumido, em cada nó, por cada decisão de encaminhamento, bem como ii) num maior consumo de RAM, em cada nó, em tabelas de encaminhamento; durante um estágio, à medida que o número de nós da DHT aumenta, o número médio de entradas (e logo de tabelas de encaminhamento) por nó diminui, o que faz decair as métricas $\overline{CPU}_{\text{hop}}$ e $\overline{RAM}_{\text{node}}$, até ao final do estágio.

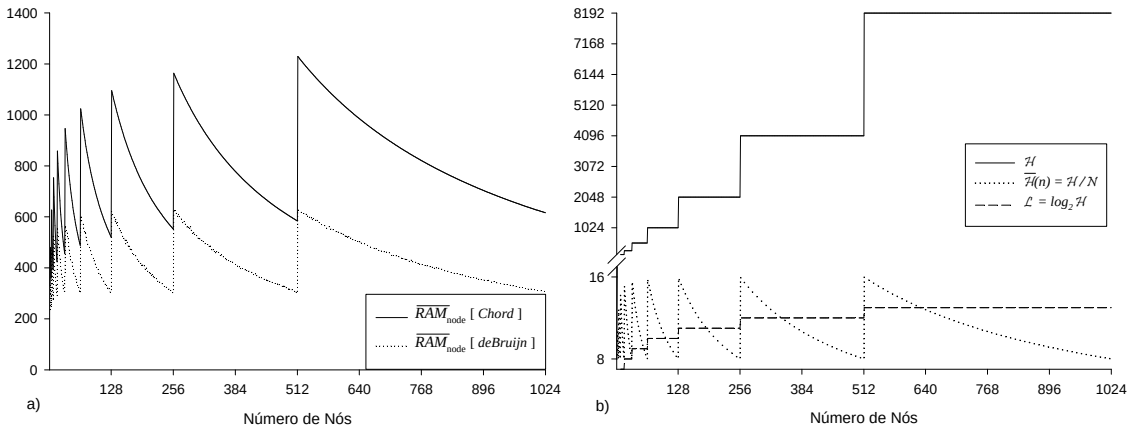


Figura 4.15: Evolução de a) $\overline{RAM}_{\text{node}}$ (em bytes) e b) grandezas básicas de M2.

4.9.5.1 Tempo de CPU por Salto

Embora o padrão evolutivo (por estágios) da métrica $\overline{CPU}_{\text{hop}}$ seja semelhante para os vários algoritmos, de ambos os grafos, os valores concretos dessa métrica variam de algoritmo para algoritmo, em resultado do diferente esforço computacional desenvolvido por cada um. De seguida, discute-se essa variação e o posicionamento relativo resultante.

Começamos então por notar que, ao contrário do observável para a métrica $\overline{RAM}_{\text{node}}$, a gama de variação dos tempos $\overline{CPU}_{\text{hop}}$ obtidos com algoritmos de ambos os grafos é semelhante[44], com um mínimo de $\approx 0.25\mu s$ e um máximo de $\approx 2.0\mu s$. Adicionalmente, consideramos esses tempos repartidos por duas zonas, com um valor fronteira de $\approx 0.5\mu s$.

Assim, com um só acesso à árvore de encaminhamento (*trie*/*rbtree*, c.f. o caso) o Encaminhamento Convencional (EC) é o algoritmo mais rápido, para ambos os grafos; de perto, segue-se o Encaminhamento Melhorado (EM), com apenas mais um acesso. Verifica-se também que, com grafos Chord, os algoritmos EC e EM são um pouco mais rápidos, reflexo não só do diferente custo computacional envolvido, como também de um acesso tendencialmente mais rápido que as *rbtrees* balanceadas oferecem sobre as *tries*. Com grafos Chord, o Encaminhamento Acelerado na sua variante E-1 (algoritmo EA-E-1) necessita igualmente de apenas dois acessos à *rbtree*, conseguindo tempos muito próximos do algoritmo EM. Todavia, comparativamente com o algoritmo EM, o algoritmo EA-E-1 consegue minimizar consideravelmente a distância média externa $\overline{d}_{ext}$ (rever figura 4.12.b)), o que sugere desde já uma boa classificação de EA-E-1 sob ponto de vista da métrica $\mathcal{R}(a)$.

Com tempos acima de $0.5\mu s$ têm-se os algoritmos de Encaminhamento Acelerado nas variantes EA-$\mathcal{L}$ e EA-*all* para grafos DeBruijn, e EA-E-$\mathcal{L}$ e EA-*all* para grafos Chord. Neste contexto, começamos por comparar os dois últimos, relativos a grafos Chord.

Assim, com grafos Chord, EA-*all* consegue tempos até 50% menores que EA-E-$\mathcal{L}$ apesar de, em cada nó, EA-*all* ter de consultar todas as $\overline{\mathcal{H}}(n)$ tabelas de encaminhamento, enquanto que EA-E-$\mathcal{L}$ é obrigado a consultar "apenas" $\mathcal{L}$ tabelas; adicionalmente, a gama de variação

[44] O que sugere que as implementações usadas para *tries* e *rbtrees* têm desempenhos semelhantes.

dos tempos produzidos por EA-*all* é fixa (entre 0.5 e 0.75μs), ao passo que os tempos produzidos por EA-E-$\mathcal{L}$ tendem a crescer; considerando a figura 4.15.b), a explicação para estas observações é a seguinte: por um lado, a tendência de $\mathcal{L}$ é crescente, enquanto que $8 \leq \overline{\mathcal{H}}(n) \leq 16$; tal faz com que se tenha tendencialmente $\mathcal{L} > \mathcal{H}(n)$, o que implica maior número de consultas para EA-E-$\mathcal{L}$ e explica a tendência crescente dos seus tempos; por outro lado, por cada tabela a consultar, EA-E-$\mathcal{L}$ comporta maior esforço computacional[45].

A lógica anterior não é aplicável na comparação dos algoritmos EA-$\mathcal{L}$ e EA-*all* para grafos DeBruijn. Neste caso, o algoritmo EA-$\mathcal{L}$ consegue ser claramente mais rápido que EA-*all*. De facto, enquanto que, com grafos Chord, o algoritmo EA-E-$\mathcal{L}$ visita a *rbtree* exactamente $\mathcal{L}$ vezes, já com grafos DeBruijn, o algoritmo EA-$\mathcal{L}$ visita a *trie* no máximo $\mathcal{L} - 1$ vezes sendo que, em termos médios, esse número de visitas será inferior; o resultado é a obtenção de tempos na ordem dos 50%, face ao algoritmo EA-*all*; além disso, os tempos para o algoritmo EA-*all* tendem a crescer, o que se poderá dever a um aumento progressivo da profundidade das *tries* já que estas, ao contrário das *rbtrees*, podem não ser balanceadas.

Em termos gerais, pode-se concluir que os tempos conseguidos pelos algoritmos para grafos Chord são tendencialmente inferiores aos dos algoritmos para grafos DeBruijn. Este facto, aliado à observação prévia de distâncias externas também inferiores, permite já adivinhar uma supremacia efectiva da abordagem Chord face à abordagem DeBruijn, nas classificações pela métrica $\mathcal{R}(a)$, especialmente se o factor RAM não estiver em jogo.

4.9.5.2 Consumo de RAM por Nó

Como referido na secção 4.9.3, a métrica $\overline{RAM}_{\text{node}}$ (consumo de RAM em tabelas de encaminhamento, por nó) depende apenas do tipo de grafo, sendo independente do algoritmo de encaminhamento. Neste contexto, a figura 4.15.a) revela que, com grafos Chord, a métrica $\overline{RAM}_{\text{node}}$ é substancialmente maior (em $\approx 100\%$) do que com grafos DeBruijn; tal compreende-se pelo facto de, nos últimos, o número de entradas sucessoras por cada entrada ser sempre 2 (fixo, portanto), ao passo que, nos primeiros, esse número é $\mathcal{L} \geq 1$, valor que cresce uma unidade entre dois estágios sucessivos (como mostra a figura 4.15.b)).

Entre estágios, $\overline{RAM}_{\text{node}}$ tende a crescer, para ambos os grafos, embora de forma mais vincada com grafos Chord; para estes, o número crescente de sucessoras por entrada é a causa primária para o aumento de $\overline{RAM}_{\text{node}}$; de facto, em qualquer estágio, o número de nós das *rbtrees* evolui sempre de forma semelhante, decaindo de $\mathcal{H}_{max}(n)$ para $\mathcal{H}_{min}(n)$; logo, só o aumento da quantidade de informação preservada pelos nós das *rbtrees* explica o aumento do consumo de RAM. Já com grafos DeBruijn, o número de folhas/entradas das *tries* também decai de $\mathcal{H}_{max}(n)$ para $\mathcal{H}_{min}(n)$, mas o número de sucessoras de cada folha é sempre 2; verifica-se, todavia, um aumento (ligeiro) de $\overline{RAM}_{\text{node}}$, entre estágios, consistente com a hipótese anteriormente formulada de aumento da profundidade das *tries*.

[45] Resultante do cálculo de um predecessor de uma certa ordem e da necessidade de visitar mais nós da *rbtree* até ao nó desejado (rever a descrição informal do algoritmo fornecida na secção 4.6.4.4); de facto, o número total de nós visitados (incluindo repetições) por EA-*all* numa só travessia *depth-first*, tende a ser menor que o número de nós visitados por EA-E-$\mathcal{L}$ nas $\mathcal{L}$ descidas diferentes que tem de efectuar na *rbtree*.

4.9.6 Tempo Total por Cadeia

As figuras 4.16 e 4.17 apresentam a evolução de $\overline{TIME}_{\text{chain}}$ (tempo médio total por cadeia de localização, em μs), para grafos DeBruijn e Chord; para cada grafo, as sub-figuras a), b), c) e d) apresentam o efeito da variação de $\overline{NET}_{\text{hop}}$ na gama {1,10,100,1000} μs.

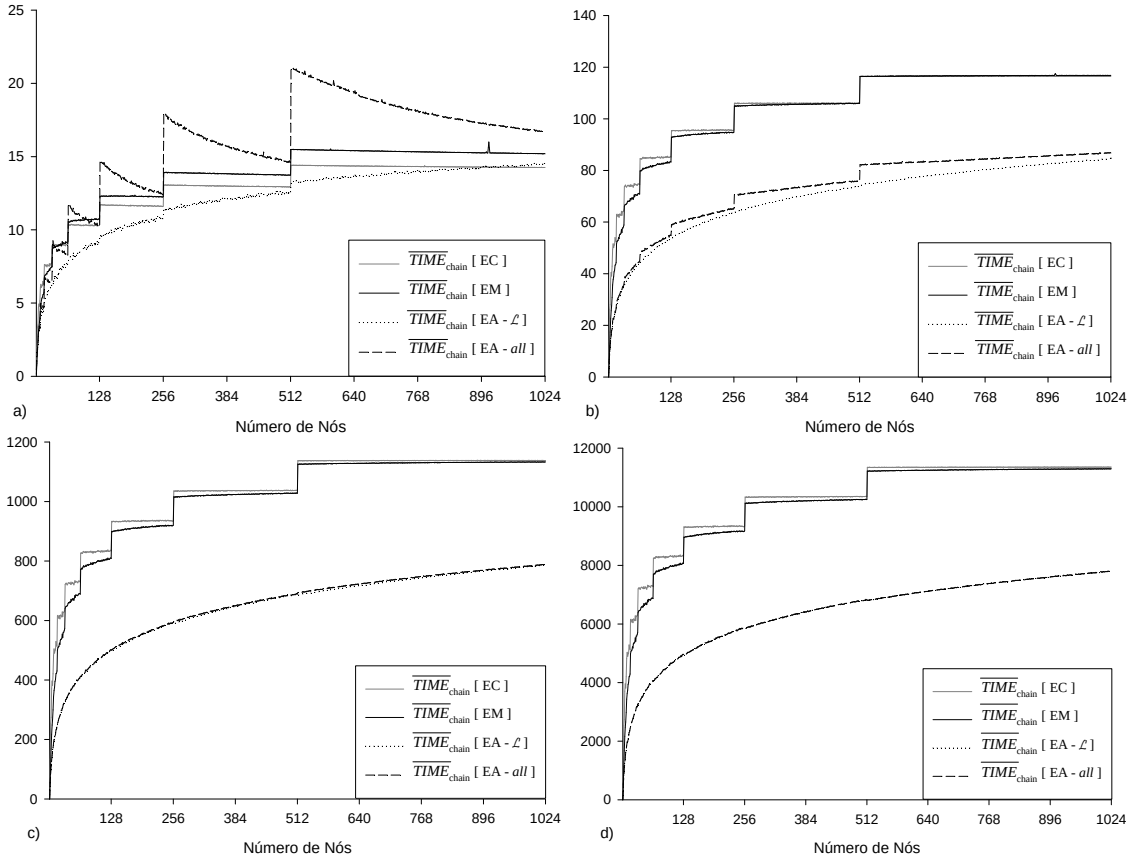


Figura 4.16: $\overline{TIME}_{\text{chain}}$ para grafos DeBruijn, com $\overline{NET}_{\text{hop}}$ = a) 1, b) 10, c) 100, d) 1000 (μs).

Em termos globais, é evidente a mais valia dos algoritmos de Encaminhamento Acelerado, menos vincada com $\overline{NET}_{\text{hop}} = 1\mu s$[46], mas que sobressai à medida que $\overline{NET}_{\text{hop}}$ aumenta. De facto, à medida que os tempos de comunicação sobem, torna-se mais importante minimizar o número de saltos, em detrimento do tempo de CPU necessário a essa minimização. Espera-se, assim, que algoritmos com valores similares da distância $\overline{d}_{\text{ext}}$, apresentem valores semelhantes de $\overline{TIME}_{\text{chain}}$, a partir de um certo valor de $\overline{NET}_{\text{hop}}$.

Por exemplo, para grafos DeBruijn, os algoritmos EA-$\mathcal{L}$ e EA-*all* exibem valores de $\overline{TIME}_{\text{chain}}$ virtualmente idênticos quando $\overline{NET}_{\text{hop}} \geq 100$, sendo que esses mesmos algoritmos exibem valores similares para as distâncias $\overline{d}_{\text{ext}}$ (figura 4.12.a)). Analogamente, para grafos Chord, o posicionamento relativo que EA-E-1, EA-E-$\mathcal{L}$ e EA-*all* apresentam para a métrica $\overline{d}_{\text{ext}}$ (figura 4.12.b)) é conservado para $\overline{TIME}_{\text{chain}}$ quando $\overline{NET}_{\text{hop}} \geq 100$.

[46]Sendo os algoritmos EA-*all* para grafos DeBruijn, e EA-E-$\mathcal{L}$ para grafos Chord, os mais demorados.

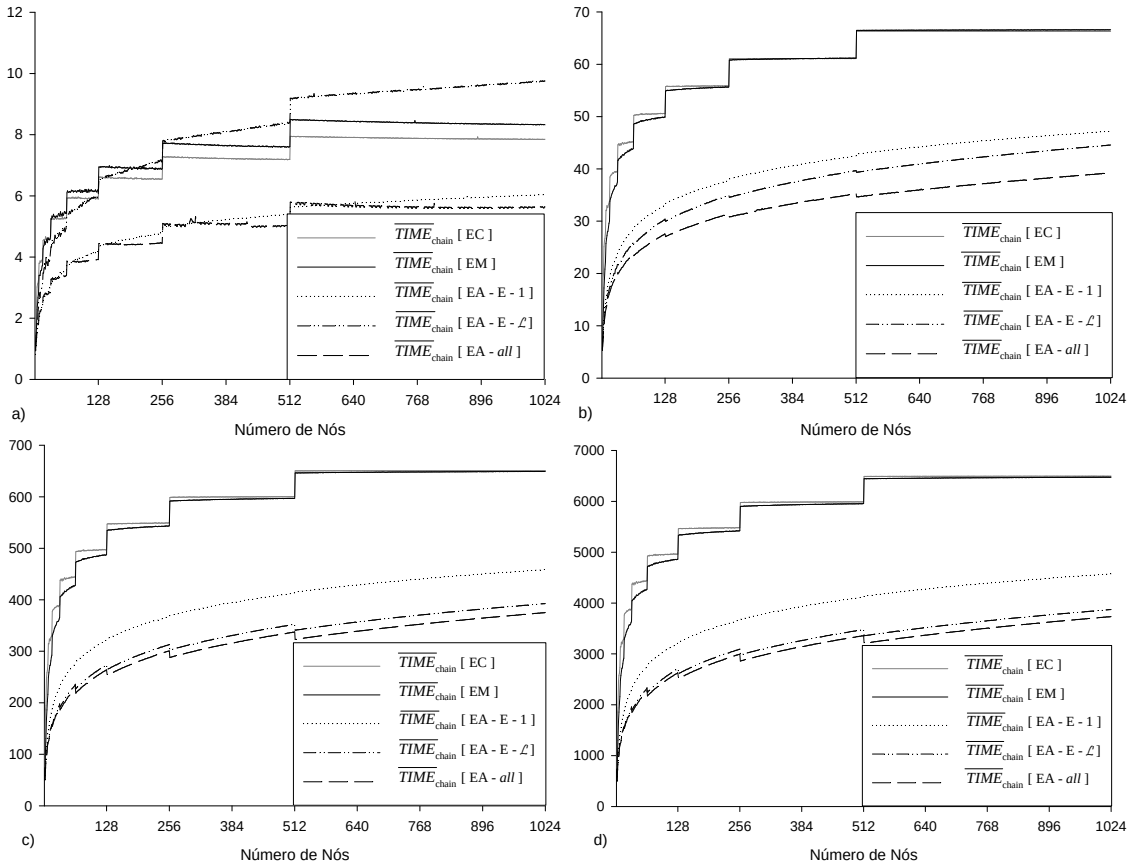


Figura 4.17: $\overline{TIME}_{\text{chain}}$ para grafos Chord, com $\overline{NET}_{\text{hop}}$ = a) 1, b) 10, c) 100, d) 1000 (μs).

Os resultados obtidos demonstram o interesse dos nossos algoritmos de Encaminhamento Acelerado para ambientes WAN (em que $\overline{NET}_{\text{hop}} \geq 1000$), o domínio usual das DHTs utilizadas em sistemas P2P. Mas, mais importante ainda, os mesmos resultados demonstram o mérito desses algoritmos em ambientes *cluster*, onde tipicamente $10 \leq \overline{NET}_{\text{hop}} \leq 100$.

A comparação dos resultados para os dois tipos de grafos evidencia a superioridade dos algoritmos de Encaminhamento Acelerado para grafos Chord, na minimização do tempo médio consumido em localização distribuída. Tomando como base de comparação os resultados conseguidos pelo algoritmo mais rápido para cada grafo, os tempos conseguidos com grafos Chord são da ordem dos 50% dos tempos conseguidos com grafos DeBruijn.

Desta forma, e a não ser que o factor RAM seja bastante relevante, a opção pela abordagem Chord é evidente. A possível influência do factor RAM é sistematizada na secção seguinte.

4.9.7 Classificação de Algoritmos

Na secção 4.9.3 definiu-se uma métrica sintética linear, $\mathcal{R}(a)$, que permite determinar o algoritmo mais competitivo, tendo em conta os factores i) desempenho da localização distribuída (veiculado pela métrica $\overline{TIME}_{\text{chain}}$) e ii) consumo de RAM (dado pela métrica

$\overline{RAM}_{\text{node}}$), afectados de pesos complementares, ω_{TIME} e ω_{RAM}, com $\omega_{TIME}+\omega_{RAM} = 1$. Na figura 4.18, apresentam-se as classificações resultantes da exercitação da métrica $\mathcal{R}(a)$, para um conjunto seleccionado de algoritmos e várias combinações de ω_{TIME} e ω_{RAM}.

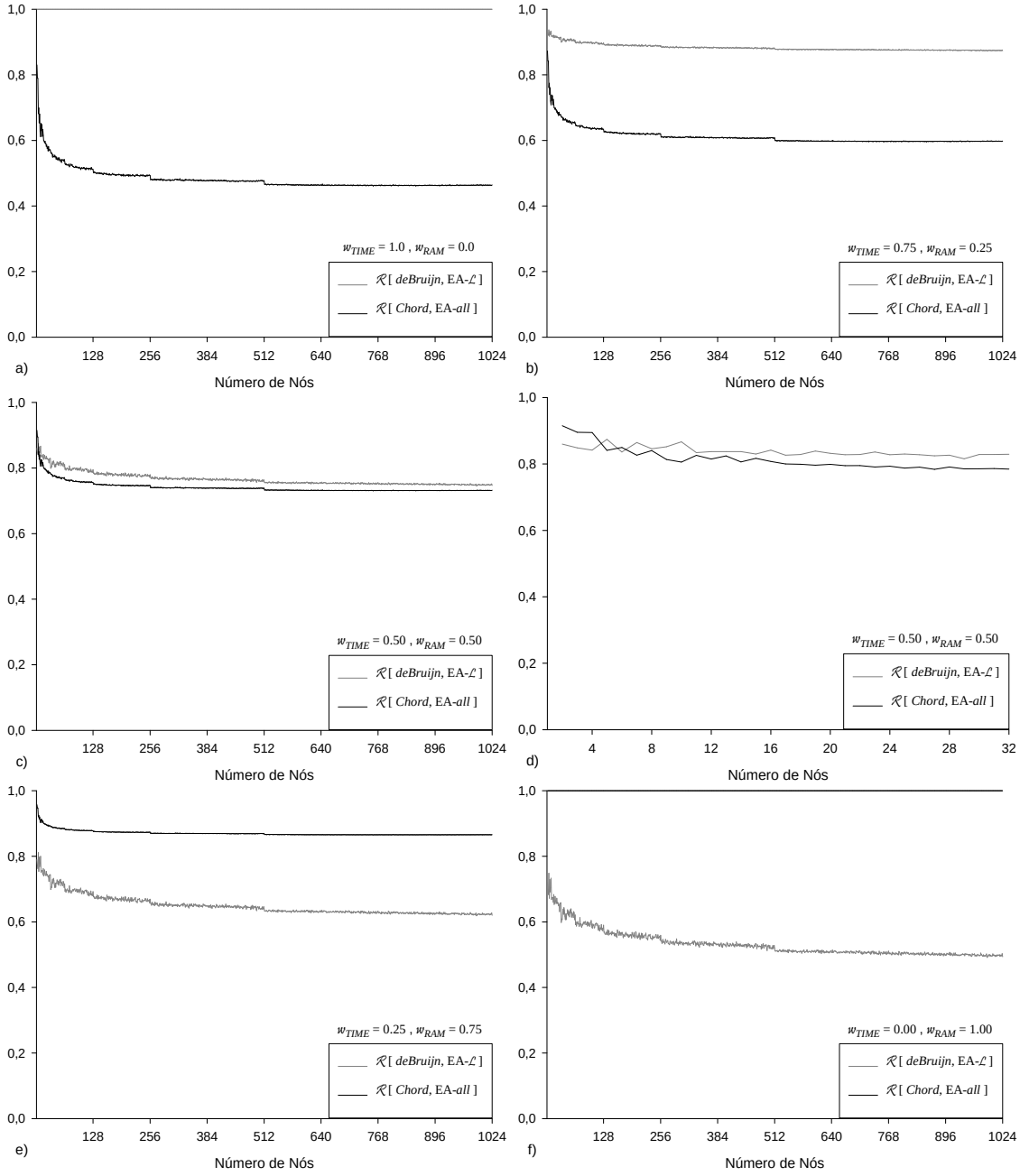


Figura 4.18: Classificação segundo a métrica $\mathcal{R}$, para os algoritmos mais rápidos.

Assim, consideraram-se apenas os algoritmos que, para cada grafo, minimizam sistematicamente $\overline{TIME}_{\text{chain}}$, independentemente do valor de $\overline{NET}_{\text{hop}}$, ou seja, o algoritmo EA-$\mathcal{L}$ para grafos DeBruijn (ver figura 4.16) e EA-*all* para grafos Chord (ver figura 4.17). A fim de reduzir o número de cenários a classificar, considerou-se apenas $\overline{NET}_{\text{hop}} = 10\mu s$.

Adicionalmente, as combinações de ω_{TIME} e ω_{RAM} seleccionadas incluem i) as duas situações extremas, em que só o factor Tempo interessa ($\omega_{TIME} = 1.0$ e $\omega_{RAM} = 0.0$) ou só o factor RAM interessa ($\omega_{TIME} = 0.0$ e $\omega_{RAM} = 1.0$), ii) e três combinações intermédias, que correspondem a flutuações de 25% em cada um dos factores. Desta forma, é possível observar a influência do aumento/diminuição progressiva do peso de cada um dos factores.

Ora, como se pode observar, só com $\omega_{RAM} > 0.5$ é que o algoritmo EA-$\mathcal{L}$ para grafos DeBruijn consegue uma classificação melhor (*ranking* inferior) que o algoritmo EA-*all* para grafos Chord (embora a figura 4.18.d), que amplia a 4.18.c) para $1 \leq \mathcal{N} \leq 32$, revele que com $\omega_{RAM} = 0.5$ e um número muito reduzido de nós, EA-$\mathcal{L}$ já é competitivo).

Dito de outra forma, só atribuindo à economia de RAM uma grande importância, é que a abordagem DeBruijn se tornaria competitiva. Todavia, a redução em 50% no consumo de RAM que resultaria da adopção da abordagem DeBruijn, e que é observável na figura 4.15.a), tem pouco significado em termos absolutos: de facto, valores máximos de ≈ 1200 bytes para grafos Chord, e ≈ 600 bytes para grafos DeBruijn, são residuais, comparados com a capacidade de RAM por nó que é habitual hoje em dia (pelo menos 512 Mbytes). A única forma de a abordagem DeBruijn ganhar competitividade será pois pela optimização agressiva dos seus algoritmos de encaminhamento, de forma a que a diminuição do tempo de CPU por salto, compense o maior número de saltos que são necessários por cadeia.

A conclusão final da nossa avaliação aponta assim para a supremacia dos grafos Chord completos face aos DeBruijn binários, como vinha sendo sugerido pelos resultados anteriores. Os algoritmos de Encaminhamento Acelerado que desenvolvemos para grafos Chord, suportados pela implementação particular de Red-Black-Trees que escolhemos, representam pois um mecanismo de aceleração da localização distribuída leve e eficaz, capaz de rentabilizar a presença de múltiplas tabelas de encaminhamento em cada nó de uma DHT.

4.10 Relação com Outras Abordagens

Em abordagens baseadas no paradigma do Hashing Consistente [KLL+97], como é o caso do Chord [SMK+01], cada nó de uma DHT gere uma partição contínua, de *hashes* estáticos, derivada a partir do identificador desse nó e constrangida por ele[47]. Noutras abordagens, como a P-Grid [Abe01, AHPS02], não existe um vínculo fixo entre os identificadores dos nós e os seus *hashes*, o que é explorado com proveito para efeitos de balanceamento de carga [ADH03, ADH05]. No nosso caso, a atribuição de partições de H descontínuas, a cada nó (computacional ou virtual) de uma DHT, tem também implícita uma dissociação entre os identificadores dos nós e os *hashes* que compõem as partições; porém, enquanto que na abordagem P-Grid cada nó é responsável por um único *hash* (com um número de bits dinâmico e variável entre nós), já no nosso caso cada nó é responsável por vários *hashes* (com um número de bits dinâmico mas igual entre nós); adicionalmente, na abordagem P-Grid, i) o *hash* atribuído a cada nó é a sequência de bits correspondente a uma folha de uma *trie* binária distribuída e ii) o nó hospedeiro de cada *hash* conhece um número logarítmico de hospedeiros de outros *hashes*, a distâncias logarítmicas; estas características

[47] Ou seja, a partição pode aumentar/diminuir de amplitude, mas sempre na vizinhança do *hash* do nó.

permitem estabelecer algum paralelismo com a visão em *trie* da evolução das nossas DHTs, e com a existência de um grafo que interliga as folhas dessa *trie* virtual (rever secção 4.7.2).

A abordagem Kademlia [MM02] partilha semelhanças com a P-Grid, no sentido de que também assenta numa *trie* binária distribuída, em que cada folha é identificada pelo menor número possível de bits, e a complexidade espacial e temporal da localização é logarítmica no número de nós da DHT; todavia, ao contrário da P-Grid, a folha da *trie* correspondente a um nó é derivada do *hash* do identificador do nó[48]; depois, é usada uma métrica XOR de distância (não euclidiana), de que basicamente resulta a associação de *hashes* dispersos (não-contínuos) a cada nó da DHT (tal como acontece na nossa abordagem, embora explorando a aleatoriedade); precisamente, a mais valia do Kademlia é conseguir que a localização distribuída continue a ser feita com base num grafo no domínio dos nós, sem que isso exija continuidade das partições de cada nó, como acontece com o Chord e derivados; no entanto, continua a existir uma associação rígida entre *hashes* e nós, o que se pode revelar pouco flexível para uma distribuição de carga de armazenamento não-uniforme.

A abordagem Kademlia preconiza também o aumento da dimensão das tabelas de encaminhamento como forma de *aceleração da localização*, conceito que, no nosso caso, está associado à exploração de várias tabelas de encaminhamento em simultâneo. Essa possibilidade, sob a designação de *encaminhamento por atalhos* (*shortcut routing*), é explorada pela primeira vez no sistema CFS [DKKM01, Dab01], um sistema de ficheiros distribuído orientado ao bloco (e apenas de leitura) para ambientes WAN, baseado no paradigma DHT oferecido pelo Chord, onde se admitem vários nós virtuais por cada nó da DHT, cada qual com sua tabela de encaminhamento; as poucas descrições da abordagem CFS [DKKM01, Dab01] são, todavia, omissas nos detalhes da técnica de aceleração utilizada.

Precisamente, apontando o CFS como exemplo, Godfrey et al. [GLS+04] referem que os atalhos de encaminhamento permitidos pelas tabelas dos nós virtuais poderão garantir localização em $O(log\mathcal{N})$ passos[49], o que está de acordo com os resultados obtidos pelos nossos algoritmos de Encaminhamento Acelerado; com efeito, embora não tenhamos usado nós virtuais nas nossas simulações, o facto é que cada nó computacional dispunha de várias tabelas de encaminhamento, correspondentes a *hashes* dispersos em H sendo que a utilização de nós virtuais no Chord assegura também uma cobertura topológica dispersa.

Posteriormente, Godfrey [GS05] salienta a sobrecarga espacial e temporal (na localização distribuída), derivada da manutenção de múltiplos nós virtuais por nó computacional, e propõe o protocolo Y_0, derivado do Chord, baseado na agregação dos nós virtuais de um mesmo nó, numa zona contígua de $[0, 1)$; dessa forma, o grafo de localização pode ser de novo construído no domínio dos nós computacionais, em vez de o ser no domínio dos nós virtuais; essa vantagem é significativa num quadro de operação com um número muito elevado de nós (computacionais e virtuais), o que não corresponde ao nosso cenário alvo. Godfrey observa ainda um efeito acelerador da heterogeneidade (no número de nós virtuais por nó computacional), no esforço de localização face a cenários homogéneos, o que não podemos comprovar dada a orientação dos nossos testes apenas a cenários deste tipo.

Nos grafos que seleccionamos para localização distribuída, exploramos apenas a variante

[48] Sendo os *hashes* limitados a 160 bits, enquanto que no P-Grid esse número é virtualmente ilimitado.
[49] Se entre m nós virtuais sucessivos (em $[0, 1)$) de um mesmo nó computacional, existirem $\mathcal{N}$ nós virtuais.

unidireccional, de forma a garantir tabelas de encaminhamento pequenas. Como constatamos na secção 4.9.5, os recursos de armazenamento (RAM) necessários para as tabelas, em cada nó da DHT, são mínimos, sendo perfeitamente viáveis tabelas com o dobro do tamanho. Uma hipótese interessante seria pois a exploração de variantes bidireccionais dos grafos, capazes de acelerar mais o processo de localização. Ganesan et al. [GM04], por exemplo, desenvolveram algoritmos óptimos, que asseguram distância média $\mathcal{L}/3$ em vez de $\mathcal{L}/2$, para grafos Chord bidireccionais em que o número de vértices é potência de 2, como no nosso caso, em que usamos grafos completos em H; quando o número original de vértices não é potência de 2, Ganesan et al. propõem a formação de grupos de vértices (com um número semelhante de vértices por grupo), de forma a que o número de grupos seja potência de 2, sendo depois construída uma topologia Chord sobre estes grupos; Ganesan et al. efectuam ainda a generalização de grafos Chord na base 2 a grafos Chord na base k (em que o número de vértices é potência de k), com distâncias médias menores.

Para grafos DeBruijn, a possibilidade de trabalhar numa base $k \neq 2$ está desde logo prevista na formulação da secção 4.5.1, bastando recorrer a um alfabeto A de $k = \#A$ símbolos. No nosso caso, uma hipótese a explorar seria, por exemplo, utilizar grafos DeBruijn k-ários com $k = \mathcal{L} = log_2\mathcal{H}$; nesse cenário, as tabelas de encaminhamento teriam $\mathcal{L}$ entradas, como no Chord, mas as distâncias médias seriam menores, de ordem $O((log\mathcal{H})/loglog\mathcal{H})$, o que é óptimo; a principal dificuldade a enfrentar seria a necessidade de criar e lidar com vértices fictícios (em adição aos $\mathcal{H}$ hashes), de forma a garantir a navegabilidade no grafo com base no algoritmo (convencional) de "erosão de símbolos", o que só seria possível com o grafo completo. Neste contexto, as técnicas utilizadas na abordagem Koorde [KK03] ou por Riley et al. [RS04] poderiam ser exploradas com proveito.

4.11 Epílogo

Na arquitectura Domus de co-operação de DHTs, introduzida no capítulo 5, prevêem-se diversos atributos das DHTs (no quadro do suporte à sua *heterogeneidade*) que lhes permitem exibir propriedades específicas, adequadas a diferentes requisitos aplicacionais. Assumindo a utilização de grafos Chord completos em H, o *algoritmo de localização*, derivado das contribuições deste capítulo, surge como um atributo configurável das DHTs.

Veremos ainda que, para além da *localização distribuída*, é possível recorrer a outros *métodos de localização* (como sejam a *localização directa*/1-HOP ou o recurso a *caches de localização*) seja de forma *isolada*, seja de forma *combinada*, o que permite definir diferentes *estratégias de localização*. Posteriormente, no capítulo 7, teremos oportunidade de observar o resultado da exercitação em ambiente real (não simulado) de várias *estratégias de localização*, no quadro da avaliação de um protótipo da arquitectura Domus.

Por fim, espera-se que a utilidade dos nossos algoritmos de *encaminhamento acelerado* não se esgote no nosso contexto aplicacional particular. Genericamente, a utilidade dos algoritmos pode ser posta à prova noutros cenários (*e.g.*, DHTs para ambientes P2P), sempre que haja informação topológica de vizinhança disponível para alimentar esses algoritmos.

Capítulo 5

Operação Conjunta de DHTs

Resumo

Neste capítulo introduz-se a arquitectura Domus de suporte à *co-operação* (operação conjunta e integrada) de múltiplas DHTs independentes, vistas como um serviço distribuído do *cluster*, orientado ao armazenamento de grandes dicionários. São apresentadas as entidades e relações da arquitectura, incluindo i) a análise em detalhe dos seus componentes e subsistemas e ii) a definição dos atributos e invariantes que regulam as suas relações.

5.1 Sinopse

A arquitectura *Domus* articula os conceitos e os mecanismos adequados ao suporte, de forma integrada, à 1) realização, 2) exploração e 3) gestão de *múltiplas* DHTs independentes, num *cluster* de nós possivelmente *heterogéneos* e *partilhados* por várias tarefas, que executam concorrentemente no *cluster*, impondo-lhe requisitos dinâmicos e imprevisíveis.

O suporte à *multiplicidade* de DHTs é enriquecido pelo suporte à sua *heterogeneidade*, assente na possibilidade de afinar, para cada DHT, um conjunto básico de atributos; dessa afinação resultam diferentes propriedades, que dão resposta a requisitos aplicacionais específicos; por exemplo, a simples possibilidade de definir Disco ou RAM como meio de armazenamento é relevante sob o ponto de vista de requisitos de persistência e fiabilidade.

A arquitectura prevê também mecanismos de ajuste/balanceamento dinâmico de carga que procuram, por um lado, a) rentabilizar a utilização dos recursos do *cluster* e, por outro, b) assegurar níveis esperados de serviço para cada DHT. Assim, as funcionalidades de *endereçamento* e de *armazenamento* dos registos de uma DHT são dissociadas, uma vez que impõem, por princípio, requisitos de natureza diferente aos nós computacionais. Essa dissociação permite que os nós do *cluster* executem aquelas funcionalidades de forma diferenciada, possibilidade que pode ser explorada para *balanceamento dinâmico* das DHTs.

Outras facilidades da arquitectura incluem a possibilidade de se i) contrair/expandir a

utilização do *cluster* e ii) congelar/reactivar uma instância (incluindo, neste caso, a possibilidade de o fazer para DHTs específicas); em particular, esta última funcionalidade permite que a execução no *cluster* de certas aplicações paralelas/distribuídas seja intermitente, ou que recursos limitados, como RAM, possam ser comutados entre DHTs. O acesso de aplicações clientes às DHTs efectua-se por intermédio de uma biblioteca que disponibiliza funcionalidades de carácter administrativo e operações usuais sobre dicionários.

Por fim, mas não menos importante, a arquitectura assenta em contribuições dos capítulos 3 e 4 (modelo M4 de distribuição heterogénea e modelos de localização distribuída, respectivamente), devidamente adaptadas para suportar a dissociação entre endereçamento e armazenamento. Essa relação com trabalho anterior é estabelecida, sempre que oportuno.

5.2 Entidades e Relações

A arquitectura Domus baseia-se num conjunto relativamente restrito de entidades: a) *clientes/aplicações Domus* (ou simplesmente *aplicações*), b) *DHTs*, c) *serviços Domus* (ou simplesmente *serviços*), d) *serviços básicos* e e) *nós computacionais* (ou simplesmente *nós*).

Aplicações e serviços são *entidades activas de software*, que executam em nós do *cluster*; DHTs são *estruturas de dados (tabelas de hash) distribuídas*, realizadas pela colaboração de serviços Domus; os serviços básicos (assumidos sempre presentes) asseguram funcionalidades de a) passagem de mensagens, b) monitorização de recursos e c) execução remota.

Uma instanciação da arquitectura Domus é designada por "cluster Domus". Um cluster Domus comporta a execução de pelo menos um serviço Domus designado de "supervisor"; todavia, essa configuração tem pouco interesse, dado que o objectivo primordial da arquitectura é realizar DHTs e isso requer a cooperação de um ou mais serviços Domus "regulares". A figura 5.1 representa um cluster Domus típico que, não sendo genérico, é suficientemente representativo das relações essenciais entre as entidades da arquitectura.

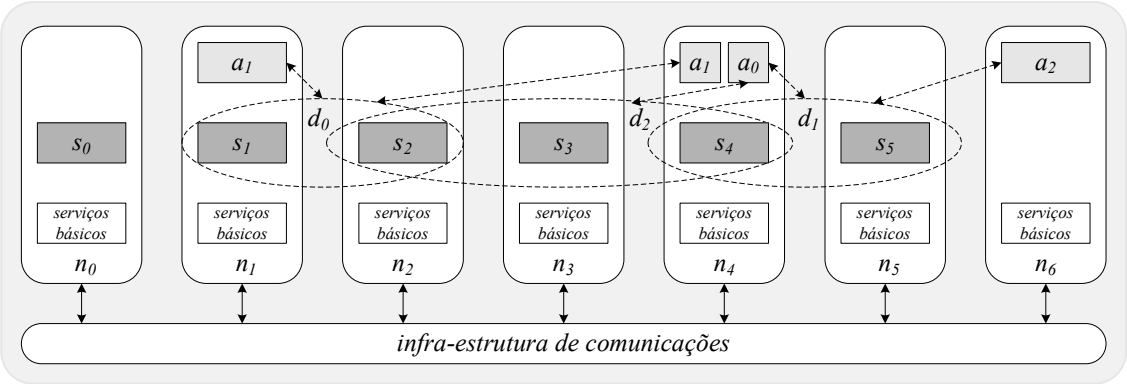


Figura 5.1: Representação de um cluster Domus típico.

Na figura 5.1, as várias entidades da arquitectura representam-se com base na seguinte notação: a) a_i denota uma aplicação Domus, b) d_j denota uma DHT, c) s_k denota um

serviço Domus e d) n_k denota um nó[1]. Adicionalmente, convencionamos representar por A o conjunto de todas as aplicações, por D o conjunto de todas as DHTs, por S o conjunto de todos os serviços e por N o conjunto de todos os nós do *cluster*[2]. Assim, para o cenário ilustrado: $A = \{a_0, a_1, a_2\}$, $D = \{d_0, d_1, d_2\}$, $S = \{s_0, s_1, ..., s_5\}$ e $N = \{n_0, n_1, ..., n_6\}$.

Na figura não são representadas, por simplicidade, aplicações e serviços exógenos à arquitectura Domus, que podem partilhar os nós do *cluster* com aplicações e serviços Domus.

5.2.1 Relações

O relacionamento entre as entidades da arquitectura segue regras relativamente simples:

$\mathcal{R}_1$: "uma aplicação Domus pode aceder, em simultâneo, a várias DHTs";

Exemplo: na figura 5.1, a_0 acede a d_2 e d_1 (os acessos representados pelas setas tracejadas são virtuais já que, na realidade, as aplicações interactuarão com os serviços que suportam as DHTs), ao passo que a_1 acede apenas a d_0, e a_2 acede apenas a d_1; na mesma figura é possível constatar que a_1 é uma aplicação cliente *paralela/distribuída*, operando a partir dos nós n_1 e n_4 (sob o ponto de vista da arquitectura, o facto de um cliente ser *centralizado* ou *distribuído* é transparente);

$\mathcal{R}_2$: "uma DHT Domus pode ser acedida, em simultâneo, por várias aplicações Domus";

Exemplo: na figura 5.1, a DHT d_1 é acedida em simultâneo pelas aplicações a_0 e a_2, ao passo que d_0 é acedida apenas pela aplicação a_1 (embora por dois processos diferentes, pois a_1 é de tipo distribuído) e d_2 é acedida apenas pela aplicação a_0;

$\mathcal{R}_3$: "uma DHT é suportada por um ou mais serviços Domus";

Exemplo: na figura 5.1, a DHT d_0 é suportada pelos serviços s_1 e s_2, a DHT d_2 é suportada pelos serviços s_2, s_3 e s_4, e a DHT d_1 é suportada pelos serviços s_4 e s_5;

$\mathcal{R}_4$: "um serviço Domus pode participar na realização de zero, uma ou mais DHTs";

Exemplo: na figura 5.1, s_0 não suporta (ainda) nenhuma DHT, os serviços s_1, s_3 e s_5 suportam todos uma só DHT e os serviços s_2 e s_4 suportam ambos duas DHTs;

$\mathcal{R}_5$: "aplicações e serviços Domus podem partilhar os nós do *cluster*";

Exemplo: na figura 5.1, os nós n_1 e n_4 suportam todos um serviço Domus e pelo menos uma aplicação cliente; alguns nós podem não executar nem aplicações nem serviços Domus (cenário não ilustrado na figura) mas a presença de serviços básicos é suficiente para permitir a sua execução futura; além disso, como já referimos, é admissível a partilha dos nós por aplicações e/ou serviços de outra natureza;

[1] O facto de s_k e n_k deitarem mão do mesmo índice k é intencional, denotando uma associação biunívoca.

[2] Note-se que N ganha, a partir deste capítulo, um significado diferente; até agora, N denotava o conjuntos de todos os nós que suportavam uma certa DHT; a partir de agora, N diz respeito ao universo dos nós do *cluster*, sendo necessária notação adicional para denotar os nós específicos de uma certa DHT.

$\mathcal{R}_6$: "por cada nó computacional existe, no máximo, uma instância de um serviço Domus", suficiente para gerir a participação do nó na realização de várias DHTs;

Observação: a notação da figura 5.1 reflecte a correspondência biunívoca entre serviços e nós computacionais: n_0 hospeda s_0, ..., n_k hospeda (eventualmente[3]) s_k; essa correspondência permite referenciá-los, em certos contextos, de forma indistinta.

5.3 Nós Computacionais

A arquitectura Domus comporta uma certa visão dos nós do *cluster*, distribuídos por diferentes categorias. Nesta secção, essa visão é formalizada, através de notação apropriada.

5.3.1 Decomposição Funcional

No âmbito da arquitectura *Domus* (e como é desde logo sugerido pela figura 5.1), os nós do *cluster* suportam, em combinações diversas, aplicações Domus, DHTs, serviços Domus e serviços básicos. Para lá do cenário representado pela figura 5.1, é possível caracterizar a diversidade funcional dos nós do *cluster* através de uma representação mais formal, como a fornecida pela figura 5.2, que sintetiza todas as combinações possíveis de funcionalidade.

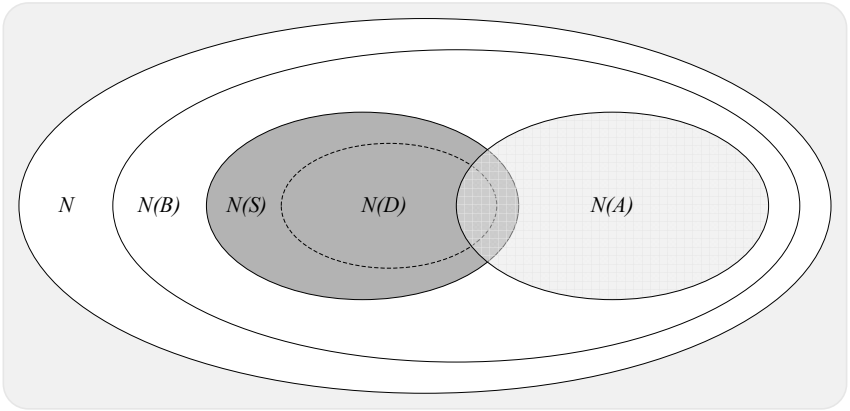


Figura 5.2: Decomposição funcional do conjunto dos nós do *cluster*.

A seguinte notação sustenta a representação formal da decomposição funcional do *cluster*.

Notação: sendo N o conjunto global (indiferenciado) de todos os nós do *cluster*, tem-se

- $N(B)$: subconjunto de N que executa serviços básicos;
- $N(A)$: subconjunto de $N(B)$ que executa aplicações Domus;
- $N(S)$: subconjunto de $N(B)$ que executa serviços Domus;

[3]Por exemplo, n_6 não hospeda (ainda) nenhum serviço Domus.

- $N(D)$: subconjunto de $N(S)$ que participa na realização de DHTs.

Exemplo: para o cenário concreto da figura 5.1, tem-se $N = \{n_0, n_1, ..., n_6\}$, $N(B) = \{n_0, n_1, ..., n_6\}$, $N(S) = \{n_0, n_1, ..., n_5\}$, $N(D) = \{n_1, n_2, ..., n_5\}$ e $N(A) = \{n_1, n_4, n_6\}$.

Notação: realizando uma caracterização mais fina dos diversos subconjuntos de N, tem-se

- $N(a)$: subconjunto de $N(A)$ que executa a aplicação a[4];
- $n(a)$: um elemento (nó computacional) de $N(a)$;
- $N(d)$: subconjunto de $N(D)$ que suporta a DHT d;
- $n(d)$: um elemento (nó computacional) de $N(d)$;
- $N(D) = \bigcup_{d \in D} N(d)$, com base nas definições anteriores.

Exemplo: na figura 5.1, tem-se $N(a_1) = \{n_1, n_4\}$, $N(a_0) = \{n_4\}$, $N(a_2) = \{n_6\}$, assim como $N(d_0) = \{n_1, n_2\}$, $N(d_2) = \{n_2, n_3, n_4\}$, $N(d_1) = \{n_4, n_5\}$ e $N(D) = \{n_1, n_2, ..., n_5\}$.

5.3.2 Dinamismo Funcional

A arquitectura Domus suporta um número dinâmico de aplicações, DHTs e serviços. Como tal, o papel que cada nó desempenha num cluster Domus pode mudar. A expansão ou contracção dos conjuntos $N(A)$, $N(S)$, $N(D)$ e $N(B)$ está apenas condicionada às fronteiras representadas na figura 5.2, ou seja: $N(B) \subseteq N$, $[N(S) \cup N(A)] \subseteq N(B)$ e $N(D) \subseteq N(S)$.

A modificação de $N(A)$ depende de mecanismos externos à arquitectura (*e.g.*, mecanismos automáticos embutidos nas aplicações, ou baseados no livre arbítrio dos utilizadores).

Analogamente, a evolução da composição de $N(B)$ é regulada por mecanismos específicos da arquitectura dos serviços básicos, cujos detalhes caiem fora do âmbito desta discussão.

Para o conjunto global (indiferenciado) de todos os nós do *cluster*, N, assume-se que é a entidade administradora do *cluster* a responsável pela gestão da sua composição.

Relativamente a $N(S)$ e $N(D)$, a sua expansão/contracção pode ser efectuada quer por razões exógenas de carácter administrativo, quer em resposta a mecanismos endógenos de balanceamento, sendo ambos os casos suportados de forma adequada pela arquitectura.

5.4 Serviços Básicos

Aplicações e serviços Domus assumem a disponibilidade de uma infra-estrutura de base, designada precisamente de *serviços básicos*, que providenciam: a) *passagem de mensagens* (de alto desempenho, idealmente), b) *monitorização de recursos* e c) *execução remota*.

[4]Sendo a convencional/centralizada, então $\#N(a) = 1$; sendo paralela/distribuída, então $\#N(a) \geq 2$.

Um mecanismo de passagem de mensagens de elevado desempenho é essencial para que aplicações e serviços Domus possam comunicar de forma expedita. Para maximizar a exploração do paralelismo potencial dos nós, as aplicações e os serviços Domus poderão ser baseados em múltiplos *fios-de-execução*, sendo portanto desejável um mecanismo de comunicação que suporte a troca de mensagens entre *fios-de-execução* em nós diferentes.

A catalogação e monitorização de recursos é também necessária tendo em vista, por um lado, a geração de distribuições iniciais balanceadas para as DHTs e, por outro, a manutenção dinâmica desse balanceamento, seja em resultado de mudanças no nível de utilização dos recursos, seja em resultado de mudanças no número de nós participantes na DHT. Estas questões serão tratadas com a profundidade devida ao longo do próximo capítulo.

Finalmente, a disponibilidade de execução remota permitirá não só efectuar o arranque de serviços Domus num conjunto inicial de nós, como ainda expandi-lo, caso seja necessário.

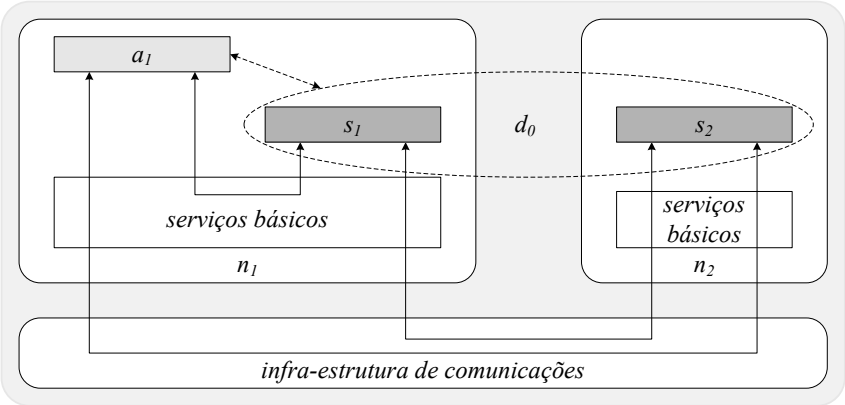


Figura 5.3: Serviços básicos como intermediários entre aplicações e serviços Domus.

A figura 5.3 amplia parte da figura 5.1, revelando que as interacções (setas bidireccionais contínuas) entre aplicações e serviços Domus são mediadas pelos serviços básicos. Na figura é também clara a distinção entre o acesso (virtual) de uma aplicação a uma DHT (seta bidireccional tracejada) e a interacção (real) da aplicação com os serviços que a suportam.

5.5 Dissociação Endereçamento – Armazenamento

Até agora assumimos, implicitamente, que a atribuição de uma certa entrada de uma DHT a um determinado nó computacional[5] é feita no pressuposto de que o nó em causa será o responsável pelo *armazenamento* dos registos associados à entrada[6]; adicionalmente, assumimos que, no âmbito da utilização de mecanismos de localização distribuída, o mesmo nó alojará as *tabelas de encaminhamento* de todas as entradas que lhe foram atribuídas.

Em tal contexto, cada nó de uma DHT assume simultaneamente a *gestão do armazenamento* e a *gestão do endereçamento* das suas entradas; equivalentemente, pode-se dizer

[5]Definição da correspondência *entrada* $\mapsto$ *nó* ou, equivalentemente, *posicionamento* da *entrada* no *nó*.
[6]Associação efectuada por via da aplicação da função de hash da DHT, às chaves dos registos.

também que cada nó realiza as *funções de armazenamento* e as *funções de endereçamento* das entradas que lhe foram atribuídas, ou seja, verifica-se uma *união/associação funcional*, na mesma entidade, entre as funções de endereçamento e de armazenamento das entradas.

Numa linha diferente, a arquitectura Domus suporta a *dissociação funcional* de endereçamento e armazenamento: se um certo (nó) serviço assume a gestão de endereçamento de uma entrada, poderá ser esse ou outro (nó) serviço a realizar a correspondente gestão de armazenamento. Neste contexto, designamos por *(nó) serviço de endereçamento* de uma entrada o (nó) serviço que efectua a sua gestão de endereçamento, e por *(nó) serviço de armazenamento* de uma entrada o (nó) serviço responsável pela gestão de armazenamento.

A dissociação funcional entre endereçamento e armazenamento permitirá que os serviços Domus executem essas funcionalidades em combinações diversas e em gradações compatíveis com as características e recursos disponíveis dos nós hospedeiros. Por exemplo, num nó as condições podem ser propícias à execução de funções de armazenamento (tipicamente *I/O bound* e dependentes de memória secundária), ao passo que podem ser, noutro nó, adequadas ao desempenho de funções de endereçamento (execução de algoritmos de encaminhamento, tipicamente *CPU bound* e sensíveis à disponibilidade de memória primária).

5.5.1 Refinamento do Conceito de Nó Virtual

A arquitectura Domus adopta o modelo M4[7] de Distribuição Heterogénea com Hashing Dinâmico do capítulo 3. Nesse modelo, um *nó virtual* equivale a um *número* de entradas[8]: 1) cada nó computacional n que participa numa DHT tem associado um certo número de nós virtuais, $\mathcal{V}(n)$; 2) por cada nó virtual v, um nó computacional n terá direito a um número de entradas, variável entre os limites $\mathcal{H}_{min}(v)$ e $\mathcal{H}_{max}(v)$; 3) no cômputo global, $\mathcal{V}(n)$ nós virtuais traduzir-se-ão em $\mathcal{H}(n)$ entradas atribuídas a n. Dado que, até agora, o endereçamento e armazenamento de uma entrada se assumiram acoplados, então n seria responsável pela gestão do endereçamento e do armazenamento das suas $\mathcal{H}(n)$ entradas.

A dissociação endereçamento-armazenamento das entradas tem implicações sobre o conceito de *nó virtual*. Em particular, implica a existência de nós virtuais de duas espécies, a) *nós virtuais de endereçamento* e b) *nós virtuais de armazenamento*, conotados com o balanceamento de *grão-grosso* (e independente) dessas funções. Neste contexto, define-se[9]:

- $\mathcal{V}^{\mathsf{e}}(n)$: número total[10] de nós virtuais de endereçamento, atribuídos a n;
- $\mathcal{V}^{\mathsf{e}}(d)$: número total de nós virtuais de endereçamento, da DHT d;
- $\mathcal{V}^{\mathsf{e}}(d, n)$: número total de nós virtuais de endereçamento da DHT d, atribuídos a n;
- $\mathcal{V}^{\mathsf{a}}(n)$, $\mathcal{V}^{\mathsf{a}}(d)$, $\mathcal{V}^{\mathsf{a}}(d, n)$: notação equivalente à anterior, mas para o armazenamento.

[7] A arquitectura também é compatível com o modelo M2 de distribuição homogénea dado que, com um número comum de nós virtuais para todos os nós/serviços da DHT (sendo suficiente, para tal, a definição $\mathcal{V}(n) = 1\ \forall n \in N$ (ou seja, um nó virtual, por cada nó)), o modelo M4 transfigura-se no modelo M2.

[8] Ao passo que na variante M4' do modelo M4 um nó virtual é visto como um *conjunto* de entradas, em que a *identidade* das entradas, para além do seu número, passam a ser relevantes (rever secção 3.7.1).

[9] Esta notação é também aplicável a serviços Domus, bastando substituir, onde for oportuno, n por s.

[10] Ver a secção 5.6.3 para uma definição formal de $\mathcal{V}^{\mathsf{e}}(n)$.

Durante a participação de um mesmo nó n numa mesma DHT d, o número adequado de nós virtuais das duas espécies, no quadro dessa participação – $\mathcal{V}^{\mathsf{e}}(d,n)$ e $\mathcal{V}^{\mathsf{a}}(d,n)$ –, poderá variar, de forma independente. Em consequência, o número total de nós virtuais das duas espécies, para a DHT d – $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ –, evoluirá também de forma independente. Todavia, essa evolução está condicionada por certos invariantes, c.f. se descreve de seguida.

A expressão 3.20 para o número global de entradas de uma DHT sob o modelo M4 é:

$$\mathcal{H} = 2^{\mathcal{L}} \text{ com } \mathcal{L} = ceil(log_2[\mathcal{V} \times \mathcal{H}_{min}(v)])$$

Considerando a existência de nós virtuais de duas espécies, da expressão anterior resulta

$$\mathcal{H}^{\mathsf{e}}(d) = 2^{\mathcal{L}^{\mathsf{e}}(d)} \text{ com } \mathcal{L}^{\mathsf{e}}(d) = ceil(log_2[\mathcal{V}^{\mathsf{e}}(d) \times \mathcal{H}_{min}(v)]) \tag{5.1}$$

$$\mathcal{H}^{\mathsf{a}}(d) = 2^{\mathcal{L}^{\mathsf{a}}(d)} \text{ com } \mathcal{L}^{\mathsf{a}}(d) = ceil(log_2[\mathcal{V}^{\mathsf{a}}(d) \times \mathcal{H}_{min}(v)]) \tag{5.2}$$

Ora, uma vez que aquilo que se pretende é suportar, de forma desacoplada, o endereçamento e o armazenamento das entradas de uma mesma DHT, então é óbvio que $\mathcal{H}^{\mathsf{e}}(d)$ e $\mathcal{H}^{\mathsf{a}}(d)$ têm que ser iguais; mas, para que tal aconteça, $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ não têm de ser iguais, pois existem intervalos de variação de $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ ao longo dos quais $\mathcal{H}^{\mathsf{e}}(d)$ e $\mathcal{H}^{\mathsf{a}}(d)$ coincidem; mais especificamente, essa coincidência verifica-se desde que $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ pertençam ao mesmo intervalo delimitado por duas potências de 2 consecutivas; ao longo desse intervalo, $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ podem evoluir independentemente; se um desses valores ultrapassar as fronteiras do intervalo actual, então o outro terá de ser aumentado/diminuído (apenas) o suficiente para transitarem conjuntamente para o próximo intervalo; este raciocínio é aplicado também durante a definição dos valores iniciais de $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$.

A tabela 5.1 apresenta um invariante que traduz a interdependência entre o número de entradas de uma DHT e o de nós virtuais de cada espécie. O invariante é o primeiro de um conjunto de outros (que serão apresentados na secção 5.8), cuja validação permanente garante a consistência das relações entre os vários componentes da arquitectura Domus.

$\mathcal{I}_0$	:	$\mathcal{H}(d) == 2^{\mathcal{L}^{\mathsf{e}}(d)} == 2^{\mathcal{L}^{\mathsf{a}}(d)}$ (com $\mathcal{L}^{\mathsf{e}}(d) = ceil[log_2(\mathcal{V}^{\mathsf{e}}(d) \times \mathcal{H}_{min}(v))]$ e $\mathcal{L}^{\mathsf{a}}(d) = ceil[log_2(\mathcal{V}^{\mathsf{a}}(d) \times \mathcal{H}_{min}(v))]$) – o número de entradas da DHT deve coincidir para as duas espécies de nós virtuais

Tabela 5.1: Invariante Base da Dissociação Endereçamento - Armazenamento.

5.5.1.1 Significado do Número de Nós Virtuais

Como referimos na secção 3.5.2, o número de nós virtuais de uma DHT, ou dos seus nós/serviços, pode ter várias leituras e resultar de lógicas diversas, desde que associadas ao suporte de uma distribuição eventualmente heterogénea da DHT, e acompanhadas de critérios para a definição do número global e individual de nós virtuais. Analisemos, então, algumas das possibilidades que fazem sentido, no contexto da arquitectura Domus:

1. $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ traduzem um certo grau de paralelismo/distribuição *potencial*, no desempenho das funções de endereçamento e armazenamento; esse grau só é *real* quando $\mathcal{N}^{\mathsf{e}}(d) = \mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{N}^{\mathsf{a}}(d) = \mathcal{V}^{\mathsf{a}}(d)$ o que, teoricamente, é possível (ver secção 5.8.3.4); neste contexto, a motivação para uma definição/limitação administrativa (*e.g.*, na criação da DHT) de $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ serão essencialmente requisitos de desempenho;

2. $\mathcal{V}^{\mathsf{a}}(d)$ traduz, indirectamente, a capacidade de armazenamento *potencial*[11] da DHT, desde que definida uma capacidade de armazenamento associada a cada nó virtual de armazenamento (ver secção 5.8.3.9); neste contexto, a motivação para uma definição administrativa de $\mathcal{V}^{\mathsf{a}}(d)$ (e da capacidade de armazenamento associada a cada nó virtual) serão essencialmente requisitos de espaço de armazenamento;

3. $\mathcal{V}^{\mathsf{e}}(d, n)$ e $\mathcal{V}^{\mathsf{a}}(d, n)$ traduzem o nível de participação de um nó n no endereçamento e no armazenamento da DHT d; a definição de $\mathcal{V}^{\mathsf{e}}(d, n)$ e de $\mathcal{V}^{\mathsf{a}}(d, n)$ obedece a mecanismos automáticos, discutidos com profundidade ao longo do capítulo 6.

5.5.1.2 Irrelevância da Identidade das Entradas de um Nó Virtual

Antes de prosseguir, aprofundamos a relevância do *número* de entradas de um nó virtual, face à irrelevância da *identidade* dessas entradas. Com efeito, na arquitectura Domus, a) o facto de um nó n da DHT d exibir/reclamar $\mathcal{V}^{\mathsf{e}}(d, n)$ nós virtuais de endereçamento e de b) isso lhe conferir o direito à gestão do endereçamento de um conjunto de entradas $H^{\mathsf{e}}(n, d)$, não implica o particionamento efectivo de $H^{\mathsf{e}}(n, d)$ em $\mathcal{V}^{\mathsf{e}}(d, n)$ subconjuntos (o mesmo raciocínio é aplicável no contexto do armazenamento, para $H^{\mathsf{a}}(n, d)$ e $\mathcal{V}^{\mathsf{a}}(d, n)$).

Dito de outra forma, as entradas não "pertencem" a nós virtuais, mas a nós computacionais (ou, equivalentemente, a serviços Domus)[12]; esta abordagem, sendo suficiente para a aplicação do modelo M4, evita a sobrecarga da gestão separada de nós virtuais do mesmo nó/serviço e, sobretudo, permite maior flexibilidade na gestão das entradas locais de uma DHT (*e.g.*, sendo necessário migrar uma entrada, à partida qualquer entrada é elegível).

5.6 Serviços Domus Regulares

Os serviços Domus são os "blocos construtores" da arquitectura Domus[13]: é a cooperação entre serviços Domus que fornece aos clientes/aplicações Domus a abstracção de uma ou mais DHTs, de dimensões virtualmente enormes. A arquitectura prevê dois tipos de serviços Domus: 1) serviços Domus "regulares/indiferenciados" (ou simplesmente *serviços*) e 2) um serviço Domus "supervisor" (ou simplesmente *supervisor*). Nesta secção detemo-nos no estudo dos serviços regulares. O papel do supervisor é discutido na secção 5.7.

[11] Porque o espaço de armazenamento não é efectivamente reservado (ver secção 6.9).

[12] Dito ainda de outra forma: não é mantida qualquer associação entre o identificador de uma entrada e um hipotético identificador de nó virtual, simplesmente porque não é necessário nomear os nós virtuais.

[13] Embora essenciais à sua realização, os serviços básicos consideram-se quase externos à arquitectura, pois a sua execução no *cluster* poderá ser justificada no quadro do suporte a outras aplicações e serviços.

5.6.1 Arquitectura Interna

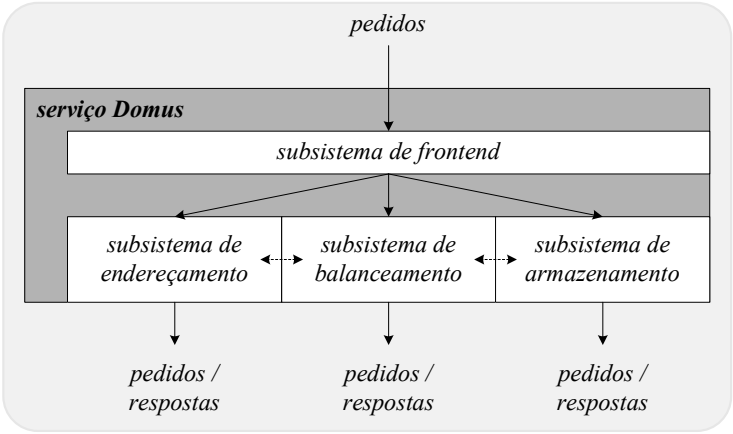


Figura 5.4: Arquitectura interna de um serviço Domus (regular).

A figura 5.4 é uma representação de alto nível da arquitectura interna de um serviço Domus. Assim, um serviço Domus comporta: a) um *subsistema de frontend* (SF), b) um *subsistema de balanceamento* (SB), c) um *subsistema de endereçamento* (SE) e d) um *subsistema de armazenamento* (SA). A definição destes subsistemas não só traduz uma estruturação modular como reflecte a filosofia de dissociação endereçamento-armazenamento.

Cada serviço Domus desempenha então funções de *endereçamento*, *armazenamento* e *balanceamento*, através de subsistemas específicos. O desempenho da função de *balanceamento* é mandatório: todos os serviços Domus participam num mecanismo global de balanceamento dinâmico que determina a forma como as funções de endereçamento e armazenamento são assumidas entre eles. As funções de endereçamento e armazenamento são facultativas e independentes (podendo ser desempenhadas em combinações diversas – ver a seguir). O subsistema de *frontend* recepciona e multiplexa as mensagens dirigidas ao serviço Domus.

Note-se que, apesar da *dissociação funcional* na origem da sua definição, o SE, SA e SB não são estanques, podendo manter interacções diversas, como sugerem as setas horizontais.

5.6.2 Flexibilidade Funcional

Num serviço Domus, o subsistema de *frontend* e o de balanceamento estão sempre activos. Os subsistemas de endereçamento e de armazenamento poderão ser activados ou desactivados, de forma independente entre si. Essa independência também se verifica na definição das DHTs suportadas (e, a um nível mais fino, na definição das entradas geridas[14]).

Assim, enquanto que noutras abordagens a DHTs, as entidades que as sustentam gozam de paridade funcional *permanente* (ou seja, desempenham todas, e sempre, o mesmo tipo de funções), na arquitectura Domus essa paridade, embora *potencial*, poderá não ser *efectiva*.

[14]Ou seja, mesmo que o subsistema de endereçamento e o subsistema de armazenamento participem na realização de uma mesma DHT, poderão fazê-lo para diferentes conjuntos de entradas dessa DHT.

Neste contexto, a regra arquitectural $\mathcal{R}_4$ pode ser refinada, veiculando de forma explícita a flexibilidade funcional permitida a um serviço Domus: "um serviço Domus poderá participar na realização de diversas DHTs, *através de múltiplas combinações de funcionalidade*".

Dos serviços Domus com apenas o subsistema de *frontend* e o de balanceamento activos, diz-se que têm uma *participação potencial* na realização de DHTs; dos que têm o subsistema de endereçamento e/ou de armazenamento activo, diz-se que têm uma *participação real* nessa realização. Naturalmente, a realização de uma DHT obriga a que haja pelo menos um serviço Domus com o respectivo subsistema de endereçamento activo e um serviço Domus (o mesmo ou outro) com o respectivo subsistema de armazenamento activo.

A figura 5.5 refina a figura 5.1, representando uma possibilidade de suporte dos subsistemas de endereçamento e armazenamento dos serviços $s_1, ..., s_5$ às DHTs d_0, d_1 e d_2.

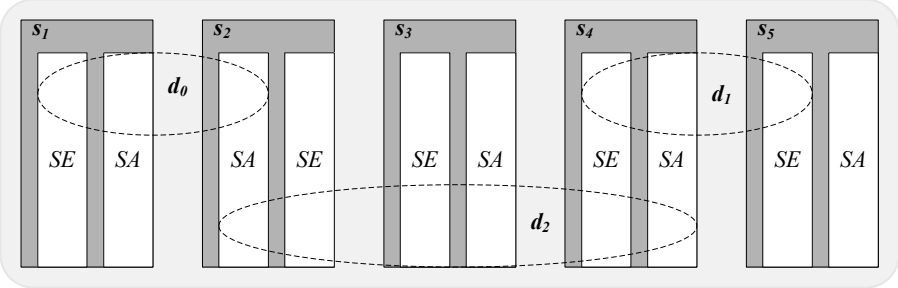


Figura 5.5: Flexibilidade Funcional (possibilidade compatível com o cenário da figura 5.1).

Assim, para a DHT d_0, o endereçamento é suportado pelo SE de s_1 e o armazenamento pelos SAs de s_1 e s_2; para a DHT d_2, o endereçamento e o armazenamento são ambos suportados pelos serviços s_2, s_3 e s_4, através dos respectivos SEs e SAs; para a DHT d_1, o endereçamento é suportado pelo SE de s_4 e s_5, e o armazenamento pelo SA de s_4. Apesar de representado, o SA de s_5 está inactivo, pois não suporta ainda nenhuma DHT. Esta descrição pode-se fazer em termos formais, recorrendo a notação apropriada (ver a seguir).

5.6.3 Decomposição Funcional

Para além de exemplos concretos, que ilustram a flexibilidade e diversidade funcional dos serviços Domus (como o fornecido pela figura 5.5), é útil dispor de notação adequada, compatível com uma descrição mais formal da arquitectura interna dos serviços Domus e da sua relação com outros componentes. No que se segue, introduz-se a notação relevante[15] com o apoio da figura 5.6, que permite visualizar parte das relações formais estabelecidas.

Notação: sendo S o conjunto global dos serviços Domus de um cluster Domus, tem-se

- $S(D)$: subconjunto de S com suporte *real/efectivo* a DHTs (*);
- $S(\emptyset)$: subconjunto de S com suporte *potencial* a DHTs (*);
- $S = S(D) \cup S(\emptyset)$.

[15] A notação assinalada com (*) é também aplicável a nós (computacionais); para o efeito, basta substituir os símbolos s (serviço) e S (conjunto de serviços) por n (nó) e N (conjunto de nós), respectivamente.

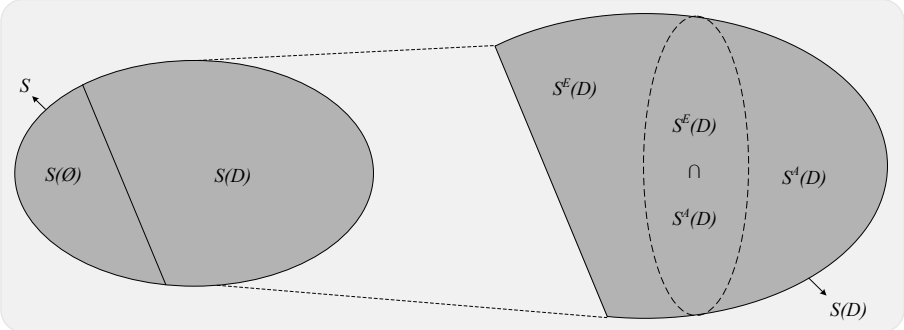


Figura 5.6: Decomposição funcional do conjunto dos serviços Domus de um cluster Domus.

Exemplo: na figura 5.1, tem-se $S(D) = \{s_1, s_2, ..., s_5\}$, $S(\emptyset) = \{s_0\}$ e $S = \{s_0, s_1, ..., s_5\}$.

Notação(*): caracterizando os serviços $S(D)$ pelas DHTs específicas suportadas, tem-se

- $S(d)$: subconjunto de $S(D)$ que suporta a DHT d;
- $s(d)$: um elemento (serviço) de $S(d)$;
- $S(D) = \bigcup_{d \in D} S(d)$.

Exemplo: reportando-nos ao cenário da figura 5.1, tem-se $S(d_0) = \{s_1, s_2\}$, $S(d_2) = \{s_2, s_3, s_4\}$, $S(d_1) = \{s_4, s_5\}$ bem como $S(D) = S(d_0) \cup S(d_2) \cup S(d_1) = \{s_1, s_2, ..., s_5\}$.

Notação(*): sendo D o conjunto global das DHTs instanciadas no cluster Domus, tem-se

- $D(s)$: DHTs de D suportadas, de alguma forma[16], pelo serviço s;
- $d(s)$: um elemento (DHT) de $D(s)$;
- $D = D(S) = \bigcup_{s \in S} D(s)$.

Exemplo: na figura 5.1, tem-se $D(s_2) = \{d_0, d_2\}$ e $D = \bigcup_{k=1}^{5} D(s_k) = \{d_0, d_2, d_1\}$.

Notação(*): sendo $H(d)$ o conjunto global das entradas da DHT d, convenciona-se

- $H(d, s)$: entradas de d suportadas, de alguma forma[16], pelo serviço $s \in S(d)$;
- $h(d, s)$: um elemento (entrada/hash) de $H(d, s)$;
- $H(d) = \bigcup_{s \in S(d)} H(d, s)$.

Notação(*): tendo em conta o desacoplamento endereçamento-armazenamento, tem-se

[16]Seja no contexto do endereçamento, seja no do armazenamento.

- $S^{\mathsf{e}}(D)$ / $S^{\mathsf{a}}(D)$: subconjunto de $S(D)$ que participa no endere./armaze. de DHTs;
- $S(D) = S^{\mathsf{e}}(D) \cup S^{\mathsf{a}}(D)$;

- $S^{\mathsf{e}}(d)$ / $S^{\mathsf{a}}(d)$: subconjunto de $S(d)$ que suporta o endere./armaze. da DHT d[17];
- $s^{\mathsf{e}}(d)$ / $s^{\mathsf{a}}(d)$: um elemento (serviço) de $S^{\mathsf{e}}(d)$ / $S^{\mathsf{a}}(d)$;
- $S(d) = S^{\mathsf{e}}(d) \cup S^{\mathsf{a}}(d)$, $S^{\mathsf{e}}(D) = \bigcup_{d \in D} S^{\mathsf{e}}(d)$ e $S^{\mathsf{a}}(D) = \bigcup_{d \in D} S^{\mathsf{a}}(d)$;

- $D^{\mathsf{e}}(s)$ / $D^{\mathsf{a}}(s)$: DHTs de D, cujo endere./armaze. é suportado pelo serviço s;
- $d^{\mathsf{e}}(s)$ / $d^{\mathsf{a}}(s)$: um elemento (DHT) de $D^{\mathsf{e}}(s)$ / $D^{\mathsf{a}}(s)$;
- $D(s) = D^{\mathsf{e}}(s) \cup D^{\mathsf{a}}(s)$;

- $H^{\mathsf{e}}(d, s)$ / $H^{\mathsf{a}}(d, s)$: entradas de d cujo endere./armaze. é gerido pelo serviço s;
- $h^{\mathsf{e}}(d, s)$ / $h^{\mathsf{a}}(d, s)$: um elemento (entrada/hash) de $H^{\mathsf{e}}(d, s)$ / $H^{\mathsf{a}}(d, s)$;
- $H(d, s) = H^{\mathsf{e}}(d, s) \cup H^{\mathsf{a}}(d, s)$;

- $\mathcal{V}^{\mathsf{e}}(s) = \sum_{d \in D^{\mathsf{e}}(s)} \mathcal{V}^{\mathsf{e}}(d, s)$ e $\mathcal{V}^{\mathsf{a}}(s) = \sum_{d \in D^{\mathsf{a}}(s)} \mathcal{V}^{\mathsf{a}}(d, s)$: número total de nós virtuais de cada espécie, atribuídos ao serviço s (complemento das definições da secção 5.5.1)

Exemplos: na figura 5.5 tem-se

- $S^{\mathsf{e}}(d_0) = \{s_1\}$, $S^{\mathsf{a}}(d_0) = \{s_1, s_2\}$, $S^{\mathsf{e}}(d_2) = \{s_2, s_3, s_4\}$, $S^{\mathsf{a}}(d_2) = \{s_2, s_3, s_4\}$, $S^{\mathsf{e}}(d_1) = \{s_4, s_5\}$, $S^{\mathsf{a}}(d_1) = \{s_4\}$, $S^{\mathsf{e}}(D) = \{s_1, ..., s_5\}$ e $S^{\mathsf{a}}(D) = \{s_1, ..., s_4\}$.
- $D(s_2) = \{d_0, d_2\}$, $D^{\mathsf{e}}(s_2) = \{d_2\}$, $D^{\mathsf{a}}(s_2) = \{d_0, d_2\}$ e $D = \bigcup_{k=1}^{5} D(s_k) = \{d_0, d_2, d_1\}$.

5.6.4 Subsistema de *Frontend*

O *subsistema de frontend* (SF) é responsável a) pela recepção das mensagens dirigidas ao serviço Domus e b) sua multiplexagem pelos restantes subsistemas (SA, SB e SE).

Em vez de uma atribuição simples, por ordem de chegada, as mensagens podem ser entregues aos subsistemas SA, SB e SE, para processamento, de forma a balancear a utilização dos recursos do nó hospedeiro. Por exemplo, espera-se que o SE seja predominantemente *CPU bound*, pelo que convém intercalar o processamento de mensagens dirigidas a esse subsistema, com outras dirigidas ao SA, o qual se espera sobretudo *IO bound*.

O SF efectuaria assim um primeiro nível de balanceamento de carga local (mais difícil de realizar com *frontends* específicos para o SB, SA e SE), complementar a outros mecanismos da arquitectura; para o efeito, uma possibilidade passa pela gestão adequada de uma fila de mensagens e da sua atribuição a fios de execução pré-associados aos outros subsistemas.

[17] Também designado por *domínio de endereçamento* / *domínio de armazenamento* – ver secção 5.8.3.4.

5.6.5 Subsistema de Endereçamento

O *subsistema de endereçamento* (SE) de um serviço Domus realiza a *gestão de endereçamento* de subconjuntos de entradas, de uma ou mais DHTs. Em termos formais: para cada DHT $d \in D^{\mathsf{e}}(s)$, o SE de s gere o endereçamento das entradas $H^{\mathsf{e}}(d,s)$, em número $\mathcal{H}^{\mathsf{e}}(d,s)$. A definição do *número* de entradas resulta da aplicação do modelo M4 de distribuição heterogénea descrito no capítulo 3, complementado pelos modelos do capítulo 6. A *identidade* das entradas é definida com base nos mecanismos descritos no capítulo 4.

Equivalentemente, o SE de s realiza a gestão de endereçamento de um número total de $\mathcal{V}^{\mathsf{e}}(n)$ *nós virtuais de endereçamento*, relativos a uma ou mais DHTs, sendo o número específico para cada $d \in D^{\mathsf{e}}(s)$ denotado por $\mathcal{V}^{\mathsf{e}}(d,s)$. Assim, é como se cada conjunto $H^{\mathsf{e}}(d,s)$ de entradas se pudesse decompor em $\mathcal{V}^{\mathsf{e}}(d,s)$ subconjuntos, com um número médio de $\overline{\mathcal{H}}^{\mathsf{e}}(d,s,v^{\mathsf{e}})$ entradas por subconjunto / nó virtual (v^{e}), dado simplesmente por

$$\overline{\mathcal{H}}^{\mathsf{e}}(d,s,v^{\mathsf{e}}) = \frac{\mathcal{H}^{\mathsf{e}}(d,s)}{\mathcal{V}^{\mathsf{e}}(d,s)} \tag{5.3}$$

No SE de s, a gestão de endereçamento de $d \in D^{\mathsf{e}}(s)$ inclui: a) a gestão da *informação de endereçamento* de $H^{\mathsf{e}}(d,s)$ e b) a participação na *localização distribuída* de entradas de d.

5.6.5.1 Informação de Endereçamento

A *informação de endereçamento* de uma entrada / hash h inclui o *identificador da entrada* (h), uma *tabela de encaminhamento* (TE) e uma *referência de armazenamento* (RA).

A *tabela de encaminhamento* específica de cada entrada regista *referências de endereçamento* (REs), ou seja, referências para o *nó/serviço de endereçamento* de outras entradas da mesma DHT, em número limitado e definidas deterministicamente. Essa definição é condicionada pelo opção de recorrer a um grafo Chord de suporte à *localização distribuída* de entradas da DHT, de acordo com a nossa própria abordagem, descrita no capítulo 4[18].

A *referência de armazenamento* de uma entrada concretiza a dissociação entre endereçamento e armazenamento: identifica o *nó/serviço de armazenamento* da entrada, que pode coincidir com o *nó/serviço de endereçamento* (referência *local*) ou não (referência *remota*).

A figura 5.7 representa alguns destes conceitos, de uma forma consistente com os papéis estabelecidos na figura 5.5 para os serviços s_3, s_4 e s_5, no suporte às DHTs d_2 e d_1.

A *informação de endereçamento* é mantida pelo SE de cada serviço em contextos específicos por DHT, representados na figura 5.7 por caixas tracejadas; um *índice de endereçamento* (IE) reúne a *informação de endereçamento* de cada contexto, indexando-a pelo identificador (h) das entradas; a representação do *índice de endereçamento* em triângulo sugere o recurso a uma estrutura de dados arborescente para a sua realização; essa estrutura pode ser uma evolução das *árvores de encaminhamento* referidas no capítulo 4, relativamente às quais é apenas necessário acrescentar o suporte às *referências de armazenamento*.

[18] Neste contexto, rever a secção 4.4.4, onde se introduz o conceito de Tabela de Encaminhamento.

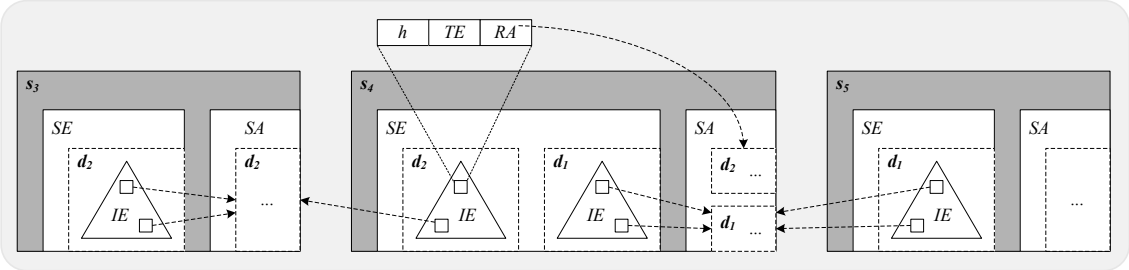


Figura 5.7: Informação de Endereçamento (possibilidade compatível com a figura 5.5).

A figura 5.7 ilustra também uma definição possível de *referências de armazenamento* de tipo local e remoto[19]: para o serviço s_3, as referências de armazenamento representadas são todas locais; para o serviço s_4, algumas referências de armazenamento relativas à DHT d_2 são locais e outras são remotas (apontando para s_3); para o serviço s_4, todas as referências de armazenamento representadas, relativas à DHT d_1, são locais; finalmente, para o serviço s_5, todas as referências de armazenamento representadas são remotas (apontando para s_4).

5.6.5.2 Localização (Distribuída) de Entradas

A dissociação entre endereçamento e armazenamento introduz uma conotação dupla no conceito de "localização de uma entrada". Assim, "localizar uma entrada" significa, em primeira instância, "localizar o seu serviço de endereçamento"; em geral, esta localização ocorrerá associada também à "localização do serviço de armazenamento" da entrada.

A localização do serviço de armazenamento de uma entrada é feita com o objectivo de aceder a registos associados à entrada, residentes no nó/serviço de armazenamento da entrada. Dado que a referência de armazenamento de uma entrada é preservada lado-a-lado com a sua tabela de encaminhamento, a localização do serviço de armazenamento é resolvida localmente (sem acesso à rede), pelo nó/serviço de endereçamento da entrada.

A localização do serviço de endereçamento de uma entrada, sem localização associada do serviço de armazenamento, também faz sentido; por exemplo, modificando-se o serviço de endereçamento de uma entrada (*e.g.*, em resultado dos mecanismos de balanceamento dinâmico descritos no capítulo 6), será preciso localizar todas as antecessoras da entrada no grafo Chord, a fim de actualizar as tabelas de encaminhamento dessas antecessoras.

Com *encaminhamento convencional*, a localização distribuída de uma "entrada alvo" inicia-se pela consulta da tabela de encaminhamento de uma "entrada inicial"; a informação topológica aí preservada permite determinar qual a próxima entrada a "visitar" (ou melhor, qual a próxima tabela de encaminhamento a consultar); o processo repete-se por cada entrada visitada, de forma a que a distância topológica à entrada alvo diminui consistentemente, até que essa entrada seja localizada; a *cadeia de encaminhamento* dá-se por concluída quando a entidade que solicitou a localização é informada do seu resultado.

Em cada nó/serviço visitado, são aplicáveis os algoritmos de *encaminhamento acelerado* que desenvolvemos no capítulo 4, sendo neste caso realizados sobre a estrutura de dados

[19]Não são ilustradas as *referências de endereçamento*, contidas nas tabelas de encaminhamento.

que implementa o *índice de endereçamento* local da DHT e que, como já se referiu, será basicamente uma evolução das *árvores de encaminhamento* introduzidas nesse capítulo.

A figura 5.8 ilustra uma *cadeia de encaminhamento*, compatível com as figuras 5.1 e 5.5.

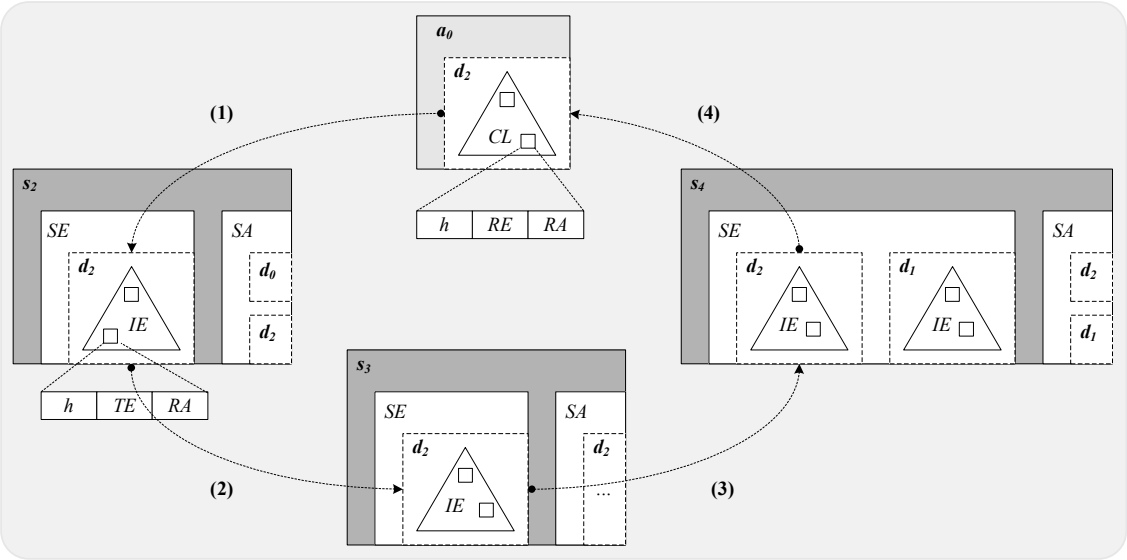


Figura 5.8: Localização Distribuída (cenário compatível com as figuras 5.1 e 5.5).

Na figura, a aplicação a_0, envolvida no acesso à DHT d_2, submete um pedido de localização de uma entrada de d_2, ao serviço s_2, que é serviço de endereçamento de d_2 (passo **1**). A informação de endereçamento disponível em s_2, acerca de d_2, permite definir s_3 como o próximo serviço de endereçamento de d_2 a visitar (passo **2**). O processo repete-se em s_3, conduzindo à conclusão de que o serviço de endereçamento da entrada de d_2, que se procura, é s_4 (passo **3**). Finalmente, s_4 informa a_0 do resultado final da localização (passo **4**)[20]; o resultado corresponde às referências de endereçamento (RE) e de armazenamento (RA) da entrada[21], usadas para alimentar uma *cache de localização* em a_0 (ver a seguir).

A cadeia de encaminhamento da figura 5.8 é classificada de *externa*, pois é originada numa aplicação Domus. Se fosse originada num serviço Domus, seria classifica de *interna* (nesta categoria cabem, por exemplo, as cadeias de localização originadas tendo em vista a actualização de tabelas de encaminhamento afectadas pela migração de entradas).

5.6.5.3 Cache de Localização

Como referido acima, as aplicações podem tirar partido de uma *cache de localização* (CL), preenchida e actualizada com os resultados dos seus pedidos de localização. Basicamente, a *cache* CL é uma colecção de registos de esquema <h,RE,RA>, em que o nó/serviço de endereçamento e de armazenamento de uma entrada h é identificado pelas referências RE

[20] Dependendo da implementação concreta, essa informação poderá ser dada pelo nó/serviço de endereçamento da "entrada alvo", ou pelo nó/serviço imediatamente anterior, na cadeia de encaminhamento. No contexto dos algoritmos de localização distribuída do capítulo anterior, aplica-se esta última hipótese.

[21] Ou seja, à identificação do nó/serviço de endereçamento e de armazenamento da entrada.

e RA. A *cache* é naturalmente indexada por h e deve ser baseada em estruturas de dados iguais às das *árvores de encaminhamento* usadas para *encaminhamento acelerado*; se assim for então, mesmo numa situação de *cache-miss*[22], os registos contidos na *cache* poderão ser usados com proveito, contribuindo para uma decisão de encaminhamento informada.

Naturalmente, o recurso a uma *cache de localização* a) implica a opção por uma certa política de substituição de registos antigos por novo registos (LRU, etc.), para além de b) comportar a necessidade de detectar registos desactualizados, que devem ser removidos.

Posteriormente (ver secção 5.9.1.3), veremos que a *cache de localização* pode ser usada como parte de uma *estratégia de localização*, combinada com outros *métodos de localização*.

5.6.6 Subsistema de Armazenamento

O *subsistema de armazenamento* (SA) de um serviço Domus é responsável pela *gestão de armazenamento* de subconjuntos de entradas, de uma ou mais DHTs. Formalmente: para cada DHT $d \in D^{\mathsf{a}}(s)$, o SA de s gere o armazenamento das entradas $H^{\mathsf{a}}(d,s)$, em número $\mathcal{H}^{\mathsf{a}}(d,s)$ (definidos com a mesma metodologia usada na definição de $H^{\mathsf{e}}(d,s)$ e $\mathcal{H}^{\mathsf{e}}(d,s)$).

No plano equivalente dos nós virtuais, o SA de s realiza a gestão de armazenamento de um número total de $\mathcal{V}^{\mathsf{a}}(n)$ *nós virtuais de armazenamento*, relativos a uma ou mais DHTs, sendo o número específico para cada $d \in D^{\mathsf{a}}(s)$ dado por $\mathcal{V}^{\mathsf{a}}(d,s)$. Em termos abstractos, é como se cada conjunto $H^{\mathsf{a}}(d,s)$ fosse particionável em $\mathcal{V}^{\mathsf{a}}(d,s)$ subconjuntos, com um número (médio) de $\overline{\mathcal{H}}^{\mathsf{a}}(d,s,v^{\mathsf{e}})$ entradas por subconjunto / nó virtual (v^{a}), dado por

$$\overline{\mathcal{H}}^{\mathsf{a}}(d,s,v^{\mathsf{a}}) = \frac{\mathcal{H}^{\mathsf{a}}(d,s)}{\mathcal{V}^{\mathsf{a}}(d,s)} \tag{5.4}$$

No SA de s, a gestão de endereçamento das entradas $H^{\mathsf{a}}(d,s)$ de $d \in D^{\mathsf{a}}(s)$ inclui: a) a gestão da sua *informação de armazenamento* e b) a gestão dos seus *repositórios* de registos.

5.6.6.1 Informação de Armazenamento

A *informação de armazenamento* de uma entrada / hash h inclui o *identificador da entrada* (h), a sua *referência de endereçamento* (RE) e a sua *referência de repositório* (RR).

Num serviço s, a *informação de armazenamento* das entradas $H^{\mathsf{a}}(d,s)$ da DHT d é consolidada num *índice de armazenamento* (IA) específico para d; tal como o *índice de endereçamento*, o *índice de armazenamento* é indexado pelo identificador das entradas e a sua realização por uma estrutura arborescente é particularmente vantajosa (ver a seguir).

A *referência de endereçamento* (RE) consubstancia a correspondência "reversa" *serviço de armazenamento* → *serviço de endereçamento*[23]; a preservação destas correspondências nos SAs agiliza a actualização das referências de armazenamento nos SEs[24] (por exemplo, se o serviço de armazenamento de uma entrada se modificar, há que actualizar a sua referência de armazenamento, conservada no serviço de endereçamento respectivo).

[22] Ausência na *cache* de um registo indexado por uma certa entrada h que se pretende localizar.
[23] Por contraposição à correspondência "directa" *serviço de endereçamento* → *serviço de armazenamento*.
[24] Outra forma, menos expedita, seria recorrer à localização distribuída dos serviços de endereçamento.

Adicionalmente, as correspondências reversas mantidas num SA podem ser exploradas pelo SE companheiro (residente no mesmo nó/serviço), a fim de acelerar uma localização distribuída. Nesse contexto, os mecanismos de *encaminhamento acelerado* seriam aplicados, adicionalmente, sobre o *índice/árvore de armazenamento* do SA, de forma análoga à sua aplicação sobre uma *cache de localização* numa aplicação; na decisão de encaminhamento final tomada pelo SE, seria eleita a rota mais curta, de entre a seleccionada pela análise i) das tabelas de encaminhamento do SE e ii) da informação de endereçamento do SA.

A *referência de repositório* de uma entrada identifica o repositório do *conjunto de registos*, do tipo <*chave*,*dados*>, atribuídos à entrada (um registo é atribuído a uma certa entrada h quando o resultado da aplicação da função de hash da DHT à *chave* coincide com h).

A figura 5.9 fornece uma representação dos conceitos anteriores, tomando como ponto de partida a figura 5.7 e ampliando o SA do serviço s_4. Assim, na figura 5.9 representam-se os índices de armazenamento (IA) das DHTs d_1 e d_2 e, no contexto desses índices, são representadas referências de repositórios (RR) e referências de endereçamento (RE) (a maioria *locais*, direccionadas para o SE de s_4, e uma *remota*, direccionada ao SE de s_5).

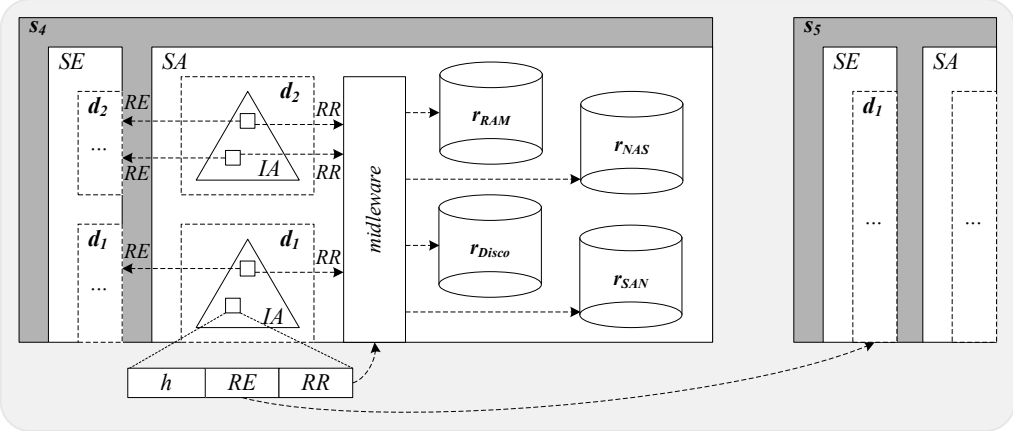


Figura 5.9: Info. de Armazenamento e Repositórios (cenário compatível com a figura 5.7).

5.6.6.2 Repositórios

No contexto da arquitectura Domus, um *repositório* é qualquer sistema de armazenamento que suporta a salvaguarda de *dicionários* (os quais, como definido no capítulo 1, são basicamente colecções de registos do tipo <*chave*,*dados*>, indexados pelo campo *chave*).

O subsistema de armazenamento de um serviço Domus é capaz de gerir múltiplos repositórios, assentes em *tecnologias de armazenamento* diversas; a designação *tecnologia de armazenamento* refere-se a uma combinação particular de uma certa *plataforma de armazenamento* de dicionários (*e.g.*, Berkeley-DB, Árvores B+, Tabelas de Hash, etc.) que deposita registos num certo *meio de armazenamento* (*e.g.*, RAM, Disco, Rede, etc.); a *plataforma de armazenamento* e o *meio de armazenamento* são atributos especificáveis individualmente para cada DHT (ver secção 5.8.3.7). Uma implementação da arquitectura Domus deverá suportar pelo menos RAM e Disco como meios de armazenamento.

Na figura 5.9, para além de representados quatro repositórios distintos (r_{RAM}, r_{Disco}, r_{NAS} e r_{SAN}), diferenciados pelo mrio de armazenamento, é também representada uma camada de *midleware*; esta intermedeia o acesso aos repositórios, através de um interface de acesso unificado que oferece as operações básicas de acesso a um dicionário (inserções, consultas e remoções); portanto, o suporte a novas *tecnologias de armazenamento* carece apenas da codificação de um conjunto mínimo de funcionalidades na camada de *midleware*.

Os repositórios podem classificar-se pela sua *granularidade*. Assim, um repositório poderá suportar i) registos de uma só entrada de uma certa DHT (*grão mínimo*), ii) vários registos de uma mesma DHT (*grão médio*) ou iii) vários registos de várias DHTs (*grão máximo*).

Um repositório de *grão fino* ou *grão médio* é portanto um repositório *dedicado* a uma só DHT, enquanto que um repositório de *grão máximo* é *partilhado* por várias DHTs; no limite, poderá haver um só repositório de tipo *partilhado*, que se designa por *unificado*.

Um repositório de grão fino permite agilizar a transferência da responsabilidade de armazenamento de uma entrada, de um nó/serviço para outro nó/serviço (já que não é necessário varrer um repositório partilhado, em buscas dos registos específicos da entrada a migrar).

Um repositório partilhado carece de processamento adicional das chaves dos registos, necessário para se distinguirem registos com a mesma chave, mas de DHTs diferentes.

5.6.7 Subsistema de Balanceamento

A atribuição de funções de endereçamento e de armazenamento aos diversos serviços Domus de um cluster Domus, é regulada por mecanismos globais de balanceamento dinâmico, em que os serviços Domus participam, através do seu *Subsistema de Balanceamento* (SB).

Como referido na secção 5.6.2, o desempenho de funções de balanceamento é obrigatório para qualquer serviço Domus, ao contrário das funções de endereçamento e armazenamento, que são facultativas; na mesma secção, definiram-se os conceitos de i) *participação potencial* e ii) *participação efectiva* de um serviço na realização de DHTs, correspondentes, respectivamente, a situações em que o serviço a) tem apenas o SF e o SB activos e b) tem o SE e/ou SA activos, em adição ao SF e ao SB; precisamente, o SB pode desempenhar diferentes funções, dependendo do *regime de participação* do serviço ao qual pertence:

- sob *participação potencial*, o SB 1) monitoriza[25] a utilização dos recursos locais do nó hospedeiro (*e.g.*, CPU, RAM, Disco e Rede), relevantes para o endereçamento e armazenamento de DHTs, 2) propaga[26] os resultados dessa monitorização ao serviço supervisor (ver secção 5.7) e, 3) cooperando com o supervisor numa *operação de balanceamento* de tipo *global*, transita para um regime de *participação efectiva*;
- sob *participação efectiva*, o SB acrescenta às tarefas anteriores 1) a monitorização (em cooperação com o SE e com o SA) da utilização específica que as DHTs fazem dos recursos locais, 2) a propagação dos resultados da monitorização ao supervisor, 3) a iniciação de *operações de balanceamento*, *locais* ou *globais* (ver a seguir).

[25] Em cooperação com os *serviços básicos*.
[26] Se necessário, pois podem ser os serviços básicos a fazê-lo.

5.6.7.1 Balanceamento Global

Uma operação de *balanceamento global* deve a sua designação ao facto de necessitar de informação de estado sobre a totalidade do cluster Domus (ver secção 5.7), informação que é suposto ser mantida pelo serviço supervisor. Uma operação desse tipo pode, no entanto, ser desencadeada quer pelo serviço *supervisor*, quer por serviços Domus indiferenciados.

Assim, num serviço Domus, o SB pode desencadear a *migração de nós virtuais* locais, com o objectivo de reduzir a sobrecarga de certos recursos do nó hospedeiro, cenário que configura uma situação de *redistribuição* de uma DHT. Adicionalmente, o serviço supervisor pode desencadear a *criação de nós virtuais* de uma certa DHT, na tentativa de assegurar uma maior dispersão da(s) carga(s) induzida(s) por essa DHT, cenário que configura uma situação de *expansão* de uma DHT. Ambas as situações envolvem a descoberta de serviços Domus, destinatários dos nós virtuais a migrar, ou hospedeiros dos nós virtuais a criar; essa descoberta é feita pelo supervisor, detentor de uma visão global do cluster Domus.

Da descrição anterior conclui-se da necessidade de dois mecanismos globais de gestão dinâmica de carga: i) um mecanismo de gestão do *Posicionamento* de nós virtuais ($mgPv$) e ii) um mecanismo de gestão do *Número* de nós virtuais ($mgNv$). Neste contexto, o termo *posicionamento* refere-se à definição do nó/serviço que deve albergar um certo número de nós virtuais[27]. O capítulo 6 é dedicado à descrição do *modus operandi* destes mecanismos.

É ainda de notar que o despoletamento de operações de balanceamento global ocorre *por necessidade*, configurando a opção por uma política de *load-shedding*, alternativamente a uma política de *load-stealling*. O objectivo é evitar as perturbações inerentes à redistribuição (seja da carga de endereçamento, seja de armazenamento) de uma DHT, até que seja realmente necessária, pois a redistribuição interfere com o acesso das aplicações à DHT.

5.6.7.2 Balanceamento Local

O SB pode instruir o SA e o SE adjacentes no sentido de se combater a sobrecarga de determinados recursos com base apenas em medidas de abrangência estritamente *local*.

Por exemplo, o SE pode comutar, se necessário, entre diferentes algoritmos de localização, afectando de forma diferente o consumo de CPU (rever os resultados da secção 4.9.5, onde se exibem as diferentes exigências de processamento de diferentes algoritmos). De facto, a alternância entre diferentes algoritmos não coloca em causa a convergência das cadeias de encaminhamento em que o SE participa; essa alternância introduz apenas maior variabilidade no número de saltos dessas cadeias dado que, ao longo de uma cadeia de encaminhamento, diferentes nós/serviços visitados podem aplicar, localmente, diferentes algoritmos (em vez de aplicarem consistentemente o algoritmo que minimiza o número de saltos mas que, tipicamente, é também aquele com maior exigência de processamento).

No SA, a sobrecarga no consumo de recursos de armazenamento poderia ser combatida pela eliminação de registos locais, em vez da criação de mais nós virtuais. Um cenário em

[27] Ou seja, o termo *posicionamento* é usado no sentido de "definição de um local", por contraposição com o termo *localização*, que se refere à "busca de um local" (definido por uma certa política de posicionamento).

que esta opção faz sentido é aquele em que uma DHT actua como uma *cache distribuída* de dimensão variável (*e.g.*, uma *cache* DNS de um *Web Crawler* paralelo/distribuído); nessa situação, assume-se que os registos eliminados poderão ser re-criados mais tarde, pela acção do(s) cliente(s) da DHT; note-se, todavia, que uma selecção não-aleatória (*e.g.*, LRU, etc.) dos registos a remover requer processamento por cada acesso aos mesmos.

A aplicação destes mecanismos pode ser regulada através de *atributos* especiais das DHTs[28].

5.6.7.3 Dinamismo Funcional

Da *flexibilidade funcional* dos serviços Domus, e da operação dos mecanismos de balanceamento, decorre *dinamismo funcional*, pois as funções desempenhadas pelos serviços podem variar ao longo do tempo. Essa variação pode mesmo conduzir, no limite, à oscilação do regime de funcionamento de um serviço Domus entre os estados *efectivo* e *potencial*.

5.7 Supervisão do Cluster Domus

Num cluster Domus, certas operações necessitam da coordenação global de um serviço especial, bem conhecido, designado por *supervisor* (na figura 5.1 apenas se representaram os serviços Domus genéricos/indiferenciados, não se tendo representado o serviço supervisor).

Basicamente, o supervisor actua como representante de todo o cluster Domus, sendo com ele que as aplicações Domus começam por interagir; posteriormente, e dependendo das operações em concreto a realizar, as aplicações poderão continuar a interagir com o supervisor ou então a interacção passará a ser feita directamente com serviços Domus regulares.

Para além do papel já referido de coordenação de operações de balanceamento global (rever a secção anterior), o supervisor gere a realização de outras operações, designadamente:

1. *criação*/*destruição* e *desactivação*/*reactivação* do cluster Domus;
2. *adição*/*remoção* e *desactivação*/*reactivação* de serviços Domus específicos;
3. *criação*/*destruição* e *desactivação*/*reactivação* de DHTs específicas.

A *criação* de um cluster Domus corresponde a arrancar um conjunto inicial de serviços Domus em regime de funcionamento *potencial*; a *destruição* implica a remoção de *informação de estado* ao nível do supervisor e dos serviços Domus[29], seguida da sua terminação[30]. A *informação de estado* inclui *informação de supervisão*, *endereçamento* e *armazenamento*.

A *desactivação* de um cluster Domus traduz-se na salvaguarda, em suporte persistente, da informação de estado do supervisor e serviços Domus, seguida da sua terminação; a *reactivação* passa, em primeiro lugar, pela ressurreição do supervisor e, a partir deste, dos serviços Domus; a ressurreição recorre à informação de estado salvaguardada previamente. Um cluster Domus inactivo/activo encontra-se num estado *offline*/*online*, respectivamente.

[28] Ver, por exemplo, o atributo `dht_attr_pld` (*política de localização distribuída*), na secção 5.8.3.6.
[29] Incluindo também, neste caso, a remoção de eventuais repositórios de DHTs.
[30] Caso se encontrem em execução. Notar que também faz sentido remover um cluster Domus desactivado.

A composição do conjunto S dos serviços Domus pode-se alterar, pela *adição* ou *remoção* de serviços regulares. As alterações afectam também $N(S)$ (conjunto dos nós que suportam serviços Domus), dada a sua relação com S. Neste contexto, e em linha com o referido na secção 5.3.2, as alterações podem ter causas *exógenas* ou *endógenas* ao cluster Domus.

Assim, a *adição* de serviços pode ter i) causas *administrativas* (exógenas), que determinam o reforço do conjunto de serviços Domus *por antecipação*, ou ii) resultar de mecanismos *automáticos* (endógenos) de balanceamento de carga, que arrancam novos serviços *por necessidade/just-in-time* a fim de, nesses serviços, se criarem ou receberem nós virtuais.

Já a *remoção* de serviços tem apenas causas *administrativas* (exógenas), como a necessidade de realizar manutenção do nó hospedeiro ou de o associar em exclusivo a outras tarefas/utilizadores; neste caso, quaisquer responsabilidades de endereçamento/armazenamento desempenhadas pelo serviço a remover devem-se transferir para outro(s), seleccionado(s) (ou até arrancado(s) para o efeito) pelo supervisor; adicionalmente, o nó associado deve ser não só removido de $N(S)$ como ainda de $N(B)$ a fim de evitar a sua utilização futura.

A *desactivação/reactivação* de serviços Domus apenas faz sentido quando realizada de forma colectiva, para todos os serviços de um cluster Domus. Basicamente, implica a terminação/arranque dos serviços, com salvaguarda/recuperação de informação de estado.

A *criação* de uma DHT envolve a verificação de *invariantes* expressos em função de *atributos* da DHT (ver secção 5.8). Parte dos atributos podem ser definidos pela aplicação que provoca a criação. A verificação dos invariantes ocorre no supervisor, que avalia a viabilidade da criação da DHT. A criação traduz-se na 1) definição (independente) de um certo número de nós virtuais de endereçamento e de armazenamento, 2) definição do número global de entradas da DHT necessário para suportar os nós virtuais das duas espécies, 3) definição do número (médio) de entradas, por cada nó virtual de cada espécie, e a 4) atribuição dos nós virtuais aos serviços Domus com recursos suficientes para os suportar[31].

A *destruição* de uma DHT comporta a remoção de toda a sua informação de estado[32] (informação de supervisão, de endereçamento e de armazenamento) e eventuais repositórios.

A *desactivação* de uma DHT corresponde a salvaguardar em suporte persistente toda a sua informação de estado, impedindo, a partir daí, o acesso de aplicações clientes à DHT. A *reactivação* corresponde ao processo inverso, o qual pressupõe o cluster Domus activo.

A possibilidade de comutar uma ou mais DHTs (no limite, o próprio cluster Domus) entre os estados activo e inactivo, permite optimizar a exploração dos recursos do *cluster* físico sem perder o conteúdo das DHTs. Por exemplo, o utilizador de um *cluster* explorado *por lotes*, pode executar um cluster Domus durante uma fatia de tempo limitada e, posteriormente, retomar a sua execução quando receber uma nova fatia de tempo; no mesmo tipo de *cluster*, ou num *cluster* em regime de exploração *partilhado*, é possível aumentar a disponibilidade de RAM desactivando uma ou mais DHTs que operem sobre esse *meio de armazenamento* e que não sejam necessárias na fase actual da resolução do problema.

[31] O que pode implicar o arranque de novos serviços (ou seja, a adição de serviços ao cluster Domus).
[32] Independentemente da DHT se encontrar activa (*online*) ou inactiva (*offline*).

5.8 Parâmetros, Atributos e Invariantes

Uma das mais valias da arquitectura Domus é o *suporte à heterogeneidade* de DHTs, traduzido na possibilidade de definir separadamente, para cada DHT, o valor de uma série de *atributos*, que regulam múltiplos aspectos da sua realização e operação. Adicionalmente, os nós/serviços Domus, ou o próprio cluster Domus, beneficiam da mesma possibilidade.

Um atributo é especificado por um par <*nome*, *valor*>. Alguns atributos têm valor *fixo* (uma vez definido, o valor do atributo permanece inalterado durante a existência da entidade a que diz respeito), enquanto que outros atributos admitem um valor *mutável*.

Os atributos podem classificar-se em *externos* ou *internos*, dependendo do domínio de execução em que é admissível a sua definição. Assim, os atributos *externos* podem ser definidos pelas aplicações Domus, no contexto de certas operações (*e.g.*, na criação de uma DHT ou de um cluster Domus) e, por isso, recebem também a designação de *atributos de nível utilizador*; estes atributos são definidos *directamente* pelo valor de *parâmetros* fornecidos à *biblioteca Domus* (a biblioteca de acesso das aplicações a um cluster Domus – ver secção 5.9.1). Os atributos *internos* (ou *atributos de nível supervisor*) são exclusivamente definidos pelo supervisor, se bem que o seu valor pode ser influenciado, de forma indirecta, pelo valor de certos atributos externos, definidos por parâmetros da *biblioteca Domus*.

O serviço supervisor armazena e gere os atributos das várias entidades do cluster Domus que supervisiona; o supervisor é também o responsável pela validação dos atributos, através do seu confronto com certos *invariantes*; estes mais não são do que condições/restrições formais a que os atributos devem, permanentemente, obedecer. A *informação de supervisão* de um cluster Domus é dada pelo valor colectivo dos atributos mantidos pelo supervisor.

De seguida, descrevem-se os atributos previstos para as entidades da arquitectura Domus.

5.8.1 Atributos de um Cluster Domus

5.8.1.1 Identificador do Cluster Domus

Um cluster Domus deve ter associado um identificador global único, que o permita referenciar univocamente, em contextos onde possam co-existir vários clusters Domus[33]. A tabela 5.2 caracteriza o atributo `attr_cluster_id`, correspondente a esse identificador.

atributo	:	`attr_cluster_id` (identificador do cluster Domus)
notação formal	:	c
tipo de definição	:	constante, externa (obrigatória)

Tabela 5.2: Caracterização do atributo Identificador do cluster Domus.

5.8.1.2 Identificador do Supervisor

O serviço supervisor actua como representante do cluster Domus. Nesse contexto, deve possuir um identificador global único, dado pelo atributo `attr_cluster_supervisor_id`,

[33]Os mecanismos capazes de garantir a unicidade destes identificadores não são aqui considerados.

que será usado para identificar o supervisor nas suas interacções com outras entidades do cluster Domus. Neste contexto, assume-se a existência de uma associação unívoca entre o identificador do supervisor e um determinado interface de rede[34]. A tabela 5.3 sintetiza a caracterização do atributo, sendo de notar que a sua definição variável é compatível com a execução do supervisor em nós diferentes, durante a existência de um cluster Domus.

atributo	:	`attr_cluster_supervisor_id` (identificador do supervisor)
tipo de definição	:	variável, externa (obrigatória)

Tabela 5.3: Caracterização do atributo Identificador do Supervisor.

5.8.1.3 Conjunto dos Nós/Serviços

O supervisor regista o conjunto dos serviços regulares do cluster Domus no atributo `attr_cluster_services`, identificando cada serviço pelo seu nome único (ver secção 5.8.2.1). O atributo `attr_cluster_services` é variável, pela adição/remoção de serviços ao cluster Domus. Cada nó/serviço Domus é caracterizado por certos atributos individuais, expostos na secção 5.8.2. A tabela 5.4 caracteriza o atributo `attr_cluster_services`.

atributo	:	`attr_cluster_services` (conjunto dos serviços do cluster Domus)
estrutura interna	:	lista/conjunto de identificadores dos serviços
notação formal	:	S (ou $N(S)$, pela correspondência unívoca entre serviços e nós)
tipo de definição	:	variável, interna
valor por omissão	:	$\emptyset$ (conjunto vazio)
invariantes associados	:	$N(S) \subseteq N(B)$ (ver figura 5.2)

Tabela 5.4: Caracterização do atributo Conjunto dos Serviços de um cluster Domus.

5.8.1.4 Conjunto das DHTs

O conjunto das DHTs instanciadas num cluster Domus é mantido pelo seu supervisor no atributo `attr_cluster_dhts`, onde cada DHT é nomeada pelo seu identificador único (ver secção 5.8.3.1). O atributo `attr_cluster_dhts` é variável, pela criação/destruição de DHTs. Cada DHT goza de vários atributos individuais (suportando a *heterogeneidade de DHTs*), expostos na secção 5.8.3. A tabela 5.5 caracteriza o atributo `attr_cluster_dhts`.

atributo	:	`attr_cluster_dhts` (conjunto das DHTs do cluster Domus)
estrutura interna	:	lista/conjunto de identificadores das DHTs
notação formal	:	D
tipo de definição	:	variável, interna
valor por omissão	:	$\emptyset$ (conjunto vazio)

Tabela 5.5: Caracterização do atributo Conjunto das DHTs de um cluster Domus.

5.8.1.5 Atributos de Gestão de Carga

Os mecanismos de gestão de carga a descrever no capítulo 6 recorrem a atributos do cluster Domus correspondentes a *limiares* globais para as taxas de utilização de certos recursos dos nós computacionais (ver secção 5.8.2.2). A tabela 5.6 caracteriza esses atributos.

[34] O que não implica, necessariamente, que o identificador do supervisor e do interface coincidam.

atributo	:	`attr_cluster_lcpu` - limiar de utilização de CPUs
notação	:	$\mathcal{U}_\tau(cpu)$

atributo	:	`attr_cluster_liodisk` - limiar de utilização (actividade E/S) de discos
notação	:	$\mathcal{U}_\tau(iodisk)$

atributo	:	`attr_cluster_lnet` - limiar de utilização de interfaces de rede
notação	:	$\mathcal{U}_\tau(net)$

atributo	:	`attr_cluster_lram` - limiar de utilização (espaço consumido) de RAM
notação	:	$\mathcal{U}_\tau(ram)$

atributo	:	`attr_cluster_ldisk` - limiar de utilização (espaço consumido) de discos
notação	:	$\mathcal{U}_\tau(disk)$

Observações:
a) para todos estes atributos, a definição é "constante, interna ou externa"

Tabela 5.6: Caracterização dos Limiares de Utilização de um cluster Domus.

5.8.2 Atributos dos Nós/Serviços

5.8.2.1 Identificador

Cada serviço Domus possui um identificador global único, dado pelo valor do seu atributo `attr_service_id`, que identifica o serviço na i) troca de mensagens, e ii) obtenção/publicação de informação de caracterização e monitorização do nó hospedeiro. A associação unívoca do identificador a um interface de rede – o "interface Domus" do serviço – permite cumprir os requisitos anteriores[35]. A tabela 5.7 caracteriza o atributo `attr_service_id`.

atributo	:	`attr_service_id` (identificador de um serviço Domus)
notação formal	:	s (ou $n(s)$, pela correspondência biunívoca entre serviços e nós)
tipo de definição	:	constante, externa (obrigatória)

Tabela 5.7: Caracterização do atributo Identificador de um serviço Domus.

5.8.2.2 Atributos de Gestão de Carga

Entre outros atributos, os mecanismos de balanceamento dinâmico do cluster Domus (ver capítulo 6) recorrem a i) atributos estáticos de caracterização de certas *capacidades* dos nós computacionais (ver secção 6.3.1) e ii) atributos dinâmicos que veiculam as taxas de utilização de recursos associados às capacidades referidas (ver secção 6.3.2). As tabelas 5.8 e 5.9 definem e caracterizam os atributos correspondentes às *capacidades* e *utilizações*.

[35] Tal como referido para o identificador do supervisor, esta associação não implica que o identificador do serviço Domus se concretize com base num endereço (MAC, IP, etc.) do interface. Numa implementação, o que é preciso é garantir que existe um interface de rede cujo endereço pode ser deduzido a partir do identificador do serviço Domus. A maneira mais fácil de o conseguir é, obviamente, fazer coincidir os dois identificadores (o do serviço e o do interface), o que acontece, aliás, no protótipo descrito no capítulo 7.

As *capacidades* poderão ser dadas a conhecer ao supervisor através de parâmetros da biblioteca Domus (definição externa), na adição de um serviço ao cluster Domus; alternativamente, o supervisor poderá recorrer a outras fontes para a sua obtenção (definição interna), como a interrogação de um sistema global de caracterização do *cluster*. As *utilizações* (de definição interna) são originalmente produzidas nos nós computacionais, assumindo-se que o supervisor dispõe de mecanismos para tomar conhecimento delas[36].

atributo	:	`attr_node_crouting` - capacidade de encaminhamento
estrutura	:	lista de pares <algoritmo de localização, capacidade de encaminhamento>
notação	:	$\mathcal{C}^{e}(a)$ (capacidade de encaminhamento com o algoritmo a)

atributo	:	`attr_node_caccess` - capacidade de acesso
estrutura	:	lista de pares <tecnologia de armazenamento, capacidade de acesso>
notação	:	$\mathcal{C}^{a}(t)$ (capacidade de acesso a repositórios de tecnologia t)

atributo	:	`attr_node_cnet` - máxima largura de banda útil do "interface Domus" (i)
notação	:	$\mathcal{C}(net)$ (ou $\mathcal{C}(i)$)

atributo	:	`attr_node_cram` - capacidade (espaço total) de memória RAM
notação	:	$\mathcal{C}(ram)$

atributo	:	`attr_node_cdisk` - capacidade (espaço total) da "partição Domus"
notação	:	$\mathcal{C}(disk)$

Observações:
a) para todos estes atributos, a definição é "constante, interna ou externa"
b) "partição Domus" é a partição de disco associada ao subsistema de armazenamento

Tabela 5.8: Caracterização dos atributos de Capacidades dos Nós Computacionais.

5.8.3 Atributos das DHTs

5.8.3.1 Identificador

Num cluster Domus, as DHTs são nomeadas por identificadores únicos, definidos pelas aplicações Domus na criação das DHTs, cabendo ao supervisor verificar a sua unicidade. A tabela 5.10 caracteriza o atributo `attr_dht_id`, correspondente a esse identificador.

5.8.3.2 Função de Hash

A *eficácia* das funções de hash varia em função do tipo de dados a que se aplicam (ver secção B.1.2). Logo, faz sentido permitir a escolha da função a utilizar pelas DHTs, de entre um conjunto de funções suportadas por uma realização da arquitectura Domus. A tabela 5.11 caracteriza o atributo `attr_dht_fh`, correspondente à função de hash.

[36] Os mecanismos para tal dependem da implementação (ver o capítulo 7 para uma abordagem possível).

atributo	:	`attr_node_ucpu` - utilização (agregada) da(s) CPU(s)
notação	:	$\mathcal{U}(cpu)$

atributo	:	`attr_node_uiodisk` - utilização (actividade E/S) do disco com a "partição Domus"
notação	:	$\mathcal{U}(iodisk)$

atributo	:	`attr_node_unet` - utilização (largura de banda consumida) do "interface Domus"
notação	:	$\mathcal{U}(net)$

atributo	:	`attr_node_uram` - utilização (espaço consumido) da memória RAM
notação	:	$\mathcal{U}(ram)$

atributo	:	`attr_node_udisk` - utilização (espaço consumido) da "partição Domus"
notação	:	$\mathcal{U}(disk)$

Observações:
a) para todos estes atributos, a definição é "variável e interna"

Tabela 5.9: Caracterização dos atributos de Utilizações dos Nós Computacionais.

atributo	:	`attr_dht_id` (identificador)
notação formal	:	d
tipo de definição	:	constante, externa (obrigatória)

Tabela 5.10: Caracterização do atributo Identificador de uma DHT Domus.

atributo	:	`attr_dht_fh` (função de hash)
notação formal	:	f
tipo de definição	:	constante, interna ou externa
valor por omissão	:	(uma função genérica)

Tabela 5.11: Caracterização do atributo Função de Hash de uma DHT Domus.

5.8.3.3 Número Mínimo de Entradas por Nó Virtual

No contexto do modelo M4 de distribuição heterogénea, a qualidade da distribuição é medida por uma grandeza do tipo "desvio-padrão-relativo" (ver fórmula 3.16). De uma forma indirecta, a qualidade da distribuição é afectada pelo parâmetro $\mathcal{H}_{min}(v)$ do modelo (número mínimo de entradas por nó virtual): quanto mais elevado for o valor de $\mathcal{H}_{min}(v)$ (com a restrição de que deve ser potência de 2), maior será a qualidade da distribuição. A tabela 5.12 caracteriza o atributo `attr_dht_nmev`, correspondente ao parâmetro $\mathcal{H}_{min}(v)$.

5.8.3.4 Restrições à Distribuição

Os atributos desta categoria permitem definir restrições a) à identidade ou número dos nós/serviços de uma DHT, b) bem como ao seu número de nós virtuais[37]. Desta forma, uma aplicação Domus (ou o utilizador associado) tem um certo nível de controlo sobre a alocação de nós/serviços a uma DHT e sobre o seu paralelismo/distribuição potencial.

[37]Neste contexto, é útil ter presente as semânticas desse número, já apresentadas na secção 5.5.1.1.

atributo	:	`attr_dht_nmev` (número mínimo de entradas por nó virtual)
notação formal	:	$\mathcal{H}_{min}(v)$
tipo definição	:	constante, interna ou externa
valor por omissão	:	8 (valor de referência para $\mathcal{H}_{min}(n)$ em M2 e $\mathcal{H}_{min}(v)$ em M4)
invariantes associados	:	$\mathcal{I}_1$ - "$\mathcal{H}_{min}(v)$ é potência de 2" $\wedge$ $\mathcal{H}_{min}(v) > 0$

Tabela 5.12: Caracterização do atributo $\mathcal{H}_{min}(v)$ de uma DHT Domus.

Através destes atributos é possível a1) limitar o endereçamento/armazenamento a certos conjuntos de nós, a2) impor um grau mínimo de paralelismo/distribuição *real* no suporte à DHT, definindo um número mínimo de nós computacionais de endereçamento/armazenamento; b1) impor um grau mínimo de paralelismo/distribuição *potencial* no suporte à DHT, definindo um número mínimo de nós virtuais de endereçamento/armazenamento.

Um exemplo que demonstra o interesse da possibilidade a1) é dado pela situação em que um subconjunto de nós do *cluster* dispõe de uma ligação a um dispositivo SAN, pretendendo-se que as funções de armazenamento da DHT sejam adstritas a esses nós; outra hipótese é criar uma DHT não persistente, especificamente sobre os nós com maior capacidade de RAM. A possibilidade de se exercer um controlo deste tipo é também útil quando existe conhecimento adicional sobre as condições actuais/futuras do *cluster*; por exemplo, sabendo-se da necessidade de manutenção de certos nós, ou da execução iminente de tarefas que vão sobrecarregar os recursos associados ao endereçamento e/ou armazenamento, então pode optar-se pela exclusão desses nós de uma DHT em criação.

Como referido na secção 5.5.1.1, a possibilidade a2) tem mais a ver com requisitos de desempenho e a b1) com requisitos de armazenamento; por exemplo, quando se tem uma ideia aproximada do espaço que vai ser necessário para suportar a totalidade dos registos de uma DHT, e sendo possível associar um certo espaço a um nó virtual de armazenamento (ver o atributo *limiar de armazenamento*, na secção 5.8.3.9), então a definição de um certo número inicial de nós virtuais de armazenamento equivale a uma declaração dos requisitos de armazenamento da DHT, sem a satisfação dos quais a DHT não chega a ser criada.

Na prática, os atributos desta categoria participam em invariantes que determinam restrições que a DHT deve respeitar, quer durante a sua *criação*, quer durante a sua *operação*. Neste contexto, introduzimos os conceitos de *domínio* e *universo* de uma DHT. Assim, o *domínio* de uma DHT é o conjunto de nós actualmente utilizados para a suportar, donde:

- $N(d)$ representa o *domínio* da DHT d;
- $N^{\mathtt{e}}(d)$ representa o *domínio de endereçamento* da DHT d;
- $N^{\mathtt{a}}(d)$ representa o *domínio de armazenamento* da DHT d.

O *universo* de uma DHT é o maior conjunto de nós a que é admissível recorrer para a suportar, ou seja, representa o limite máximo de expansão do *domínio*[38] da DHT. Assim:

[38] O conceito de *universo* que aqui introduzimos, é oriundo da Teoria de Conjuntos, onde representa o maior de todos os *superconjuntos* de um conjunto (o conceito de *superconjunto* é dual de *subconjunto*).

atributo	:	`attr_dht_ue` (universo de endereçamento)
notação formal	:	$N^{\mathsf{e}}_{max}(d)$
valor por omissão	:	$N(S)$ (conjunto de todos os nós actuais do cluster Domus)
invariantes associados	:	$\mathcal{I}_5$, $\mathcal{I}_{10}$, $\mathcal{I}_{20}$

atributo	:	`attr_dht_nmne` (número mínimo de nós de endereçamento)
notação formal	:	$\#N^{\mathsf{e}}_{min}(d)$ ou $\mathcal{N}^{\mathsf{e}}_{min}(d)$
valor por omissão	:	1
invariantes associados	:	$\mathcal{I}_{10}$, $\mathcal{I}_{30}$

atributo	:	`attr_dht_ua` (universo de armazenamento)
notação formal	:	$N^{\mathsf{a}}_{max}(d)$
valor por omissão	:	$N(S)$ (conjunto de todos os nós actuais do cluster Domus)
invariantes associados	:	$\mathcal{I}_6$, $\mathcal{I}_{11}$, $\mathcal{I}_{21}$

atributo	:	`attr_dht_nmna` (número mínimo de nós de armazenamento)
notação formal	:	$\#N^{\mathsf{a}}_{min}(d)$ ou $\mathcal{N}^{\mathsf{a}}_{min}(d)$
valor por omissão	:	1
invariantes associados	:	$\mathcal{I}_{11}$, $\mathcal{I}_{31}$

atributo	:	`attr_dht_nmve` (número mínimo de nós virtuais de endereçamento)
notação formal	:	$\#V^{\mathsf{e}}_{min}(d)$ ou $\mathcal{V}^{\mathsf{e}}_{min}(d)$
valor por omissão	:	`attr_dht_nmne` (número mínimo de nós de endereçamento)
invariantes associados	:	$\mathcal{I}_{60}$, $\mathcal{I}_{61}$

atributo	:	`attr_dht_nmva` (número mínimo de nós virtuais de armazenamento)
notação formal	:	$\#V^{\mathsf{a}}_{min}(d)$ ou $\mathcal{V}^{\mathsf{a}}_{min}(d)$
valor por omissão	:	`attr_dht_nmna` (número mínimo de nós de armazenamento)
invariantes associados	:	$\mathcal{I}_{60}$, $\mathcal{I}_{61}$

Observações:
a) para todos estes atributos, a definição é "constante, interna ou externa"

Tabela 5.13: Caracterização dos atributos de Restrições à Distribuição de uma DHT Domus.

- $N_{max}(d)$ representa o *universo* da DHT d;
- $N^{\mathsf{e}}_{max}(d)$ representa o *universo de endereçamento* da DHT d;
- $N^{\mathsf{a}}_{max}(d)$ representa o *universo de armazenamento* da DHT d;
- $N(d) \subseteq N_{max}(d)$, $N_{max}(d) = N^{\mathsf{e}}_{max}(d) \cup N^{\mathsf{a}}_{max}(d)$ e $N_{max}(d) \subseteq N(S)$.

A tabela 5.13 sistematiza a caracterização dos atributos de Restrições à Distribuição de uma DHT, e a tabela 5.14 apresenta os invariantes em que esses atributos participam.

5.8.3.5 Tabelas de Distribuição

O modelo M4 apoia-se numa *tabela de distribuição* (TD) que regista, para cada nó n de uma DHT, o seu número de entradas, $\mathcal{H}(n)$, e de nós virtuais, $\mathcal{V}(n)$ – rever secção 3.5.5.

$\mathcal{I}_5$	:	$N^{\mathsf{e}}_{max}(d) \subseteq N(S)$
		– o universo de endereçamento, $N^{\mathsf{e}}_{max}(d)$, é restrito aos nós do cluster Domus, $N(S)$
$\mathcal{I}_6$	:	$N^{\mathsf{a}}_{max}(d) \subseteq N(S)$
		– o universo de armazenamento, $N^{\mathsf{a}}_{max}(d)$, é restrito aos nós do cluster Domus, $N(S)$

$\mathcal{I}_{10}$	:	$\#N^{\mathsf{e}}_{max}(d) \geq \#N^{\mathsf{e}}_{min}(d) \geq 1$
		– o universo de endereçamento, $N^{\mathsf{e}}_{max}(d)$, deve ter pelo menos $\#N^{\mathsf{e}}_{min}(d)$ nós
		– o universo de endereçamento, $N^{\mathsf{e}}_{max}(d)$, não pode ser vazio
$\mathcal{I}_{11}$	:	$\#N^{\mathsf{a}}_{max}(d) \geq \#N^{\mathsf{a}}_{min}(d) \geq 1$
		– o universo de armazenamento, $N^{\mathsf{a}}_{max}(d)$, deve ter pelo menos $\#N^{\mathsf{a}}_{min}(d)$ nós
		– o universo de armazenamento, $N^{\mathsf{a}}_{max}(d)$, não pode ser vazio

$\mathcal{I}_{20}$	:	$N^{\mathsf{e}}(d) \subseteq N^{\mathsf{e}}_{max}(d)$
		– o domínio de endereçamento, $N^{\mathsf{e}}(d)$, é restrito ao universo de endereçamento
$\mathcal{I}_{21}$	:	$N^{\mathsf{a}}(d) \subseteq N^{\mathsf{a}}_{max}(d)$
		– o domínio de armazenamento, $N^{\mathsf{a}}(d)$, é restrito ao universo de armazenamento

$\mathcal{I}_{30}$	:	$\#N^{\mathsf{e}}(d) \geq \#N^{\mathsf{e}}_{min}(d) \geq 1$
		– o domínio de endereçamento, $N^{\mathsf{e}}(d)$, deve ter pelo menos $\#N^{\mathsf{e}}_{min}(d)$ nós
		– o domínio de endereçamento, $N^{\mathsf{e}}(d)$, não pode ser vazio
$\mathcal{I}_{31}$	:	$\#N^{\mathsf{a}}(d) \geq \#N^{\mathsf{a}}_{min}(d) \geq 1$
		– o domínio de armazenamento, $N^{\mathsf{a}}(d)$, deve ter pelo menos $\#N^{\mathsf{a}}_{min}(d)$ nós
		– o domínio de armazenamento, $N^{\mathsf{a}}(d)$, não pode ser vazio

$\mathcal{I}_{60}$	:	$\mathcal{V}^{\mathsf{e}}(d) \geq \mathcal{V}^{\mathsf{e}}_{min}(d) \geq \#N^{\mathsf{e}}(d)_{min}$
		– deve haver nós virtuais suficientes para $\#N^{\mathsf{e}}(d)_{min}$ nós de endereçamento
$\mathcal{I}_{61}$	:	$\mathcal{V}^{\mathsf{a}}(d) \geq \mathcal{V}^{\mathsf{a}}_{min}(d) \geq \#N^{\mathsf{a}}(d)_{min}$
		– deve haver nós virtuais suficientes para $\#N^{\mathsf{a}}(d)_{min}$ nós de armazenamento

Tabela 5.14: Invariantes associados dos atributos de Restrições à Distribuição.

A dissociação entre endereçamento e armazenamento tem como consequência a necessidade de aplicar o modelo M4 em separado para essas funções. Desta forma, para cada DHT d, é necessário operar com duas tabelas: 1) uma *tabela de distribuição do endereçamento* ($TD^{\mathsf{e}}(d)$) e 2) uma *tabela de distribuição do armazenamento* ($TD^{\mathsf{a}}(d)$). A primeira guarda tuplos $<n, \mathcal{V}^{\mathsf{e}}(d,n), \mathcal{H}^{\mathsf{e}}(d,n)>$ e a segunda guarda tuplos $<n, \mathcal{V}^{\mathsf{a}}(d,n), \mathcal{H}^{\mathsf{a}}(d,n)>$.

Estas tabelas são geridas pelo supervisor, representando os atributos de uma DHT que sumarizam o resultado da distribuição pesada das responsabilidades de endereçamento e de armazenamento. Essa distribuição pode basear-se num número global de nós virtuais de endereçamento, $\mathcal{V}^{\mathsf{e}}(d)$, diferente do número global de nós virtuais de armazenamento, $\mathcal{V}^{\mathsf{a}}(d)$; o número global de entradas a repartir pelas duas espécies de nós virtuais é $\mathcal{H}(d)$.

Ao contrário dos atributos já apresentados, as tabelas $TD^{\mathsf{e}}(d)$ e $TD^{\mathsf{a}}(d)$ não são definíveis directamente; assim, estas representam atributos *internos* sendo a sua definição influenciada pelos parâmetros/atributos de Qualidade da Distribuição e de Restrições à Distribuição; essa influência é visível nos invariantes associados, fornecidos na tabela 5.16.

As tabelas 5.15 e 5.16 caracterizam as Tabelas de Distribuição e invariantes associados.

atributo	:	`attr_dht_tde` (tabela de distribuição do endereçamento)
notação formal	:	$TD^{e}(d)$
tipo de definição	:	variável, interna
invariantes associados	:	$\mathcal{I}_{40}$, $\mathcal{I}_{42}$, $\mathcal{I}_{50}$, $\mathcal{I}_{52}$

atributo	:	`attr_dht_tda` (tabela de distribuição do armazenamento)
notação formal	:	$TD^{a}(d)$
tipo de definição	:	variável, interna
invariantes associados	:	$\mathcal{I}_{41}$, $\mathcal{I}_{43}$, $\mathcal{I}_{51}$, $\mathcal{I}_{53}$

Tabela 5.15: Caracterização das Tabelas de Distribuição de uma DHT Domus.

$\mathcal{I}_{40}$	:	$\mathcal{V}^{e}(d,n) > 0,\ \forall n \in TD^{e}(d)$
		– se n é nó de endereçamento, o seu número de nós virtuais de endereçamento é positivo
$\mathcal{I}_{41}$	:	$\mathcal{V}^{a}(d,n) > 0,\ \forall n \in TD^{a}(d)$
		– se n é nó de armazenamento, o seu número de nós virtuais de armazena. é positivo
$\mathcal{I}_{42}$	:	$\mathcal{H}^{e}(d,n) > 0,\ \forall n \in TD^{e}(d)$
		– se n é nó de endereçamento, o número de entradas endereçadas por n é positivo
$\mathcal{I}_{43}$	:	$\mathcal{H}^{a}(d,n) > 0,\ \forall n \in TD^{a}(d)$
		– se n é nó de armazenamento, o número de entradas armazenadas por n é positivo
$\mathcal{I}_{50}$	:	$\sum_{n \in\ TD^{e}(d)} \mathcal{V}^{e}(d,n) == \mathcal{V}^{e}(d)$
		– não pode haver nós virtuais de endereçamento por atribuir
$\mathcal{I}_{51}$	:	$\sum_{n \in\ TD^{a}(d)} \mathcal{V}^{a}(d,n) == \mathcal{V}^{a}(d)$
		– não pode haver nós virtuais de armazenamento por atribuir
$\mathcal{I}_{52}$	:	$\sum_{n \in\ TD^{e}(d)} \mathcal{H}^{e}(d,n) == \mathcal{H}(d)$
		– não pode haver entradas da DHT por endereçar
$\mathcal{I}_{53}$	:	$\sum_{n \in\ TD^{a}(d)} \mathcal{H}^{a}(d,n) == \mathcal{H}(d)$
		– não pode haver entradas da DHT por armazenar

Tabela 5.16: Invariantes associados às Tabelas de Distribuição de uma DHT Domus.

5.8.3.6 Atributos da Localização Distribuída

Os atributos desta categoria permitem influenciar a forma como se realiza, num cluster Domus, a localização distribuída de uma DHT, com base nas contribuições do capítulo 4.

Assim, o atributo `attr_dht_pld` permite especificar uma *política de localização distribuída*, que pode ser `global` ou `local`; no primeiro caso, todos os nós/serviços de endereçamento da DHT aplicarão o mesmo *algoritmo global de localização distribuída*, definido através do atributo `attr_dht_agld` (os valores possíveis deste atributo são baseados nas designações atribuídas aos algoritmos de localização discutidos no capítulo 4); no segundo caso, cada nó/serviço de endereçamento poderá comutar entre algoritmos com diferentes exigências de processamento, numa óptica de balanceamento local de recursos (rever secção 5.6.7.2). A tabela 5.17 sintetiza a caracterização dos atributos `attr_dht_pld` e `attr_dht_agld`.

atributo	:	`attr_dht_pld` (política de localização distribuída)
tipo de definição	:	constante, interna ou externa
valor por omissão	:	`global`
valores possíveis	:	{ `global`, `local` }

atributo	:	`attr_dht_agld` (algoritmo global de localização distribuída)
tipo de definição	:	constante, interna ou externa
valor por omissão	:	`EA-E-L` (melhor compromisso entre consumo de CPU e distância média)
valores possíveis	:	{ `EC`, `EM`, `EA-E-1`, `EA-E-L`, `EA-all` }

Tabela 5.17: Caracterização dos Atributos da Localização Distribuída de uma DHT.

5.8.3.7 Atributos dos Repositórios

Como referido na secção 5.6.6.2, um serviço Domus suporta *repositórios* de registos, baseados numa certa combinação de *plataforma de armazenamento* e *meio de armazenamento*. Além disso, deve existir a possibilidade de especificar a *granularidade do repositório*, de acordo com a semântica também definida na mesma secção. Estas qualidades do repositório de uma DHT são dadas pelos atributos `attr_dht_pa`, `attr_dht_ma` e `attr_dht_gr` – ver tabela 5.18; os valores admissíveis destes atributos dependem da implementação.

atributo	:	`attr_dht_pa` (plataforma de armazenamento)
tipo de definição	:	constante, interna ou externa
valor por omissão	:	(uma plataforma sobre `ram`; dependente da implementação)
valores possíveis	:	(dependentes da implementação)

atributo	:	`attr_dht_ma` (meio de armazenamento)
tipo de definição	:	constante, interna ou externa
valor por omissão	:	`ram`
valores possíveis	:	pelo menos `ram` e `disco` (outros dependem da implementação)

atributo	:	`attr_dht_gr` (granularidade do repositório)
tipo de definição	:	constante, interna ou externa
valor por omissão	:	`mínima`
valores possíveis	:	{ `mínima`, `média`, `máxima` }

Tabela 5.18: Caracterização dos Atributos dos Repositórios de uma DHT Domus.

5.8.3.8 Política de Evolução da DHT

Apesar do desenho da arquitectura Domus ser orientado ao suporte de DHTs *dinâmicas* (em que o *número* e o *posicionamento* das entradas pode evoluir), poderá ter interesse, em certas situações, instanciar DHTs de tipo *estático* (em que o *número* e o *posicionamento* das entradas, uma vez estabelecidos na criação, é fixado); mais uma vez, uma decisão deste tipo poderá ser condicionada por conhecimento adicional sobre o estado actual ou futuro do *cluster*, ou por requisitos específicos das aplicações clientes das DHTs; por exemplo, recorrendo aos parâmetros que permitem delimitar o *universo* da DHT, é possível confiná-la a uma partição exclusiva do *cluster*; se nessa partição não for permitida a execução de outras tarefas, poderá então dispensar-se o balanceamento dinâmico da DHT (independentemente de se ter realizado uma distribuição inicial pesada, da DHT, nessa partição).

Define-se assim um atributo `attr_dht_pe`, que suporta a definição de uma *política de evolução* da DHT, `estática` ou `dinâmica`, de acordo com a semântica que acabamos de descrever. A tabela 5.19 sistematiza a caracterização do atributo `attr_dht_pe`.

atributo	:	`attr_dht_pe` (política de evolução)
tipo de definição	:	constante, interna ou externa
valor por omissão	:	`dinâmica`
valores possíveis	:	{ `dinâmica`, `estática` }

Tabela 5.19: Caracterização do atributo Política de Evolução de uma DHT Domus.

5.8.3.9 Atributos de Gestão de Carga

Para além dos atributos dos nós, introduzidos na secção 5.8.2.2, os mecanismos de balanceamento do capítulo 6 necessitam, para a sua operação, do conjunto de atributos de caracterização das DHTs apresentados na tabela 5.20. Estes atributos, em conjunto com os primeiros, permitem gerir os vários tipos de carga gerada na operação das DHTs activas.

Assim, a gestão da *carga de endereçamento* de uma DHT (relacionada com a frequência de operações de encaminhamento associadas à DHT, nos seus serviços de endereçamento) implica a comparação de uma "taxa global (média) de operações de encaminhamento da DHT", com um "limiar de endereçamento" dado por um atributo `attr_dht_le` (ver secção 6.6.1). Adicionalmente, a gestão da *carga de acesso* (associada à frequência de operações de acesso aos repositórios de uma DHT, nos seus serviços de armazenamento) implica o confronto de uma "taxa média (global) de operações de acesso à DHT", com um "limiar de acesso" dado por um atributo `attr_dht_la` (ver secção 6.6.2). Finalmente, a gestão da *carga de armazenamento* (relacionada com o consumo do *meio de armazenamento* definido para a DHT, nos seus serviços de armazenamento) implica a comparação de uma "quantidade média (global) consumida, do *meio de armazenamento* da DHT, por cada nó virtual de armazenamento", com um "limiar de armazenamento" fornecido por um atributo `attr_dht_lm` (ver secção 6.6.4). Note-se que a influência dos três atributos de gestão de carga será nula se a *política de evolução* da DHT for `estática` (rever secção anterior).

5.9 Aplicações Domus

Uma aplicação Domus é uma aplicação que interactua com um ou mais clusters Domus. A interacção pode realizar-se i) com fins *administrativos* ou, ii) mais frequentemente, com o objectivo de *aceder* aos dicionários distribuídos implementados pelas DHTs Domus.

As interacções de *administração* estão associadas às operações de supervisão do cluster Domus que descrevemos na secção 5.7: 1) criação, destruição, desactivação e reactivação de um cluster Domus; 2) adição, remoção, desactivação e reactivação de serviços Domus específicos; 3) criação, destruição, desactivação e reactivação de DHTs específicas. Estas operações são solicitadas directamente ao supervisor, que se encarrega da sua prossecução.

As interacções de *acesso* solicitam as operações usuais sobre dicionários: inserção, consulta e remoção de registos de esquema $<chave, dados>$; com DHTs Domus, estas operações são precedidas, obrigatoriamente, da descoberta do serviço de armazenamento dos registos.

atributo	:	`attr_dht_le` (limiar de endereçamento)
notação formal	:	$\lambda^{\mathtt{e}}_{\tau}$
tipo de definição	:	constante, interna ou externa
valor por omissão	:	1.0
valores possíveis	:	]0.0,1.0]

atributo	:	`attr_dht_la` (limiar de acesso)
notação formal	:	$\lambda^{\mathtt{a}}_{\tau}$
tipo de definição	:	constante, interna ou externa
valor por omissão	:	1.0
valores possíveis	:	]0.0,1.0]

atributo	:	`attr_dht_lm` (limiar de armazenamento)
notação formal	:	$\mathcal{Q}^{\mathtt{a}}_{\tau}$
tipo de definição	:	constante, interna ou externa
valor por omissão	:	(dimensão da página virtual do Sistema Operativo)
valores possíveis	:	(inteiros positivos)

Tabela 5.20: Caracterização dos Atributos de Gestão de Carga de uma DHT Domus.

Tipicamente, um conjunto de interacções é precedido/sucedido de operações de associação/desassociação (*open*/*close*) a/de um cluster Domus ou a/de uma DHT, c.f. o caso; estas operações possibilitam a gestão de múltiplos *contextos de interacção*, permitindo a uma aplicação Domus lidar, de forma consistente, com múltiplas DHTs e/ou clusters Domus.

5.9.1 Biblioteca Domus

As interacções de uma aplicação com um cluster Domus processam-se através de uma biblioteca de funcionalidades *administrativas* e de *acesso*; dependendo do tipo de funcionalidade, a biblioteca comunica com serviços supervisores ou regulares; para o efeito, o nó onde a aplicação executa carece do pacote de *serviços básicos* previsto na arquitectura.

A figura 5.10 apresenta um visão abstracta da biblioteca Domus, de suporte à sua descrição.

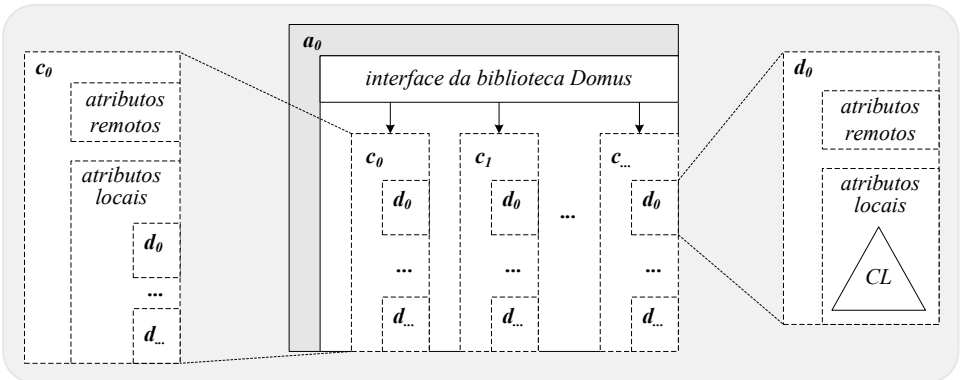


Figura 5.10: Visão Abstracta da Biblioteca Domus: Contextos de Interacção.

5.9.1.1 Contextos de Interacção

A figura 5.10 ilustra o suporte da biblioteca à interacção com vários clusters Domus e, para cada um deles, com várias DHTs. Um contexto de interacção com um cluster Domus é denotado por $c_{...}$, notação formal para o atributo Identificador de um cluster Domus (rever secção 5.8.1.1); analogamente, um contexto de interacção com uma DHT é denotado por $d_{...}$, notação formal para o atributo Identificador de uma DHT (rever secção 5.8.3.1).

5.9.1.2 Atributos de Interacção

A mesma figura revela atributos necessários, em cada contexto, à interacção com as entidades em causa. Esses atributos podem ser de dois tipos: i) atributos *locais* ou ii) atributos *remotos*. Os atributos *remotos* mimetizam certos atributos mantidos pelos supervisores, incluindo atributos *internos* e *externos* (rever secção 5.7). Os atributos *locais* são apenas instanciados na biblioteca (não são exportáveis para os supervisores), podendo ser valorados de forma diferente pelas aplicações clientes; na figura representam-se apenas atributos *locais* de tipo *externo*, ou seja, directamente definíveis por parâmetros da biblioteca Domus.

A interacção com um cluster Domus carece de atributos (*remotos*) que garantem à aplicação cliente que o seu interlocutor é o cluster Domus pretendido: Identificador do cluster Domus (`attr_cluster_id`) e Identificador do Supervisor (`attr_cluster_supervisor_id`).

Na interacção com uma DHT, os atributos *remotos* necessários são: i) o Identificador da DHT (`attr_dht_id`), ii) a Função de Hash (`attr_dht_fh`), iii) as Tabelas de Distribuição do Endereçamento e Armazenamento (`attr_dht_tde` e `attr_dht_tda`) e iv) o Algoritmo Global de Localização Distribuída (`attr_dht_agld`). Estes atributos cumprem a seguinte função: i) `attr_dht_id` desambigua as transacções com o supervisor; ii) `attr_dht_fh` identifica a função de hash a aplicar à chave dos registos para a localização do seu serviço de armazenamento; iii,iv) `attr_dht_tde`, `attr_dht_tda` e `attr_dht_agld` são usados na operacionalização de uma certa *estratégia de localização* (c.f. descrito na secção 5.9.1.3). Os atributos *locais* incluem: i) Estratégia de Localização (`attr_dht_el`), ii) Limiar de Desacerto c/ Método Directo (`attr_dht_ldmd`) e iii) Dimensão da Cache de Localização (`attr_dht_dcl`); estes atributos regulam, em conjunto, uma certa *estratégia de localização*. A tabela 5.21 sistematiza a caracterização dos Atributos Locais de Interacção com DHTs.

5.9.1.3 Estratégias de Localização

Uma *estratégia de localização* corresponde a uma certa sequência ordenada dos seguintes *métodos de localização*: i) *directo* (mD), ii) *baseado em cache* (mC) e iii) *aleatório* (mA).

Definida a estratégia (sequência), uma localização inicia-se pela exploração do primeiro método previsto; se esse método for infrutífero, explora-se o próximo método da sequência, e assim sucessivamente. De todas as sequências possíveis, apenas são válidas as seguintes: <mD,mA>, <mD,mC,mA>, <mC,mA> e <mA>. A ordenação anterior representa, em geral, uma progressão de métodos mais rápidos e menos precisos, para métodos mais lentos e mais precisos. No que se segue, descreve-se o princípio de funcionamento de cada método.

atributo	:	`attr_dht_el` (estratégia de localização)
tipo de definição:	:	constante, interna ou externa
valor por omissão	:	<mD,mC,mA>
valores possíveis	:	{<mD,mC,mA>, <mC,mA>, <mA> }

atributo	:	`attr_dht_ldmd` (limiar de desacerto com o método directo)
tipo de definição:	:	constante, interna ou externa
valor por omissão	:	50%
valores possíveis	:	]0%,100%[

designação	:	`attr_dht_dcl` (dimensão da *cache de localização*)
tipo de definição:	:	constante, interna ou externa
valor por omissão	:	1024 entradas
valores possíveis	:	(números inteiros positivos)

Tabela 5.21: Caracterização dos Atributos Locais de Interacção com DHTs.

Método Directo (ou 1-HOP) Este método procura explorar a possibilidade de a DHT ainda não ter sido redistribuída desde a sua criação (ou, quando muito, de essa redistribuição ter sido pouco significativa). Dessa forma, i) conhecendo-se o critério usado para definir a distribuição inicial (*i.e.*, *contígua*, *rotativa* ou outra) e ii) conhecendo-se as Tabelas de Distribuição do Endereçamento e do Armazenamento, iii) é possível deduzir a localização de qualquer entrada, pelo que apenas um salto de rede separará o cliente, do serviço Domus correcto (razão pela qual o método directo também se pode designar de 1-HOP). Recorde-se que a possibilidade de explorar um método deste tipo tinha sido prevista aquando da discussão da problemática da Atribuição Inicial de Entradas, na secção 4.3.1.

A utilidade de uma cópia local das Tabelas de Distribuição do Endereçamento e do Armazenamento da DHT é assim justificada, pois viabiliza a aplicação do método directo. Todavia, para que essa aplicação seja convenientemente gerida, é necessário um atributo capaz de desactivar o método se a taxa de desacerto for demasiado alta; trata-se precisamente do atributo "Limiar de Desacerto com o Método Directo", de valor percentual.

Método baseado em Cache de Localização A estrutura e o modo de funcionamento de uma *cache de localização* (dependente, basicamente, de localização distribuída) foram amplamente discutidos na secção 5.6.5.3. Resta acrescentar que, estando inicialmente vazia, a *cache* será inicializada através do método aleatório (ver a seguir); este é também o método prosseguido automaticamente sempre que a *cache* produz um erro de localização e é necessário corrigir o seu conteúdo; um atributo de Dimensão da Cache de Localização permite às aplicações configurar a capacidade da *cache* (em número de registos).

Método Aleatório Tal como o método anterior, também este recorre a localização distribuída, com a diferença de que o serviço inicial de uma cadeia de localização é escolhido de forma aleatória; essa escolha pode ser feita tirando partido da cópia local da Tabela de Distribuição do Endereçamento ou, no caso de esta já ser demasiado obsoleta, por solicitação directa ao supervisor; em todo o caso, a ideia é efectuar uma escolha aleatória mas pesada, pelo número de entradas, de forma a privilegiar os serviços com mais tabelas de encaminhamento (logo com maior probabilidade de a cadeia aí iniciada ser mais curta).

Pelo facto de poder sobrecarregar o supervisor, o método aleatório faz sobressair as vantagens da *cache de localização*, como abordagem adaptativa à evolução da DHT (ao contrário do método directo) e que coloca a maior parte da carga inicial de localização no cliente.

5.10 Relação com Outras Abordagens

A dissociação Endereçamento-Armazenamento, quase ausente em DHTs de primeira geração, e mais frequente em DHTs de segunda geração (rever secção a 2.9), é na arquitectura Domus levada ao extremo, atravessando todos os componentes e subsistemas, de forma que os vários serviços Domus podem desempenhar as funções de endereçamento e armazenamento em qualquer combinação, para qualquer DHT e dinamicamente; o objectivo da dissociação é duplo: por um lado, facilitar uma melhor utilização dos recursos do *cluster* e, por outro, assegurar certos níveis mínimos de serviço, das DHTs, aos seus clientes.

Embora na arquitectura Domus se assuma o uso de grafos Chord completos para localização distribuída (na sua formulação do capítulo 4), a verdade é que não há nenhuma restrição arquitectural à utilização de outra abordagem, desde que compatível com os modelos de particionamento do capítulo 3 (como se demonstrou ser o caso dos grafos De-Bruijn binários). Neste sentido, e tendo também em conta a dissociação Endereçamento-Armazenamento assumida, é possível encontrar algum paralelismo com a arquitectura Dipsea [Man04], onde *particionamento*, *localização* e *armazenamento* são independentes.

A visão da "DHT como serviço" (rever secção 2.10), corresponde ao paradigma de operação prosseguido pela arquitectura Domus. Neste sentido, a visão alternativa da "DHT como biblioteca", não se deve confundir com o conceito de "biblioteca de acesso à DHT" previsto pela arquitectura Domus; neste último caso, a biblioteca é utilizada pelas aplicações para interactuarem com as DHTs, assentes em serviços distribuídos (ou seja, não é a aplicação, mas sim a biblioteca de acesso à DHT, que determina quais o(s) serviço(s) a contactar).

A co-operação de DHTs suportada pela arquitectura Domus traduz-se na co-existência de múltiplas DHTs independentes e sem interacção. Esta aproximação é diferente de outras abordagens (rever secção 2.11), onde i) se embebem [HJS+03] / virtualizam [KRRS04] múltiplas DHTs numa DHT de base, ou ii) se permitem operações de fusão/composição, subdivisão/decomposição ou de interligação (*bridging*) [HGRW06, COJ+06, CJO+07].

A existência de um *supervisor* na arquitectura Domus, responsável pela centralização de certas decisões, encontra também algum paralelismo noutras abordagens. Riley et al. [RS04], por exemplo, definem "sistema/DHT P2P supervisionado(a)" como "aquele(a) em que a composição do(a) sistema/DHT é gerida por um supervisor, sendo o resto das actividades independentes dele"; neste contexto, propõem a aplicação da sua abordagem (SPON-DHT) a ambientes Grid, onde a centralização de certas funções é admissível.

5.11 Epílogo

O próximo capítulo é inteiramente dedicado à descrição aprofundada dos mecanismos de balanceamento dinâmico da arquitectura Domus. Depois, no capítulo 7, apresenta-se o *protótipo Domus*, uma implementação parcial da arquitectura; nesse capítulo apresentam-se também os resultados de uma série de testes de avaliação/exercitação do protótipo.

Capítulo 6

Gestão Dinâmica de Carga

Resumo

Neste capítulo descrevem-se, de forma aprofundada, os mecanismos de gestão dinâmica de carga previstos na arquitectura Domus: i) um mecanismo de gestão do *posicionamento* de nós virtuais, e ii) um mecanismo de gestão do *número* de nós virtuais. A descrição destes mecanismos gira em torno do papel que desempenham nos momentos fundamentais da evolução das DHTs (criação, expansão e migração). Concomitantemente, descrevem-se os atributos e as medidas em que os mecanismos se apoiam para tomar as suas decisões.

6.1 Prólogo

Este capítulo conclui a apresentação da arquitectura Domus, pela descrição aturada dos seus mecanismos de gestão de carga, introduzidos no capítulo anterior. A descrição desses mecanismos carece de uma caracterização precisa das entidades da arquitectura Domus envolvidas. Assim, uma boa parte deste capítulo é dedicada a essa caracterização, realizada à custa de certos *atributos* e *medidas*, conceitos aos quais associamos o seguinte significado:

- *atributo*: qualidade de uma entidade, definida por um par <*nome*,*valor*>; um atributo faz parte da caracterização da entidade, podendo o seu valor ser *fixo* ou *variável*;
- *medida*: grandeza determinada *a pedido* (*just-in-time*); pode ser dependente de valores de atributos, mas também pode servir para defini-los (directa ou indirectamente).

Os mecanismos em causa apenas se encontram parcialmente implementados no protótipo da arquitectura (ver capítulo 7) carecendo assim de uma validação experimental completa. Julgamos, todavia, que a descrição aqui fornecida não só constitui o ponto de apoio indispensável à implementação total, como é suficiente para uma avaliação de nível conceptual.

6.2 Mecanismos de Gestão de Carga

De acordo com a sua especificação, apresentada no capítulo anterior, a arquitectura Domus prevê dois mecanismos globais de gestão dinâmica de carga (rever secção 5.6.7.1.):

- um mecanismo de gestão do *Posicionamento* de nós virtuais (mgPv);
- um mecanismo de gestão do *Número* de nós virtuais (mgNv).

A operação do mgPv e do mgNv corresponde à execução de tarefas específicas de gestão dos nós virtuais das DHTs, com objectivos diferentes mas complementares (neste contexto, recorde-se que, na arquitectura Domus, 1) um *nó virtual* corresponde a um certo número de *hashes* e 2) a dissociação endereçamento-armazenamento implica a existência de 2a) *nós virtuais de endereçamento* e 2b) *nós virtuais de armazenamento* – rever secção 5.5) .

O mgPv controla o *posicionamento* dos nós virtuais das DHTs, assegurando a sua atribuição aos nós mais adequados. A actividade do mgPv é portanto centrada na rentabilização dos recursos dos nós do cluster Domus, tendo a capacidade de reagir a sobrecargas induzidas pelos serviços Domus ou por aplicações/serviços exógenos, que com eles co-habitam.

O mgNv gere o *número* global de nós virtuais das DHTs[1]. Assim, no âmbito do mgNv monitoriza-se a carga induzida pelas DHTs (mais especificamente, a carga média associada a cada nó virtual) e, ultrapassados certos limiares, altera-se o número global de nós virtuais. A visão do mgNv é pois centrada nas DHTs, procurando garantir-lhes certos *níveis mínimos de serviço* (*e.g.*, assegurando um grau de dispersão suficiente para não se ultrapassar uma certa carga de localização, acesso e armazenamento – ver secção 6.6).

Os mecanismos mgPv e mgNv intervêm em diferentes momentos da vida de uma DHT, de forma individual ou combinada. Durante a *criação* de uma DHT, cabe ao mgPv a definição da sua distribuição inicial. O mgPv volta a intervir durante a *expansão* de uma DHT: esta é despoletada pelo mgNv, mas caberá ao mgPv decidir do posicionamento dos novos nós virtuais. Finalmente, a *migração* de nós virtuais de uma DHT solicita novamente a intervenção do mgPv, para que se definam os nós hospedeiros dos nós virtuais a migrar.

Na operação do mgPv e mgNv, os protagonistas principais são i) o supervisor do cluster Domus e ii) o Subsistema de Balanceamento (SB) dos serviços Domus; o Subsistema de Endereçamento (SE) e o Subsistema de Armazenamento (SA) também intervêm, mas num plano secundário. Recordando o exposto na secção 5.6.7.1: a) é o SB que despoleta a *migração de nós virtuais* mas é o supervisor que decide o seu posicionamento; b) é o supervisor que despoleta a *criação de nós virtuais*, decidindo também o seu posicionamento; c) a informação necessária à actividade do supervisor é proveniente do SB dos vários serviços Domus e parte dessa informação é obtida, em primeira instância, pelo SE ou pelo SA.

Note-se que a actuação do mgPv e mgNv é limitada aos nós que alojam os serviços Domus – $N(S)$ – e, mesmo aí, a sua margem de manobra pode ser reduzida. Com efeito, apenas a carga induzida pelos serviços Domus pode ser redistribuída por aqueles mecanismos;

[1] E, indirectamente, o seu número global de entradas – ver secção 6.7.

serviços e aplicações exógenos ao cluster Domus, mas que partilham com os serviços Domus os nós hospedeiros, estão fora da alçada do mgPv e mgNv; apesar disso, as entidades exógenas acabam por tirar partido, indirectamente, da gestão de carga efectuada pelos serviços Domus, já que se parte do princípio de que cabe a estes a necessária adaptação às condições dinâmicas do *cluster*, mesmo que os recursos sejam sobrecarregados por terceiros.

6.3 Caracterização dos Nós Computacionais

A operação de um cluster Domus depende, em primeiro lugar, de algum conhecimento sobre o conjunto dos nós do *cluster* que o vão suportar, $N(S)$. No máximo, esse conjunto é $N(B)$, formado pelos nós que executam *serviços básicos* (rever secções 5.3.1 e 5.4). Para simplificar o discurso, assumiremos doravante que $N(S) = N(B) = N$ (*i.e.*, $N(S)$ abarca a totalidade do *cluster* físico). Para cada $n \in N$ é então necessária a caracterização i) de certos recursos base (ou do seu desempenho potencial) e ii) dos seus níveis de utilização.

6.3.1 Capacidades

Os atributos que caracterizam um nó n, em termos de recursos ou seu desempenho potencial, definem as *capacidades* de n (ver tabela 6.1). Uma vez que as capacidades descrevem características tipicamente *estáticas*, a sua identificação necessita de ser feita apenas uma vez, antes da primeira implantação de um cluster Domus sobre um *cluster* físico.

Capacidades	**Significado**	**Unidades**
$\mathcal{C}^{\mathsf{e}}(a, n)$	capacidade de *encaminhamento* com o algoritmo a	Koperações/s
$\mathcal{C}^{\mathsf{a}}(t, n)$	capacidade de *acesso* a repositórios de tecnologia t	Koperações/s
$\mathcal{C}(i, n)$	máxima largura de banda útil do "interface Domus"	Mbps
$\mathcal{C}(ram, n)$	capacidade (espaço total) de memória RAM	Mbytes
$\mathcal{C}(disk, n)$	capacidade (espaço total) da "partição Domus"	Gbytes

Tabela 6.1: Capacidades de um Nó Computacional n.

Os atributos $\mathcal{C}^{\mathsf{e}}(a, n)$, $\mathcal{C}^{\mathsf{a}}(t, n)$ e $\mathcal{C}(i, n)$ reflectem o desempenho potencial do nó n, resultando de avaliações especializadas, designadas por *micro-benchmarks* (ver secção 7.6.4).

Assim, o atributo $\mathcal{C}^{\mathsf{e}}(a, n)$ representa a máxima capacidade do nó n para realizar operações de encaminhamento com o algoritmo de localização a. Essas operações serão essencialmente dependentes da capacidade computacional de n e da sua disponibilidade de RAM.

O atributo $\mathcal{C}^{\mathsf{a}}(t, n)$ reflecte o máximo potencial do nó n para a realização de operações de acesso a repositórios baseados numa tecnologia de armazenameno $t =< p, m >$, sendo p a plataforma de armazenamento e m o meio de armazenamento utilizados. Se m for RAM, essas operações serão mais dependentes dos recursos associados ao encaminhamento; se m for Disco, serão mais dependentes do desempenho do acesso ao Disco e da sua capacidade.

O atributo $\mathcal{C}(i, n)$ representa a máxima largura de banda útil suportada pelo interface de Rede i do nó n, escolhido como "interface Domus" de acordo com a definição da secção 5.8.2.1. Definindo como $N(i, n)$ o conjunto dos nós do *cluster* acessíveis a n através do interface i, e podendo $N(i, n)$ comportar nós de categorias diversas (em termos de configurações de hardware) então, para obter o valor de $\mathcal{C}(i, n)$, é preciso considerar a

troca de mensagens entre n e um nó representante de cada uma daquelas categorias, uma vez que o desempenho da comunicação é influenciado pelo hardware do interlocutor. Por exemplo, havendo $\#N_k(i,n)$ nós da espécie k acessíveis a partir de n via interface de Rede i, e sendo $\mathcal{C}(i,n,N_k)$ a máxima largura de banda útil entre n e um nó da espécie k então,

$$\mathcal{C}(i,n) = \sum_k \left[\frac{\#N_k(i,n)}{\#N(i,n)} \times \mathcal{C}(i,n,N_k) \right] \tag{6.1}$$

, assumindo um modelo probabilístico simples, em que a probabilidade de n trocar mensagens com um nó da espécie k é $\#N_k(i,n)/\#N(i,n)$, entre outros critérios possíveis.

Os atributos $\mathcal{C}(ram,n)$ e $\mathcal{C}(disk,n)$ caracterizam, respectivamente, as capacidades de armazenamento primário e secundário do nó, necessárias à sua participação no cluster Domus; a caracterização é trivial, *e.g.*, interrogando o sistema operativo do nó. $\mathcal{C}(disk,n)$ é a capacidade de uma certa partição – a "partição Domus" –, escolhida para satisfazer as necessidades de armazenamento em disco durante a participação do nó no cluster Domus.

As capacidades dos nós que hospedam todos os serviços Domus, de um cluster Domus, são registadas pelo seu serviço supervisor. Adicionalmente, cada serviço Domus mantém informação sobre as capacidades do nó hospedeiro. Esta informação, em conjunção com outra (ver adiante), é utilizada durante a criação, expansão e redistribuição de DHTs.

6.3.2 Utilizações

Como o nome sugere, *utilizações* são medidas que traduzem o nível de utilização (percentual) dos recursos dos nós (ver tabela 6.2). Os seus valores são *dinâmicos*, variando de forma imprevisível, pois admite-se a partilha dos nós por múltiplas aplicações e serviços.

Utilizações	**Significado**
$\mathcal{U}(cpu,n)$	utilização (agregada) das CPUs
$\mathcal{U}(iodisk,n)$	utilização (actividade E/S) do disco com a "partição Domus"
$\mathcal{U}(i,n)$	utilização (largura de banda consumida) do "interface Domus"
$\mathcal{U}(ram,n)$	utilização (espaço consumido) da memória RAM
$\mathcal{U}(disk,n)$	utilização (espaço consumido) da "partição Domus"

Tabela 6.2: Utilizações de um Nó Computacional n.

As utilizações são dadas por médias móveis exponenciais, que sintetizam um histórico de valores. A amplitude/dimensão do histórico, e o peso da história recente face à passada, são parâmetros do processo de cálculo, que admite ainda diversas fórmulas possíveis[2].

À semelhança das capacidades, as utilizações dos nós que participam num cluster Domus são também registadas pelo serviço supervisor e utilizadas durante a criação e migração de nós virtuais. Adicionalmente, cada serviço Domus monitoriza as utilizações do seu nó hospedeiro, em função das quais pode despoletar a migração de nós virtuais (ver a seguir).

[2]Ver o conceito em `http://en.wikipedia.org/wiki/Weighted_moving_average`. A ideia do recurso a médias deste tipo (aplicados também ao cálculo de outras métricas, no âmbito dos nossos mecanismos de balanceamento) é evitar fenómenos de *thrashing*, pela tomada de decisões de redistribuição precipitadas.

6.3.2.1 Limiares de Utilização

Sendo $\mathcal{U}(r,n)$ a utilização de um recurso r, então $\mathcal{U}_\tau(r)$ é um *limiar* de forma que a condição $\mathcal{U}(r,n) \geq \mathcal{U}_\tau(r)$ desencadeia um processo de migração de nós virtuais (ver secção 6.8).

Os limiares são atributos globais do cluster Domus (rever secção 5.8.1.5), sendo definidos quando o cluster Domus é criado, e propagados do supervisor para os restantes serviços.

6.3.2.2 Disponibilidades

Em face das definições anteriores, $\mathcal{A}(r,n) = 1 - \mathcal{U}(r,n)$ traduz a *disponibilidade local* de um recurso r em n, e $\mathcal{A}_\tau(r,n) = \mathcal{U}_\tau(r) - \mathcal{U}(r,n)$ traduz a *disponibilidade útil* correspondente.

A disponibilidade local de um recurso é medida no contexto global do seu nó, independentemente de quem utiliza o recurso; a disponibilidade útil traduz a folga do recurso sob o ponto de vista Domus já que, quando $\mathcal{U}(r,n) \geq \mathcal{U}_\tau(r)$, o recurso considera-se sobrecarregado; ambas as disponibilidades coincidem apenas quando $\mathcal{U}_\tau(r) = 1$ (*i.e.*, 100%).

6.4 Caracterização dos Serviços Domus

As capacidades e utilizações dos nós computacionais permitem definir medidas para a caracterização do conjunto S de serviços de um cluster Domus. Assim, no contexto do suporte a uma DHT específica, é possível definir para cada $s \in S$: a) o seu *potencial de nós virtuais* e b) a sua *atractividade*; estas medidas serão (re)calculadas pelo supervisor, sempre que for necessário (re)definir o serviço hospedeiro de nós virtuais da DHT. Antes de se definirem estas medidas convém recordar (rever secções 5.5.1, 5.6.3, 5.6.5 e 5.6.6):

- $\mathcal{V}^{\mathsf{e}}(d,s)$: nº de nós virtuais de endereçamento da DHT d, atribuídos ao serviço s;
- $\mathcal{V}^{\mathsf{a}}(d,s)$: nº de nós virtuais de armazenamento da DHT d, atribuídos ao serviço s;
- $\mathcal{H}^{\mathsf{e}}(d,s)$: nº de entradas da DHT d cujo endereçamento é gerido pelo serviço s;
- $\mathcal{H}^{\mathsf{a}}(d,s)$: nº de entradas da DHT d cujo armazenamento é gerido pelo serviço s;
- $\overline{\mathcal{H}}^{\mathsf{e}}(d,s,v^{\mathsf{e}})$: nº médio de entradas de d, por cada nó virtual de endere. v^{e}, em s;
- $\overline{\mathcal{H}}^{\mathsf{a}}(d,s,v^{\mathsf{a}})$: nº médio de entradas de d, por cada nó virtual de armaze. v^{a}, em s.

Dado que o modelo M4 acaba por uniformizar o número de entradas por cada nó virtual, os valores $\overline{\mathcal{H}}^{\mathsf{e}}(d,s,v^{\mathsf{e}})$ e $\overline{\mathcal{H}}^{\mathsf{a}}(d,s,v^{\mathsf{a}})$ são semelhantes a valores médios globais, ou seja,

$$\overline{\mathcal{H}}^{\mathsf{e}}(d,s,v^{\mathsf{e}}) \approx \overline{\mathcal{H}}^{\mathsf{e}}(d,v^{\mathsf{e}}) = \frac{\mathcal{H}(d)}{\mathcal{V}^{\mathsf{e}}(d)} \tag{6.2}$$

e

$$\overline{\mathcal{H}}^{\mathsf{a}}(d,s,v^{\mathsf{a}}) \approx \overline{\mathcal{H}}^{\mathsf{a}}(d,v^{\mathsf{a}}) = \frac{\mathcal{H}(d)}{\mathcal{V}^{\mathsf{a}}(d)} \tag{6.3}$$

Recordamos ainda que, de acordo com a regra $\mathcal{R}_6$ da arquitectura Domus (rever secção 5.2.1), "admite-se (no máximo) um só serviço s, por cada nó n". Como também se definiu a associação de um único interface de Rede i a cada serviço s (o "interface Domus" – rever secção 5.8.2.1), a associação unívoca entre *serviço*, *nó* e *interface* permite que se utilize notação do tipo $n(s)$, $i(s)$, etc., para referenciar uma dessas entidades em função de outra.

6.4.1 Potencial de Nós Virtuais

Num dado instante, cada serviço dispõe, por via das capacidades e disponibilidades dos recursos de armazenamento do nó hospedeiro, de um potencial de suporte a um certo número de nós virtuais. Esse potencial é calculado de forma diferente e independente, para o *endereçamento* e *armazenamento*, funções com requisitos específicos: para as funções de endereçamento, o recurso relevante é a RAM; para as de armazenamento, o recurso relevante poderá ser RAM ou Disco, conforme a opção tomada na criação da DHT. O potencial depende ainda de outros parâmetros, com influência na realização dessas funções.

6.4.1.1 Potencial de Nós Virtuais de Endereçamento

A fórmula 6.4 fornece[3] o (número) potencial de nós virtuais de endereçamento da DHT d, suportáveis pelo serviço s, face à capacidade e disponibilidade actual de RAM do nó $n(s)$:

$$\rho^{\mathsf{e}}(d,s) \approx int\left[\mathcal{A}_{\tau}(ram,n(s)) \times \frac{\mathcal{C}(ram,n(s))}{\overline{\mathcal{Q}^{\mathsf{e}}}(ram,v^{\mathsf{e}},d)}\right] \tag{6.4}$$

Assim, a disponibilidade útil de RAM em $n(s)$, $\mathcal{A}_{\tau}(ram,n(s))$, limita o número máximo de nós virtuais de endereçamento da DHT d, comportável por $n(s)$; o máximo é dado pelo quociente entre a capacidade instalada de RAM em $n(s)$, $\mathcal{C}(ram,n(s))$, e a quantidade de RAM gasta (em média) por cada nó virtual de endereçamento, $\overline{\mathcal{Q}^{\mathsf{e}}}(ram,v^{\mathsf{e}},d)$, dada por

$$\overline{\mathcal{Q}^{\mathsf{e}}}(ram,v^{\mathsf{e}},d) \approx \overline{\mathcal{H}}^{\mathsf{e}}(d,v^{\mathsf{e}}) \times \mathcal{Q}(ram,<h,TE,RA>) \tag{6.5}$$

, em que $\mathcal{Q}(ram,<h,TE,RA>)$ é a quantidade de RAM por cada tuplo de esquema <*hash*, *tabela de encaminhamento*, *referência de armazenamento*> (rever secção 5.6.5.1).

O potencial de nós virtuais de endereçamento será nulo ou negativo se $\mathcal{A}_{\tau}(ram,n(s)) \leq 0$, o que acontecerá se o consumo de RAM atingir ou ultrapassar o máximo admissível; nesse caso, não deverá ser admitida a criação de nós virtuais de endereçamento de d em s.

6.4.1.2 Potencial de Nós Virtuais de Armazenamento

A fórmula 6.6 fornece o (número) potencial de nós virtuais de armazenamento da DHT d, suportáveis pelo serviço s, face à capacidade e disponibilidade do recurso $m(d)$ em $n(s)$:

[3] A notação *int* representa a truncagem de um valor real para o inteiro não superior mais próximo.

$$\rho^{\mathsf{a}}(d,s) \approx int\left[\mathcal{A}_{\tau}(m(d),n(s)) \times \frac{\mathcal{C}(m(d),n(s))}{\overline{\mathcal{Q}^{\mathsf{a}}}(m(d),v^{\mathsf{a}},d)}\right] \tag{6.6}$$

Na fórmula 6.6, a disponibilidade útil do recurso de armazenamento $m(d) \in \{ram, disk\}$ em $n(s)$, dada por $\mathcal{A}_{\tau}(m(d),n(s))$, limita o máximo comportável por $n(s)$, de nós virtuais de armazenamento da DHT d; o máximo é dado pelo quociente entre a capacidade instalada do recurso $m(d)$ em $n(s)$, $\mathcal{C}(m(d),n(s))$, e a quantidade máxima de espaço consumível por um nó virtual de armazenamento da DHT d, $\overline{\mathcal{Q}^{\mathsf{a}}}(m(d),v^{\mathsf{a}},d)$; esta quantidade é o *limiar de armazenamento*, um atributo das DHT Domus (rever secção 5.8.3.9 e ver secção 6.6.4).

Se $\mathcal{A}_{\tau}(m(d),n(s)) \leq 0$, é sinal de que o consumo do recurso $m(d)$ ultrapassou o limiar $\mathcal{U}_{\tau}(m(d),n(s))$; em consequência, o potencial de nós virtuais de armazenamento, $\rho^{\mathsf{a}}(d,s)$, será nulo/negativo, prevenindo a criação de nós virtuais de armazenamento de d em $n(s)$.

6.4.2 Atractividade

A atractividade de um serviço Domus, para o desempenho de funções de endereçamento/armazenamento de uma DHT, é função das disponibilidades e capacidades dos recursos do nó hospedeiro, relevantes para o exercício daquelas funções: para as funções de endereçamento, e para as funções de armazenamento em RAM, definem-se como recursos associados mais relevantes a CPU, RAM e Interfaces de Rede; para o armazenamento em Disco, consideram-se o Disco e os Interfaces de Rede como os recursos mais relevantes.

O serviço mais atractivo para o desempenho de um certa função será aquele em que a folga/disponibilidade (útil) dos recursos associados à função é a maior, e essa folga tem o maior "interesse". Dito de outra forma: a ideia base do cálculo da atractividade é pesar as disponibilidades dos recursos em causa, com a importância desses recursos, num determinado universo de nós. Com efeito, a disponibilidade de um recurso num nó pode ser elevada e, naquele universo, o valor relativo dessa disponibilidade reduzido (e vice-versa).

Antes de prosseguir, relembramos conceitos e notação úteis, introduzidos na secção 5.8.3.4:

- $N^{\mathsf{f}}(d)$ – *domínio da funcionalidade* f para a DHT d: trata-se do conjunto dos nós que desempenham a função f para a DHT d ($\mathsf{f} = \mathsf{e}$ para a função de endereçamento e $\mathsf{f} = \mathsf{a}$ para a função de armazenamento); em particular, $N^{\mathsf{e}}(d)$ e $N^{\mathsf{a}}(d)$ definem, respectivamente, o *domínio de endereçamento* e o *domínio de armazenamento* de d;

- $N^{\mathsf{f}}_{max}(d)$ – *universo da funcionalidade* f para a DHT d: limite superior de $N^{\mathsf{f}}(d)$; em particular, $N^{\mathsf{e}}_{max}(d)$ é o *universo de endereçamento* e $N^{\mathsf{a}}_{max}(d)$ é o *universo de armazenamento* da DHT d, ambos atributos de DHTs Domus (rever secção 5.8.3.4).

A associação unívoca entre nós computacionais, serviços Domus e interfaces de rede, permite ainda recorrer, quando conveniente, à notação $S^{\mathsf{f}}(d)$ e $S^{\mathsf{f}}_{max}(d)$, bem como a $I^{\mathsf{f}}(d)$ e $I^{\mathsf{f}}_{max}(d)$, para a representação dos respectivos domínios e universos, de serviços e interfaces.

6.4.2.1 Atractividade de Endereçamento

Seja s um serviço alojado pelo nó $n(s)$ e associado ao interface $i(s)$. A atractividade de s para o endereçamento da DHT d com base no algoritmo de localização distribuída $a(d)$ é

$$\alpha^{\mathsf{e}}(d,s) = \begin{cases} \alpha^{\mathsf{e}}(cpu,d,s) + \alpha^{\mathsf{e}}(i(s),d,s) & \textbf{se} \quad \alpha^{\mathsf{e}}(cpu,d,s) > 0 \quad \textbf{e} \quad \alpha^{\mathsf{e}}(i(s),d,s) > 0 \\ 0 & \textbf{se} \quad \alpha^{\mathsf{e}}(cpu,d,s) \leq 0 \quad \textbf{ou} \quad \alpha^{\mathsf{e}}(i(s),d,s) \leq 0 \end{cases} \tag{6.7}$$

, em que $\alpha^{\mathsf{e}}(cpu,d,s)$ denota a atractividade específica para o recurso CPU, dada por

$$\alpha^{\mathsf{e}}(cpu,d,s) = \mathcal{A}_{\tau}(cpu,n(s)) \times \frac{\mathcal{C}^{\mathsf{e}}(a(d),n(s))}{\mathcal{C}^{\mathsf{e}}_{max}(a(d),N^{\mathsf{e}}_{max}(d))} \tag{6.8}$$

, e $\alpha^{\mathsf{e}}(i(s),d,s)$ denota a atractividade específica para o interface de Rede $i(s)$, dada por

$$\alpha^{\mathsf{e}}(i(s),d,s) = \mathcal{A}_{\tau}(i(s),n(s)) \times \frac{\mathcal{C}(i(s),n(s))}{\mathcal{C}_{max}(\ I^{\mathsf{e}}_{max}(d)\ ,\ N^{\mathsf{e}}_{max}(d)\ )} \tag{6.9}$$

No cálculo de $\alpha^{\mathsf{e}}(cpu,d,s)$, a disponibilidade útil de CPU em n, dada por $\mathcal{A}_{\tau}(cpu,n(s))$, é pesada pela razão entre i) a capacidade de encaminhamento de n para DHTs com algoritmo de localização distribuída $a(d)$ – $\mathcal{C}^{\mathsf{e}}(a(d),n(s))$ – e ii) a maior capacidade de encaminhamento conhecida, no universo $N^{\mathsf{e}}_{max}(d)$, com o mesmo algoritmo – $\mathcal{C}^{\mathsf{e}}_{max}(a(d),N^{\mathsf{e}}_{max}(d))$.

Paralelamente, no cálculo da atractividade $\alpha^{\mathsf{e}}(i(s),d,s)$, a disponibilidade útil do interface $i(s)$ em $n(s)$, dada por $\mathcal{A}_{\tau}(i(s),n(s))$, é pesada pela razão entre i) a largura de banda útil de $i(s)$, dada por $\mathcal{C}(i(s),n(s))$, e ii) a maior largura de banda útil no universo de interfaces $I^{\mathsf{e}}_{max}(d)$ pertencentes ao universo de nós $N^{\mathsf{e}}_{max}(d)$, dada por $\mathcal{C}_{max}(I^{\mathsf{e}}_{max}(d),\ N^{\mathsf{e}}_{max}(d))$.

A atractividade de endereçamento dada pela fórmula 6.7 só é positiva se a atractividade individual de todos os recursos associados for positiva; caso contrário, a atractividade é nula. Por sua vez, a atractividade individual de cada recurso só será positiva se a sua disponibilidade útil for positiva. Ou seja: basta que um dos recursos relevantes para o endereçamento se encontre em sobrecarga (com disponibilidade não positiva), para que o serviço respectivo deixe de se considerar atractivo para o suporte de nós virtuais de endereçamento. Este raciocínio aplica-se também às atractividades de armazenamento.

6.4.2.2 Atractividade de Armazenamento

Seja $t(d) =<p(d),m(d)>$ a tecnologia de armazenamento escolhida para a DHT d, baseada numa plataforma de armazenamento $p(d)$ e num meio de armazenamento $m(d)$. A atractividade do serviço s para o armazenamento da DHT d quando $m(d) = ram$ é então

$$\alpha^{\mathsf{a}}(d,s) = \begin{cases} \alpha^{\mathsf{a}}(cpu,d,s) + \alpha^{\mathsf{a}}(i(s),d,s) & \textbf{se} \quad \alpha^{\mathsf{a}}(cpu,d,s) > 0 \quad \textbf{e} \quad \alpha^{\mathsf{a}}(i(s),d,s) > 0 \\ 0 & \textbf{se} \quad \alpha^{\mathsf{a}}(cpu,d,s) \leq 0 \quad \textbf{ou} \quad \alpha^{\mathsf{a}}(i(s),d,s) \leq 0 \end{cases} \tag{6.10}$$

Complementarmente, a atractividade para o armazenamento de d no meio $m(d) = disk$ é

$$\alpha^{\mathsf{a}}(d,s) = \begin{cases} \alpha^{\mathsf{a}}(iodisk,d,s) + \alpha^{\mathsf{a}}(i(s),d,s) & \textbf{se} \quad \alpha^{\mathsf{a}}(iodisk,d,s) > 0 \quad \textbf{e} \quad \alpha^{\mathsf{a}}(i(s),d,s) > 0 \\ 0 & \textbf{se} \quad \alpha^{\mathsf{a}}(iodisk,d,s) \leq 0 \quad \textbf{ou} \quad \alpha^{\mathsf{a}}(i(s),d,s) \leq 0 \end{cases} \tag{6.11}$$

Se $m(d) = ram$ (fórmula 6.10) a atractividade de armazenamento resulta da soma de duas atractividades: a) uma para o recurso CPU, e b) outra para o recurso Rede. Se $m(d) = disk$ (fórmula 6.11), a atractividade de armazenamento resulta da soma da a) atractividade para o recurso Disco, com b) a atractividade para o recurso Rede. A fórmula 6.11 é assim semelhante à 6.10, considerando que o recurso Disco substitui o recurso CPU.

As atractividades de armazenamento específicas para os recursos em causa são dadas por

$$\alpha^{\mathsf{a}}(cpu,d,s) = \mathcal{A}_{\tau}(cpu,n(s)) \times \frac{\mathcal{C}^{\mathsf{a}}(t(d),n(s))}{\mathcal{C}^{\mathsf{a}}_{max}(t(d),N^{\mathsf{a}}_{max}(d))} \tag{6.12}$$

$$\alpha^{\mathsf{a}}(iodisk,d,s) = \mathcal{A}_{\tau}(iodisk,n(s)) \times \frac{\mathcal{C}^{\mathsf{a}}(t(d),n(s))}{\mathcal{C}^{\mathsf{a}}_{max}(t(d),N^{\mathsf{a}}_{max}(d))} \tag{6.13}$$

$$\alpha^{\mathsf{a}}(i(s),d,s) = \mathcal{A}_{\tau}(i(s),n(s)) \times \frac{\mathcal{C}(i(s),n(s))}{\mathcal{C}_{max}(\ I^{\mathsf{a}}_{max}(d)\ ,\ N^{\mathsf{a}}_{max}(d)\ )} \tag{6.14}$$

No cálculo da atractividade para o recurso CPU (fórmula 6.12), a sua disponibilidade útil em n é agora pesada pela razão entre i) a capacidade máxima de n para a realização de acesso a repositórios de tecnologia $t(d) =<p(d),m(d)>$, dada por $\mathcal{C}^{\mathsf{a}}(t(d),n(s))$, e ii) a maior dessas capacidades, no universo de nós $N^{\mathsf{a}}_{max}(d)$, dada por $\mathcal{C}^{\mathsf{a}}_{max}(t(d),N^{\mathsf{a}}_{max}(d))$. A mesma razão é usada para pesar a folga útil do Disco, em termos de actividade E/S, a fim de se calcular, no mesmo contexto, a atractividade específica do Disco (fórmula 6.13).

No cálculo da atractividade do recurso Rede (fórmula 6.14), i) a largura de banda útil do interface $i(s)$, dada por $\mathcal{C}(i(s),n(s))$, é dividida ii) pela maior largura de banda útil conhecida, dos interfaces $I^{\mathsf{a}}_{max}(d)$ do universo de nós $N^{\mathsf{a}}_{max}(d)$, dada por $\mathcal{C}_{max}(I^{\mathsf{a}}_{max}(d),\ N^{\mathsf{a}}_{max}(d))$.

6.5 Definição da Distribuição Inicial de uma DHT

Uma vez definidos os conceitos necessários à caracterização dos nós computacionais e dos serviços, ilustramos agora a sua aplicação à definição da distribuição inicial de uma DHT.

Na escolha dos nós/serviços que irão alojar os nós virtuais e entradas iniciais, é preciso considerar os diferentes requisitos das funções de endereçamento e armazenamento, com o objectivo de seleccionar os nós com as condições de carga mais adequadas a tais funções.

Basicamente, o que está em causa é a definição inicial da *tabela de distribuição do ende-*

reçamento ($TD^{\mathsf{e}}(d)$), de esquema $< n, \mathcal{V}^{\mathsf{e}}(d,n), \mathcal{H}^{\mathsf{e}}(d,n) >$, e da *tabela de distribuição do armazenamento* ($TD^{\mathsf{a}}(d)$), de esquema $< n, \mathcal{V}^{\mathsf{a}}(d,n), \mathcal{H}^{\mathsf{a}}(d,n) >$ (rever secção 5.8.3.5). Essa definição configura uma actuação do mgPv, ao nível do supervisor, com base nos atributos de Qualidade da Distribuição e de Restrições à Distribuição; esses atributos, e respectivos invariantes, foram anteriormente discutidos, nas secções 5.8.3.3 e 5.8.3.4.

6.5.1 Definição do Número Global de Nós Virtuais e de Entradas

Como referido na secção 5.8.3.4, é possível definir limites mínimos $\mathcal{V}^{\mathsf{e}}_{min}(d)$ e $\mathcal{V}^{\mathsf{a}}_{min}(d)$ para o número global de nós virtuais, através dos atributos `attr_dht_nmve` e `attr_dht_nmva`, respectivamente; estes começam por fornecer valores provisórios de $\mathcal{V}^{\mathsf{e}}_{min}(d)$ e $\mathcal{V}^{\mathsf{a}}_{min}(d)$, podendo ser necessário proceder a ajustamentos no sentido de cumprir certos invariantes.

Assim, $\mathcal{V}^{\mathsf{e}}_{min}(d)$ e $\mathcal{V}^{\mathsf{a}}_{min}(d)$ devem cumprir os invariantes $\mathcal{I}_{60}$ e $\mathcal{I}_{61}$, de forma a garantir nós virtuais suficientes para o número mínimo de nós computacionais, $\mathcal{N}^{\mathsf{e}}_{min}(d)$ e $\mathcal{N}^{\mathsf{a}}_{min}(d)$ (rever secção 5.8.3.4); esta exigência poderá impor reajustamentos em $\mathcal{V}^{\mathsf{e}}_{min}(d)$ e $\mathcal{V}^{\mathsf{a}}_{min}(d)$.

Adicionalmente, é necessário garantir que $\mathcal{V}^{\mathsf{e}}_{min}(d)$ e $\mathcal{V}^{\mathsf{a}}_{min}(d)$ se encontram no mesmo intervalo entre duas potências de 2 consecutivas, o que pode originar novo reajustamento; cumprido esse requisito, $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$ podem assumir os valores de $\mathcal{V}^{\mathsf{e}}_{min}(d)$ e $\mathcal{V}^{\mathsf{e}}_{min}(d)$.

Nas condições anteriores, é também assegurado o cumprimento do invariante $\mathcal{I}_0$ (rever secção 5.5.1), o qual permite definir o número inicial global de entradas da DHT, $\mathcal{H}(d)$.

6.5.2 Definição do Número Local de Nós Virtuais e de Entradas

Com base nos valores iniciais de $\mathcal{V}^{\mathsf{e}}(d)$, $\mathcal{V}^{\mathsf{a}}(d)$ e $\mathcal{H}(d)$, e numa visão geral do estado dos nós (decorrente do conhecimento de capacidades e disponibilidades), o supervisor do cluster Domus efectua uma distribuição pesada dos nós virtuais e entradas, através dos universos de endereçamento e armazenamento da DHT. Efectuada independentemente nos dois universos, a distribuição obedece, em ambos os casos, ao algoritmo (genérico) 6.13. Nas secções seguintes detemo-nos em cada um dos seus passos, dissecando-os c.f. necessário.

6.5.2.1 Passo 1

O passo 1 sugere, desde logo, a relevância das medidas de *potencial* e *atractividade* introduzidas na secção 6.4, para a definição do número de nós virtuais e entradas de uma DHT, a atribuir a cada serviço. Essa definição é efectivamente concretizada nos passos 2, 3 e 4.

6.5.2.2 Passos 2 e 3

Em conjunto, os passos 2 e 3 definem, para cada serviço $s \in S^{\mathsf{f}}_{max}(d)$, o número $\mathcal{V}^{\mathsf{f}}(d,s)$ de nós virtuais atribuídos para realização da função f sendo que, inicialmente, $\mathcal{V}^{\mathsf{f}}(d,s) \leftarrow$

Algoritmo 6.13: Definição da Distribuição Inicial de uma DHT.

1. definir, para cada serviço $s \in S^{\mathsf{f}}_{max}(d)$
   (a) o potencial de nós virtuais, $\rho^{\mathsf{f}}(d,s)$
   (b) a atractividade, $\alpha^{\mathsf{f}}(d,s)$
2. considerando apenas os serviços $s \in S^{\mathsf{f}}_{max}(d)$ com $\rho^{\mathsf{f}}(d,s) > 0$ e $\alpha^{\mathsf{f}}(d,s) > 0$, atribuir *um* nó virtual aos $\#S^{\mathsf{f}}_{min}(d)$ serviços mais atractivos
3. considerando apenas os serviços $s \in S^{\mathsf{f}}_{max}(d)$ com $\rho^{\mathsf{f}}(d,s) > 0$ e $\alpha^{\mathsf{f}}(d,s) > 0$, distribuir o *resto* dos nós virtuais proporcionalmente a $\alpha^{\mathsf{f}}(d,s)$
4. definir o *número de entradas* $\mathcal{H}^{\mathsf{f}}(d,s)$ inicial de cada $s \in S^{\mathsf{f}}(d)$

0. Também neste contexto, denotamos por $\mathcal{V}^{\mathsf{f}}_{assigned}(d)$ o número total de nós virtuais (iniciais) já atribuídos a serviços de $S^{\mathsf{f}}_{max}(d)$, sendo que, inicialmente, $\mathcal{V}^{\mathsf{f}}_{assigned}(d) \leftarrow 0$.

O passo 2 garante que um certo número de nós virtuais é atribuído a serviços diferentes, assegurando um nível mínimo de distribuição no suporte à DHT, especificado de forma independente, no contexto do endereçamento ou armazenamento. Esse nível é denotado, respectivamente, por $\#S^{\mathsf{e}}_{min}(d)$ e $\#S^{\mathsf{a}}_{min}(d)$ que, como já referimos (rever secção 5.8.3.4), constituem atributos de uma DHT, definidos na sua génese. Após o passo 2, $\mathcal{V}^{\mathsf{f}}_{assigned}(d)$ é acrescido do número total de nós virtuais atribuídos nesse passo e, para cada $s \in S^{\mathsf{f}}_{max}(d)$, $\mathcal{V}^{\mathsf{f}}(d,s)$ é somado do número de nós virtuais atribuídos a s nesse passo (um, no máximo).

No passo 3, os nós virtuais ainda por atribuir, em número dado pela diferença $\mathcal{V}^{\mathsf{f}}(d) - \mathcal{V}^{\mathsf{f}}_{assigned}(d)$, devem ser distribuídos por serviços do universo $S^{\mathsf{f}}_{max}(d)$. À partida, essa distribuição só pode contemplar serviços cujo potencial de nós virtuais ainda não tenha sido consumido, ou seja, serviços para os quais $\rho^{\mathsf{f}}(d,s) - \mathcal{V}^{\mathsf{f}}(d,s) > 0$, sendo esses serviços denotados por $S^{\mathsf{f}}_{+}(d)$, com $S^{\mathsf{f}}_{+}(d) \subseteq S^{\mathsf{f}}_{max}(d)$. O número de nós virtuais atribuídos neste passo a qualquer $s_+ \in S^{\mathsf{f}}_{+}(d)$ deverá ainda ser função da atractividade $\alpha^{\mathsf{f}}(d,s_+)$. Após o passo 3, os serviços $s \in S^{\mathsf{f}}_{max}(d)$ para os quais se tem $\mathcal{V}^{\mathsf{f}}(d,s) > 0$ definem $S^{\mathsf{f}}(d)$[4].

O algoritmo 6.14 formaliza o passo 3. A lógica subjacente é simples: distribuir os nós virtuais remanescentes do passo 2, privilegiando os serviços com a atractividade mais elevada; se, durante o processo, o potencial de um serviço é esgotado, os nós virtuais não atribuídos são reservados para uma próxima iteração, onde é feita nova distribuição, proporcionalmente à atractividade dos serviços que ainda dispõem de potencial não utilizado.

6.5.2.3 Passo 4

No início deste passo, já se encontram definidos os *domínios de funcionalidade* iniciais, $S^{\mathsf{f}}(d,s)$, assim como o número de nós virtuais atribuídos a cada nó/serviço desses domínios,

[4]E, equivalentemente, $N^{\mathsf{f}}(d)$ e $I^{\mathsf{f}}(d)$, pela relação de um-para-um entre serviços, nós e interfaces.

Algoritmo 6.14: Passo 3 do Algoritmo 6.13.

1. $\mathcal{V}^{\mathsf{f}}_{assigned'}(d) \leftarrow \mathcal{V}^{\mathsf{f}}_{assigned}(d)$
2. **enquanto** $\mathcal{V}^{\mathsf{f}}_{assigned}(d) < \mathcal{V}^{\mathsf{f}}(d)$ **fazer**
   (a) calcular a soma das atractividades dos serviços $S^{\mathsf{f}}_{+}(d)$ desta iteração: $\alpha^{\mathsf{f}}(S^{\mathsf{f}}_{+}(d)) = \sum_{s_+ \in S^{\mathsf{f}}_{+}(d)} \alpha^{\mathsf{f}}(d, s_+)$
   (b) **para** $s_+ \in S^{\mathsf{f}}_{+}(d)$ **fazer**
      i. calcular a quota de nós virtuais a atribuir a s_+ nesta iteração: $Q^{\mathsf{f}}_{iteration}(d, s_+) = \alpha^{\mathsf{f}}(d, s_+) \; / \alpha^{\mathsf{f}}(S^{\mathsf{f}}_{+}(d))$
      ii. definir o número de nós virtuais a atribuir a s_+ nesta iteração: $\mathcal{V}^{\mathsf{f}}_{iteration}(d, s_+) = int \; [\; Q^{\mathsf{f}}_{iteration}(d, s_+) \times (\mathcal{V}(d) - \mathcal{V}^{\mathsf{f}}_{assigned}(d)) \;]$ **se** $\rho^{\mathsf{f}}(d, s_+) - \mathcal{V}^{\mathsf{f}}(d, s_+) < \mathcal{V}^{\mathsf{f}}_{iteration}(d, s_+)$ **então** $\mathcal{V}^{\mathsf{f}}_{iteration}(d, s_+) \leftarrow \rho^{\mathsf{f}}(d, s_+) - \mathcal{V}^{\mathsf{f}}(d, s_+)$ **fse**
      iii. actualizar o número total de nós virtuais já atribuído a s_+: $\mathcal{V}^{\mathsf{f}}(d, s_+) \leftarrow \mathcal{V}^{\mathsf{f}}(d, s_+) + \mathcal{V}^{\mathsf{f}}_{iteration}(d, s_+)$
      iv. actualizar o número total de nós virtuais já atribuído: $\mathcal{V}^{\mathsf{f}}_{assigned'}(d) \leftarrow \mathcal{V}^{\mathsf{f}}_{assigned'}(d) + \mathcal{V}^{\mathsf{f}}_{iteration}(d, s_+)$
      v. terminar se todos os nós virtuais tiverem sido atribuídos: **se** $\mathcal{V}^{\mathsf{f}}_{assigned'}(d) = \mathcal{V}(d)$ **então** *end* **fse**
   fpara
   (c) $\mathcal{V}^{\mathsf{f}}_{assigned}(d) \leftarrow \mathcal{V}^{\mathsf{f}}_{assigned'}(d)$

$\mathcal{V}^{\mathsf{f}}(d, s)$. Para finalizar o preenchimento das *tabelas de distribuição* $TD^{\mathsf{f}}(d)$, resta apenas definir o número de entradas a atribuir a cada nó/serviço, $\mathcal{H}^{\mathsf{f}}(d, s)$; essa definição é feita aplicando duas vezes o Procedimento de (Re)Distribuição do modelo M4 (rever as secções 3.6.3 e 3.5.5): uma vez sobre a tabela $TD^{\mathsf{e}}(d)$, e uma outra vez sobre a tabela $TD^{\mathsf{a}}(d)$.

6.5.3 Implantação da Distribuição Inicial

Definidas as *tabelas de distribuição* iniciais, resta definir a identidade das entradas atribuídas a cada serviço, $H^{\mathsf{f}}(d, s)$, o que se consegue através dos mecanismos previstos na secção 4.3.1; estes mecanismos permitem que cada serviço de endereçamento da DHT deduza a sua *informação de endereçamento* (tuplos < *hash*, *tabela de encaminhamento*, *referência de armazenamento* > – rever secção 5.6.5.1); os mesmos mecanismos possibilitam que cada

serviço de armazenamento instancie a sua *informação de armazenamento* (tuplos < *hash*, *referência de repositório*, *referência de endereçamento* > – rever secção 5.6.6.1), sendo ainda necessário efectuar a criação de repositórios ou preparar a sua utilização pela DHT.

6.6 Caracterização das DHTs

Os mecanismos mgPv e mgNv dependem, para a sua operação, de *atributos de gestão* (rever secção 5.8.3.9) e *medidas de caracterização* da carga induzida pelas DHTs. Os atributos são valorados na criação das DHTs, sendo registados pelo supervisor, como parte da *informação de supervisão* das DHTs (rever secção 5.8). O Subsistema de Endereçamento (SE) e o Subsistema de Armazenamento (SA) dos serviços Domus que suportam as DHTs produzem medidas locais de carga que, processadas no supervisor, originam medidas sintéticas/globais, que integrarão também a *informação de supervisão* das DHTs.

A carga induzida por uma DHT *activa* pode ser: 1) *carga de uso corrente* derivada do processamento das operações habituais sobre dicionários (inserções, consultas e remoções de registos), solicitadas por aplicações clientes (rever secção 5.9) e 2) *carga de administração*.

A *carga de uso corrente* subdivide-se em *carga de localização*, *de acesso* e *de armazenamento*. A carga de localização incidirá, essencialmente, sobre os nós $N^{\mathtt{e}}(d)$, que alojam os nós virtuais de endereçamento de d. A carga de acesso e a de armazenamento incidirão, maioritariamente, sobre os nós $N^{\mathtt{a}}(d)$, que alojam os nós virtuais de armazenamento de d.

As actividades administrativas associadas a uma DHT podem ser de origem *endógena* ou *exógena*. As actividades endógenas são despoletadas automaticamente pelo mgNv e mgPv, incluindo a *expansão* do número de nós virtuais e/ou a sua *migração*[5]. As actividades exógenas têm origem em *interacções de administração* desencadeadas por aplicações Domus e incluem a criação, destruição, desactivação e reactivação da DHT (rever secção 5.9).

A *carga de administração* é considerada sempre transitória, não sendo por isso objecto de gestão. A *carga de uso corrente* depende do padrão de utilização das DHTs pelas aplicações clientes. Com efeito, são essas aplicações que determinam o ritmo a que uma DHT é consultada ou actualizada, assim como o volume de dados que nela é armazenada.

O mgNv gere a *carga de uso corrente* das DHTs através da expansão do número de nós virtuais: redistribuindo a mesma carga por maior número de nós virtuais, a carga média por nó virtual irá diminuir. Para que a expansão possa ser despoletada no momento próprio, as cargas de localização, acesso e armazenamento têm que ser devidamente monitorizadas.

As medidas resultantes da monitorização da carga ligada ao uso corrente das DHTs são também úteis sob o ponto de vista do mgPv. Este mecanismo gere a carga global dos nós do *cluster* socorrendo-se do reposicionamento dos nós virtuais das DHTs. Precisamente, na escolha desses nós virtuais, é utilizada informação resultante daquela monitorização, e são respeitados certos constrangimentos, alguns determinados por atributos das DHTs.

[5] A migração também pode ter origem externa, pela remoção administrativa de um serviço Domus.

6.6.1 Taxas e Limiares de Encaminhamento

Seja $D^{\mathsf{e}}(s)$ o conjunto das DHTs cujo endereçamento é suportado pelo serviço s, no nó $n(s)$. Então, para cada DHT $d \in D^{\mathsf{e}}(s)$, o Subsistema de Endereçamento (SE) de s produz a medida $\overline{\lambda^{\mathsf{e}}}(d,s)$ – "taxa *média* de operações de *encaminhamento*[6], para a DHT d, em s". A taxa $\overline{\lambda^{\mathsf{e}}}(d,s)$ é medida em operações por segundo, tal como as capacidades $\mathcal{C}^{\mathsf{e}}(a(d),n(s))$.

$\overline{\lambda^{\mathsf{e}}}(d,s)$ é uma taxa *média*, calculada com base numa média móvel exponencial, alimentada por uma certa janela temporal de taxas amostrais $\lambda^{\mathsf{e}}(d,s)$, obtidas em sessões amostrais sucessivas, de igual duração; cada taxa $\lambda^{\mathsf{e}}(d,s)$ obtem-se dividindo 1) o número de operações de encaminhamento registadas na sessão amostral, pela 2) duração da sessão amostral.

Para cada taxa $\overline{\lambda^{\mathsf{e}}}(d,s)$, conhece-se um máximo absoluto, $\lambda^{\mathsf{e}}_{max}(d,s)$ (ver secção 6.6.3).

No supervisor, as taxas $\overline{\lambda^{\mathsf{e}}}(d,s)$ oriundas dos serviços de endereçamento da DHT d, são combinadas em $\overline{\lambda^{\mathsf{e}}}(d)$ – "taxa *média global* de operações de *encaminhamento* da DHT d":

$$\overline{\lambda^{\mathsf{e}}}(d) = \sum_{s \in S^{\mathsf{e}}(d)} \overline{\lambda^{\mathsf{e}}}(d,s) \tag{6.15}$$

Adicionalmente, a "taxa *máxima global* de operações de *encaminhamento* da DHT d" é:

$$\lambda^{\mathsf{e}}_{max}(d) = \sum_{s \in S^{\mathsf{e}}(d)} \lambda^{\mathsf{e}}_{max}(d,s) \tag{6.16}$$

Sempre que a taxa média $\overline{\lambda^{\mathsf{e}}}(d)$ ultrapassar uma certa fracção $\lambda^{\mathsf{e}}_{\tau}(d)$ da taxa máxima, *i.e.*,

$$\overline{\lambda^{\mathsf{e}}}(d) \geq \lambda^{\mathsf{e}}_{\tau}(d) \times \lambda^{\mathsf{e}}_{max}(d) \tag{6.17}$$

o supervisor provocará a expansão do *domínio de endereçamento* da DHT d, para que a *carga de endereçamento* da DHT se disperse por mais nós/serviços de endereçamento; neste contexto, é importante notar que a *carga de endereçamento*, imposta pelos clientes da DHT, poderá continuar a ser a mesma em termos globais; todavia, ao aumentar $\lambda^{\mathsf{e}}_{max}(d)$, o reforço do *domínio de endereçamento* torna mais difícil a ultrapassagem do valor $\lambda^{\mathsf{e}}_{\tau}(d) \times \lambda^{\mathsf{e}}_{max}(d)$.

A expansão do *domínio de endereçamento* comporta a selecção de novos nós de endereçamento e o cálculo do seu número de nós virtuais de endereçamento, que vão contribuir para aumentar o número global de nós virtuais de endereçamento; menos frequentemente, este aumento poderá ditar o incremento do número global de entradas da DHT e, em consequência, do número global de nós virtuais de armazenamento (rever secção 5.5.1).

A grandeza $\lambda^{\mathsf{e}}_{\tau}(d)$ designa-se por *limiar de encaminhamento/endereçamento*, sendo um atributo de *gestão da carga de endereçamento*, das DHTs Domus (rever secção 5.8.3.9).

[6]Isto é, operações que fazem parte do processo de localização do SE e do SA de uma entrada.

6.6.2 Taxas e Limiares de Acesso

A lógica e metodologia anteriores são extensíveis ao contexto do armazenamento: sendo $D^{\mathsf{a}}(s)$ o conjunto das DHTs cujo armazenamento é suportado pelo serviço s, o Subsistema de Armazenamento (SA) de s produz, para cada $d \in D^{\mathsf{a}}(s)$, a medida $\overline{\lambda^{\mathsf{a}}}(d,s)$, correspondente à "taxa *média* de operações de *acesso*[7] a registos da DHT d, em s". A taxa $\overline{\lambda^{\mathsf{a}}}(d,s)$ é medida em operações por segundo, tal como o são as capacidades $\mathcal{C}^{\mathsf{a}}(t(d),n(s))$.

De forma análoga a $\overline{\lambda^{\mathsf{e}}}(d,s)$, $\overline{\lambda^{\mathsf{a}}}(d,s)$ resulta também de uma média móvel exponencial (ver acima) e assume-se que é conhecido um máximo absoluto $\lambda^{\mathsf{a}}_{max}(d,s)$ (ver secção 6.6.3).

No supervisor, a gestão da carga de acesso é similar à da carga de endereçamento, baseando-se nos valores $\overline{\lambda^{\mathsf{a}}}(d)$ ("taxa *média global* de operações de *acesso* da DHT d") e $\lambda^{\mathsf{e}}_{max}(d)$ ("taxa *máxima global* de operações de *acesso* da DHT d"), e no teste da condição

$$\overline{\lambda^{\mathsf{a}}}(d) \geq \lambda^{\mathsf{a}}_{\tau}(d) \times \lambda^{\mathsf{a}}_{max}(d) \tag{6.18}$$

cuja verificação determina a expansão do *domínio de armazenamento* da DHT d, como forma de dispersar a *carga de acesso*. Na expressão 6.18, $\lambda^{\mathsf{a}}_{\tau}(d)$ denota o *limiar de acesso*, um atributo de *gestão da carga de acesso*, para as DHTs Domus (rever secção 5.8.3.9).

6.6.3 Máximos Absolutos para os Limiares de Encaminhamento e Acesso

6.6.3.1 Cenário 1

Num cenário simplificado, em que cada nó n executa apenas um serviço Domus s, e este suporta apenas funções de endereçamento, de uma só DHT d, com base no algoritmo $a(d)$,

$$\lambda^{\mathsf{e}}_{max}(d,n(s)) = \mathcal{C}^{\mathsf{e}}(a(d),n(s)) \tag{6.19}$$

Num cenário similar, mas para o armazenamento com a tecnologia $t(d) =< p(d), m(d) >$,

$$\lambda^{\mathsf{a}}_{max}(d,n(s)) = \mathcal{C}^{\mathsf{a}}(t(d),n(s)) \tag{6.20}$$

6.6.3.2 Cenário 2

Admitamos agora que, em cada nó n, o serviço s endereça várias DHTs, que denotamos por $D^{\mathsf{e}}(s)$, podendo cada DHT $d \in D^{\mathsf{e}}(s)$ utilizar algoritmos $a(d)$ eventualmente diferentes. Nestas condições, será mais difícil a uma taxa média $\overline{\lambda^{\mathsf{e}}}(d,s)$ convergir para a capacidade $\mathcal{C}^{\mathsf{e}}(a(d),n(s))$, uma vez que os recursos de cada nó serão partilhados pelas várias DHTs.

Neste cenário, o serviço s suporta o endereçamento de um total de $\mathcal{H}^{\mathsf{e}}(s)$ entradas, denotando $\mathcal{H}^{\mathsf{e}}(d,s)$ o número específico de entradas associado a cada $d \in D^{\mathsf{e}}(s)$, pelo que

[7] Isto é, operações de consulta, inserção, modificação ou remoção de registos.

$$\mathcal{H}^{\mathsf{e}}(s) = \sum_{d \in D^{\mathsf{e}}(s)} \mathcal{H}^{\mathsf{e}}(d, s) \tag{6.21}$$

Assumindo que o acesso às DHTs $D^{\mathsf{e}}(s)$ é uniforme, então a carga de endereçamento induzida por cada $d \in D^{\mathsf{e}}(s)$ em s será proporcional a $\frac{\mathcal{H}^{\mathsf{e}}(d,s)}{\mathcal{H}^{\mathsf{e}}(s)}$, o que nos permite estabelecer

$$\lambda^{\mathsf{e}}_{max}(d, n(s)) = \frac{\mathcal{H}^{\mathsf{e}}(d, s)}{\mathcal{H}^{\mathsf{e}}(s)} \times \mathcal{C}^{\mathsf{e}}(a(d), n(s)) \tag{6.22}$$

Basicamente, o que transparece da fórmula 6.22 é que, em $n(s)$, a DHT d apenas pode contar, para o seu endereçamento, com a fracção $\frac{\mathcal{H}^{\mathsf{e}}(d,s)}{\mathcal{H}^{\mathsf{e}}(s)}$ da capacidade máxima $\mathcal{C}^{\mathsf{e}}(a(d), n(s))$.

A fórmula 6.23 oferece uma perspectiva alternativa da fórmula 6.22. Assim, representando $\frac{1}{\mathcal{C}^{\mathsf{e}}(a(d),n(s)}$ o custo, em segundos, de *uma* operação de encaminhamento em $n(s)$, com o algoritmo $a(d)$, então $\lambda^{\mathsf{e}}_{max}(d, n(s))$ fornece o número máximo dessas operações num segundo, de forma a não se consumir mais que a fracção $\frac{\mathcal{H}^{\mathsf{e}}(d,s)}{\mathcal{H}^{\mathsf{e}}(s)}$ do segundo.

$$\frac{1}{\mathcal{C}^{\mathsf{e}}(a(d), n(s)} \times \lambda^{\mathsf{e}}_{max}(d, n(s)) = \frac{\mathcal{H}^{\mathsf{e}}(d, s)}{\mathcal{H}^{\mathsf{e}}(s)} \tag{6.23}$$

No contexto do armazenamento, este cenário corresponde, em cada serviço s, ao armazenamento de registos de várias DHTs, denotadas por $D^{\mathsf{a}}(s)$, com tecnologias de armazenamento $t(d)$, escolhidas para cada DHT. Cada serviço s suportará o armazenamento de $\mathcal{H}^{\mathsf{a}}(s)$ entradas, sendo $\mathcal{H}^{\mathsf{a}}(d, s)$ o número específico de entradas associado a $d \in D^{\mathsf{a}}(s)$. Nestas condições, a aplicação da lógica anterior (e seus pressupostos) permite estabelecer

$$\lambda^{\mathsf{a}}_{max}(d, n(s)) = \frac{\mathcal{H}^{\mathsf{a}}(d, s)}{\mathcal{H}^{\mathsf{a}}(s)} \times \mathcal{C}^{\mathsf{a}}(t(d), n(s)) \tag{6.24}$$

6.6.3.3 Cenário 3

Se admitirmos que um nó possa desempenhar em simultâneo funções de endereçamento e de armazenamento, o grau de partilha dos recursos do nó será maior e, em consequência, o valor dos máximos absolutos $\lambda^{\mathsf{e}}_{max}(d, n(s))$ e $\lambda^{\mathsf{a}}_{max}(d, n(s))$ será necessariamente menor.

Neste cenário, um serviço s suporta um total de $\mathcal{H}(s) = \mathcal{H}^{\mathsf{e}}(s) + \mathcal{H}^{\mathsf{a}}(s)$ entradas. Destas, $\mathcal{H}^{\mathsf{e}}(s)$ entradas são de DHTs de $D^{\mathsf{e}}(s)$, e as restantes $\mathcal{H}^{\mathsf{a}}(s)$ entradas são de DHTs do conjunto $D^{\mathsf{a}}(s)$. Assim, para $d \in D^{\mathsf{e}}(s)$, o máximo absoluto $\lambda^{\mathsf{e}}_{max}(d, n(s))$ é dado por

$$\lambda^{\mathsf{e}}_{max}(d, n(s)) = \frac{\mathcal{H}^{\mathsf{e}}(d, s)}{\mathcal{H}(s)} \times \mathcal{C}^{\mathsf{e}}(a(d), n(s)) \tag{6.25}$$

Analogamente, para as DHTs $d \in D^{\mathsf{a}}(s)$, o máximo absoluto $\lambda^{\mathsf{a}}_{max}(d, n(s))$ é dado por

$$\lambda^{\mathsf{a}}_{max}(d, n(s)) = \frac{\mathcal{H}^{\mathsf{a}}(d, s)}{\mathcal{H}(s)} \times \mathcal{C}^{\mathsf{a}}(t(d), n(s)) \tag{6.26}$$

Para a mesma DHT d, com o mesmo número de entradas $\mathcal{H}^{\mathsf{f}}(d, s)$ associado ao mesmo serviço s, os máximos absolutos $\lambda^{\mathsf{f}}_{max}(d, n(s))$ serão agora menores que os dados pelas equações 6.22 e 6.24, pois as fracções $\frac{\mathcal{H}^{\mathsf{f}}(d,s)}{\mathcal{H}(s)}$ serão agora menores que as fracções $\frac{\mathcal{H}^{\mathsf{f}}(d,s)}{\mathcal{H}^{\mathsf{f}}(s)}$.

6.6.3.4 Cenário 4

No cenário anterior constatámos a necessidade de reduzir o valor dos máximos absolutos $\lambda^{\mathsf{f}}_{max}(d, n(s))$, ao permitir a coexistência, no mesmo serviço, de suporte ao endereçamento e ao armazenamento de DHTs. Essa necessidade resultou, como referido, de um maior grau de partilha dos recursos do nó hospedeiro do serviço. Esse grau de partilha será ainda maior se, no mesmo nó, permitirmos a execução de outras aplicações/serviços, fora da alçada dos nossos mecanismos de balanceamento, e com requisitos dinâmicos e imprevisíveis.

Nestas circunstâncias, os máximos absolutos $\lambda^{\mathsf{f}}_{max}(d, n(s))$ do cenário 3 serão mais dificilmente alcançáveis. O papel complementar do mgPv relativamente ao mgNv revela-se nestas situações: quando os recursos de um nó ficam sobrecarregados, independentemente de quem os sobrecarrega (entidades Domus ou exógenas), o mgPv movimentará nós virtuais para nós mais aliviados; nesses nós, a convergência para os máximos absolutos será mais fácil, logo será maior a probabilidade de a DHT expandir, pela criação de nós virtuais.

6.6.4 Quantidades e Limiares de Armazenamento

Para cada $d \in D^{\mathsf{a}}(s)$, o SA de s produz a medida $\overline{\mathcal{Q}}^{\mathsf{a}}(d, s)$, que é a "quantidade *média* consumida, do recurso $m(d)$, por cada nó virtual de armazenamento da DHT d, em s".

A média $\overline{\mathcal{Q}}^{\mathsf{a}}(d, s)$ obtém–se dividindo a) a quantidade de espaço de armazenamento consumido pelos registos <*chave*, *dados*> atribuídos às entradas locais, $H^{\mathsf{a}}(d, s)$, pelo b) número de nós virtuais de armazenamento locais, $\mathcal{V}^{\mathsf{a}}(d, s)$. Para este cálculo, é apenas relevante o espaço útil/nominal consumido pelos registos (soma da dimensão das chaves e dos dados associados), ignorando-se as sobrecargas induzidas pelas estruturas de dados de suporte e o espaço consumido pelo *índice de armazenamento* de SA relativo às entradas $H^{\mathsf{a}}(d, s)$.

Subjacente ao papel da média $\overline{\mathcal{Q}}^{\mathsf{a}}(d, s)$ está a assumpção de que o desvio-padrão associado é reduzido, o que acontecerá se a dimensão dos registos <*chave*, *dados*> e a sua dispersão pelos nós virtuais de armazenamento da DHT, seguirem ambos uma distribuição uniforme[8].

As médias $\overline{\mathcal{Q}}^{\mathsf{a}}(d, s)$ produzidas pelos serviços de armazenamento de uma DHT d são comunicadas ao supervisor; este produz a média pesada $\overline{\mathcal{Q}}^{\mathsf{a}}(d)$, "quantidade *média global* consumida, do recurso $m(d)$, por cada nó virtual de armazenamento da DHT d", dada por

[8] Uma forma de o garantir seria através de uma camada de *fragmentação* (ver secção 6.10)

$$\overline{\mathcal{Q}^{\mathsf{a}}}(d) = \sum_{s \in S^{\mathsf{a}}(d)} \left[\overline{\mathcal{Q}^{\mathsf{a}}}(d,s) \times \frac{\mathcal{V}^{\mathsf{a}}(d,s)}{\mathcal{V}^{\mathsf{a}}(d)} \right] \tag{6.27}$$

Sempre que $\overline{\mathcal{Q}^{\mathsf{a}}}(d)$ ultrapassar um *limiar de armazenamento*[9] $\mathcal{Q}^{\mathsf{a}}_{\tau}(d)$, ou seja, sempre que

$$\overline{\mathcal{Q}^{\mathsf{a}}}(d) \geq \mathcal{Q}^{\mathsf{a}}_{\tau}(d) \tag{6.28}$$

, o supervisor provocará a criação de mais nós virtuais de armazenamento; a criação poderá ocorrer nos serviços que formam o actual *domínio de armazenamento* da DHT ou, se necessário, o domínio poderá expandir-se; os nós virtuais de armazenamento adicionais acomodarão parte dos registos dos outros nós virtuais, reduzindo-se o valor de $\overline{\mathcal{Q}^{\mathsf{a}}}(d)$.

$\mathcal{Q}^{\mathsf{a}}_{\tau}(d)$ é um atributo de *gestão da carga de armazenamento* das DHTs (rever secção 5.8.3.9).

Se o limiar $\mathcal{Q}^{\mathsf{a}}_{\tau}(d)$ não for alcançável, por escassez de meios de armazenamento $m(d)$, o mgPv acabará por migrar os nós virtuais de armazenamento para outros nós, com maior disponibilidade desses recursos; a probabilidade de o mgNv actuar, por esgotamento da capacidade dos nós virtuais, será então maior. A lógica subjacente a esta interacção entre o mgNv e o mgPv é semelhante à que referimos, a propósito do Cenário 4 da secção 6.6.3.

Neste contexto, torna-se evidente que não há *reserva* efectiva de recursos, mas antes *partilha*; ou seja, o facto de se atribuir um nó virtual de endereçamento ou armazenamento a um nó/serviço significa apenas que, no instante dessa atribuição, havia recursos suficientes para tal; se, entretanto, esses recursos forem consumidos por outras entidades, que coexistem no nó, então acabará por ser necessário migrar os nós virtuais para outros nós[10].

6.6.5 Largura de Banda Consumida

Para cada DHT $d \in D^{\mathsf{e}}(s)$, o SE de s produz a medida $\overline{\beta^{\mathsf{e}}}(d,s)$, correspondente à "largura de banda *média* consumida pelas operações de *encaminhamento* de d, em s". Análogamente, para cada DHT $d \in D^{\mathsf{a}}(s)$, o SA de s produz a medida $\overline{\beta^{\mathsf{a}}}(d,s)$, que representa a "largura de banda *média* consumida pelas operações de *acesso* a d, em s".

A obtenção destas médias, que são também médias móveis exponenciais, prossegue um método semelhante ao descrito previamente para as taxas de encaminhamento e de acesso.

Assim, no contexto do endereçamento, s regista e acumula o tamanho útil das mensagens de localização recebidas e enviadas, para cada $d \in D^{\mathsf{e}}(s)$; após cada sessão amostral, calcula-se a taxa amostral $\beta^{\mathsf{e}}(d,s)$, dividindo-se 1) o tamanho acumulado, pela 2) duração da sessão amostral, usando-se essa taxa para actualizar a média móvel exponencial $\overline{\beta^{\mathsf{e}}}(d,s)$.

No contexto do armazenamento, a lógica é similar, estando em causa o registo da dimensão útil de pedidos de acesso a registos <*chave*,*dados*>, e suas respostas, para as DHTs $D^{\mathsf{a}}(s)$.

Ao contrário das medidas anteriores, $\overline{\beta^{\mathsf{e}}}(d,s)$ e $\overline{\beta^{\mathsf{a}}}(d,s)$ não são comunicadas ao supervisor, sendo utilizadas apenas pelo Subsistema de Balanceamento (SB) local (ver secção 6.8.1.4).

[9] Na secção 6.4.1.2, esse limiar foi denotado por $\overline{\mathcal{Q}^{\mathsf{a}}}(m(d), v^{\mathsf{a}}, d)$ por consistência com a notação $\overline{\mathcal{Q}^{\mathsf{e}}}(ram, v^{\mathsf{e}}, d)$ da secção 6.4.1.1. Aqui usa-se a forma mais abreviada, $\mathcal{Q}^{\mathsf{a}}_{\tau}(d)$, introduzida na secção 5.8.3.9.

[10] Esta questão será abordada com mais profundidade na secção 6.9.

6.7 Redistribuição de DHTs por Criação de Nós Virtuais

Na secção anterior introduzimos as medidas, atributos e condições que, em tempo de execução, permitem i) caracterizar a carga de localização, acesso e armazenamento das DHTs, e ii) determinar da necessidade da sua *expansão*. Nesta secção descreve-se o processo de expansão propriamente dito; este socorre-se, cumulativamente, dos atributos e medidas associados à caracterização de nós computacionais e serviços, introduzidos inicialmente.

Neste contexto, convém recordar que o termo *expansão* refere-se ao *aumento do número de nós virtuais* de determinada espécie, podendo esse aumento ter, como efeito colateral, i) o aumento do número global de entradas da DHT, ii) o aumento do número de nós virtuais da outra espécie, iii) o aumento do número de nós computacionais de qualquer espécie (configurando uma expansão do *domínio de endereçamento* e/ou do *de armazenamento*).

Uma *política de evolução estática* previne a re-distribuição de uma DHT por criação de nós virtuais, pois impõe um *número fixo* de nós virtuais e entradas (rever secção 5.8.3.8).

Em suma, o mgNv recorre ao aumento do *número* de nós virtuais de uma DHT para redistribuir a carga ligada ao seu uso corrente (localização, acesso e armazenamento). Para o efeito, o mgNv socorre-se das métricas geradas pelos Subsistemas de Endereçamento (SE) e de Armazenamento (SA), dos serviços Domus, para decidir quando desencadear a expansão, e conta com a colaboração do mgPv para se definir qual o nó/serviço que irá alojar o(s) novo(s) nó(s) virtual(s). A expansão de uma DHT é assim um processo que depende da cooperação entre os dois mecanismos de balanceamento previstos na arquitectura Domus. De seguida, descreve-se o processo de expansão e as suas implicações, com mais detalhe.

6.7.1 Redefinição do Número Global de Nós Virtuais e de Entradas

Seja f_1 a espécie do nó virtual a criar e f_* a espécie complementar (ou seja, se $\mathsf{f}_1 = \mathsf{e}$ (função de endereçamento) então $\mathsf{f}_* = \mathsf{a}$ (função de armazenamento), e vice-versa).

Assim, na criação de um (e um só) nó virtual da espécie f_1, de uma certa DHT d, começa-se por verificar a necessidade de se duplicar o número global de entradas da DHT, $\mathcal{H}(d)$; essa duplicação será necessária se, antes da criação do nó virtual da espécie f_1, o número global desses nós virtuais, $\mathcal{V}^{\mathsf{f}_1}(d)$, for uma potência de 2; neste caso, a única forma de cumprir o invariante $\mathcal{I}_0$ (rever secção 5.5.1) será duplicar $\mathcal{H}(d)$; segue-se que, por efeito de arrastamento, o número global de nós virtuais da espécie complementar, $\mathcal{V}^{\mathsf{f}_*}(d)$, terá também de ser aumentado (uma ou mais unidades), a fim de não se violar o invariante $\mathcal{I}_0$.

Se, em alternativa, não for necessário duplicar $\mathcal{H}(d)$, então $\mathcal{V}^{\mathsf{f}_*}(d)$ mantém-se constante (o símbolo $*$ pretende traduzir o facto de que o aumento de $\mathcal{V}^{\mathsf{f}_*}(d)$ pode ser nulo ou não).

Impacto da Dissociação Endereçamento-Armazenamento A eventual duplicação de $\mathcal{H}(d)$ tem ainda outras consequências, ainda não discutidas, no quadro da dissociação entre funções de endereçamento e armazenamento, preconizada pela arquitectura Domus.

Assim, no caso do endereçamento, são aplicáveis os mecanismos previstos na secção 4.7, os quais permitem a dedução do grafo $G^{H(d)}(\mathcal{L}+1)$ a partir do grafo $G^{H(d)}(\mathcal{L})$, realizável de forma autónoma pelos serviços de endereçamento da DHT, uma vez notificados para tal, pelo supervisor; todavia, a par com a duplicação e actualização de *tabelas de encaminhamento*, esses serviços necessitam também de actualizar as *referências de armazenamento* que acompanham cada tabela (rever secção 5.6.5.1); para tal, assume-se que o *conjunto de registos* de uma entrada (rever secção 5.6.6.1) ascendente (de nível $\mathcal{L}$) é partido em dois conjuntos, correspondentes às entradas descendentes (de nível $\mathcal{L}+1$); de forma coerente com o assumido na secção 4.7.1, esses conjuntos são mantidos no mesmo serviço de armazenamento da entrada ascendente, imediatamente após a duplicação de $\mathcal{H}(d)$ e logo as *referências de armazenamento* das entradas descendentes herdam o mesmo valor da ascendente; por seu turno, nos serviços de armazenamento, as *referências de endereçamento* das entradas descendentes coincidem também com a da entrada ascendente, pois as *tabelas de encaminhamento* das descendentes são mantidas no mesmo serviço da ascendente.

Nos serviços de armazenamento é ainda preciso levar em conta o impacto sobre os *repositórios* (rever secção 5.6.6.2). Se a DHT em causa usar repositórios de *grão fino*, então o desdobramento dos registos de uma entrada ascendente nos subconjuntos das descendentes implica a criação de pelo menos um novo repositório local, para onde devem ser movidos os registos de uma das entradas descendentes; se este requisito for relaxado (*e.g.*, para evitar interferir com acessos concorrentes aos registos), o repositório passa a *partilhado*.

6.7.2 Redefinição do Número Local de Nós Virtuais e de Entradas

Recalculados os novos valores de $\mathcal{H}(d)$, $\mathcal{V}^{\mathsf{e}}(d)$ e $\mathcal{V}^{\mathsf{a}}(d)$, é necessário derivar e implantar as novas *tabelas de distribuição do endereçamento* ($TD^{\mathsf{e}}(d)$) e do *armazenamento* ($TD^{\mathsf{a}}(d)$). Tal pode ser conseguido através do algoritmo (genérico) 6.15, onde $\mathcal{V}^{\mathsf{f}+}$ denota o número adicional de nós virtuais da espécie f (se $\mathsf{f}_+ = \mathsf{f}_1$ então $\mathcal{V}^{\mathsf{f}+} = 1$; se $\mathsf{f}_+ = \mathsf{f}_*$ então $\mathcal{V}^{\mathsf{f}+} \geq 0$). Os vários passos do algoritmo são discutidos, individualmente, nas subsecções seguintes.

Algoritmo 6.15: Redistribuição de uma DHT por Criação de Nós Virtuais.

0. se necessário, duplicar o *número de entradas* $\mathcal{H}^{\mathsf{f}}(d,s)$ de cada $s \in S^{\mathsf{f}}(d)$,
1. definir, para cada serviço $s \in S^{\mathsf{f}}_{max}(d)$
   (a) o potencial de nós virtuais, $\rho^{\mathsf{f}}(d,s)$
   (b) a atractividade, $\alpha^{\mathsf{f}}(d,s)$
2. considerando apenas os serviços $s \in S^{\mathsf{f}}_{max}(d)$ com $\rho^{\mathsf{f}}(d,s) > 0$ e $\alpha^{\mathsf{f}}(d,s) > 0$, distribuir entre eles $\mathcal{V}^{\mathsf{f}+}$ nós virtuais, proporcionalmente a $\alpha^{\mathsf{f}}(d,s)$
3. (re)definir o *número de entradas* $\mathcal{H}^{\mathsf{f}}(d,s)$ de cada $s \in S^{\mathsf{f}}(d)$

6.7.2.1 Passo 0

A actualização de uma tabela $TD^{\mathsf{f}}(d)$ começa por reflectir a eventual duplicação prévia do número global de entradas da DHT, $\mathcal{H}(d)$, realizada ainda antes da execução do algoritmo, em conformidade com os mecanismos discutidos na secção 6.7.1. A eventual duplicação de $\mathcal{H}(d)$ é incorporada numa tabela $TD^{\mathsf{f}}(d)$ duplicando simplesmente o número individual de entradas de cada nó/serviço, $\mathcal{H}^{\mathsf{f}}(d,s)$. Numa implementação da arquitectura Domus, este passo pode ser realizado como parte integrante das tarefas previstas na secção 6.7.1.

6.7.2.2 Passos 1 e 2

Os passos 1 e 2 fazem incidir na tabela de distribuição $TD^{\mathsf{f}}(d)$ o acréscimo $\mathcal{V}^{\mathsf{f}+}$ do número de nós virtuais da espécie f, eventualmente decidido pelos mecanismos da secção 6.7.1.

Para se decidir quais os serviços onde vão ser criados os $\mathcal{V}^{\mathsf{f}+}$ nós virtuais adicionais de cada espécie f, o algoritmo 6.13 começa por actualizar, no passo 1, a caracterização do *potencial* e da *atractividade* do *universo* de serviços da DHT para a funcionalidade f, $\mathcal{S}^{\mathsf{f}}_{max}(d)$.

Depois, no passo 2, concretiza-se a escolha dos nós/serviços pelos quais se vão distribuir os $\mathcal{V}^{\mathsf{f}+}$ nós virtuais adicionais; a escolha é feita de forma semelhante à realizada no passo 3 do algoritmo 6.13, podendo resultar na expansão do *domínio* de funcionalidade f, $\mathcal{S}^{\mathsf{f}}(d)$.

6.7.2.3 Passo 3

No passo 3 aplica-se o Procedimento de (Re)Distribuição do modelo M4 (rever secções 3.6.3 e 3.5.5) sobre a tabela $TD^{\mathsf{f}}(d)$, resultando no reacerto do número de entradas atribuídas aos serviços, necessário pelo reacerto do seu número de nós virtuais no passo 2.

6.7.3 Implantação da Redistribuição

Comparando a nova versão da tabela $TD^{\mathsf{f}}(d)$ com a anterior, definem-se quais os serviços que devem ceder/receber entradas e em que *número*, de forma a implantar a nova distribuição da DHT; neste contexto, a estratégia a seguir foi já delineada na secção 3.3.4.2.

Assim, cada serviço $s^{\mathsf{f}}_{\_}$ obrigado a ceder um certo número $\mathcal{H}^{\mathsf{f}}_{\_}(d,s_{\_})$ de entradas, terá de escolhê-las a partir do seu conjunto local de entradas, $H^{\mathsf{f}}(d,s_{\_})$, podendo a definição da *identidade* das entradas $H^{\mathsf{f}}_{\_}(d,s_{\_})$ ser aleatória. Depois, a migração destas entradas materializa-se de forma diferente, c.f. f seja a função de endereçamento ou armazenamento.

No contexto do endereçamento, a migração das entradas $H^{\mathsf{e}}_{\_}(d,s_{\_})$ implica a 1) transferência das suas *tabelas de encaminhamento* e a 2) actualização das *referências de endereçamento* direccionadas às tabelas; neste último caso, para além da necessidade de se manter a conectividade total do grafo da DHT (o que implica actualizar um conjunto de outras tabelas de encaminhamento – rever secção 4.8), é também necessário actualizar o conjunto das *referências de endereçamento*, mantidas pelos serviços de armazenamento de $H^{\mathsf{e}}_{\_}(d,s_{\_})$.

No contexto do armazenamento, a migração das entradas $H^{\mathtt{a}}_{\_}(d, s\_)$ corresponde 1) à migração do *conjunto de registos* das entradas[11] e 2) à actualização das respectivas *referências de armazenamento*, mantidas pelos serviços de endereçamento das entradas $H^{\mathtt{a}}_{\_}(d, s\_)$.

Como previsto na secção 3.3.4.2, a transferência de entradas é coordenada pelo supervisor, sendo paralelizada quando possível. De facto, o grau de autonomia de cada serviço na prossecução destas operações é algo limitado, devido à teia complexa de interdependências que podem surgir, designadamente no que diz respeito à actualização de referências de endereçamento e armazenamento; daí a necessidade/conveniência de o processo ser coordenado centralizadamente, de forma a assegurar a consistência final de todas as referências.

6.8 Redistribuição de DHTs por Migração de Nós Virtuais

A par da definição do posicionamento dos i) nós virtuais iniciais e ii) dos criados subsequentemente, o mgPv intervém também no seu eventual iii) reposicionamento/migração; este pode ter a) causas exógenas/administrativas (*e.g.*, a necessidade de excluir um nó, dos domínios da DHT), ou b) resultar de mecanismos endógenos/automáticos de gestão de carga; neste último caso, é o Subsistema de Balanceamento (SB) que, em cada serviço Domus, despoleta a migração de nós virtuais, como reacção à sobrecarga de recursos locais.

Uma *política de evolução estática* (rever secção 5.8.3.8) inibe a migração automática/endógena de nós virtuais, mas não impede a migração por razões exógenas/administrativas. Neste último caso, a escolha dos serviços hospedeiros dos nós virtuais a migrar prossegue um algoritmo semelhante ao algoritmo 6.16 para migrações automáticas (ver secção 6.8.2).

A condição que, num serviço Domus s, catalisa a migração automática de um nó virtual, foi já introduzida na secção 6.3.2.1: quando pelo menos uma das utilizações $\mathcal{U}(r, n(s))$ da tabela 6.2 atingir ou ultrapassar o limiar $\mathcal{U}_\tau(r, n(s))$ respectivo, o processo é despoletado. Tomada a decisão de migrar um nó virtual local é necessário 1) escolher a DHT à qual o nó virtual pertence e 2) definir o serviço hospedeiro mais apropriado para esse nó virtual.

No contexto da migração de nós virtuais, o mgPv recorre a todas as métricas apresentadas anteriormente, para a caracterização de nós, serviços e DHTs. Assim, para cada serviço Domus, o mgPv deve ter acesso a) ao estado dos recursos do nó hospedeiro do serviço, para poder a1) decidir da necessidade de migrar um nó virtual local, bem como a2) escolher convenientemente o seu novo nó hospedeiro; adicionalmente, o mgPv deve também ter acesso a b) informação sobre o padrão de utilização dos recursos dos nós, pelas DHTs locais, sendo essa informação útil na b1) eleição da DHT a que se refere o nó virtual a migrar. As decisões/escolhas a1) e b1) são efectuadas pelo Subsistema de Balanceamento do serviço que vai perder o nó virtual, enquanto que a escolha a2) é feita pelo supervisor.

Nesta secção explicamos apenas a forma como o mgPv realiza as suas escolhas. Todavia, antes de descrevermos o processo de escolha propriamente dito, recordamos alguma notação útil, introduzida na secção 5.6.3, apropriada à expressão da "dupla personalidade" das DHTs Domus e suas entradas, derivada da dissociação endereçamento-armazenamento:

[11] Sendo a migração agilizada se esses conjuntos forem mantidos em repositórios separados, por entrada.

- $D(s)$ denota as DHTs de D suportadas, de alguma forma[12], pelo serviço s;
- $d(s)$ denota um elemento (DHT) de $D(s)$;
- $D^{\mathsf{f}}(s)$ denota as DHTs de D, cuja função f é suportada pelo serviço s;
- $d^{\mathsf{f}}(s)$ denota um elemento (DHT) de $D^{\mathsf{f}}(s)$;
- $D(s) = D^{\mathsf{e}}(s) \cup D^{\mathsf{a}}(s)$.

6.8.1 Selecção da DHT a Redistribuir

Esta eleição segue um princípio simples: num serviço s, executando no nó $n(s)$, será migrado um nó virtual daquela DHT $d_{-} \in D(s)$ que apresentar a *relação* mais forte entre a) a utilização dada especificamente por d_{-} aos recursos de $n(s)$, e b) os níveis de utilização locais desses recursos em $n(s)$. Se $d_{-} \in D^{\mathsf{e}}(s)$, será migrado um nó virtual de endereçamento de d_{-}; se $d_{-} \in D^{\mathsf{a}}(s)$, será migrado um nó virtual de armazenamento. Nas secções seguintes elaboramos sobre as métricas necessárias para escolher a DHT d_{-}.

6.8.1.1 Carga Induzida por uma DHT num só Recurso

O "nível de utilização do recurso r, para realizar a função f, para a DHT d, no nó $n(s)$" é:

$$\mu(r, d^{\mathsf{f}}, n(s)) = \mathcal{U}(r, n(s)) \times \mu(r, d^{\mathsf{f}}, D(s)) \tag{6.29}$$

Na fórmula anterior, i) $\mathcal{U}(r, n(s))$ representa o "nível de utilização do recurso r, no contexto do nó $n(s)$" (definido na secção 6.3.2) e ii) $\mu(r, d^{\mathsf{f}}, D(s))$ representa o "nível de utilização do recurso r, para realizar a função f, para a DHT d, no contexto das DHTs $D(s)$".

A fórmula 6.29 pode ser interpretada da seguinte forma: dado o nível de utilização global $\mathcal{U}(r, n(s))$ de um recurso r do nó $n(s)$ então, num cenário hipotético, em que o recurso r seria dedicado exclusivamente ao suporte das DHTs $D(s)$, a medida $\mu(r, d^{\mathsf{f}}, D(s))$ forneceria a fracção de $\mathcal{U}(r, n(s))$ imputável à DHT d^{f}; o produto $\mathcal{U}(r, n(s)) \times \mu(r, d^{\mathsf{f}}, D(s))$ seria assim uma medida da utilização local do recurso r pela DHT d^{f}, naquelas condições.

Apesar de o cenário considerado ser hipotético, a medida $\mu(r, d^{\mathsf{f}}, n(s))$ é útil, dado que o que está em causa é o posicionamento relativo das diversas DHTs $D(s)$ face ao consumo do recurso r, e essa ordenação é introduzida, de forma natural, pelo factor $\mu(r, d^{\mathsf{f}}, D(s))$.

6.8.1.2 Carga Induzida por uma DHT sobre vários Recursos

Seja $R(d^{\mathsf{f}}, s)$ o conjunto dos recursos de que depende $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$ para a função $\mathsf{f} \in \{\mathsf{e}, \mathsf{a}\}$. Então, de acordo com assumpções prévias[13], é possível definir os seguintes conjuntos:

[12] Seja no contexto do endereçamento, seja no do armazenamento.

[13] Rever secções 6.3 e 6.4. Em particular, note-se que a definição de $R(d^{\mathsf{f}}, s)$ referencia os mesmos recursos envolvidos no cálculo das medidas de *atractividade* e *potencial de nós virtuais*, introduzidas na secção 6.4.

- $R(d^{\mathsf{e}}, s) = \{cpu, i(s), ram\}$;
- $R(d^{\mathsf{a}}, s) = R(d^{\mathsf{e}}, s)$, quando $m(d^{\mathsf{a}}) = ram$;
- $R(d^{\mathsf{a}}, s) = \{iodisk, i(s), disk\}$, quando $m(d^{\mathsf{a}}) = disk$.

Estes conjuntos incluem os recursos mais relevantes para o exercício de diferentes funções: a) $R(d^{\mathsf{e}}, s)$ inclui os recursos para a função de endereçamento e b) $R(d^{\mathsf{a}}, s)$ inclui os recursos para a função de armazenamento, diferentes conforme o meio de armazenamento $m(d)$.

Seja então $\mu(d^{\mathsf{f}}, n(s))$ o "nível de utilização dos recursos $R(d^{\mathsf{f}}, s)$, por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, no contexto do nó $n(s)$". Numa primeira aproximação, $\mu(d^{\mathsf{f}}, n(s))$ pode ser dado pela fórmula

$$\mu(d^{\mathsf{f}}, n(s)) = \sum_{r \in R(d^{\mathsf{f}}, s)} \mu(r, d^{\mathsf{f}}, n(s)) \tag{6.30}$$

Na fórmula 6.30, a importância relativa dos vários recursos de $R(d^{\mathsf{f}}, s)$ é igual, tal como assumido previamente, no cálculo da atractividade (rever secção 6.4.2); se essa importância não for uniforme, os pesos considerados deverão ser os mesmos em ambos os contextos.

Se for baseada na soma de valores produzidos pela fórmula 6.29, a fórmula 6.30 poderá não ser suficientemente sensível aos recursos em sobrecarga, de forma a destacar nitidamente as DHTs que induzem maior carga nesses recursos. Assim, em alternativa, deve-se usar

$$\mu(r, d^{\mathsf{f}}, n(s)) = \left[\frac{\mathcal{U}(r, n(s))}{\mathcal{U}_\tau(r)}\right]^2 \times \mu(r, d^{\mathsf{f}}, D(s)) \tag{6.31}$$

Na fórmula anterior, o quociente $\frac{\mathcal{U}(r,n(s))}{\mathcal{U}_\tau(r)}$ será superior a 1 para os recursos em sobrecarga e o quadrado amplificará o peso que esse quociente fornece ao factor $\mu(r, d^{\mathsf{f}}, D(s))$; para os recursos ainda com disponibilidade útil positiva, o quociente $\frac{\mathcal{U}(r,n(s))}{\mathcal{U}_\tau(r)}$ será inferior a 1 e o quadrado reduzirá substancialmente o peso que o quociente fornece ao factor $\mu(r, d^{\mathsf{f}}, D(s))$.

6.8.1.3 Critério de Selecção da DHT a Redistribuir

Representando $\mu(d, n(s))$ o esforço induzido por d, nos recursos locais do nó $n(s)$, então a DHT $d' \in D(s)$ que maximizar $\mu(d, n(s))$ deverá ceder um nó virtual para migração.

6.8.1.4 Definição de $\mu(r, d^{\mathsf{f}}, D(s))$

Para ambas as fórmulas 6.29 e 6.31, os valores de $\mathcal{U}(r, n(s))$ são assumidos como disponíveis, restando a necessidade de definir a métrica $\mu(r, d^{\mathsf{f}}, D(s))$, para cada $r \in R(d^{\mathsf{f}}, s)$.

Definição de $\mu(cpu, d^{\mathsf{f}}, D(s))$: Esta medida deverá fornecer a proporção do tempo de CPU consumido por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, face ao tempo de CPU gasto por todas as DHTs de $D(s)$.

Em primeiro lugar, recordem-se as medidas $\overline{\lambda^{\mathsf{e}}}(d,s)$ e $\overline{\lambda^{\mathsf{a}}}(d,s)$, definidas nas secções 6.6.1 e 6.6.2, e que denotam, respectivamente, a "taxa média de operações de *encaminhamento* da DHT d, em s" e a "taxa média de operações de *acesso* à DHT d, em s", expressas em operações/s. Estas medidas são denotáveis, abreviadamente, por $\overline{\lambda^{\mathsf{f}}}(d,s)$, com $\mathsf{f} \in \{\mathsf{e},\mathsf{a}\}$.

Em segundo lugar, recorde-se que, como definido no Cenário 2 da secção 6.6.3, o custo temporal unitário (*i.e.*, por cada operação) é dado, no caso das operações de *encaminhamento*, por $1/\mathcal{C}^{\mathsf{e}}(a(d),n(s))$ e, no caso das operações *de acesso*, por $1/\mathcal{C}^{\mathsf{a}}(t(d),n(s))$. Estas métricas são denotáveis, de forma mais sintética, por $1/\mathcal{C}^{\mathsf{f}}(d,n(s))$, que assim representa "o custo temporal unitário, na realização da função f, para a DHT d, no nó $n(s)$".

Uma aproximação ao "tempo de CPU consumido por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, num segundo", será pois

$$\mathcal{T}(cpu, d^{\mathsf{f}}, s) = \frac{1}{\mathcal{C}^{\mathsf{f}}(d, n(s))} \times \overline{\lambda^{\mathsf{f}}}(d,s) \tag{6.32}$$

A fórmula 6.32 só é válida para a) $\mathsf{f} = \mathsf{e}$, ou para b) $\mathsf{f} = \mathsf{a}$ se $m(d^{\mathsf{a}}) = ram$. Quando $\mathsf{f} = \mathsf{a}$ e $m(d) = disk$, o recurso CPU não se considera relevante, sendo então $\mathcal{T}(cpu, d^{\mathsf{a}}, s) = 0$.

Uma aproximação ao "tempo de CPU consumido pelas DHTs $D^{\mathsf{f}}(s)$, num segundo" será

$$\mathcal{T}(cpu, D^{\mathsf{f}}(s)) = \sum_{d^{\mathsf{f}} \in D^{\mathsf{f}}(s)} \mathcal{T}(cpu, d^{\mathsf{f}}, s) \tag{6.33}$$

, logo o "tempo de CPU consumido pelas DHTs $D(s)$, num segundo" será aproximado por

$$\mathcal{T}(cpu, D(s)) = \mathcal{T}(cpu, D^{\mathsf{e}}(s)) + \mathcal{T}(cpu, D^{\mathsf{a}}(s)) \tag{6.34}$$

, o que permite definir $\mu(cpu, d^{\mathsf{f}}, D(s))$ de forma expedita, através da seguinte fórmula:

$$\mu(cpu, d^{\mathsf{f}}, D(s)) = \frac{\mathcal{T}(cpu, d^{\mathsf{f}}, s)}{\mathcal{T}(cpu, D(s))} \tag{6.35}$$

Definição de $\mu(iodisk, d^{\mathsf{f}}, D(s))$: Esta métrica fornecerá a proporção do tempo de serviço do Disco, consumido por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, face ao tempo consumido pelas DHTs $D(s)$.

Recorrendo aos conceitos em que nos apoiámos para definir $\mu(cpu, d^{\mathsf{f}}, D(s))$, uma aproximação ao "tempo de serviço do Disco, consumido por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, num segundo", será

$$\mathcal{T}(iodisk, d^{\mathsf{f}}, s) = \frac{1}{\mathcal{C}^{\mathsf{f}}(d, n(s))} \times \overline{\lambda^{\mathsf{f}}}(d,s) \tag{6.36}$$

A fórmula 6.36 só é válida se $\mathsf{f} = \mathsf{a}$ e $m(d) = disk$. Nos restantes casos, $\mathcal{T}(iodisk, d^{\mathsf{f}}, s) = 0$.

O "tempo de serviço do Disco consumido pelas DHTs $D(s)$, num segundo" será dado por

$$\mathcal{T}(iodisk, D(s)) = \sum_{d \in D(s)} \mathcal{T}(iodisk, d, s) \tag{6.37}$$

, o que permite definir $\mu(iodisk, d^{\mathsf{f}}, D(s))$ de forma expedita, através da seguinte fórmula:

$$\mu(iodisk, d^{\mathsf{f}}, D(s)) = \frac{\mathcal{T}(iodisk, d^{\mathsf{f}}, s)}{\mathcal{T}(iodisk, D(s))} \tag{6.38}$$

Definição de $\mu(i(s), d^{\mathsf{f}}, D(s))$: Está em causa a proporção da largura de banda do interface $i(s)$, consumida por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, face à largura de banda consumida por $D(s)$.

Recordemos as medidas $\overline{\beta^{\mathsf{e}}}(d^{\mathsf{e}}, s)$ e $\overline{\beta^{\mathsf{a}}}(d^{\mathsf{a}}, s)$, definidas na secção 6.6.5, e que denotam, respectivamente, a "largura de banda média consumida em *encaminhamento* para a DHT d^{e}, em s" e a "largura de banda média consumida no *acesso* a registos da DHT d^{a}, em s".

A "largura de banda de $i(s)$, consumida pelas DHTs $D(s)$, num segundo" será dada por

$$\beta(i(s), D(s)) = \sum_{d \in D(s)} \overline{\beta}(d, s) \tag{6.39}$$

, o que permite definir $\mu(i(s), d^{\mathsf{f}}, D(s))$ de forma expedita, através da seguinte fórmula:

$$\mu(i(s), d^{\mathsf{f}}, D(s)) = \frac{\overline{\beta^{\mathsf{f}}}(d, s)}{\beta(i(s), D(s))} \tag{6.40}$$

Definição de $\mu(d^{\mathsf{f}}, ram, D(s))$: Esta métrica indicará a proporção da memória RAM consumida por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, face à memória RAM consumida por todas as DHTs $D(s)$.

Como se estabeleceu na secção 6.4.1, para uma DHT $d^{\mathsf{e}} \in D^{\mathsf{e}}(s)$, a RAM consumida (média) por cada nó virtual de endereçamento, $\overline{\mathcal{Q}}(ram, v^{\mathsf{e}}, d^{\mathsf{e}})$, é dada pela fórmula 6.5.

Assim, denotando $\mathcal{V}^{\mathsf{e}}(d^{\mathsf{e}}, s)$ o número de nós virtuais de endereçamento da DHT d^{e} em s, então, "a memória RAM consumida, num dado instante, por $d^{\mathsf{e}} \in D^{\mathsf{e}}(s)$" será dada por

$$\mathcal{Q}(ram, d^{\mathsf{e}}, s) = \overline{\mathcal{Q}}(ram, v^{\mathsf{e}}, d^{\mathsf{e}}) \times \mathcal{V}^{\mathsf{e}}(d^{\mathsf{e}}, s) \tag{6.41}$$

, e a "memória RAM consumida pelas DHTs $D^{\mathsf{e}}(s)$, num dado instante" será dada por

$$\mathcal{Q}(ram, D^{\mathsf{e}}(s)) = \sum_{d^{\mathsf{e}} \in D^{\mathsf{e}}(s)} \mathcal{Q}(ram, d^{\mathsf{e}}, s) \tag{6.42}$$

No contexto do armazenamento a memória RAM é consumida por *conjuntos de registos*, mas só pelas DHTs $d^{\mathsf{a}} \in D^{\mathsf{a}}(s)$ com $m(d^{\mathsf{a}}) = ram$; para as restantes DHTs, esse consumo

é nulo. Ora, na secção 6.6.4 definimos que, para uma DHT $d^{\mathsf{a}} \in D^{\mathsf{a}}(s)$, o consumo (médio) do recurso $m(d^{\mathsf{a}})$, por cada nó virtual de armazenamento v^{a}, é uma das medidas produzidas pelo Subsistema de Armazenamento de s, que denotamos aqui por[14] $\overline{\mathcal{Q}}(m(d^{\mathsf{a}}), v^{\mathsf{a}}, d^{\mathsf{a}}, s)$.

Assim, denotando $\mathcal{V}^{\mathsf{a}}(d^{\mathsf{a}}, s)$ o número de nós virtuais de armazenamento da DHT d^{a} em s, então, "a memória RAM consumida, num dado instante, por $d^{\mathsf{a}} \in D^{\mathsf{a}}(s)$" será dada por

$$\mathcal{Q}(ram, d^{\mathsf{a}}, s) = \begin{cases} \overline{\mathcal{Q}}(m(d^{\mathsf{a}}), v^{\mathsf{a}}, d^{\mathsf{a}}, s) \times \mathcal{V}^{\mathsf{a}}(d^{\mathsf{a}}, s) & \text{se } m(d^{\mathsf{a}}) = ram \\ 0 & \text{se } m(d^{\mathsf{a}}) = disk \end{cases}$$

, e a "memória RAM consumida pelas DHTs $D^{\mathsf{a}}(s)$, num dado instante" será dada por

$$\mathcal{Q}(ram, D^{\mathsf{a}}(s)) = \sum_{d^{\mathsf{a}} \in D^{\mathsf{a}}(s)} \mathcal{Q}(ram, d^{\mathsf{a}}, s) \tag{6.43}$$

Assim, a "memória RAM consumida pelas DHTs $D(s)$, num dado instante" será dada por

$$\mathcal{Q}(ram, D(s)) = \mathcal{Q}(ram, D^{\mathsf{e}}(s)) + \mathcal{Q}(ram, D^{\mathsf{a}}(s)) \tag{6.44}$$

o que, finalmente, permite definir a métrica $\mu(d^{\mathsf{f}}, ram, D(s))$ através da seguinte fórmula:

$$\mu(d^{\mathsf{f}}, ram, D(s)) = \frac{\mathcal{Q}(ram, d^{\mathsf{f}}, s)}{\mathcal{Q}(ram, D(s))} \tag{6.45}$$

Definição de $\mu(disk, d^{\mathsf{f}}, D(s))$: Esta métrica indicará a proporção do espaço em Disco consumido por $d^{\mathsf{f}} \in D^{\mathsf{f}}(s)$, face ao espaço em Disco consumido por todas as DHTs $D(s)$.

Em primeiro lugar, recorde-se que o espaço em Disco apenas é consumido pelas DHTs $d^{\mathsf{a}} \in D^{\mathsf{a}}(s)$ para as quais $m(d^{\mathsf{a}}) = disk$; para as restantes DHTs, o consumo de Disco é nulo. Em segundo lugar, para $m(d^{\mathsf{a}}) = disk$, o consumo (médio) por cada nó virtual de armazenamento v^{a}, é agora representado por $\overline{\mathcal{Q}}(m(d^{\mathsf{a}}), v^{\mathsf{a}}, d^{\mathsf{a}}, s)$ – ver nota de rodapé 14.

Assim, denotando $\mathcal{V}^{\mathsf{a}}(d^{\mathsf{a}}, s)$ o número de nós virtuais de armazenamento da DHT d^{a} em s, então, "o espaço em Disco consumido, num dado instante, por $d^{\mathsf{a}} \in D^{\mathsf{a}}(s)$" será dado por

$$\mathcal{Q}(disk, d^{\mathsf{a}}, s) = \begin{cases} \overline{\mathcal{Q}}(m(d^{\mathsf{a}}), v^{\mathsf{a}}, d^{\mathsf{a}}, s) \times \mathcal{V}^{\mathsf{a}}(d^{\mathsf{a}}, s) & \text{se } m(d^{\mathsf{a}}) = disk \\ 0 & \text{se } m(d^{\mathsf{a}}) = ram \end{cases}$$

, e "o espaço em Disco consumido pelas DHTs $D^{\mathsf{a}}(s)$, num dado instante" será dado por

$$\mathcal{Q}(disk, D^{\mathsf{a}}(s)) = \sum_{d^{\mathsf{a}} \in D^{\mathsf{a}}(s)} \mathcal{Q}(disk, d^{\mathsf{a}}, s) \tag{6.46}$$

[14] Na secção 6.6.4 usa-se a notação $\overline{\mathcal{Q}}^{\mathsf{a}}(d, s)$, mas $\overline{\mathcal{Q}}(m(d^{\mathsf{a}}), v^{\mathsf{a}}, d^{\mathsf{a}}, s)$ é mais adequada ao contexto presente.

O "espaço em Disco consumido pelas DHTs $D(s)$, num dado instante" será então dado por

$$\mathcal{Q}(disk, D(s)) = \mathcal{Q}(disk, D^{\mathsf{e}}(s)) + \mathcal{Q}(disk, D^{\mathsf{a}}(s)) = \mathcal{Q}(disk, D^{\mathsf{a}}(s)) \tag{6.47}$$

, o que permite finalmente definir a métrica $\mu(d^{\mathsf{f}}, disk, D(s))$ através da seguinte fórmula:

$$\mu(disk, d^{\mathsf{f}}, D(s)) = \frac{\mathcal{Q}(disk, d^{\mathsf{f}}, s)}{\mathcal{Q}(disk, D(s))} \tag{6.48}$$

6.8.2 Definição da Redistribuição

Escolhida a DHT $d_{\_}$, fornecedora do nó virtual a migrar, é preciso definir o seu novo serviço hospedeiro e o número de entradas a migrar. Para tal, o supervisor executa o algoritmo 6.16, no qual $s_{\_}$ denota o actual hospedeiro do nó virtual e s_{+} é o futuro hospedeiro. Os vários passos do algoritmo são discutidos, individualmente, nas secções seguintes.

Algoritmo 6.16: Redistribuição de uma DHT por Migração de um Nó Virtual.

1. definir, para cada serviço $s \in S^{\mathsf{f}}_{max}(d_{\_})$
   (a) o potencial de nós virtuais, $\rho^{\mathsf{f}}(d_{\_}, s)$
   (b) a atractividade, $\alpha^{\mathsf{f}}(d_{\_}, s)$
2. considerando apenas os serviços $s \in S^{\mathsf{f}}_{max}(d_{\_})$ com $\rho^{\mathsf{f}}(d_{\_}, s) > 0$ e $\alpha^{\mathsf{f}}(d_{\_}, s) > 0$, doar um nó virtual ao serviço mais atractivo (s_{+}) e retirar um nó virtual a $s_{\_}$
3. reajustar o *número de entradas* de $s_{\_}$ e s_{+}

6.8.2.1 Passos 1 e 2

No passo 1 calcula-se o *potencial* e a *atractividade* do *universo* de serviços da DHT, permitindo que, no passo 2, se efectue a escolha do nó/serviço para onde se vai migrar o nó virtual; essa escolha é feita com base no mesmo critério usado para a escolha dos nós/serviços onde se criam nós virtuais iniciais ou adicionais (rever algoritmos 6.13 e 6.15); o passo 2 reflecte-se na modificação da *tabela de distribuição* da DHT $d_{\_}$, $TD^{\mathsf{f}}(d_{\_})$, traduzida na subtracção de um nó virtual a $\mathcal{V}^{\mathsf{f}}(d_{\_}, s_{\_})$ e adição de um nó virtual a $\mathcal{V}^{\mathsf{f}}(d_{\_}, s_{+})$.

6.8.2.2 Passo 3

No passo 3 aplica-se o Procedimento de (Re)Distribuição do modelo M4 sobre a $TD^{\mathsf{f}}(d_{\_})$, ajustando o *número de entradas* de $s_{\_}$ e s_{+} ao *número de nós virtuais* definido no passo 2.

6.8.2.3 Implantação da Redistribuição

A comparação entre a nova tabela $TD^{\mathsf{f}}(d_{\_})$ e a versão anterior à aplicação do algoritmo 6.16, permite determinar a (eventual) necessidade de migrar entradas (a fim de materializar a migração do nó virtual[15]), aplicando-se a descrição fornecida antes, na secção 6.7.3.

Da escolha realizada no passo 2 do algoritmo 6.16, podem resultar vários tipos de decisão: pode decidir-se a) migrar o nó virtual a1) para um dos serviços do domínio actual da DHT ($S^{\mathsf{f}}(d_{\_})$), ou a2) para um dos serviços do seu universo ($S^{\mathsf{f}}_{max}(d_{\_})$), mas ainda ausente do domínio (o que provocará a expansão do domínio); adicionalmente, pode também decidir-se b) não migrar o nó virtual pois, dos serviços do universo $S^{\mathsf{f}}_{max}(d_{\_})$ com *potencial* positivo, aquele com maior *atractividade* para o nó virtual continua a ser o seu hospedeiro actual; nesta situação, pode-se usar um algoritmo de *backoff* exponencial, para determinar o momento da próxima reavaliação da carga local e decisão da necessidade de migração.

A realizar-se, uma consequência possível da migração é a remoção do serviço $s^{\mathsf{f}}_{\_}$ do domínio $S^{\mathsf{f}}(d_{\_})$, o que acontecerá se o número $\mathcal{V}^{\mathsf{f}}(d_{\_}, s^{\mathsf{f}}_{\_})$ de nós virtuais locais de d descer a zero (sinal de que também o número $\mathcal{H}^{\mathsf{f}}(d_{\_}, s^{\mathsf{f}}_{\_})$ de entradas locais de $d_{\_}$ é nulo); neste caso, o serviço $s^{\mathsf{f}}_{\_}$ ainda pertence ao universo $S^{\mathsf{f}}_{max}(d_{\_})$ da DHT, podendo vir a integrar novamente o seu domínio $S^{\mathsf{f}}(d_{\_})$ (excepto quando $s^{\mathsf{f}}_{\_}$ é removido administrativamente do universo).

6.9 Partilha de Recursos

Como é evidenciado pelos algoritmos 6.13, 6.15 e 6.16, a determinação do número *real* de nós virtuais de uma DHT[16], atribuído a um determinado serviço Domus, é função de medidas de *potencial* e *atractividade*. Em particular, o cálculo das medidas de *potencial* depende de *estimativas* sobre a) a quantidade de recursos disponíveis no nó hospedeiro do serviço, e b) sobre a quantidade de recursos que os nós virtuais necessitam. Ora, estes recursos, designadamente RAM e Disco, não são efectivamente reservados para serem utilizados, em exclusividade, pelos nós virtuais. Consequentemente, diferentes eventos de criação de nós virtuais, munidos de uma visão semelhante do estado dos recursos do *cluster*, poderão resultar na selecção de hospedeiros comuns para esses nós virtuais. Os recursos desses hospedeiros serão então partilhados pelos vários nós virtuais, e sujeitos a taxas de utilização/ocupação eventualmente diferentes, consoante a DHT a que os nós virtuais digam respeito. Neste cenário, o mgPv desencadeará, se necessário for, migrações de nós virtuais, a fim de aliviar a carga dos recursos sobrecarregados (partilhados ou não).

Em suma, a escolha dos nós que suportam uma DHT (seja para endereçamento, seja para armazenamento) é baseada na disponibilidade instantânea dos recursos relevantes e as migrações de nós virtuais ocorrerão, automaticamente, à medida que as disponibilidades desses recursos descerem abaixo de certos limiares. Esta abordagem tem a vantagem de permitir uma utilização progressiva de recursos e de ser compatível com padrões de acesso diferentes por DHT (representando um contributo para o suporte da sua heterogeneidade).

[15] No plano abstracto, a "migração de um nó virtual" traduz-se na modificação da tabela $TD^{\mathsf{f}}(d_{\_})$, nas quantidades $\mathcal{V}^{\mathsf{f}}(d_{\_}, s_{\_})$, $\mathcal{V}^{\mathsf{f}}(d_{\_}, s_{+})$, $\mathcal{H}^{\mathsf{f}}(d_{\_}, s_{\_})$ e $\mathcal{H}^{\mathsf{f}}(d_{\_}, s_{+})$. Na prática, haverá movimentação de dados.

[16] No contexto desta discussão, a espécie (armazenamento/endereçamento) dos nós virtuais é irrelevante.

6.10 Relação com Outras Abordagens

Dois problemas que também se colocam no domínio das DHTs/DDSs, e que Lee et al. [LKO+00] identificam no contexto das Bases de Dados Paralelas auto-adaptativas, são a determinação i) da necessidade de migração de registos e ii) dos registos a migrar. Para resolver o problema i), preconiza-se a monitorização da carga (entre outras métricas locais) em cada nó da Base de Dados (BD); ultrapassado certo limiar, despoleta-se a migração (ou seja, tal como na nossa abordagem, são condições de carga do nó como um todo – e não apenas relativas ao seu suporte à BD paralela – que provocam a migração); essa decisão é tomada por um nó com visão global do estado do nós da BD (de papel semelhante ao supervisor da nossa arquitectura Domus), que ordena e serializa as migrações. Tendo em conta que, em cada nó, os registos locais são indexados/armazenados numa árvore B+, a solução proposta para o problema ii) passa pela monitorização do padrão de acesso (taxa de acesso) a essa árvore B+, com diferentes níveis de granularidade (na arquitectura Domus, cada índice/árvore de endereçamento ou armazenamento, é também sujeita a monitorização, sendo as entradas de cada DHT monitorizadas em conjunto). Adicionalmente, a abordagem de Lee prevê a migração *incremental* ou *em bloco* dos registos das árvores B+, de novo com vários níveis de granularidade (na arquitectura Domus, a unidade mínima de migração, adequada a migração incremental, é a entrada[17] e a unidade máxima, adequada a migração em bloco, é o nó virtual, visto como conjunto de entradas).

As DHTs de primeira geração [LNS93a, Dev93, HBC97, GBHC00] auto-ajustam o número de entradas/contentores às necessidades efectivas de armazenamento, sendo capazes de acomodar distribuições não-uniformes do *número* e *dimensão* dos registos. Na arquitectura Domus, a evolução por estágios determina um número global fixo de entradas/contentores por cada estágio, assumindo-se uma distribuição uniforme do *número* e *dimensão* dos registos pelas entradas, ao longo de cada estágio; porém, com registos de dimensão variável, o recurso à *fragmentação* (*e.g.*, ao nível da biblioteca Domus, logo transparente para os serviços Domus), em *blocos* de igual dimensão (esta configurável como mais um atributo das DHTs) permitiria ainda uniformizar o consumo dos recursos de armazenamento.

Face às DHTs acima referidas, a arquitectura Domus encontra-se mais próxima de abordagens como a SNOWBALL [VBW98] (rever secção 2.8.1). Nesta, para além do balanceamento da carga de armazenamento, procura-se também balancear a carga de acesso aos registos, em resposta a *surtos de acesso* (*hot-spots*); este balanceamento é realizado com a preocupação em manter um certo nível de Qualidade de Serviço (QoS) no acesso à generalidade dos registos (incluindo os mais populares); na arquitectura Domus, o mecanismo mgNv desempenha um papel semelhante. Outras particularidades da abordagem SNOWBALL incluem: a) o armazenamento apenas em Disco[18] (a arquitectura Domus também suporta RAM), b) a admissão da partilha do mesmo nó computacional por clientes e serviços do Dicionário Distribuído (tal como na arquitectura Domus), c) o suporte a um número variável de nós, com escalabilidade linear do desempenho[19] (no capítulo seguinte, demonstra-se o mesmo tipo de escalabilidade num protótipo da arquitectura Domus).

[17] a) um tuplo $< h, TE, RA >$ ou b) um tuplo $<h, RR, RE >$ mais o *conjunto de registos* associado a h.
[18] Como acontece, aliás, na generalidade das DHTs/DDSs de primeira geração.
[19] Especialmente importante numa óptica de custo-benefício, segundo os mesmos autores [VBW94].

Na arquitectura Domus, i) a definição do número de nós virtuais proporcionalmente a capacidades dos nós computacionais, e ii) a sua migração para balanceamento de carga, encontram paralelismos no contexto das DHTs de segunda geração e suas aplicações. Por exemplo, o CFS [DKKM01, Dab01] (um sistema de ficheiros distribuído, já anteriormente referenciado), contempla a definição e a migração referidas (rever secções 2.6.2.2 e 2.8.2).

Relativamente aos esquemas de balanceamento dinâmico de carga investigados por Rao et al. [RLS+03] (rever secção 2.8.2), também baseados na movimentação de nós virtuais, é de salientar que, face aos nossos, apresentam limitações: a) suportam o balanceamento dinâmico de apenas um tipo de recurso; b) não são compatíveis com dinamismo na composição do sistema (em termos de nós computacionais) e no conteúdo da DHT (em termos de registos). Por outro lado, a nossa aproximação assenta na manutenção de informação *global* sobre o estado dos recursos e coordenação *centralizada* do processo de balanceamento, algo que Rao et al. pretendem evitar com as abordagens *one-to-one load-stealing* e *one-to-many load-shedding*, mas para o que acabam por convergir, com a abordagem *many-to-many*, no sentido de conseguirem um balanceamento mais eficaz. Esta aproximação de uma estratégia de balanceamento puramente distribuída, a uma mais centralizada, verifica-se também na abordagem de Godfrey et al. [GLS+04] (rever secção 2.8.2).

Aplicando a terminologia introduzida na secção 2.8, os mecanismos de balanceamento dinâmico da arquitectura Domus actuam ao *nível aplicacional* (caso do mgNv) e ao *nível sistema* (caso do mgPv). Além disso, i) são *cientes-dos-recursos* do *cluster* (como é evidenciado pela uso de medidas de *capacidade*, *utilização*, *potencial* e *atractividade*), ii) são adequados a *clusters heterogéneos* e iii) toleram a co-execução concorrente de outras aplicações. Neste contexto, a nossa abordagem apresenta, claramente, afinidades com o modelo DRUM (*Dynamic Resource Utilization Model*) [Fai05, TFF05] (rever secção 2.8.3). Todavia, o DRUM é essencialmente orientado ao balanceamento de tarefas de computação científica, considerando apenas relevantes as capacidades dos nós em termos de i) poder de processamento e ii) largura de banda; noutra linha, o foco na arquitectura Domus é o armazenamento distribuído, pelo que também são relevantes as capacidades de armazenamento (principal e secundário). O DRUM assenta numa visão hierárquica, em árvore, do ambiente de execução; em contraste, na arquitectura Domus a visão do *cluster* é plana. Finalmente, ao contrário dos mecanismos de gestão dinâmica de carga da arquitectura Domus, o DRUM não efectua, por si só, balanceamento dinâmico de carga embora, como referido na secção 2.8.3, as métricas que produz possam alimentar sistemas desse tipo.

A aplicação de um modelo *linear* ao cálculo dinâmico de capacidades em *clusters* heterogéneos e partilhados é também demonstrada pela abordagem de Sinha et al. [SP01] (rever secção 2.8.3); o foco é, mais uma vez, o cálculo científico, monitorizando-se a RAM, além de CPU e largura de banda; na abordagem Domus, orientada ao armazenamento distribuído, a actividade E/S dos subsistemas de memória secundária é também monitorizada.

6.11 Epílogo

Consumada a descrição dos mecanismos de gestão dinâmica de carga, dá-se por concluída a apresentação da arquitectura Domus, iniciada no capítulo anterior. No próximo capítulo descreve-se um protótipo da arquitectura e discutem-se os resultados da sua avaliação.

Capítulo 7

Protótipo da Arquitectura Domus

Resumo

Neste capítulo descrevem-se os aspectos mais relevantes de um protótipo da arquitectura Domus, no seu estágio actual de realização. A descrição contempla i) os principais componentes de software e sua relação com os componentes da arquitectura, ii) os mecanismos implementados para caracterização e monitorização do *cluster* físico, iii) o acesso às funcionalidades do protótipo por programadores e administradores. O capítulo termina com a apresentação e discussão dos resultados de um conjunto de testes de avaliação do protótipo.

7.1 Prólogo

Um protótipo é, por definição, uma realização parcial de uma certa especificação, através da qual se procura demonstrar a exequibilidade de certos modelos. Neste contexto, o protótipo da arquitectura Domus (ou, simplesmente, "protótipo Domus") que apresentamos neste capítulo, permite a exercitação da *co-operação de DHTs*, mas apenas suporta parte dos conceitos e mecanismos ligados à *gestão dinâmica de carga*, descritos no capítulo 6.

Em particular, a implementação actual permite a criação, exploração e gestão de múltiplas DHTs que, embora estáticas, beneficiam de i) distribuição inicial pesada, ii) suporte à maior parte dos atributos previstos, iii) suporte à totalidade das operações de acesso e administração. Adicionalmente, os testes realizados permitem concluir que a plataforma desenvolvida, apesar de ser ainda um protótipo, revela elevada estabilidade e níveis de desempenho que a tornam competitiva com outros repositórios de dicionários comparáveis.

7.2 Ferramentas e Tecnologias Utilizadas

Como previsto na secção 5.4, a arquitectura Domus pressupõe a disponibilidade de um conjunto de funcionalidades base de a) passagem de mensagens, b) monitorização de re-

cursos e c) execução remota. Neste contexto, o protótipo deitou mão, respectivamente, dos seguintes mecanismos: a) *sockets* BSD sobre TCP/IP, b) sistema Ganglia [DSKMC03] (apoiado por serviços Domus de monitorização – ver secção 7.6.2), c) serviços Secure-Shell.

A principal linguagem de desenvolvimento do protótipo foi a linguagem de alto nível Python [pyt], opção que permitiu grande produtividade na codificação e exercitação do protótipo. Para a codificação de certos componentes, com maiores requisitos de desempenho, recorreu-se à linguagem C, tendo-se usado a ferramenta SWIG [swi] para interligar o código produzido em ambas as linguagens (ver secção 7.3.1). Foram ainda utilizadas diversas bibliotecas e plataformas de código aberto; destas, destaca-se a plataforma de armazenamento Berkeley-DB [OBS99, ber], entre outras a referenciar oportunamente.

O sistema de exploração alvo do protótipo foi o Linux, mais especificamente a distribuição ROCKS [roc], actualmente um referencial no contexto dos vários ambientes de exploração e gestão de *clusters* existentes (rever secção 2.12). O recurso a tecnologia VMware [vmw] para a virtualização do ambiente distribuído numa estação de trabalho portátil permitiu, sempre que necessário, independência da disponibilidade efectiva de um *cluster* físico.

7.3 Componentes do Protótipo

O protótipo Domus actual baseia-se nos componentes de software representados na figura 7.1. Na figura, as setas contínuas veiculam o recurso de um componente (ponto de partida da seta) às funcionalidades de um outro (ponto de chegada da seta), e as setas tracejadas unem os pares de componentes envolvidos em trocas de mensagens (sendo evidenciado que essa troca se processa recorrendo a funcionalidades do componente `domus_libsys.py`).

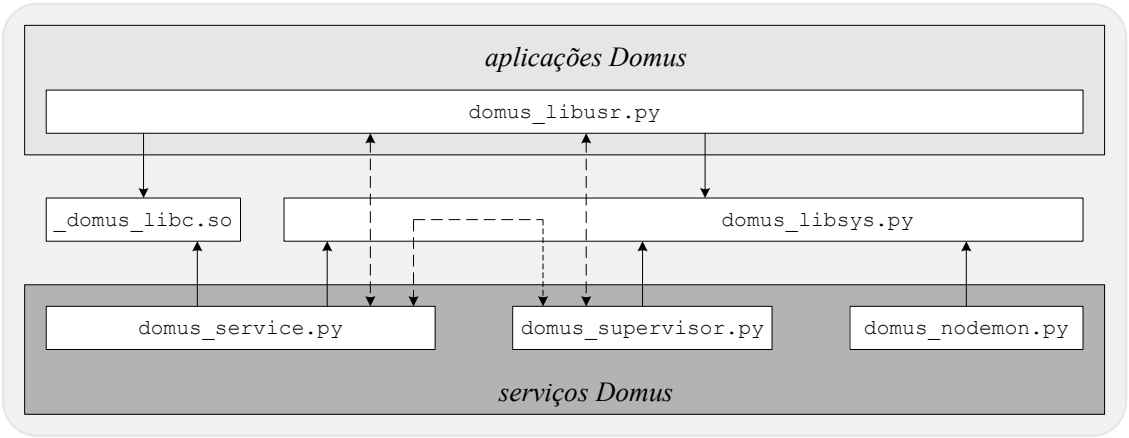


Figura 7.1: Componentes de Software do protótipo Domus (dependências e comunicação).

O protótipo foi maioritariamente codificado em Python, com a excepção do módulo `_domus_libc.so`, codificado em C. O papel dos componentes do protótipo é o seguinte:

- `domus_libusr.py`: biblioteca Domus, prevista pela arquitectura (rever secção 5.9.1);

- `_domus_libc.so`: biblioteca de funcionalidades de endereçamento e armazenamento;
- `domus_libsys.py`: constantes e métodos re-utilizados pelos outros componentes;
- `domus_service.py`: serviço Domus regular;
- `domus_supervisor.py`: serviço Domus supervisor;
- `domus_nodemon.py`: serviço Domus de monitorização, complementar ao Ganglia.

A operação dos componentes que realizam as entidades da arquitectura (aplicações e serviços Domus) é discutida ao longo do capítulo. Os componentes de software auxiliares (módulos `_domus_libc.so` e `domus_libsys.py`) são descritos de seguida, abreviadamente.

7.3.1 Biblioteca `_domus_libc.so`

A biblioteca `_domus_libc.so` suporta um conjunto de funcionalidades de endereçamento e armazenamento codificadas em C; essas funcionalidades assentam na exploração de estruturas de dados em árvore do tipo Red-Black Tree, realizadas por uma outra biblioteca específica [Ive03], também de interface em C (esta biblioteca é a mesma que foi utilizada nas simulações dos algoritmos de localização sobre grafos Chord, relatadas no capítulo 4).

A utilização das funcionalidades da biblioteca `_domus_libc.so`, de interface em C, a partir dos módulos `domus_libusr.py` e `domus_service.py`, em Python, é intermediada por código gerado automaticamente pela ferramenta SWIG [swi], com base numa especificação.

Registe-se que a utilização das funcionalidades da biblioteca `_domus_libc.so` a partir de módulos Python obriga a cuidados especiais: um objecto Python não deve ser eliminado/reciclado sem antes se terem libertado os recursos que solicitou à biblioteca, dado que esses recursos estão fora do alcance dos mecanismos de *garbage collection* do Python.

7.3.2 Biblioteca `domus_libsys.py`

Este módulo comporta constantes e funcionalidades utilizadas pelos outros componentes:

- valores admissíveis (e por omissão), dos atributos Domus suportados pelo protótipo;
- códigos de retorno dos métodos públicos e explicações eventualmente associadas;
- valores por omissão de parâmetros e atributos da i) utilização do sistema de ficheiros pelos serviços Domus e ii) interacção com o sistema de monitorização Ganglia;
- funcionalidades de i) arranque de serviços em segundo plano, ii) interrogação ao sistema Ganglia (ver secção 7.6.2.2) e iii) gestão da comunicação (ver secção 7.10.1);

Alguns dos parâmetros cujos valores por omissão são definidos na biblioteca, podem ser também valorados através de um ficheiro de configuração, a discutir na secção 7.12.2.

7.4 Suporte ao Endereçamento

7.4.1 Índices de Endereçamento

A arquitectura prevê (rever secção 5.6.5.1) que o Subsistema de Endereçamento de um serviço Domus concentre a *informação de endereçamento* local de cada DHT num *índice de endereçamento*, compatível com os algoritmos de localização definidos no capítulo 4.

No protótipo, cada *índice de endereçamento* é uma árvore Red-Black Tree, gerida pela biblioteca `_domus_libc.so`. Mais especificamente, a biblioteca suporta uma "árvore de *índices de endereçamento*", realizada como "Red-Black Tree de Red-Black Trees", a fim de gerir, de forma integrada, todos os *índices de endereçamento* de um serviço Domus.

As funcionalidades relevantes associadas aos *índices de encaminhamento* não se limitam às relacionadas com localização distribuída (ver a seguir), incluindo também métodos de salvaguarda/serialização e recuperação/reconstrução das suas árvores em/de disco, essenciais para o suporte das operações de desactivação/reactivação previstas pela arquitectura.

7.4.2 Localização Distribuída

O protótipo realiza todos os algoritmos de localização distribuída previstos pela especificação (rever secção 5.8.3.6); essa realização é também concretizada pela biblioteca `_domus_libc.so`, tirando partido dos *índices/árvores de endereçamento* nela alojados.

A localização distribuída assenta na utilização do protocolo UDP. Para que a resposta a um pedido de localização seja enviada à entidade originadora, o corpo (*payload*) do pedido de localização inclui um par <endereço IP,porto> a usar como destino da resposta.

7.4.3 Estratégias de Localização

O protótipo também implementa as *estratégias de localização* previstas pela especificação, para as aplicações Domus (rever secção 5.9.1.3). A implementação é feita pela biblioteca Domus (`domus_libusr.py`) em parceria com a biblioteca auxiliar `_domus_libc.so`.

Em particular, a biblioteca `_domus_libc.so` realiza as *caches de localização* (rever secção 5.6.5.3), de forma semelhante às *árvores/índices de endereçamento* (ver acima); uma *cache de localização* assenta pois numa estrutura do tipo Red-Black Tree, sobre a qual ainda são aplicáveis os algoritmos de localização do capítulo 4; na realidade, em vez de uma, são usadas duas Red-Black Trees, sobrepostas no mesmo conjunto de registos da *cache*; numa delas, a ordenação dos registos é apropriada à aplicação dos algoritmos de encaminhamento (ordenação simples, pelo campo *hash* dos registos); na outra, a ordenação é baseada em marcas temporais de acesso (uma por registo), suportando uma política de gestão LRU.

7.5 Suporte ao Armazenamento

7.5.1 Tecnologias de Armazenamento

A arquitectura Domus prevê a possibilidade de o Subsistema de Armazenamento de um serviço Domus recorrer a *tecnologias de armazenamento* diversas, para realizar *repositórios*

locais de dicionários (rever secção 5.6.6.2). De acordo com a especificação da arquitectura, uma *tecnologia de armazenamento* corresponde a uma combinação < *plataforma de armazenamento*, *meio de armazenamento* >, seleccionável para cada DHT em particular, através dos atributos `attr_dht_pa` e `attr_dht_ma`, respectivamente (rever secção 5.8.3.7).

As tecnologias de armazenamento actualmente suportadas pelo protótipo são as seguintes:

1. <`python-dict`, `ram`> : dicionários nativos da linguagem Python, sobre RAM; são dicionários não persistentes, bastante eficientes e de extrema versatilidade (guardam correspondências entre objectos de qualquer tipo, sem necessidade de serialização);

2. <`python-cdb`, `disk`> : dicionário "constante", sobre Disco, gerido pelo módulo Python `cdb` [cdb]; é um dicionário persistente, de acesso muito eficiente (ver secção 7.13.1) mas versatilidade limitada, suportando apenas um padrão de acesso *Write-once-Read-many* (este pressupõe duas fases distintas de operação: na 1ª ocorrem apenas inserções e na 2ª ocorrem apenas consultas[1], não se permitindo remoções);

3. <`python-bsddb-btree`/`hash`, `ram`/`disk`>: dicionário BerkeleyDB[2] gerido pelo módulo Python `bsddb` [bsd]; realizável com uma Árvore B+ (`python-bsddb-btree`) ou com uma Tabela de Hash Dinâmica (`python-bsddb-hash`), sobre RAM ou Disco;

4. <`domus-bsddb-btree`/`hash`, `ram`/`disk`>: tecnologias semelhantes às anteriores mas mais eficientes (ver secção 7.13.1); o acesso aos dicionários BerkeleyDB processa-se pelo seu interface em C, a partir da biblioteca `_domus_libc.so` (ver secção 7.5.3).

As tecnologias das categorias 2, 3 e 4 implicam todas a serialização (encapsulamento numa sequência de caracteres) prévia dos campos de um registo <*chave*, *dados*> antes da sua inserção num repositório, sendo necessário efectuar o processo reverso antes da devolução dos resultados de uma consulta à entidade que a solicitou. A perda de desempenho em que incorre este processo é compensada pela flexibilidade derivada da possibilidade de os registos serem um par de objectos Python de qualquer tipo (desde que serializáveis[3]).

Ao contrário da tecnologia <`python-cdb`, `disk`>, as das categorias 1, 3 e 4 suportam todas as operações básicas de acesso a dicionários (inserções, consultas e remoções), com a única restrição de que o repositório respectivo deve estar acessível/*activo* (*online*); um repositório diz-se activo/inactivo em função do estado activo/inactivo da DHT associada.

7.5.2 Granularidade dos Repositórios

No protótipo, cada entrada de uma DHT é suportada por um repositório específico, representando a adopção de uma abordagem de *grão fino*, pela classificação da secção 5.6.6.2. Uma vez que o protótipo não permite acesso concorrente intra-repositório (embora o permita inter-repositório), a abordagem de *grão fino* tem a vantagem de oferecer maior paralelismo potencial no acesso às DHTs; além disso, permitirá agilizar o processo de redistribuição das DHTs, uma vez que os registos de cada entrada estarão à partida isolados num

[1] No protótipo, a transição da 1ª para a 2ª fase é despoletada pela inserção do registo <`None`,`None`>.

[2] Na modalidade mais eficiente fornecida pela plataforma BerkeleyDB; nessa modalidade, não há suporte a acesso concorrente, sendo o problema resolvido por mecanismos próprios do protótipo – ver secção 7.10.2.

[3] Certos objectos, como por exemplo *trincos* (*locks*), não são serializáveis.

repositório dedicado, não necessitando de serem extraídos de um repositório partilhado. Uma desvantagem óbvia é a maior *sobrecarga* espacial dos recursos de armazenamento.

7.5.3 Índices de Armazenamento

De acordo com a especificação da arquitectura, o Subsistema de Armazenamento de um serviço Domus deverá concentrar a *informação de armazenamento* local de cada DHT num *índice de armazenamento*; esse índice faz corresponder a cada *hash* um par <*RE*,*RR*>, em que *RE* (*referência de endereçamento*) identifica o serviço de endereçamento do *hash* e RR (*referência de repositório*) é uma referência para o repositório associado ao *hash* (rever secção 5.6.6.1). No protótipo, a realização do *índice de armazenamento* é influenciada pela adopção de granularidade fina dos repositórios, e pela tecnologia de armazenamento usada.

Assim, para as tecnologias baseadas nas plataformas `python-dict`, `python-cdb` e `pyhton-bsddb-*` (ou para qualquer a adoptar, desde que com acesso nativo a partir do Python), o *índice de armazenamento* é um simples dicionário Python, que associa cada *hash* a um par <*RE*,*RR*>, em que RR é uma referência *interna* para o repositório específico do *hash*.

Para tecnologias baseadas nas plataformas `domus-bsddb-*` (ou para qualquer outra que se venha a suportar, em que o acesso ao repositório recorra a um interface em C), os *índices de armazenamento* residem no contexto da biblioteca `_domus_libc.so` e os serviços Domus mantêm referências *externas* para eles; é também a partir desse contexto que são operados os repositórios que são, neste caso, dicionários Berkeley-DB. De forma análoga aos *índices de endereçamento*, i) um *índice de armazenamento* é realizado pela biblioteca `_domus_libc.so` através de uma árvore Red-Black Tree e ii) a mesma biblioteca gere uma "árvore de *índices de armazenamento*", realizada como "Red-Black Tree de Red-Black Trees", suportando assim vários *índices de armazenamento* de um mesmo serviço Domus.

7.6 Caracterização e Monitorização de Recursos

Como referido anteriormente, o protótipo ainda não suporta a redistribuição de DHTs. Todavia, a sua criação com base nos modelos preconizados pela especificação exige um nível mínimo de caracterização de certas entidades do cluster Domus. Essa caracterização passa pela determinação de certos atributos/medidas de *capacidades* e *utilizações* dos nós computacionais, bem como de *potencial* e *atractividade* dos serviços Domus, baseadas nas primeiras. Nesta secção concentra-mo-nos apenas nos mecanismos usados ao nível do protótipo para a determinação, disponibilização e consulta das *capacidades* e *utilizações*[4].

7.6.1 Abrangência da Caracterização e Monitorização

Os nós computacionais a caracterizar são os que poderão vir a alojar serviços Domus e que correspondem, em termos formais, ao conjunto dos nós que suportam serviços básicos, $N(B)$ (ou, então a um subconjunto deste). Inclusivamente, é suposto que nesses nós execute um serviço de monitorização, em consonância com o previsto pela arquitectura.

[4]O protótipo também suporta a definição dos *limiares* previstos nas secções 5.8.1.5 e 6.3.2; todavia, na ausência de suporte à redistribuição, a definição dos *limiares* não tem, por enquanto, consequências.

7.6.2 Serviços de Caracterização e Monitorização

O ambiente de execução do protótipo, fornecido pelo sistema ROCKS de exploração de *clusters*, garante a presença do serviço de monitorização Ganglia em todos os nós do *cluster* (N), sendo da responsabilidade do administrador de um cluster Domus garantir a execução adicional de um serviço Domus de monitorização (`domus_nodemon.py`) em $N(B)$. Este serviço justifica-se pelo facto de o Ganglia não produzir, originalmente, certas métricas necessárias ao protótipo, tendo-se optado por desenvolver um serviço complementar[5]. O serviço injecta as métricas por si produzidas no sistema Ganglia, permitindo tirar partido das melhores funcionalidades deste: i) um repositório com todas as métricas produzidas no *cluster* e ii) um *frontend* WWW que permite acompanhar a evolução dessas métricas.

Aparentemente, seria mais adequado que as funcionalidades desempenhadas pelos serviços `domus_nodemon.py` fossem executadas pelo Subsistema de Balanceamento dos serviços Domus regulares[6]. Todavia, a execução dessas funcionalidades por um serviço dedicado i) evita a sua duplicação numa situação em que existem várias instâncias do protótipo em execução com partilha de nós computacionais; ii) torna mais leve um serviço Domus regular. O funcionamento de um serviço `domus_nodemon.py` é, portanto, agnóstico no que diz respeito aos clusters Domus em execução[7], limitando-se a produzir métricas sobre recursos eventualmente partilhados entre eles; depois, caberá aos serviços de cada cluster Domus seleccionar as métricas que lhes interessam. Por estas razões, um serviço `domus_nodemon.py` está mais próximo dos "serviços básicos" do que dos "serviços Domus".

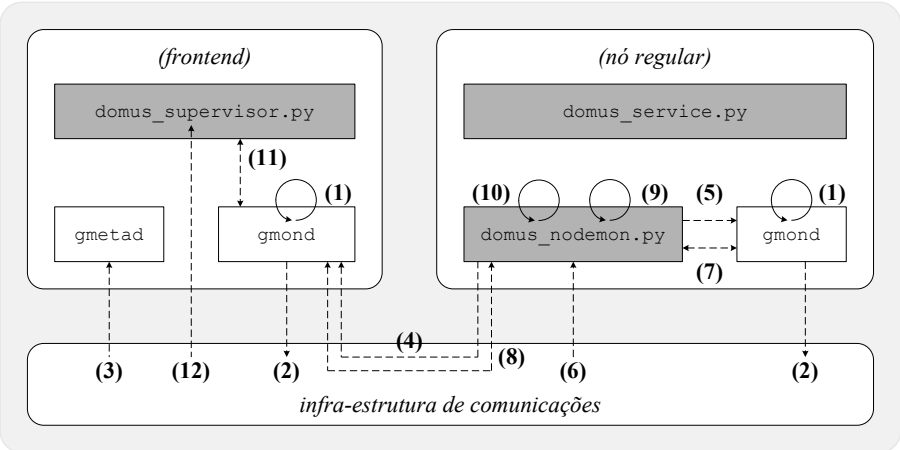


Figura 7.2: Recolha, Publicação e Acesso a Métricas.

A figura 7.2 ilustra a forma como o sistema Ganglia e os serviços Domus interagem na recolha, publicação e acesso a várias métricas, servindo de apoio à descrição que se segue.

[5]Esta complementaridade surge também noutros sistemas, como no DRUM [Fai05], onde o serviço distribuído de monitorização NWS [WSH99] complementa a operação de fios-de-execução de monitorização do DRUM. No contexto do nosso protótipo, desenvolveram-se inicialmente *plugins* Ganglia, baseados em *scripts* de injecção de métricas, escalonadas com base no serviço `cron` (abordagem clássica); porém, a falta de um mecanismo de gestão integrada das *scripts* espalhadas pelo *cluster*, motivou a mudança de estratégia.

[6]E, numa fase inicial do desenvolvimento do protótipo, assim acontecia.

[7]Ou seja, a produção, o nome e o valor das métricas é independente dos clusters Domus em execução.

7.6.2.1 Operação dos Serviços Ganglia

Em traços gerais, o Ganglia opera da seguinte forma num ambiente ROCKS[8]: em cada nó do *cluster* um serviço `gmond` recolhe periodicamente certas métricas (**1**) e difunde-as, via *multicast IP* (**2**); as métricas assim produzidas são acumuladas num serviço `gmetad` (**3**), residente no *frontend* do *cluster*; o serviço `gmetad` concentra assim informação de monitorização que lhe proporciona uma visão global do estado dos recursos do *cluster*.

7.6.2.2 Acesso às Métricas Ganglia

No âmbito do protótipo existem duas hipóteses de aceder às métricas produzidas pelo Ganglia: 1) inquirição a um serviço `gmond`; 2) intercepção das suas mensagens de difusão.

Na hipótese 1), a interrogação é feita recorrendo a funcionalidades da biblioteca `domus_libsys.py` e pode processar-se de duas formas distintas: 1.i) através de conexão TCP e filtragem de um relatório XML obtido como resposta; 1.ii) através de execução local/remota, no hospedeiro do serviço `gmond`, do comando de interrogação `ganglia`, sendo o resultado capturado pelo processo invocador através da primitiva `popen`. De facto, durante a experimentação com o protótipo, a opção 1.i) revelou-se pouco estável, o que ditou a necessidade da opção 1.ii) que, apesar de estável, tem a desvantagem de ser mais lenta.

Na hipótese 2), as entidades interessadas nas métricas (os serviços `domus_nodemon.py` e `domus_supervisor.py`) têm de se associar ao grupo *multicast IP* do Ganglia, a fim de interceptarem as mensagens de monitorização; essas mensagens vêm codificados em representação XDR, sendo abertas recorrendo aos bons ofícios do módulo Python `xdrlib`. Todavia, este método só é realmente efectivo para métricas dinâmicas, actualizadas e difundidas frequentemente; para conhecer métricas estáticas ou de actualização pouco frequente, acabará por ser indispensável a interrogação de serviços `gmond` (hipótese 1)).

7.6.2.3 Operação dos Serviços `domus_nodemon.py`

As métricas produzidas por um serviço `domus_nodemon.py` são, por omissão, injectadas directamente no serviço `gmond` (**4**) do *frontend*, recorrendo à execução remota (via `ssh`) do comando `gmetric` do Ganglia; alternativamente, essas métricas poderiam ser injectadas no serviço `gmond` local (**5**), recorrendo ao mesmo comando, sendo depois difundidas juntamente com as restante métricas do serviço (**2**); em ambos os casos, a execução do comando `gmetric` carece de permissões de superutilizador[9], facilmente geríveis para casos particulares através de mecanismos `sudo`; este requisito de segurança acaba por favorecer a primeira abordagem (**4**), na qual é apenas necessário intervir na máquina de *frontend*.

Um serviço `domus_nodemon.py` gera e publica "métricas Domus" que veiculam as *capacidades* e *utilizações* previstas pela especificação, sendo diversas as fontes que alimentam a produção dessas métricas. Assim: i) algumas dessas métricas podem depender de outras geradas pelo serviço `gmond` vizinho, tendo o serviço `domus_nodemon.py` a capacidade de i-a) as interceptar (**6**) quando se dá a sua difusão ou i-b) de interrogar o `gmond` compa-

[8]Esta descrição é simplificada, adaptada ao nosso cenário de operação. Com efeito, o sistema Ganglia tem pontencialidades adicionais que lhe permitem, por exemplo, monitorizar federações de *clusters*.

[9]Uma restrição imposta pelo ROCKS, e não pelo sistema Ganglia em si.

nheiro (**7**) ou mesmo o `gmond` do *frontend* (**8**); a abordagem i-a) é apropriada à obtenção de métricas dinâmicas, difundidas periodicamente pelo `gmond` local, e que são usadas no cálculo de métricas de *utilizações*; já a abordagem i-b) é mais adequada à obtenção de métricas estáticas, difundidas menos frequentemente, e usadas na definição de *capacidades*; ii) a definição de algumas *utilizações* depende de métricas dinâmicas obtidas localmente pelo serviço `domus_nodemon.py` (**9**); iii) o valor de algumas *capacidades* é directamente extraído de repositórios externos[10], que guardam o resultado de *micro-benchmarks* (**10**).

7.6.2.4 Acesso às Métricas Domus

Para se poderem criar DHTs, o supervisor de um cluster Domus necessita de conhecer as *capacidades* e *utilizações* dos nós computacionais que executam serviços Domus. Assim, quando um cluster Domus é criado, ou quando um novo serviço Domus lhe é adicionado, o supervisor interroga o serviço `gmond` do *frontend* sobre as métricas dos nós computacionais em causa (**11**). Adicionalmente, o supervisor escuta o canal *multicast IP* do Ganglia, filtrando as métricas de *utilizações* que dizem respeito aos nós do cluster Domus (**12**). Basicamente, estes dois mecanismos correspondem aos descritos na secção 7.6.2.2, dado que as "métricas Domus", uma vez inseridas no Ganglia, serão também "métricas Ganglia".

Suportando-se a redistribuição de DHTs, os serviços Domus regulares passariam a ser, ao mesmo tempo, produtores e consumidores de métricas (produtores de métricas de caracterização de DHTs (rever secção 6.6) e consumidores de *capacidades* e *utilizações* (rever secção 6.8)), podendo recorrer aos métodos aqui descritos para a sua disseminação/obtenção.

7.6.2.5 Tempo de Vida das Métricas Domus

O sistema Ganglia suporta a definição de um tempo de vida limitado para as métricas, findo o qual as métricas são descartadas (eliminadas). Por outro lado, os vários serviços Domus também suportam a definição de um tempo de vida limitado, após o que terminam (ver secção 7.12.2). Neste contexto, o tempo de vida das métricas Domus corresponde ao tempo de vida dos serviços `domus_nodemon.py` que as produzem[11]. O objectivo é evitar a sobrecarga do sistema Ganglia pela manutenção duradoura de métricas que são inúteis.

7.6.2.6 Frequências da Caracterização

Nos serviços `gmond` do Ganglia, a frequência máxima de difusão de métricas é de uma vez por minuto, para certas métricas dinâmicas (taxas de utilização de recursos). Para métricas estáticas, o intervalo entre difusões sucessivas pode chegar a ser de horas. No Ganglia, as frequências de difusão são configuráveis, mas isso exige privilégios de super-utilizador e, eventualmente, intervenção manual (no caso de não ser possível propagar automaticamente a reconfiguração). Por outro lado, cada vez que se injecta externamente (via comando `gmetric`) uma métrica em serviços `gmond`, a métrica é difundida instantaneamente, para além de também ser possível definir, dessa forma, a frequência de re-difusão; estas particularidades podem ser exploradas para contornar a necessidade de reconfigurar

[10] Repositórios necessários por não haver suporte à persistência de estado dos serviços `domus_nodemon.py` entre execuções sucessivas, e o tempo de vida das métricas por eles geradas ser limitado (ver secção 7.6.2.5).

[11] Sendo suposto que os clientes das métricas não têm um tempo de vida superior ao das métricas.

globalmente os serviços `gmond`. Além disso, convém notar que a frequência de difusão não corresponde necessariamente à frequência de extracção/amostragem das métricas na fonte: no Ganglia, a frequência máxima de extracção é de uma vez por minuto, tornando inútil uma difusão mais frequente (*e.g*, de meio em meio minuto) de métricas colectadas internamente pelo Ganglia, já que serão iguais dois ou mais valores sucessivos das métricas.

Nos serviços `domus_nodemon.py`, a frequência de amostragem e a de publicação das métricas são iguais. As métricas estáticas são obtidas e publicadas uma só vez. A frequência de amostragem/publicação das métricas dinâmicas é configurável, em função do tipo de recurso do nó; esta frequência é, por omissão, de uma vez por minuto; todavia, como veremos na secção 7.6.5.3, certos recursos exigem frequências mais elevadas. Como o serviço `domus_nodemon.py` recorre ao comando `gmetric` para injectar as suas métricas no Ganglia, beneficia da sua difusão imediata, a qual ocorre à cadência determinada por si.

7.6.3 Identificação das Métricas Domus no Ganglia

Para distinguir[12] as métricas geradas nativamente pelo sistema Ganglia, das "métricas Domus", que veiculam *capacidades* e *utilizações*, é usada uma nomenclatura própria para a sua identificação, resumida nas tabelas 7.1 e 7.2. A nomenclatura procura evitar que o valor das métricas seja estruturado, dado que valores unos e escalares facilitam a publicação das métricas no sistema Ganglia e posterior recuperação. Por seu turno, o nome das métricas é estruturado em torno do símbolo `#`, permitindo parametrizar esse nome de forma compatível com a notação formal das métricas: todos os identificadores incorporam o *nome primário* do nó computacional[13] (*node*) bem como outros parâmetros necessários (*algorithm*, *technology*, *interface* e *partition*). A estruturação adoptada permite também extrair, facilmente, os parâmetros embebidos nos identificadores, conforme necessário[14].

Capacidades	**Métricas Domus**	**Unidades**
$\mathcal{C}^{e}$(*algorithm,node*)	`domus#static#`*node*`#routing_throughput#`*algorithm*	Koperações/s
$\mathcal{C}^{a}$(*technology,node*)	`domus#static#`*node*`#storage_throughput#`*technology*	Koperações/s
$\mathcal{C}$(*interface,node*)	`domus#static#`*node*`#net_bandwith#`*interface*	Mbps
$\mathcal{C}$(*ram,node*)	`domus#static#`*node*`#ram_total`	Mbytes
$\mathcal{C}$(*partition,node*)	`domus#static#`*node*`#part_total#`*partition*	Gbytes

Tabela 7.1: Correspondência entre Atributos e Métricas de Capacidades.

7.6.4 Caracterização das Capacidades dos Nós Computacionais

No protótipo, as métricas de *capacidades* previstas na secção 6.3.1 podem ser determinadas por dois meios: a) por *micro-benchmarks* (realizados apenas uma vez por cada espécie de nós do *cluster*); b) por interrogações ao serviço `gmond` do Ganglia e/ou ao sistema operativo.

[12] A distinção facilita o reconhecimento das métricas no *frontend* WWW do Ganglia e o seu acesso via `gmond`.

[13] Dado pelo comando UNIX `hostname`, ou pela primitiva `gethostname`.

[14] Por exemplo, para se saber quais os *interfaces* de um determinado *node*, basta filtrar, no relatório XML do Ganglia, as linhas que obedecem ao padrão *`domus#static#`*node*`#net_bandwith#`*. Esta informação é relevante, por exemplo, para o supervisor, para a definição apropriada de estruturas de dados internas.

Utilizações	Métricas Domus
$\mathcal{U}$(*cpu,node*)	`domus#dinamic#`*node*`#cpu_utilization`
$\mathcal{U}$(*iopartition,node*)	`domus#dinamic#`*node*`#iodisk_utilization#`*partition*
$\mathcal{U}$(*interface,node*)	`domus#dinamic#`*node*`#net_utilization#`*interface*
$\mathcal{U}$(*ram,node*)	`domus#dinamic#`*node*`#ram_utilization`
$\mathcal{U}$(*partition,node*)	`domus#dinamic#`*node*`#disk_utilization#`*partition*

Tabela 7.2: Correspondência entre Atributos e Métricas de Utilizações.

7.6.4.1 Capacidades de Encaminhamento e Acesso

Tendo em vista a determinação das métricas correspondentes às *capacidades de encaminhamento* e às *capacidades de acesso* de cada nó, recorre-se à execução dos *micro-benchmarks* `domus_benchmark_routing_throughput` e `domus_benchmark_storage_throughput`.

A aplicação `domus_benchmark_routing_throughput` re-utiliza parte do código desenvolvido para a realização das simulações dos algoritmos de encaminhamento[15], discutidas no capítulo 4; as simulações então conduzidas permitiram determinar uma métrica de "Tempo de CPU por Salto" ($\overline{CPU}_{\text{hop}}$), para cada algoritmo de encaminhamento (rever secção 4.9.5); os valores inversos dessa métrica são as capacidades de encaminhamento.

A aplicação `domus_benchmark_storage_throughput` re-utiliza o código desenvolvido no protótipo, no quadro do Suporte ao Armazenamento (ver secção 7.5), sendo assim de aplicabilidade limitada às tecnologias de armazenamento presentemente suportadas. O tipo de resultados (posicionamento relativo e valores absolutos) que este *micro-benchmark* permite obter é coerente com os resultados da avaliação apresentada na secção 7.13.1.4[16].

7.6.4.2 Máxima Largura de Banda Útil

A nomenclatura adoptada para a métrica da *máxima largura de banda útil* (correspondente a `domus#static#`*node*`#net_bandwith#`*interface*) suporta a caracterização de qualquer interface de rede de um nó. Por outro lado, um serviço Domus será associado a um único interface (designado, na secção 5.8.2.1, por "interface Domus"). A escolha dos interfaces de rede a caracterizar é então uma decisão administrativa, baseada em expectativas/previsões acerca dos interfaces de rede que se esperam vir a usar para suportar serviços Domus.

Para o cálculo da *máxima largura de banda útil* de um interface de rede recorreu-se à ferramenta Iperf [ipe]. Na sua utilização, convém ter presente o referido na secção 6.3.1: a partir de um interface de rede i, um nó n poderá aceder a $N(i,n)$ nós, no mesmo segmento de rede; num *cluster* heterogéneo, $N(i,n)$ poderá comportar nós de diferentes espécies, o que exige a avaliação da largura de banda entre o nó n e um nó qualquer de cada espécie.

[15] Código que viria a ser também utilizado pelo protótipo no Suporte ao Endereçamento, c.f. já referido.

[16] Partindo-se do princípio de que o desempenho do acesso ao disco é semelhante para as várias partições do sistema de ficheiros de um nó; caso contrário, seria necessário repetir, para cada partição, a avaliação das tecnologias de armazenamento sobre Disco, bem como incorporar a identificação *partition* da partição no nome da métrica, que seria então `domus#static#`*node*`#storage_throughput#`*technology*`#`*partition*.

7.6.4.3 Capacidade de Memória Primária (RAM)

O Ganglia disponibiliza uma métrica `mem_total`, que fornece a informação pretendida. Todavia, por questões de nomenclatura, a métrica é republicada pelos serviços `domus_nodemon.py` com o nome `domus#static#`*node*`#ram_total`. O valor publicado é o da métrica `mem_total`, solicitada pelos serviços `domus_nodemon.py` aos serviços `gmond`.

7.6.4.4 Capacidade de Memória Secundária (Partições)

Assim como um nó pode suportar vários interfaces de rede, o mesmo pode acontecer com partições do sistema de ficheiros. Neste contexto, colocam-se o mesmo tipo de questões levantadas para os interfaces: i) no protótipo, as necessidades de armazenamento secundário de um serviço Domus serão satisfeitas a partir de uma única partição, designada de "partição Domus"; ii) a selecção das partições a monitorizar é uma decisão administrativa.

Neste caso, o sistema Ganglia é inútil, pois contempla métricas que fornecem apenas a capacidade total (`disk_total`) e livre (`disk_free`) de memória secundária de um nó, sem discriminar partições. Consequentemente, essa caracterização mais fina tem de ser feita pelo serviço `domus_nodemon.py`, servindo-se para o efeito dos resultados do comando `df`. Por omissão, são caracterizadas todas as partições, mas é possível definir um subconjunto.

É ainda de realçar que, no protótipo, o termo partição não tem uma conotação necessariamente local; uma designação mais correcta seria *ponto de acesso* (*mount point*), o que é compatível com a exploração, através da rede, de suportes de armazenamento remotos. Porém, esta possibilidade comporta a realização da necessária avaliação da *capacidade de acesso*, para além de não ser suportada pela ferramenta `iostat` de monitorização de actividade E/S, a qual actua apenas sobre discos e partições locais (ver secção 7.6.5.2).

7.6.4.5 Repositórios Externos dos Resultados dos Benchmarks

No protótipo, os resultados dos *micro-benchmarks* de avaliação de *capacidades* são actualmente armazenados em ficheiros de texto, de localização bem conhecida e baseados numa sintaxe simples (que usa a nomenclatura adoptada). Numa próxima iteração do protótipo, deverá ser utilizada a base de dados MySQL que o ROCKS mantém no *frontend* do *cluster* (o que exige, todavia, privilégios administrativos especiais; neste sentido, outra hipótese seria o recurso a um único ficheiro, *e.g.*, em formato XML, como no DRUM [TFF05]).

7.6.5 Monitorização das Utilizações dos Nós Computacionais

No protótipo, as métricas de *utilização* previstas pela especificação (rever secção 6.3.2) são todas produzidas pelos serviços `domus_nodemon.py`, i) nalguns casos recorrendo a métricas geradas pelo sistema Ganglia, e ii) noutros pela monitorização local dos recursos. As métricas de *utilizações* apresentam todas valores reais adimensionais, no intervalo]0.0,1.0[. Os valores das *utilizações* são médias móveis exponenciais, como previsto na secção 6.3.2.

7.6.5.1 Utilização da CPU

A métrica de *utilização de cpu* (`domus#dinamic#`*node*`#cpu_utilization`) é alimentada pela soma das métricas `cpu_user`, `cpu_system` e `cpu_nice` do Ganglia; numa fase inicial

do desenvolvimento do protótipo, estas métricas eram capturadas do canal *multicast IP*, onde são injectadas pelos serviços `gmond` de minuto em minuto; todavia, constatou-se que nem sempre era possível capturar todas as três métricas da mesma janela amostral, o que impedia a publicação da *utilização de cpu* até à próxima difusão; assim, optou-se por interrogar explicitamente o Ganglia, o que é mais demorado mas produz resultados fiáveis.

7.6.5.2 Utilização E/S dos Discos

Para definir métricas `domus#dinamic#`*node*`#iodisk_utilization#`*partition* é necessário determinar o nível de actividade E/S dos discos locais e discriminá-lo por partições. Uma vez que o Ganglia não suporta métricas adequadas, o serviço `domus_nodemon.py` recorre ao comando `iostat` do pacote de monitorização SYSSTAT [sys], fornecido pelo ROCKS.

7.6.5.3 Utilização dos Interfaces de Rede

O Ganglia gera métricas `bytes_in` e `bytes_out`, que traduzem a quantidade total de bytes que um nó transaccionou com a rede, mas acumulada para todos os interfaces, prevenindo a sua utilização na produção de métricas `domus#dinamic#`*node*`#net_utilization#`*interface*. Este constrangimento determina que a monitorização da utilização dos interfaces de rede tenha de ser feita pelo serviço `domus_nodemon.py`. Para o efeito, efectua-se a amostragem de contadores de tráfego específicos para cada interface, acessíveis via `/proc/net/dev`.

A amostragem dos contadores disponibilizados em `/proc/net/dev` exige cuidados especiais. De facto, com débitos da ordem dos Gbps, os contadores podem sofrer rapidamente *overflow*[17]; nessas circunstâncias, se o intervalo entre amostras for suficientemente pequeno, nunca ocorrerá mais de uma situação de *overflow* no intervalo, cenário que ainda permite usar os valores dos contadores para determinar a quantidade de tráfego que circulou; por exemplo, com um interface de 1Gbps a funcionar em *full-duplex* numa máquina de 32 bits, o tempo (teórico) necessário para *overflow* de um contador inteiro será de $2^{32}/[(2 \times 10^9)/8] \approx 17.18s$, pelo que o período amostral deve ser inferior a $2 \times 17.18 = 34.36s$; tendo em conta que o débito real é inferior ao nominal, este período amostral poderá ser um pouco maior[18]; em todo o caso, um período amostral conservador, de 30s, é adoptado por omissão pelo protótipo, para a amostragem dos contadores associados aos interfaces de rede, podendo esse período ser reconfigurado (ver secção 7.12.2).

7.6.5.4 Utilização da Memória Primária

Os sistemas operativos modernos fazem uma gestão da memória RAM que dificulta a sua caracterização em termos simplistas, de "memória total livre" e "memória total ocupada". Evidência disso é que, mesmo em repouso, é reduzida a memória classificada como efectivamente livre, uma vez que o sistema operativo tende a maximizar a utilização da memória como *cache* do sistema de ficheiros e para outros propósitos afins. Neste contexto, qualquer medida de memória total livre ou ocupada será, necessariamente, uma aproximação.

Com base em métricas do Ganglia, uma medida aproximada da utilização da RAM será: 1 - [(`mem_buffers` + `mem_cached` + `mem_free`) / `mem_total`]. Esta abordagem produz

[17] Este fenómeno foi induzido e constatado durante o desenvolvimento do protótipo.

[18] O interesse da avaliação da capacidade dos interfaces de rede é assim reforçado por este cenário.

valores um pouco superiores (na ordem dos 5% a 10%) a outra também de uso frequente: através do comando `ps`, é possível obter a percentagem da RAM usada por cada processo do sistema, sendo a soma dessas percentagens uma medida aproximada da utilização global da RAM. Em ambos os casos, a memória partilhada pelos processos é contabilizada mais do que uma vez, contribuindo para valores de utilização mais elevados que os reais. Por este motivo, usamos a segunda abordagem, prosseguida pelo serviço `domus_nodemon.py`.

7.6.5.5 Utilização do Espaço dos Discos

Na linha da estratégia prosseguida para a caracterização da capacidade das partições, a monitorização da sua utilização (espaço consumido) é também realizada pelo serviço `domus_nodemon.py` por intermédio do comando `df`, uma vez que as métricas `disk_total` e `disk_free` geradas pelo Ganglia não permitem discriminar a utilização por partição.

7.6.5.6 Médias Móveis Exponenciais

Os valores das métricas de *utilizações* resultam de médias móveis exponenciais, como previsto na secção 6.3.2. Existindo várias fórmulas de cálculo possíveis, o protótipo adapta a fórmula utilizada para o cálculo da carga (*load average*) em sistemas UNIX [Gun03]:

$$\mathcal{U}_t(r) = \mathcal{U}_{t-1}(r) \times e^{-\frac{\mathsf{T}}{\triangle\mathsf{T}}} + (1 - e^{-\frac{\mathsf{T}}{\triangle\mathsf{T}}}) \times u_t(r) \tag{7.1}$$

Assim, na fórmula anterior: i) $\mathcal{U}_t(r)$ é o valor actual (instante t) da média móvel exponencial da utilização do recurso r; ii) $\mathcal{U}_{t-1}(r)$ sintetiza a história passada, pois é o anterior (no instante $t-1$) valor da média móvel exponencial; iii) $u_t(r)$ é a amostra actual da utilização do recurso r; iv) T é o período amostral (ou, equivalentemente, $1/\mathsf{T}$ é a frequência amostral); v) $\triangle\mathsf{T}$ é a abrangência temporal da média móvel exponencial, expressa como múltiplo de T, sendo $\triangle\mathsf{T}/\mathsf{T}$ o correspondente número de amostras. Basicamente, quanto maior for o número de amostras consideradas, mais suave é a evolução da média $\mathcal{U}_t(r)$.

No protótipo, assumem-se como valores por omissão $\triangle\mathsf{T} = 300s$ e $\mathsf{T} = 60s$. Para interfaces de rede, tem-se $\triangle\mathsf{T} = 300s$ e $\mathsf{T} = 60s$, ou $\triangle\mathsf{T} = 150s$ e $\mathsf{T} = 30s$, conforme a arquitectura alvo seja de 64 bits ou de 32 bits; estes valores são adequados a interfaces de débito nominal de 1Gbps operando em *full duplex* (rever a secção 7.6.5.3), mas admitem reconfiguração (ver secção 7.12.2), permitindo suportar interfaces de débitos diferentes. Em todo o caso, deve ser mantida a proporção $\mathsf{T}/\triangle\mathsf{T} = 1/5$; esta pode ser interpretada de várias maneiras: 1) o peso atribuído ao passado, $\mathcal{U}_{t-1}(r)$, é de $e^{-\frac{\mathsf{T}}{\triangle\mathsf{T}}} = e^{-\frac{1}{5}} \approx 0,82 \approx 82\%$; 2) só após 5 amostras consecutivas de igual valor é que a média móvel exponencial assume esse valor.

7.7 Atributos da Especificação Suportados

Sendo o protótipo uma implementação parcial da arquitectura Domus, ainda sem suporte à redistribuição dinâmica das DHTs, o protótipo suporta apenas um subconjunto dos atributos previstos no capítulo 5 (para as várias entidades da arquitectura), necessários à criação das DHTs e subsequente exploração; fora desse subconjunto (que representa a quase totalidade dos atributos previstos) ficam assim os Atributos de Gestão de Carga das DHTs

(rever secção 5.8.3.9), com excepção do atributo `attr_dht_lm` (*limiar de armazenamento*), necessário para se realizar a distribuição inicial dos nós virtuais de armazenamento.

Adicionalmente, o suporte a certos atributos é limitado a um conjunto restrito de valores:

- `attr_dht_fh` (Função de Hash): disponível apenas a função genérica do Python;
- `attr_dht_pe` (Política de Evolução): suportada apenas uma política `estática`;
- `attr_dht_pld` (Política de Localização Distribuída): apenas de tipo `global`;
- `attr_dht_gr` (Granularidade do Repositório): apenas granularidade `mínima`;
- <`attr_dht_pa`, `attr_dht_ma`> (Tecnologia de Armazenamento): ver secção 7.5.1.

Foram também realizadas algumas opções que importa referir, designadamente no que toca ao tipo de valores de atributos de identificação das entidades da arquitectura. Os atributos de identificação de um cluster Domus (`attr_cluster_id`) e de uma DHT (`attr_dht_id`) são meras sequências de caracteres. Os atributos de identificação do serviço supervisor (`attr_supervisor_id`) e de serviços regulares (`attr_service_id`) são pares <endereço IP, porto>, face à opção por *sockets* BSD sobre TCP/IP para a passagem de mensagens.

7.8 Biblioteca Domus

O módulo Python `domus_libusr.py` realiza a biblioteca Domus, prevista pela especificação, para que aplicações Domus (aplicações clientes) possam explorar e administrar as abstracções de um cluster Domus. A figura 7.3 representa os principais componentes e funcionalidades da biblioteca, e as suas interacções mais relevantes com serviços Domus. A figura servirá de apoio à descrição da biblioteca, recorrendo às suas referências a negrito.

A biblioteca `domus_libusr.py` expõe as suas funcionalidades através de duas classes:

- a classe `cDomusUsrProxy` (abreviatura de "cluster Domus User-level Proxy");
- a classe `dDomusUsrProxy` (abreviatura de "dht Domus User-level Proxy").

Como a sua designação sugere, as classes suportam a interacção entre aplicações, DHTs e clusters Domus, baseada na intermediação de objectos "representantes" (*proxies*); estes tornam transparente a interacção das aplicações com as várias abstracções Domus (DHTs e serviços específicos, ou a totalidade do cluster Domus), encapsulando todos os detalhes da interacção com objectos pares remotos, residentes no ambiente de execução dos serviços Domus. Em resumo, e na terminologia própria da Teoria de Padrões de Desenho (de Software) [GHJV95], a biblioteca obedece a um "padrão de desenho de tipo *proxy*". Neste contexto, um "representante cDomus" designa um objecto da classe `cDomusUsrProxy` e, por analogia, um "representante dDomus" designa um objecto da classe `dDomusUsrProxy`.

A biblioteca suporta, em simultâneo, múltiplos *contextos de interacção* com DHTs e clusters Domus, como previsto na secção 5.9.1.1. Basicamente, cada objecto das classes `cDomusUsrProxy` e `dDomusUsrProxy` corresponde, respectivamente, a um *contexto* desse tipo.

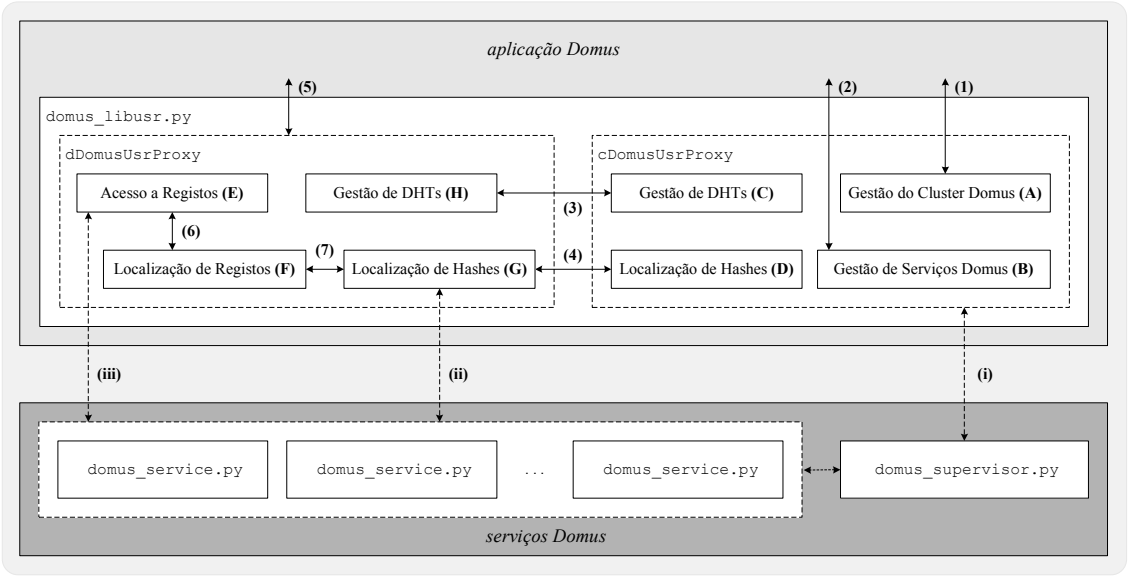


Figura 7.3: Principais Componentes e Interacções da Biblioteca `domus_libusr.py`.

7.8.1 Classe `cDomusUsrProxy`

A classe `cDomusUsrProxy` comporta métodos adequados às interacções *administrativas* previstas na especificação da arquitectura (secção 5.9); na figura 7.3, as interacções dividem-se pela Gestão do cluster Domus (**A**), Gestão de serviços Domus (**B**) e Gestão de DHTs (**C**).

As funcionalidades de Gestão de DHTs da classe `cDomusUsrProxy` não são directamente expostas às aplicações, podendo apenas ser invocadas (**3**) a partir da classe `dDomusUsrProxy`; as funcionalidades de Gestão de serviços Domus e de Gestão do cluster Domus fazem parte dos métodos públicos da classe `cDomusUsrProxy`, podendo ser invocadas directamente (**1**, **2**) pelas aplicações clientes; um representante cDomus tem como interlocutor externo (**i**) o serviço supervisor do cluster Domus de que esse objecto é representante.

A figura 7.3 revela ainda que um representante cDomus pode ser chamado (**4**) a intervir na Localização de Hashes (**D**) . Essa intervenção foi anteriormente prevista, na secção 5.9.1.3, no quadro do recurso ao Método Aleatório, o qual implica o acesso à versão mais actual da Tabela de Distribuição de uma DHT, residente no supervisor do seu cluster Domus.

A tabela 7.3 apresenta uma listagem dos métodos públicos da classe `cDomusUsrProxy`, acompanhada de uma descrição sumária dos mesmos. Para os métodos `cluster_create` / `_destroy`, `cluster_restart` / `_shutdown` e `service_add` / `_remove`, a secção 5.7 da especificação da arquitectura fornece uma descrição de alto nível das operações subjacentes.

7.8.2 Classe `dDomusUsrProxy`

A classe `dDomusUsrProxy` oferece métodos conotados com as interacções de *acesso* a uma DHT (rever secção 5.9): métodos de Acesso a Registos (**E**) e métodos de Localização (**F** e **G**), que servem os primeiros; a classe inclui também métodos de Gestão de DHTs (**H**)

Gestão do cluster Domus

`cluster_ping ()`
- verifica se o supervisor e serviços de um cluster Domus estão activos
`cluster_open ()`
- estabelece uma associação com o supervisor de um cluster Domus activo
`cluster_close ()`
- termina uma associação com o supervisor de um cluster Domus activo (*)
`cluster_create ()`
- cria um novo cluster Domus, no estado activo
`cluster_destroy ()`
- destrói um cluster Domus, activo ou inactivo
`cluster_restart ()`
- reactiva um cluster Domus inactivo
`cluster_shutdown ()`
- desactiva um cluster Domus activo (*)
`cluster_attget (attName)`
- consulta o atributo `attName` de um cluster Domus activo
`cluster_attset (attName, attValue)`
- define o atributo `attName` com o valor `attValue`

Gestão de serviços Domus

`supervisor_ping ()`
- verifica se o supervisor do cluster Domus está activo
`service_ping (srvAddress)`
- verifica se no cluster Domus existe um serviço activo, associado ao interface `srvAddress` (*)
`service_add (srvAddress)`
- acrescenta ao cluster Domus um novo serviço Domus, associado ao interface `srvAddress` (*)
`service_remove (srvAddress)`
- remove do cluster Domus o serviço Domus associado ao interface `srvAddress` (*)
`service_attget (srvAddress, attName)`
- consulta o atributo `attName` de um serviço Domus activo
`service_attset (srvAddress, attValue)`
- define o atributo `attName` com o valor `attValue`

Observações: a) as operações assinaladas a (*) carecem de associação prévia a um supervisor; b) o método construtor da classe, público por definição, é discutido na secção 7.9.1.

Tabela 7.3: Métodos Públicos da Classe `cDomusUsrProxy`.

que viabilizam as interacções *administrativas* com DHTs, também previstas na secção 5.9.

Como já havia sido referido na secção anterior, a Localização de Hashes (**G**) na classe `dDomusUsrProxy`, com base no Método Aleatório, pode implicar a invocação (**4**) de funcionalidades desse tipo (**D**) na classe `cDomusUsrProxy`; alternativamente, o Método Aleatório pode contactar directamente (**ii**) serviços Domus de endereçamento, sem intermediação do procurador cDomus, como acontece com o Método baseado em Cache de Localização (rever secção 5.9.1.3); utilizando-se o Método Directo para Localização de Hashes, então a localização não carece de qualquer transacção de rede (rever também secção 5.9.1.3).

Qualquer Acesso a Registos (**E**) é precedido (**6**) de uma operação de Localização de Registos (**F**) que, por sua vez, invoca (**7**) funcionalidades de Localização de Hashes (**G**). Uma vez determinado (**6** e **7**) o serviço Domus de armazenamento de um registo, as operações de Acesso a Registos (**E**) podem desenrolar-se directamente com esse serviço (**iii**).

A tabela 7.4 apresenta os métodos públicos da classe `dDomusUsrProxy`, com uma descrição sumária. A secção 5.7 da especificação da arquitectura fornece informação adicional sobre as operações subjacentes aos métodos `dht_create/_destroy` e `dht_restart/_shutdown`.

Gestão da DHT
`dht_ping ()` - verifica se uma DHT está activa
`dht_probe ()` - verifica se uma DHT pertence a um cluster Domus
`dht_open ()` - estabelece uma associação com uma DHT activa
`dht_close ()` - termina uma associação com uma DHT activa (*)
`dht_create ()` - cria uma nova DHT, no estado activo
`dht_destroy ()` - destrói uma DHT, activa ou inactiva
`dht_restart ()` - reactiva uma DHT inactiva
`dht_shutdown ()` - desactiva uma DHT activa
`dht_attget (attName)` - consulta o atributo `attName` de uma DHT activa
`dht_attset (attName, attValue)` - define o atributo `attName` com o valor `attValue`
Acesso a Registos (*)
`dht_record_probe (key)` - verifica se existe, na DHT, um registo indexado por `key`
`dht_record_put (key, data)` - insere, na DHT, o registo <`key`,`data`>
`dht_record_get (key)` - recupera, da DHT, a componente `data` do registo indexado por `key`
`dht_record_del (key)` - remove, da DHT, o registo indexado por `key`
Localização de Registos (*)
`dht_record_lookup (key)`: - retorna o *hash*, o serviço de endereçamento e o de armazenamento, do registo indexado por `key`
Localização de Hashes (*)
`dht_hash_lookup (hash)`: - retorna o serviço de endereçamento e o de armazenamento, associado a um `hash`
Observações a) as operações assinaladas a (*) carecem de associação prévia a um supervisor; b) os parâmetros `key` e `data` podem ser quaisquer objectos Python serializáveis; c) o método construtor da classe, público por definição, é discutido na secção 7.9.1.

Tabela 7.4: Métodos Públicos da Classe `dDomusUsrProxy`.

7.9 Desenvolvimento de Aplicações Domus

7.9.1 Metodologia de Desenvolvimento

A metodologia de utilização da biblioteca `domus_libusr.py` é relativamente simples:

1. Começa-se por criar um representante cDomus, recorrendo a código como o seguinte:

```
_CDOMUS_USRPROXY = domus_libusr.cDomusUsrProxy( \
                   cluster_id = "myClusterDomus", \
                   cluster_supervisor_id = ("192.168.0.1",7571) )
```

   (o método construtor suporta também um parâmetro `conf_file`, que localiza um ficheiro de configuração; este permite alterar certas pré-definições internas do módulo `domus_libsys.py`, relevantes na operação de um cluster Domus – ver secção 7.12.2)

2. Depois, é necessário associar o representante cDomus ao supervisor do cluster Domus; a associação resulta da execução dos métodos `cluster_open`, `cluster_create` ou `cluster_restart`; a associação quebra-se em resultado da execução dos métodos `cluster_close`, `cluster_destroy` ou `cluster_shutdown`; quebrada a associação, não é permitido realizar mais operações através do objecto "representante cDomus";

3. Para operar com uma DHT, cria-se um representante dDomus e fornece-se-lhe i) a referência para um representante cDomus já associado e ii) o identificador da DHT:

```
_DDOMUS_USRPROXY = domus_libusr.dDomusUsrProxy( \
                   dht_id = "myDhtDomus", \
                   cluster_proxy = _CDOMUS_USRPROXY)
```

   (o método construtor suporta também um parâmetro `comm_protocol`, que pode assumir os valores `udp` ou `tcp` (sendo este último o valor por omissão), permitindo seleccionar o protocolo de comunicações preferencial[19] na interacção com a DHT)

4. Depois, é preciso associar o representante dDomus à DHT pretendida, o que pode ser feito através dos métodos `dht_open`, `dht_create` ou `dht_restart`; a associação quebra-se em resultado da execução dos métodos `dht_close`, `dht_destroy` ou `dht_shutdown`; quebrada a associação, não é permitido realizar mais operações através do representante dDomus; a quebra da associação é essencial para que se possam libertar recursos reservados através da biblioteca `_domus_libc.so` (ver secção 7.3.1);

5. Todos os métodos públicos retornam um par (`val1,val2`), com o seguinte significado:

   (a) se a operação é bem sucedida, `val1` assume o valor inteiro zero; dependendo do tipo de operação em causa, `val2` pode retornar dados resultantes da operação, como acontece, por exemplo, pela invocação do método `dht_record_get`;

   (b) se a operação não foi bem sucedida, `val1` é um inteiro diferente de zero, correspondente a um código de erro definido na biblioteca `domus_libsys.py`, de forma que `domus_libsys.STRERROR[val1]` retorna uma mensagem de erro elucidativa; nalguns casos, `val2` define um código de erro adicional, *e.g.*, retornado por tentativas falhadas de execução remota via `ssh` ou resultante de operações exógenas, como as efectuadas no seio da biblioteca `_domus_libc.so`; nesse caso, `domus_libsys.STRERROR[val2]` retorna a mensagem de erro associada;

[19] Para certas mensagens, o protocolo é fixo, não sendo possível re-configurá-lo – ver secção 7.10.1.

6. Um representante cDomus é partilhável por vários representantes dDomus. A partilha suporta acesso concorrente, se activado o código necessário (ver secção 7.12.1).

No apêndice E fornecem-se dois exemplos de código comentado, que ilustram a utilização da biblioteca `domus_libusr.py`. A documentação mais actual da API, gerada automaticamente pela ferramenta Doxygen [dox], encontra-se em `http://www.ipb.pt/~rufino/domus`.

7.9.2 Gestão de Atributos

As classes anteriores oferecem métodos para a definição (`*_attset`) e consulta (`*_attget`) dos atributos da especificação suportados pelo protótipo (rever secção 7.7). Como já referimos, cada objecto dessas classes representa um *contexto de interacção* (com um cluster Domus ou com uma DHT); segue-se que os atributos desses objectos correspondem aos atributos de interacção previstos na secção 5.9.1.2 da especificação, sendo conveniente recordar que parte desses atributos são *locais*, existindo apenas no domínio da aplicação cliente, e outra parte são *remotos*, sendo réplicas de atributos mantidos pelo supervisor.

A consulta permite obter o valor de um atributo obedecendo ao seguinte algoritmo genérico: "**se** o atributo for *local* **então** retornar o seu valor imediatamente **senão** pedir o seu valor ao supervisor, actualizar a (eventual[20]) réplica local e retornar o seu valor **fse**".

Só é permitida a definição de atributos *locais* ou *remotos externos*; os últimos, recorde-se (rever secção 5.9.1.2), são o subconjunto dos atributos de uma entidade Domus (DHT, serviço ou cluster) que, sendo mantidos pelo supervisor (*remotos*), admitem a sua definição inicial através de parâmetros fornecidos à biblioteca Domus (*externos*). Todavia, a sua definição é limitada ao período que antecede a criação da entidade; depois, na criação, os atributos eventualmente definidos são exportados para o supervisor. O atributo de identificação de uma DHT ou de um cluster Domus pode ser fornecido como parâmetro ao construtor da classe respectiva, dispensando assim a sua definição via `dht/cluster_attset`.

7.10 Gestão da Comunicação e da Concorrência

7.10.1 Gestão da Comunicação

Como referido anteriormente, a troca de mensagens entre componentes do protótipo recorre ao mecanismo de *sockets* BSD sobre TCP/IP. Mais especificamente, todas as trocas são intermediadas por uma classe da biblioteca `domus_libsys.py`, com funcionalidades de gestão de contextos de comunicação. Basicamente, a classe mantém colecções de *sockets* TCP e UDP, reutilizáveis em trocas sucessivas de mensagens com a mesma entidade remota. Esta simples facilidade de reutilização, conjugada com a utilização da primitiva `select` nos serviços Domus (ver abaixo) permitiu ganhos dramáticos de estabilidade e desempenho da comunicação (até 10 vezes), face a versões iniciais do protótipo, baseadas

[20] De facto, este procedimento também permite consultar atributos que não são replicados localmente ...

na criação sistemática de um *socket* para cada transacção. A classe tem também a capacidade de manter conexões TCP duradouras, o que requer um suporte correspondente nas entidades destino das conexões: essas entidades só podem ser serviços Domus (supervisor e regulares), uma vez que as aplicações Domus nunca são destino de conexões TCP oriundas dos serviços; nos serviços, cada conexão TCP duradoura é associada a um *fio-de-execução*.

Nos serviços, a recepção de mensagens recorre a três *sockets* de *frontend*, para os protocolos TCP, UDP e *multicast IP*. O processamento pode ser feito com base num único fio de execução, ou com múltiplos, sendo essa opção actualmente definida no acto da instalação do protótipo (ver secção 7.12.1). Com um único fio de execução, os três *sockets* são geridos de forma assíncrona, com base na primitiva `select`. Com múltiplos fios de execução, são criados à partida um fio para cada *socket* de *frontend* e depois um fio para processar cada pedido (mecanismo *fork-on-request*). Ao nível das aplicações, já se referiu a possibilidade de estas especificarem o tipo de protocolo a usar nas transacções associadas a uma DHT, através de um atributo `comm_protocol` da classe `dDomusUsrProxy` (rever secção 7.9.1). Além disso, certas transacções usam um protocolo fixo, como é caso das associadas a operações de supervisão (assentes em TCP) e de localização distribuída (assente em UDP).

7.10.2 Gestão da Concorrência

O suporte a múltiplos *fios-de-execução* é opcional. A razão de fundo prende-se com o fraco desempenho dos *fios-de-execução* em Python, incapaz de verdadeiro paralelismo: o interpretador Python mantém um trinco global que previne dois fios de execução de acederem ao mesmo objecto em simultâneo[21] [MH05]. Neste contexto, a execução dos serviços Domus com base num único *fio-de-execução*, associado ao mecanismo `select`, provou ser a abordagem com melhor desempenho, em testes realizados com o protótipo.

Nos serviços Domus, a possibilidade de execução de múltiplos *fios-de-execução* é suportada por código *thread-safe*. Essa propriedade é também assegurada pelas classes da biblioteca `domus_libusr.py`, permitindo a sua utilização por aplicações Python com mais do que um *fio-de-execução*. As necessárias garantias de consistência são, nos serviços e na biblioteca, fornecidas por um mecanismo similar, baseado numa hierarquia de trincos *um-escritor-múltiplos-leitores*. Estes trincos surgem como atributos das várias classes do código, tendo havido o cuidado de garantir que a sua utilização não gera situações de *deadlock*. Nos serviços, a profundidade máxima de execução concorrente permitida pelos trincos atinge-se à entrada de um repositório, onde é permitido o acesso por um único *fio-de-execução*.

7.11 Utilitários de Administração e Acesso

O protótipo inclui um conjunto de aplicações desenvolvidas sobre a biblioteca Domus, para a gestão dos serviços de monitorização e das várias entidades de um cluster Domus: a aplicação `manage_domus_nodemons.py` e as aplicações `manage_domus_cluster.py`

[21] Aparentemente, esta propriedade dispensa a utilização de trincos de alto nível, o que se deve encarar com reservas, pois depende de detalhes de implementação do interpretador, que pode evoluir noutro sentido.

`manage_domus_supervisor.py`, `manage_domus_service.py` e `manage_domus_dht.py`. A aplicação `manage_domus_dht.py` pode também ser usada para realizar operações de *acesso* (inserções, remoções, consultas, verificações de existência e localizações) a registos <`key`,`data`> de uma DHT, desde que `key` e `data` sejam exprimíveis na linha de comando[22].

Em conjunto, estas aplicações fornecem uma via alternativa de acesso às funcionalidades da biblioteca Domus para programas não codificados em Python, desde que estes tenham a possibilidade de invocar essas aplicações; o desempenho desta abordagem deverá ser, naturalmente, inferior à execução directa ou com base em mecanismos como o SWIG; o grande benefício é que virtualmente qualquer aplicação pode operar um cluster Domus.

```
[domus@omega domus#source]$ ./manage_domus_nodemons.py
Usage: ./manage_domus_nodemons.py
-h | { [-N node_name] -o operation [options] [-c conf_file] [-d] }
operation:        start | kill | getpid | rmpid | log | ls
options:          -o kill -k kill_signal
kill_signal:      TERM (=> exit) | USR1 (=> shutdown) | USR2 (=> auto-destroy)
```

```
[domus@omega domus#source]$ ./manage_domus_cluster
Usage: ./manage_domus_cluster.py
 -h | { -C cluster_id -S supervisor_interface -o operation [options] [-c conf_file] [-d] }
operation:         ping| create| destroy| shutdown| restart| kill| rm| getpid| log| ls
options:           { -o create [-e] } | { -o kill -k kill_signal }
kill_signal:       TERM (=> exit) | USR1 (=> shutdown) | USR2 (=> auto-destroy)
```

```
[domus@omega domus#source]$ ./manage_domus_supervisor.py
Usage: ./manage_domus_supervisor.py
-h | { -C cluster_id -S supervisor_interface -o operation [options] [-c conf_file] [-d] }
operation:        ping | kill | rm | getpid | log | ls
options:          -o kill -k kill_signal
kill_signal:      TERM (=> exit) | USR1 (=> shutdown) | USR2 (=> auto-destroy)
```

```
[domus@omega domus#source]$ ./manage_domus_service.py
Usage: ./manage_domus_service.py
-h | { -C cluster_id -S supervisor_interface -o operation [options] [-c conf_file] [-d]
       -I service_interface }
operation:        ping | add | remove | kill | rm | getpid | log | ls
options:          -o kill ... [-k kill_signal]
kill_signal:      TERM (=> exit) | USR1 (=> shutdown) | USR2 (=> auto-destroy)
```

```
[domus@omega domus#source]$ ./manage_domus_dht.py
Usage: ./manage_domus_dht.py
-h | { -C cluster_id -S supervisor_interface -o operation [options] [-c conf_file] [-d]
       -D dht_id }
operation:        ping | probe | create | destroy | restart | shutdown | ls |
                  record_probe | record_put | record_get | record_del |  record_lookup |
                  hash_lookup
options:          { -o record_probe -k key } | { -o record_put -k key -d data } |
                  { -o record_get -k key } | { -o record_del -k key } |
                  { -o record_lookup -k key } | { -o hash_lookup -H hash (0x...) }
```

Figura 7.4: Interface das Aplicações de Gestão e Acesso.

[22] À partida, qualquer objecto que seja serializável numa sequência de caracteres compatível com a *shell*.

A figura 7.4 revela o interface das aplicações de administração e acesso. Assim, para além das operações suportadas pela biblioteca Domus para as várias entidades de um cluster Domus, as aplicações em causa suportam um conjunto de outras operações convenientes:

- `start`: arranque de um ou mais serviços (só para serviços `domus_nodemon.py`);
- `kill`: envia sinais a um ou mais serviços, com diferentes consequências (`exit` implica a terminação imediata; `shutdown` implica a terminação com salvaguarda prévia de todo o estado volátil em suporte persistente; `auto-destroy` implica a terminação com remoção prévia de todo o estado eventualmente já em suporte persistente);
- `getpid`: obtém o *process ID* de um ou mais serviços supostamente em execução;
- `rmpid`: remove o(s) ficheiro(s) usado(s) para guardar o *process ID* do(s) serviço(s);
- `log`: lança terminais X que apresentam, em tempo real, o *output* do(s) serviço(s);
- `ls`: lista recursivamente a(s) directoria(s) com o estado persistente do(s) serviço(s);
- `rm`: remoção forçada da(s) directoria(s) com o estado persistente do(s) serviço(s).

Adicionalmente, existem também uma série de parâmetros cujo significado importa reter:

- `-c conf_file`: localização do ficheiro de configuração referido na secção 7.12.2;
- `-d`: activação das facilidades de depuração dos serviços (efectiva apenas se o código de depuração não foi extirpado dos serviços, na instalação – rever secção 7.12.1);
- `-e`: impõe a criação de um cluster Domus vazio (só com o supervisor); senão, será criado com os serviços listados no ficheiro definido pelo atributo `CLUSTER_INTERFACES` da configuração, com valor `~/domus#interfaces` por omissão (ver secção 7.12.2).

O protótipo inclui ainda uma consola de administração (`domus_console.py`) que concentra a funcionalidade dos restantes utilitários num único ambiente de utilização mais amigável.

7.12 Instalação e Configuração

7.12.1 Processo de Instalação

A instalação do protótipo é feita através de uma aplicação dedicada (`domus_install.py`) que suporta um conjunto de operações relevantes: a) pre-processamento do código, através do qual é possível a1) desactivar código de depuração e/ou código de controle de concorrência, assim como a2) activar o suporte adequado à arquitectura alvo (32 bits ou 64 bits); b) instalação remota em múltiplos nós do *cluster*. A selecção da arquitectura alvo apropriada é imprescindível no sentido de assegurar o correcto funcionamento das funcionalidades associadas à a2.1) produção de *hashes* e sua manipulação através de máscaras, e a2.2) produção de métricas de utilização de interfaces de rede (rever secção 7.6.5.3).

```
#############################################################################################
DEBUG           0                                   # 1 == True ; 0 == False; default: 0
HOME            /home/domus/domus#home              # where (to install) / (to find) Domus;
                                                    ## default: /home/domus/domus#home
#############################################################################################
GANGLIA_HOST            omega                       # Ganglia host where to PUBLISH/QUERY;
                                                    ## default: localhost
GANGLIA_CHANNEL         228.46.60.78                # default: 239.2.11.71
GANGLIA_PORT            8649                        # default: 8649
#############################################################################################
NODEMON_ROOT                /home/domus/domus#home  # root dir for the "nodemon" daemon;
                                                    ## default: /home/domus
NODEMON_LIFETIME            0                       # default: 0 (infinite)
NODEMON_SAMPLE_PERIOD_NET   30                      # default: 30s for 32bit; 60s for 64bit
NODEMON_SAMPLE_WIDTH_NET    150                     # default: 150s for 32bit;300s for 64bit
#############################################################################################
CLUSTER_INTERFACES    /home/domus/domus#interfaces  # default: ~/domus#interfaces
CLUSTER_LIFETIME                      0             # default: 0 (infinite)
CLUSTER_LIFETIME_END_ACTION           exit          # exit, shutdown, destroy
#############################################################################################
SUPERVISOR_ROOT     /home/domus/domus#home          # root dir for the "supervisor" daemon;
                                                    ## default: /home/domus
SUPERVISOR_PORT     7571                            # default: 7571
#############################################################################################
SERVICE_ROOT        /state/partition1               # root dir for the "service" daemons;
                                                    ## default: /home/domus
SERVICE_PORT        8379                            # default: 7571
```

Figura 7.5: Exemplo de Ficheiro de Configuração de um Cluster Domus.

7.12.2 Ficheiro de Configuração

Através de um ficheiro de configuração, aplicações e serviços Domus podem modificar certas constantes originalmente definidas na biblioteca `domus_libsys.py`, e que têm influência sobre aspectos diversos do funcionamento de um cluster Domus. Para as aplicações de administração apresentadas antes, vimos que pode ser usado um parâmetro `-c conf_file` para veicular a localização desse ficheiro; por seu turno, os serviços Domus (incluindo os de monitorização) aceitam um parâmetro semelhante. No contexto do desenvolvimento de aplicações Domus, referiu-se também a existência de um parâmetro `conf_file` para o método construtor de um procurador cDomus (rever secção 7.9.1). A figura 7.5 mostra o conteúdo de um ficheiro deste tipo. Os atributos dividem-se em vários grupos e, embora o seu papel seja relativamente evidente, merecem uma breve descrição:

- `*_ROOT`: directoria sob a qual os serviços gerem o seu estado persistente; no caso dos serviços Domus regulares, a sua escolha (através do atributo `SERVICE_ROOT`) deve ser feita atendendo à *partição* desejada para os repositórios das DHTs; a definição de uma directoria remota é possível, mas desaconselhável (rever secção 7.6.4.4);
- `*_LIFETIME`: duração dos serviços de monitorização e do cluster Domus (não devendo os serviços de monitorização ter uma duração inferior ao cluster Domus); no término, as acções possíveis são iguais às provocadas com sinais, pelos utilitários (ver acima);

- `NODEMON_SAMPLE_PERIOD_NET`: período amostral da utilização dos interfaces de rede (rever secção 7.6.5.3), não sendo indiferente na definição a arquitectura da máquina;
- `NODEMON_SAMPLE_WIDTH_NET`: abrangência da utilização dos interfaces de rede;
- `CLUSTER_INTERFACES`: listagem (de interfaces) de serviços a usar num cluster Domus.

7.13 Avaliação do Protótipo

Este capítulo conclui pela apresentação e discussão dos resultados de um conjunto de testes de avaliação do protótipo [RPAE07b, RPAE07a]. Os testes foram conduzidos num *cluster* ROCKS 4.0.0, de 16 nós homogéneos (um deles de *frontend*) com a seguinte configuração de hardware: placa-mãe de chipset i865, CPU Pentium 4 de 32bits a 3 GHz, 1GB de RAM DDR400, disco SATA-I de 80 GB, interface de rede Ethernet de 1Gbps (integrado na placa mãe). A interligação dos nós baseou-se num comutador Ethernet *full duplex* de 1Gbps.

Em todos os testes i) o serviço Domus supervisor executou sempre no nó de *frontend*, ii) os pedidos foram atendidos por um só *fio-de-execução*, através de `select`, iii) os acessos às DHTs recorreram ao protocolo UDP e, salvo indicação em contrário, foi sempre usada a estratégia de localização <mD,mC,mA>, baseada na primazia do *método directo* (1-HOP).

7.13.1 Avaliação das Tecnologias de Armazenamento

O primeiro conjunto de testes teve como objectivo avaliar, a dois níveis, as tecnologias de armazenamento suportadas pelo protótipo (rever secção 7.5.1): i) *desempenho do acesso* (ou seja, de operações básicas de acesso a dicionários) e ii) *sobrecarga de armazenamento* (pela comparação do espaço de armazenamento nominal com o efectivamente consumido).

Para além do nó de *frontend*, esta avaliação incluiu a utilização de apenas dois nós (sempre os mesmos) do *cluster*: um nó para albergar uma aplicação Domus, geradora de acessos a DHTs baseadas em diferentes tecnologias de armazenamento, e outro nó para albergar o único serviço Domus dos clusters Domus criados. Esta configuração simples, ponto-a-ponto, é suficiente para avaliar o mérito relativo das várias tecnologias de armazenamento.

Cada tecnologia de armazenamento foi avaliada com base no seguinte procedimento: 1) criação de um cluster Domus com um único serviço regular; 2) criação de uma DHT (inicialmente vazia, portanto, e suportada pelo único serviço regular); 3) acessos à DHT.

O acesso a cada DHT a partir da aplicação cliente prosseguiu a seguinte sequência de operações: 3a) inserções (teste *put1*); 3b) consultas (teste *get*); 3c) sobreposições (teste *put2*); 3d) remoções (teste *del*). Imediatamente a seguir ao teste *get*, as DHTs foram desactivadas (via `dht_shutdown`) e foi registado o consumo de espaço em disco resultante da desactivação, para serem logo de seguida reactivadas (via `dht_restart`), antes do teste *put2*. Cada um dos quatros testes de acesso foi realizado 8M = 8×2^{20} vezes, utilizando-se os inteiros do intervalo {0,...,8M-1} para valorar os registos <*chave*,*dados*> das DHTs (para os testes *put1* e *put2*, *dados*=*chave*). Com registos de tão pequena dimensão (8 bytes para nós de 32 bits), a sobrecarga de armazenamento emerge de forma mais notória.

7.13.1.1 Desempenho do Acesso por Tipo de Operação

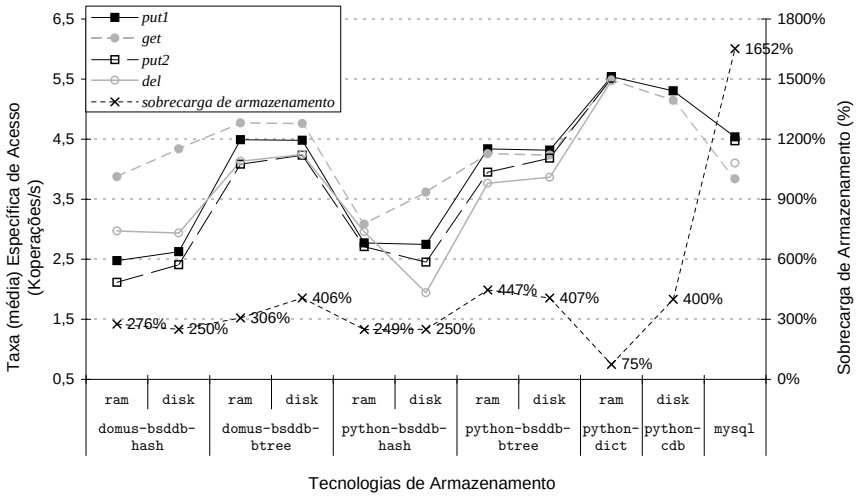


Figura 7.6: Taxas Específicas de Acesso e Sobrecarga de Armazenamento.

Para cada operação $o \in \{put1, get, put2, del\}$ e para cada tecnologia de armazenamento t suportada pelo protótipo, é possível definir uma *taxa (média) específica de acesso*, $\overline{\lambda}(o,t)$. A taxa calcula-se dividindo o número total de operações (8M, para todas as operações), pelo tempo total necessário à sua realização (que será variável em função de o e de t). A figura 7.6 representa $\overline{\lambda}(o,t)$ em Koperações/s, na escala dada pelo eixo vertical esquerdo.

Como é observável, a tecnologia <`python-dict`,`ram`> oferece o melhor desempenho, seguida de <`python-cdb`,`disk`>; esta, apesar de assente em suporte secundário, consegue um desempenho muito próximo da primeira, assente em RAM; no entanto, convém recordar (rever secção 7.5.1) que a operação da plataforma `python-cdb` é limitada ao modo *Write-once-Read-many* (o que, aliás, viabilizou apenas a execução dos testes *put1* e *get*).

Neste contexto, o melhor compromisso em termos de "variedade de meios de armazenamento suportados" e "variedade de operações de acesso a dicionários suportadas" é assegurado pelas plataformas assentes em dicionários BerkeleyDB (`domus-bsddb-*` e `python-bsddb-*`). O seu desempenho é, como esperado, inferior ao das plataformas `python-dict` e `python-cdb`, e a diferença de desempenho depende do substrato usado pelos dicionários BerkeleyDB: com árvores B+ (plataformas `*-bsddb-btree`), o desempenho é substancialmente melhor do que com Tabelas de Hash Dinâmicas (plataformas `*-bsddb-hash`). Em todo o caso, verifica-se uma vantagem consistente das plataformas `domus-bsddb-*` fornecidas pela biblioteca `_domus_libc.so`, face às plataformas `python-bsddb-*`, acedidas "directamente" a partir do Python, o que comprova o mérito da nossa implementação.

Curiosamente, os resultados obtidos com a plataforma `domus-bsddb-hash` sobre o meio `ram` são um pouco piores do que sobre o meio `disk`; o mesmo se passou para o teste *get* com a plataforma `python-bsddb-hash` e, de forma pontual e bastante menos vincada, para plataformas `*-bsddb-btree`. O facto é que a plataforma BerkeleyDB continua a recorrer intensivamente a armazenamento em Disco (criando repositórios temporários em `/var/tmp`, por omissão) mesmo quando instruída para usar RAM como meio de armaze-

namento e, como efeito secundário, isso parece contribuir para degradar o desempenho em certas situações. Estes resultados contra-intuitivos determinam a necessidade de repetir estes testes em máquinas de hardware diferente, uma vez que a sua repetição noutras máquinas (iguais) do mesmo *cluster* usado na avaliação, produziu resultados similares.

Numa perspectiva global, é possível retirar algumas conclusões gerais sobre o posicionamento relativo do desempenho das operações de acesso às DHTs, independentemente do tipo de tecnologia usada: 1) as operações mais rápidas são as de consulta (teste *get*); 2) as inserções de registos pela primeira vez (teste *put1*) são mais rápidas que a sua sobreposição (teste *put2*), o que demonstra que o desempenho de operações de inserção não deve ser julgado apenas com base nas "primeiras inserções". A remoção (teste *del*) exibe uma posicionamento mais irregular, o que demonstra a necessidade de um critério mais rigoroso de classificação das diferentes tecnologias de armazenamento. Esse critério deve entrar em linha de conta com o tipo de operações mais frequentes, entre outros – ver secção 7.13.1.3.

Comparação com o MySQL Para compreender a significância do desempenho das tecnologias de armazenamento suportadas pelo protótipo, realizou-se uma avaliação do mesmo tipo com a plataforma de base de dados MySQL [mys], sobejamente conhecida.

Para o efeito, instalou-se o MySQL no nó usado para albergar o serviço Domus regular e criou-se uma base de dados, com uma só tabela, de apenas duas colunas de tipo inteiro, apropriadas para suportar os campos dos registos <*chave*,*dados*> a inserir pelo cliente.

A partir de um cliente em Python, e recorrendo ao módulo `MySQLdb`, realizou-se a sequência de testes $\langle put1, get, put2, del\rangle$. A codificação do cliente em Python, a sua execução no mesmo nó usado para alojar os clientes Domus, e o cuidado em evitar a agregação de operações[23], torna os resultados desta avaliação comparáveis com os das "nossas" tecnologias.

A figura 7.6 revela que os resultados obtidos com o MySQL situam-se na mesma faixa de valores dos obtidos com as tecnologias `*-bsddb-btree`, o que reforça o mérito das nossas tecnologias de armazenamento, tanto mais que dispomos de alternativas que, embora menos flexíveis, conseguem desempenho um pouco melhor (`python-dict` e `python-cdb`).

7.13.1.2 Sobrecarga de Armazenamento por Tipo de Tecnologia

Cada tecnologia t incorre numa *sobrecarga de armazenamento*, que denotamos por $\phi(t)$. Sendo $\mathcal{S}_i$=64Mbytes o espaço nominal/ideal necessário para os 8M registos[24], e sendo $\mathcal{S}_r(t)$ o espaço efectivo/real que acaba por ser necessário com a tecnologia t (em face das necessidades próprias das estruturas de dados de t que suportam os registos), então $\phi(t) = 1 - \mathcal{S}_r(t)/\mathcal{S}_i$. A figura 7.6 representa $\phi(t)$ em percentagem, na escala do eixo direito. Por exemplo, $\mathcal{S}_r$(<`python-dict`,`ram`>)=112 Mbytes, donde ϕ(< `python-dict`,`ram`>) ≈75%.

Como referido anteriormente, a medição de $\mathcal{S}_r(t)$ é feita em Disco, após desactivação das DHTs. De facto, todas as tecnologias de armazenamento usadas permitem a desacti-

[23] Encapsulamento de vários pedidos numa só mensagem, o que não é suportado pela biblioteca Domus.

[24] Com nós de 32 bits, cada número inteiro consome 4 bytes, logo um registo <*chave*,*dados*> consome 8 bytes, donde o "espaço nominal" correspondente a 8M registos é de 8 × 8Mbytes = 64Mbytes.

vação das DHTs para Disco. Por outro lado, a contabilização do consumo de RAM só faria sentido para as tecnologias baseadas nesse meio de armazenamento, para além de necessitar de instrumentação adequada, nem sempre disponibilizada pelas plataformas de armazenamento ou pelo ambiente de execução (*e.g.*, o Python esconde esses detalhes).

Como se pode observar na figura 7.6, os dicionários `python-dict` são também, em termos de sobrecarga espacial de armazenamento, a alternativa mais atractiva sendo que, no extremo oposto, situa-se o MySQL, com uma sobrecarga bastante elevada (1650%); esta justifica-se, em boa parte, pelo suporte transaccional (que obriga ao registo das operações efectuadas), e pela manutenção de índices auxiliares para acelerar o acesso aos dados.

As plataformas `*-bsddb-btree`, com melhor desempenho que as plataformas `*-bsddb-hash`, são agora penalizadas na sobrecarga de armazenamento, substancialmente superior. Num patamar similar de maior sobrecarga, situa-se também a plataforma `python-cdb`.

7.13.1.3 Métrica de Selecção de Tecnologias de Armazenamento

Os resultados da avaliação do *desempenho do acesso* e da *sobrecarga de armazenamento* podem ser usados para alimentar uma métrica $\mathcal{R}(t)$ que, levando em conta esses dois factores, permite classificar tecnologias de armazenamento t de acordo com requisitos aplicacionais específicos. Por exemplo, para certas aplicações, a maximização do desempenho pode ser o único factor relevante, ao passo que, para outras, pode ser a minimização da sobrecarga de armazenamento, se bem que, a maioria, estará provavelmente interessada num bom compromisso entre os dois factores. A fórmula 7.2 fornece a definição da métrica referida:

$$\mathcal{R}(t) = \omega_\lambda \times \mathcal{R}_\lambda(t) + \omega_\phi \times \mathcal{R}_\phi(t), \text{ com } \mathcal{R}(t) \in [0,1] \tag{7.2}$$

Na fórmula anterior, ω_λ e ω_ϕ são pesos complementares ($\omega_\lambda + \omega_\phi = 1$) das classificações $\mathcal{R}_\lambda(t)$ e $\mathcal{R}_\phi(t)$; estas classificações denotam o posicionamento da tecnologia t face às outras, em termos de *desempenho do acesso* e *sobrecarga de armazenamento*, respectivamente. A tecnologia de armazenamento t mais apropriada a uma aplicação será a que maximiza $\mathcal{R}(t)$, ou seja, a que "maximiza o desempenho" e "minimiza a sobrecarga de armazenamento".

A classificação $\mathcal{R}_\phi(t)$ é dada simplesmente por 1 - $\phi(t)/\max[\phi(*)]$, em que $\max[\phi(*)]$ é o valor máximo de $\phi(t)$ registado, para todas as tecnologias de armazenamento avaliadas[25].

A classificação $\mathcal{R}_\lambda(t)$ resulta de uma avaliação mais complexa. De facto, a análise da figura 7.6 revela que, para diferentes tecnologias de armazenamento, as operações básicas de acesso a dicionários posicionam-se de forma diferente em termos de *desempenho do acesso*; essa diferença de posicionamento tem a ver não só com a) a ordenação relativa, como também com b) a maior ou menor variância do desempenho das várias operações; por exemplo: a) a remoção (teste *del*) tanto pode ser a segunda operação mais rápida (plataformas `domus-bsddb-hash` e `mysql`), como pode ser a mais lenta (restantes plataformas); b) nalgumas plataformas, as diferentes operações têm desempenho similar (*e.g.*,

[25]Note-se que se $\phi(t) \to 0$, então $\mathcal{R}_\phi(t) \to 1$; reciprocamente, se $\phi(t) \to \max[\phi(*)]$, então $\mathcal{R}_\phi(t) \to 0$; desta forma, a tecnologia que minimizar a sobrecarga de armazenamento terá a maior classificação $\mathcal{R}_\phi(t)$.

platafomas `python-dict` e `python-cdb`), ao passo que noutras plataformas as diferenças são maiores, podendo mesmo ser acentuadas (como nas plataformas `*-bsddb-hash`).

As observações anteriores sugerem a conveniência em definir padrões de acesso às DHTs, assentes na probabilidade das operações. Essas probabilidades servem para pesar o desempenho das diferentes operações de acesso a dicionários, produzindo-se assim uma métrica combinada de desempenho, $\mathcal{R}_\lambda(t)$, de que as aplicações se podem servir para seleccionar a tecnologia de armazenamento que melhor se adapta ao seu padrão de acesso às DHTs:

$$\begin{aligned}\mathcal{R}_\lambda(t) \;=\; & p(put1) \times \frac{\overline{\lambda}(put1,t)}{\max[\overline{\lambda}(put1,*)]} + p(put2) \times \frac{\overline{\lambda}(put2,t)}{\max[\overline{\lambda}(put2,*)]} \\ & + p(del) \times \frac{\overline{\lambda}(del,t)}{\max[\overline{\lambda}(del,*)]} \;+\; p(get) \times \frac{\overline{\lambda}(get,t)}{\max[\overline{\lambda}(get,*)]}\end{aligned} \tag{7.3}$$

Na fórmula 7.3, i) $p(o)$ denota a probabilidade da operação $o \in \{put1, get, put2, del\}$ (com $\sum p(o) = 1$) e ii) $\max[\overline{\lambda}(o,*)]$ é o valor máximo de $\overline{\lambda}(o,t)$ para todas as tecnologias t. Na prática, as probabilidades $p(o)$ actuam como pesos, tal como os factores ω da fórmula 7.2.

Exercitamos agora as fórmulas anteriores com dois dos padrões de acesso suportados pelas tecnologias do protótipo, o que implica definir as probabilidades correspondentes:

- padrão *Write-once-Read-many* (WoRm), suportado por todas as tecnologias; neste caso $p(put2) = p(del) = 0$ e $p(get) >> p(put1)$, donde $p(put1) \approx 0$ e $p(get) \approx 1$;
- padrão *Write-many-Read-many-Delete-many* (WmRmDm), suportado por todas as plataformas, excepto a `python-cdb` (que suporta apenas o padrão WoRm); neste caso, assumimos que consultas, remoções e inserções são equiprováveis, ou seja, $p(get) = p(del) = p(put) = 1/3$ com $p(put) = p(put1) + p(put2)$; no caso das inserções, assumimos um modelo simples em que $p(put1) = p(put2) = (1/3)/2 = 1/6$.

A figura 7.7 apresenta a classificação $\mathcal{R}(t)$ das tecnologias do protótipo, para padrões de acesso WoRm e WmRmDm. Para cada padrão apresentam-se duas classificações: uma para $\omega_\lambda = 1.0$ e $\omega_\phi = 0.0$ (apenas o desempenho importa); outra para $\omega_\lambda = 0.5$ e $\omega_\phi = 0.5$ (desempenho e sobrecarga de armazenamento têm igual importância). A ordenação resultante de uma classificação com $\omega_\lambda = 0.0$ e $\omega_\phi = 1.0$ (apenas a sobrecarga de armazenamento importa) é dedutível da curva da *sobrecarga de armazenamento* da figura 7.6 (note-se que, nesse contexto, é irrelevante o padrão de acesso das aplicações às DHTs).

Globalmente, os resultados das classificações podem ser sistematizados da seguinte forma:

1. para *repositórios não persistentes*, a tecnologia <`python-dict`,`ram`> é a melhor opção, uma vez que é a que maximiza o *desempenho do acesso* (independentemente do *padrão de acesso*) e minimiza a *sobrecarga de armazenamento* (rever figura 7.6); a segunda melhor opção seria, neste contexo, a tecnologia <`domus-bsddb-btree`,`ram`> (a alguma distância da tecnologia mais comparável, <`python-bsddb-btree`,`disk`>);

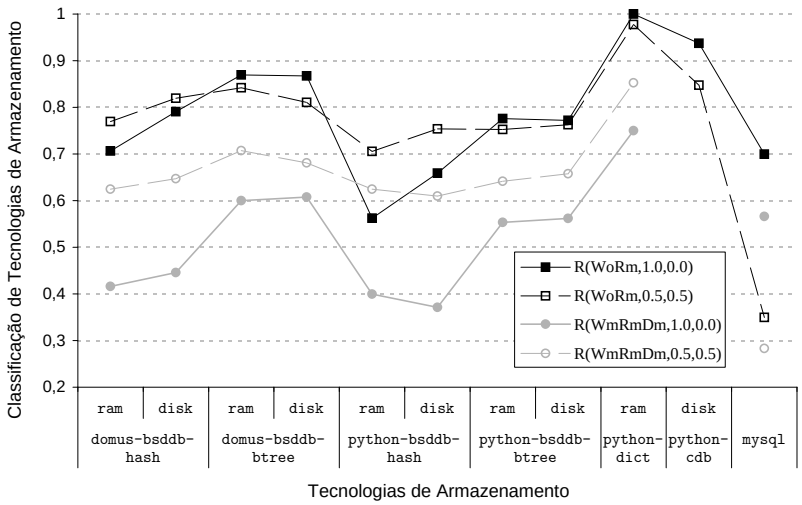


Figura 7.7: Classificação de Tecnologias de Armazenamento.

2. para *repositórios persistentes*, se o padrão de acesso for WoRM, a tecnologia <`python-cdb`,`disk`> é claramente a melhor opção, e se o padrão de acesso for WmDmRm, a melhor opção é sempre a tecnologia <`domus-bsddb-btree`,`disk`> (neste caso a distância reduzida da tecnologia comparável, <`python-bsddb-btree`,`disk`>)[26].

7.13.1.4 Desempenho Combinado do Acesso (Capacidades de Acesso)

As probabilidades de cada operação, definidas para cada padrão de acesso, permitem calcular uma *taxa (média) combinada de acesso*, $\overline{\lambda}(t)$, característica de cada tecnologia t:

$$\overline{\lambda}(t) = \sum \left[\; p(o) \times \overline{\lambda}(o,t) \; \right], \text{ com } o \in \{put1, get, put2, del\} \tag{7.4}$$

A taxa $\overline{\lambda}(t)$ representa uma aproximação a $\mathcal{C}^{\mathtt{a}}(t,n)$ ("capacidade de acesso a repositórios de tecnologia t, pelo nó n"), um dos atributos de nós computacionais previstos pela arquitectura Domus (rever secção 6.3.1), e suportado pelo protótipo (rever secção 7.6.4.1).

A figura 7.8 apresenta os valores da taxa $\overline{\lambda}(t)$, para os padrões de acesso estudados. A figura ainda permite ordenar o desempenho das tecnologias de forma coerente com $\mathcal{R}$(WoRm,1,0) e $\mathcal{R}$(WmRmDm,1,0) mas, mais importante, permite representar uma única medida sintética de desempenho, mais conveniente que as quatro medidas específicas de cada operação, exibidas na figura 7.6. Pela observação dos resultados expressos na figura 7.8, comprova-se também, mais uma vez, a supremacia das plataformas `python-dict` e `python-cdb` para um uso mais especializado, e de `domus-bsddb-btree` para uma utilização mais genérica.

[26]Esta observação ajuda a colocar na perspectiva correcta os reais ganhos do acesso a repositórios BerkleyDB via `_domus_libc.so`, em vez do acesso via Python: os ganhos podem ser pouco significativos.

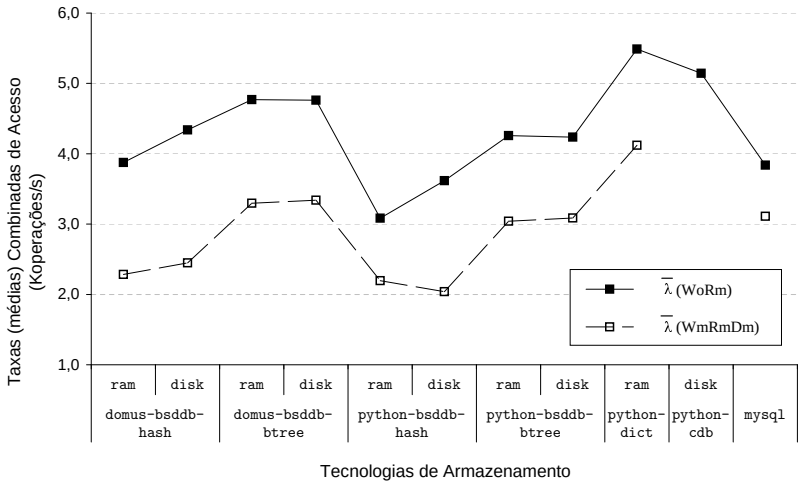


Figura 7.8: Taxas Combinadas de Acesso.

7.13.1.5 Evolução do Desempenho do Acesso

As medidas $\overline{\lambda}(o,t)$ e $\overline{\lambda}(t)$ definidas anteriormente são médias globais, obtidas levando em consideração o tempo total gasto para realizar as operações de acesso sobre os 8M registos. Essas medidas globais pouco ou nada revelam sobre a forma como o desempenho das operações evolui à medida que se enche/consulta/esvazia completamente um repositório, sendo esse conhecimento também relevante na selecção de uma tecnologia de armazenamento. Por exemplo: 1) a mesma operação com duas tecnologias diferentes pode apresentar a mesma taxa global $\overline{\lambda}(o,t)$, mas num caso pode haver oscilações consideráveis no ritmo de execução da operação, e noutro esse ritmo pode ser relativamente constante; 2) certas operações podem executar rapidamente enquanto o número de registos do repositório não atingir um certo limite, após o que o desempenho se pode degradar, pelo que é conveniente conhecer o padrão de execução da operação em função do volume esperado do repositório.

Os exemplos anteriores demonstram então a conveniência do estudo da evolução da taxa de realização das operações, por forma a identificar tendências para além do curto prazo. Assim, durante a realização dos testes às tecnologias de armazenamento, registaram-se "taxas instantâneas", $\lambda(o,t)$, de 1024 em 1024 acessos. Nos gráficos das figuras 7.9 à 7.13 representam-se os resultados desse estudo[27]. Para facilitar a análise dos resultados, as tecnologias de armazenamento foram divididas em vários grupos, discutidos de seguida.

A figura 7.9, concentra as tecnologias de armazenamento que apresentam as taxas de operação mais estáveis/sustentadas; note-se que este grupo inclui as tecnologias do protótipo que oferecem melhor desempenho (<`python-dict`,`ram`> e <`python-cdb`,`disk`>).

As figuras 7.10 e 7.11 mostram a evolução das taxas de acesso para plataformas `*-bsddb-hash`, sobre meios `ram` e `disk`, respectivamente. Em ambas pode-se observar claramente a evolução em serra da taxa de operações *put1*; trata-se de uma consequência da utilização de Hashing Dinâmico pela plataforma BerkeleyDB, que determina a duplicação periódica

[27] A escala não é uniforme, para facilitar a distinção entre as curvas; estas foram também suavizadas.

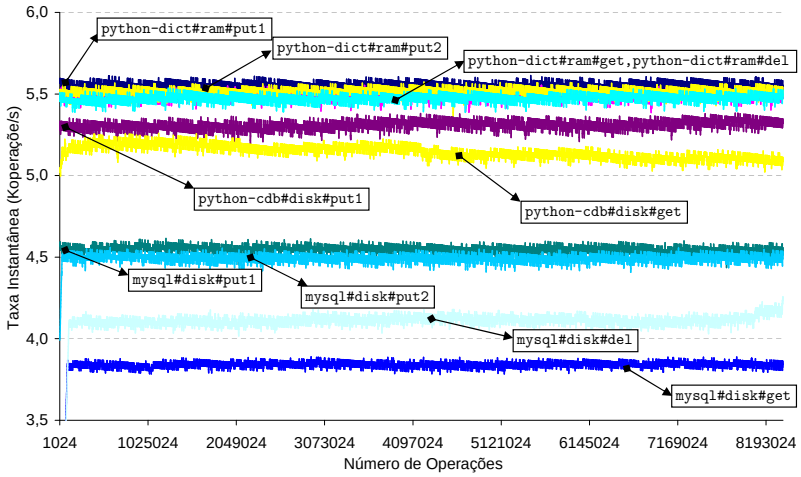


Figura 7.9: Taxas Instantâneas de Acesso (`python-dict`, `python-cdb` e `mysql`).

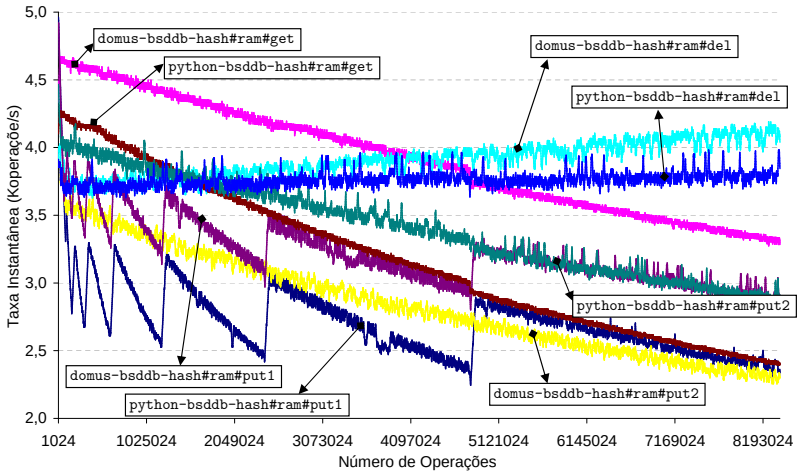


Figura 7.10: Taxas Instantâneas de Acesso (tecnologias <`*-bsddb-hash`,`ram`>).

do número de contentores (*buckets*) da Tabela de Hash; à medida que o conjunto actual de contentores esgota a sua capacidade, o desempenho das inserções decresce, até que aumenta de novo com a duplicação do número de contentores (se bem que a tendência global seja decrescente); além disso, como o espalhamento dos registos pelos contentores não é absolutamente uniforme, a duplicação ocorre desfasada dos momentos ideais, em que o número de registos inseridos perfaz uma potência de 2. Em relação às restantes operações, pode-se observar, por exemplo, que as taxas das re-inserções (operações *put2*), apesar de apresentarem a mesma tendência global decrescente das inserções (operações *put1*), não exibem a evolução destas em serra (essa evolução é causada pelo esforço de criação de contentores, que não se regista nas re-inserções). As operações de consulta (*get*) seguem uma tendência semelhante à das re-inserções, mas com níveis de desempenho superiores. A evolução crescente das taxas de remoção sugere que as operações de remoção

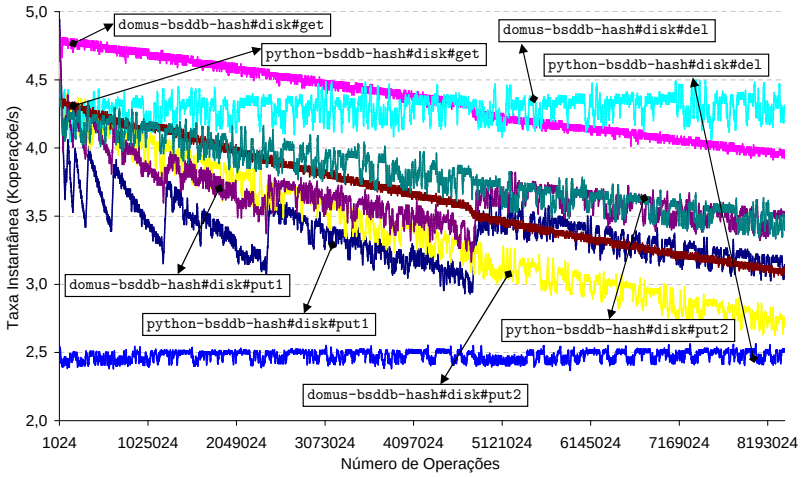


Figura 7.11: Taxas Instantâneas de Acesso (tecnologias <`*-bsddb-hash`,`disk`>).

(*del*) beneficiam da diminuição progressiva do número global de registos dos repositórios[28].

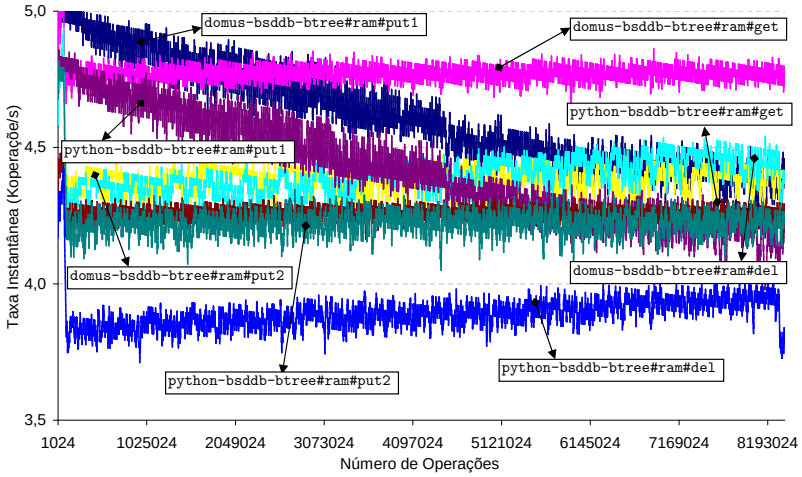


Figura 7.12: Taxas Instantâneas de Acesso (tecnologias <`*-bsddb-btree`,`ram`>).

As figuras 7.12 e 7.13 revelam que a utilização de Árvores B+ pela plataforma BerkeleyDB assegura taxas de operação sustentadas e, em média, superiores às permitidas pela utilização de Tabelas de Hash Dinâmicas (as taxas de inserções ainda decaem, mas de forma mais suave). A vantagem das plataformas `domus-bsddb-btree` sobre as `python-bsddb-btree` é também mais evidente (ao passo que as diferenças entre as plataformas `domus-bsddb-hash` e `python-bsddb-hash` são menos pronunciadas) justificando-se assim a eleição anterior de `domus-bsddb-btree` como a terceira melhor plataforma do protótipo.

Em suma, as plataformas melhor classificadas pela métrica $\mathcal{R}(t)$ da secção 7.13.1.3 (`python-dict`, `python-cdb` e `domus-bsddb-btree`), apresentam todas taxas de operação susten-

[28] É de realçar que, no BerkeleyDB, as Tabelas de Hash são do tipo *grow-only*, ou seja, uma vez reservado o espaço para os contentores, a remoção de registos não se traduz na libertação efectiva de contentores.

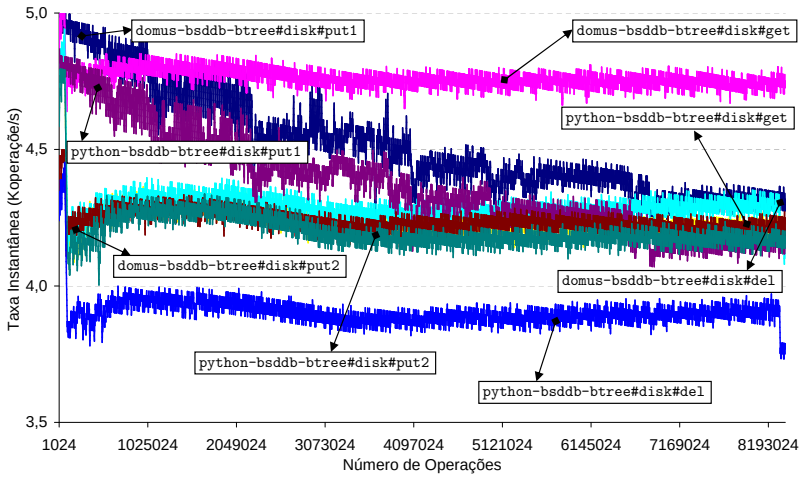


Figura 7.13: Taxas Instantâneas de Acesso (tecnologias <`*-bsddb-btree`,`disk`>)

tadas, com excepção da operação de inserção (*put1*) da plataforma `domus-bsddb-btree`; neste caso, a tendência decrescente das taxas instantâneas deixa adivinhar taxas médias inferiores às medidas nas nossas experiências, para um número de registos superior a 8M.

7.13.2 Avaliação da Escalabilidade sob Saturação

O segundo conjunto de testes procurou avaliar a escalabilidade do protótipo. A avaliação incidiu apenas sobre as tecnologias <`python-dict`,`ram`> e <`domus-bsddb-btree`,`disk`>. Para além de suportarem diferentes níveis de persistência, as duas tecnologias em causa foram as avaliadas por serem, de acordo com o estudo da secção 7.13.1, as que apresentam melhor desempenho, de entre as que suportam todas as operações de acesso a dicionários[29].

Além disso, a avaliação restringiu-se à operação de inserção (*put1*), dado que as operações de consulta (*get*), re-inserções (*put2*) e remoções (*del*) apresentam um desempenho similar, permitindo extender os resultados obtidos, a essas operações, isoladas ou combinadas.

A avaliação da escalabilidade foi feita em condições de *saturação*: durante cada teste, cada cliente Domus gerou pedidos de inserção (baseados no método `dht_record_put` da biblioteca Domus) o mais rapidamente possível, sem qualquer processamento adicional entre pedidos sucessivos, ou seja, os clientes tentaram saturar os serviços Domus. Este tipo de testes dificilmente retrata uma situação real, em que os clientes Domus intercalam os acessos a DHTs com processamento adicional. Todavia, i) permitem conhecer os limites superiores do desempenho do protótipo, ii) o cenário de teste é facilmente reproduzível e iii) são independentes dos padrões de acesso específicos de diferentes tipos de clientes.

Foram testadas duas *classes* distintas de *configurações clientes–serviços*. Neste contexto, uma *configuração clientes–serviços* (ou, simplesmente, *configuração*) refere-se a uma combinação particular de i) número e posicionamento (definição do nó hospedeiro) de clientes Domus, com ii) número e posicionamento de serviços Domus. As classes testadas foram:

[29] Este requisito exclui automaticamente a tecnologia <`python-cdb`,`disk`> do estudo da escalabilidade.

- classe *N_Xcli_M_1srv* (classe de *configurações disjuntas*): N "nós clientes", executando X clientes Domus cada, e M "nós servidores" independentes, executando 1 serviço Domus cada; no total, a classe *N_Xcli_M_1srv* exige $N + M$ nós diferentes, para a execução de $x = N \times X$ clientes Domus e $y = M \times 1 = M$ serviços Domus;

- classe *N_Xcli1srv* (classe de *configurações partilhadas*): N nós clientes, cada um partilhado por X clientes Domus e 1 serviço Domus; no total, exige N nós diferentes, para a execução de $x = N \times X$ clientes Domus e $y = N \times 1 = 1$ serviços Domus.

Com a avaliação destas classes procurou-se averiguar o efeito de duas situações: uma em que clientes e serviços ocupam conjuntos disjuntos de nós (classe *N_Xcli_M_1srv*) e outra em que partilham os nós (classe *N_Xcli1srv*). Mais uma vez, trata-se de situações extremas, mas cuja avaliação poderá fornecer indicações sobre a utilização preferencial a dar ao *cluster* físico quando nele se pretendem operar e explorar clusters Domus. Em particular, o estudo das classes deverá permitir determinar qual a configuração que maximiza o desempenho para um certo número de nós envolvidos, o que é particularmente importante em *clusters* pequenos ou quando se atribuem subconjuntos limitados de nós a cada utilizador.

Para cada classe testaram-se várias configurações de x clientes e y serviços, com base no seguinte procedimento: 1) criação de um cluster Domus com y serviços regulares; 2) criação de uma DHT, com funções de endereçamento e armazenamento equalitariamente suportadas pelos y serviços; 3) arranque simultâneo das inserções na DHT, pelos x clientes.

Cada cliente inseriu $2 \times 2^{20} = 2$M registos $<chave,dados>$ na DHT, valorados por números inteiros sequenciais, diferentes entre todos os clientes (para evitar re-inserções). O número de registos a inserir foi definido de forma a permitir inserções contínuas durante 210s, tempo de duração de cada teste. Este tempo é suficiente para se observar a estabilização, nos valores de pico, das médias móveis exponenciais das *utilizações* dos recursos, geradas pelos serviços Domus de monitorização (`domus_nodemon.py`), quando configurados com um período amostral $\mathsf{T} = 30s$ e uma amplitude amostral $\triangle\mathsf{T} = 150s$ (rever secção 7.6.5.6)[30].

Para cada configuração, da soma das *taxas específicas de inserção* dos x clientes activos resultou uma *taxa agregada de inserção*, $\overline{\lambda}(put1)$. A análise das taxas $\overline{\lambda}(put1)$ obtidas com diferentes configurações da mesma classe permitiu estudar a escalabilidade dessa classe.

7.13.2.1 Avaliação com a Tecnologia <python-dict,ram>

Configurações Disjuntas

Os pontos da figura 7.14 são *taxas agregadas de inserção* obtidas por configurações da classe *N_Xcli_M_1srv*. As taxas são representadas em função do total de nós, $N + M$.

Cada curva da figura representa uma sub-classe de *N_Xcli_M_1srv*, em que apenas N é variável. Para obter uma curva, fixou-se M e, para cada valor possível de N, fez-se variar X até encontrar o valor de X que maximiza a *taxa agregada de inserção*. Como o *cluster*

[30] Para abreviar a duração do teste, definiu-se $\mathsf{T} = 30s$ e $\triangle\mathsf{T} = 150s$ para todos os recursos.

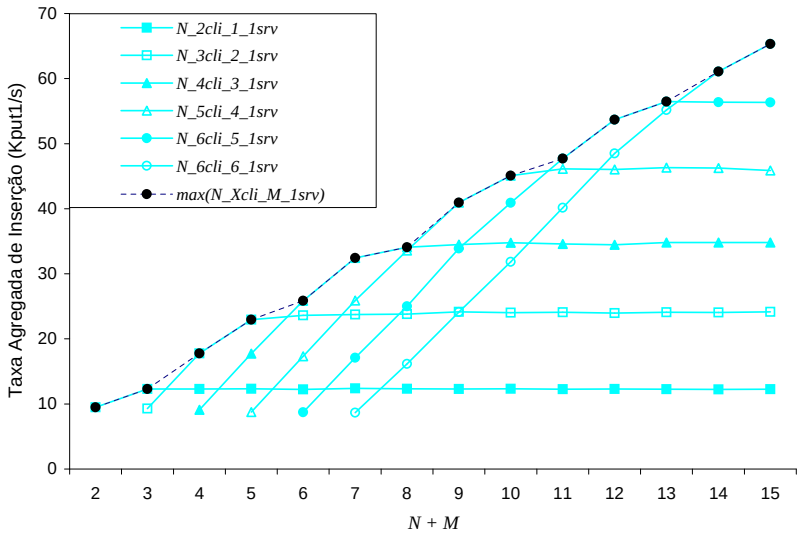


Figura 7.14: $\overline{\lambda}(put1)$ com a Tecnologia <`python-dict`,`ram`> e Configurações Disjuntas.

tinha apenas 15 nós, então N e M estão sujeitos ao invariante $N + M \leq 15$, juntamente com $N \geq 1$ e $M \geq 1$; dessa forma, a fixação de M estabelece automaticamente a banda de variação $1 \leq N \leq 15 - M$, e as curvas só apresentam pontos que cumpram os invariantes.

O comportamento das curvas é típico de um cenário de saturação: as taxas crescem de forma linear com o aumento do número de nós clientes (N), até que o seu valor estabiliza.

Na figura é visível um conjunto de pontos, $max(N\_Xcli\_M\_1srv)$, representados a negrito; cada um desses pontos corresponde à máxima taxa $\overline{\lambda}(put1)$ obtida com um certo número $N+M$ de nós; por exemplo, a sub-classe *N_2cli_M_1srv* fornece pontos para $N+M \in \{2,3\}$ e a sub-classe *N_3cli_M_2srv* fornece pontos para $N+M \in \{4,5\}$. Como se pode observar, $max(N\_Xcli\_M\_1srv)$ cresce linearmente com $N + M$. Por outras palavras: a escolha adequada dos parâmetros N, X e M proporciona escalabilidade linear da taxa $\overline{\lambda}(put1)$.

Constrangimentos à Escalabilidade A análise das *utilizações* dos recursos dos nós clientes e servidores, registadas durante os testes, permite determinar quais os recursos limitadores da escalabilidade das taxas $\overline{\lambda}(put1)$. Considerando que os recursos críticos para uma DHT baseada em RAM são CPU, Rede e RAM (rever secção 6.4.2), e que a RAM é suficiente[31], analisamos de seguida a utilização das CPUs e dos Interfaces de Rede.

A figura 7.15 divide-se em seis secções, correspondentes às seis sub-classes de configurações da figura 7.14; em cada secção existem duas curvas, uma para nós clientes, e outra para nós servidores; cada ponto de cada curva representa a média dos picos de utilização de CPU

[31] A sub-classe que exige mais RAM por servidor é *N_2cli_1_1srv*, com apenas 1 servidor; quando $N = 14$, haverá 14*2=28 clientes activos, inserindo 2M registos cada, num total de 56M registos; cada registo consome 8 bytes, logo o espaço nominal correspondente a 56M registos será $\mathcal{S}_i$=448Mbytes; tendo em conta a sobrecarga de armazenamento da tecnologia <`python-dict`,`ram`> de 75% (rever secção 7.13.1.2), o espaço real necessário será $\mathcal{S}_r$=784Mbytes, aquém da capacidade instalada de 1Gbyte (na prática, $\mathcal{S}_r$ deverá ser inferior, pois a sobrecarga foi calculada após serialização para disco, a qual tende a ser onerosa).

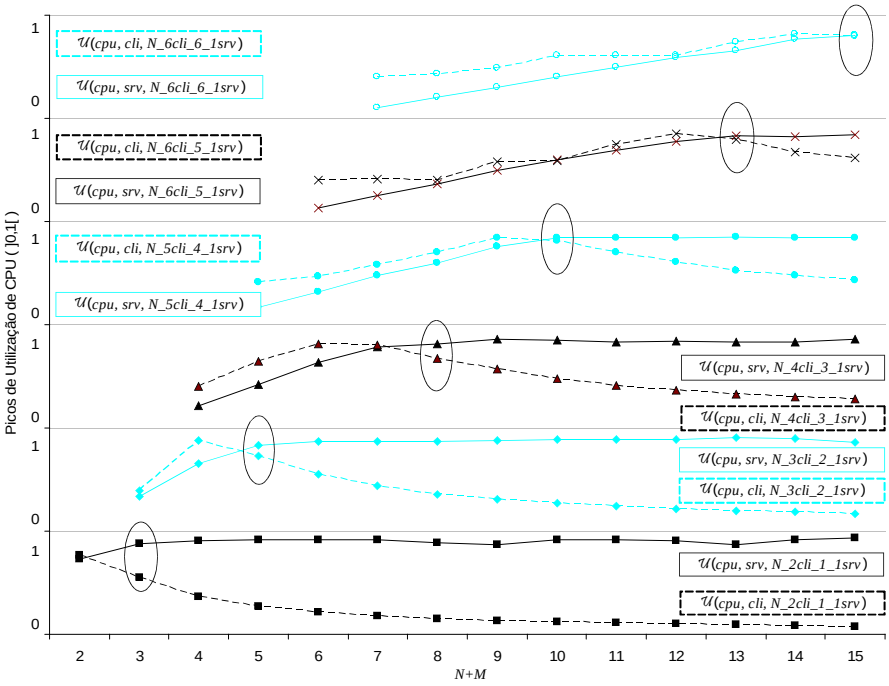


Figura 7.15: Picos de $\mathcal{U}(cpu)$, para os testes da figura 7.14.

registados pelos nós (clientes ou servidores) que participaram no teste de uma configuração; por exemplo, a curva $\mathcal{U}(cpu, srv, N\_2cli\_1\_1srv)$ é composta pelos valores médios dos picos de utilização dos nós servidores de configurações da sub-classe *N_2cli_1_1srv*, e a curva $\mathcal{U}(cpu, cli, N\_2cli\_1\_1srv)$ desempenha o mesmo papel para os nós clientes.

Na figura 7.15 observa-se o mesmo padrão de evolução das curvas de utilização, nas seis secções, à medida que o número de nós clientes (N) aumenta: i) a utilização de CPU dos nós clientes aumenta até certo ponto e depois decresce; ii) a utilização de CPU dos nós servidores (em número fixo, M) também aumenta até certo ponto e depois estabiliza; a estabilização coincide com a descida da utilização de CPU dos clientes para valores inferiores à dos servidores, como assinalado pelas elipses; o número total de nós envolvidos ($N + M$) quando as utilizações de clientes e servidores se cruzam é também o número total de nós envolvidos quando a *taxa de inserção agregada* estabiliza (ver figura 7.14); a estabilização desta taxa está portanto relacionada com a saturação da CPU dos servidores.

A figura 7.16 tem uma organização semelhante à da figura 7.15, com uma secção por cada sub-classe de configurações, mas com uma só curva por secção; cada ponto dessa curva representa a soma das utilizações dos interfaces de rede de todos os nós (clientes e servidores) envolvidos numa configuração; a soma justifica-se porque a Rede é um recurso partilhado por todos os nós; por exemplo, a curva $\mathcal{U}(net, N\_2cli\_1\_1srv)$ é composta pela utilização agregada da Rede, feita por todos os nós de configurações da sub-classe *N_2cli_1_1srv*.

A análise da figura 7.16 permite concluir que a Rede está longe de ser um factor limitante da escalabilidade, uma vez que, no pior caso (quando há N=9 e M=6 nós trocando mensagens), a utilização não ultrapassa os 35%. A disponibilidade de CPU nos nós servidores

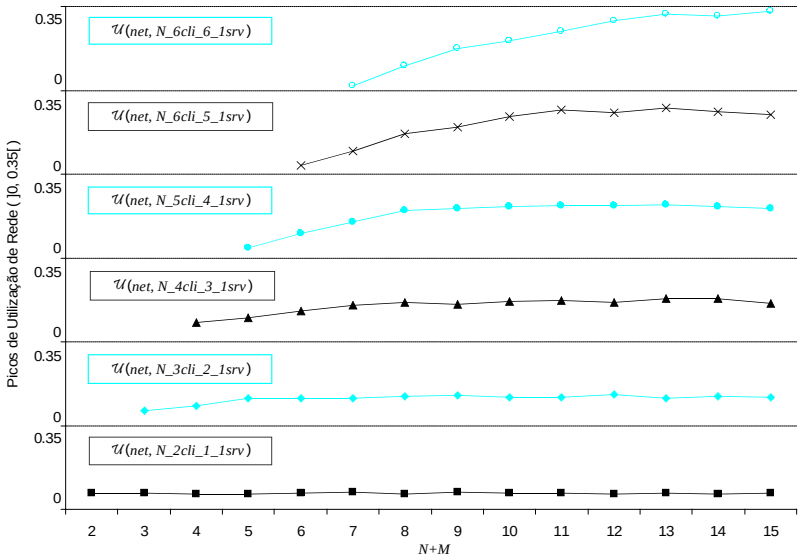


Figura 7.16: Picos de $\mathcal{U}(net)$, para os testes da figura 7.14.

é assim o factor crítico para assegurar a escalabilidade linear da *taxa agregada de inserção*.

Impacto das Estratégias de Localização Salvo outra indicação, todos os resultados da avaliação do protótipo foram obtidos com a *estratégia de localização* <mD,mC,mA> (a utilizada por omissão), maximizadora do desempenho do acesso em DHTs estáticas, pela prioridade dada à exploração do Método Directo (mD). Todavia, o protótipo também suporta as outras estratégias previstas pela arquitectura (rever secção 5.9.1.3): <mC,mA> e <mA> (assentes na utilização do Método baseado em Cache de Localização (mC) e/ou do Método Aleatório (mA)). Assim, para se ter uma ideia do impacto de diferentes estratégias de localização no desempenho do acesso às DHTs, repetiu-se a avaliação das *taxas agregadas de inserção* (que deram origem à figura 7.14), com as estratégias <mC,mA> e <mA>. A estratégia <mC,mA> foi avaliada duas vezes, com duas dimensões diferentes da *cache de localização*, correspondentes a 25% e a 50% do total de *hashes* da DHT[32].

A figura 7.17 apresenta os resultados da re-avaliação da *taxa agregada de inserção*, juntamente com os obtidos anteriormente. Em ambos os casos, são apresentados apenas os valores de pico obtidos para um certo número $N + M$ de nós utilizados por uma configuração disjunta. Assim: i) a curva *max*(*N_Xcli_M_1srv*, <mD,mC,mA>) corresponde à curva *max*(*N_Xcli_M_1srv*) da figura 7.14; ii) as curvas *max*(*N_Xcli_M_1srv*, <mC,mA>, 50%) e *max*(*N_Xcli_M_1srv*, <mC,mA>, 25%) correspondem aos resultados da avaliação da estratégia <mC,mA> com *caches de localização* de diferente dimensão; iii) a curva *max*(*N_Xcli_M_1srv*, <mA>) fornece os resultados da avaliação da estratégia <mA>.

Como esperado, a figura 7.17 comprova i) a supremacia da estratégia <mD,mC,mA>, ii) o pior desempenho por parte da estratégia <mA> e iii) um desempenho intermédio por parte das estratégias <mC,mA>. As estratégias agora avaliadas asseguram também a

[32]Recorde-se que existe a possibilidade de parametrizar a dimensão da *cache* (rever secção 5.9.1.2).

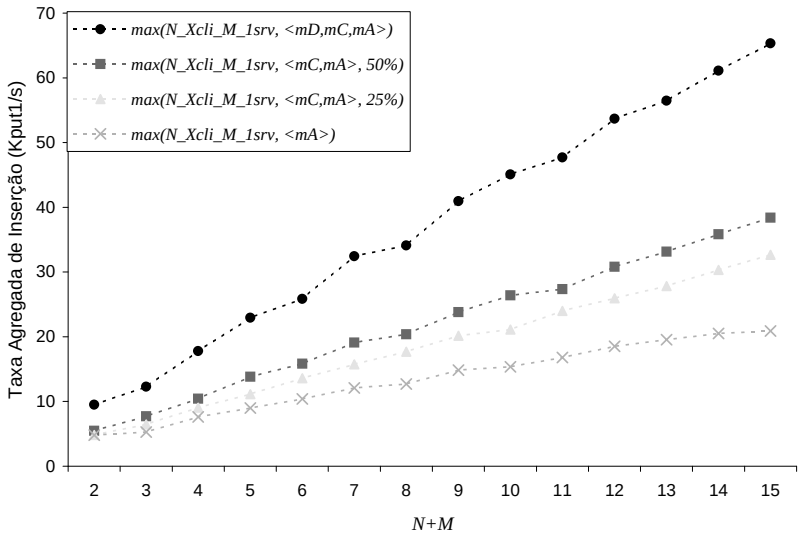


Figura 7.17: Efeito da Estratégia de Localização no cenário da figura 7.14.

escalabilidade linear das *taxas agregadas de inserção*. O efeito de uma *cache de localização* maior é também evidente, pela comparação dos resultados das estratégias <mC,mA>, embora os resultados obtidos com *caches* de diferente dimensão não sejam proporcionais ao diferencial de dimensão das *caches*: utilizar uma *cache* a 25% ainda garante desempenhos da ordem dos 60% (em média) dos obtidos com uma *cache* a 50%, provavelmente em resultado da utilização de algoritmos de *encaminhamento acelerado*. Por outro lado, numa estratégia <mC,mA>, utilizar uma *cache* a 50% não se traduz num desempenho de 50% face à estratégia <mD>, mas sim de 35% (em média), uma vez que: i) pesquisar a *cache* é inerentemente mais lento que resolver uma localização através de um cálculo determinístico (essência do Método Directo); ii) numa situação de *cache-miss*, a penalização temporal resultante do envio do pedido de inserção ao serviço Domus errado é agravada pela necessidade de recorrer a uma localização distribuída (essência do Método Aleatório).

Na prática, os resultados desta avaliação reforçam a importância da escolha da estratégia de localização adequada: para DHTs estáticas, ou dinâmicas de curta-duração (logo com poucas oportunidades de re-distribuição), uma estratégia assente na primazia do Método Directo é a que oferece o melhor desempenho. Adicionalmente, a avaliação permite vislumbrar o tipo de penalização que deriva do recurso aos outros métodos de localização.

Configurações Partilhadas

A figura 7.18 apresenta os resultados do estudo da escalabilidade da taxa $\overline{\lambda}(put1)$ com a tecnologia <`python-dict`,`ram`> e configurações da classe *N_Xcli1srv*, em que cada um de N nós é partilhado por X clientes e 1 serviço Domus. Os resultados obtiveram-se fazendo variar X para cada $N \in \{1, ..., 15\}$, até encontrar o valor de X que maximiza a *taxa agregada de inserção*. Neste caso, esse valor é $X = 2$ para todos os valores de N, pelo que a curva formada pelos valores de pico, *max(N_Xcli1srv)*, coincide com a curva *N_2cli1srv*.

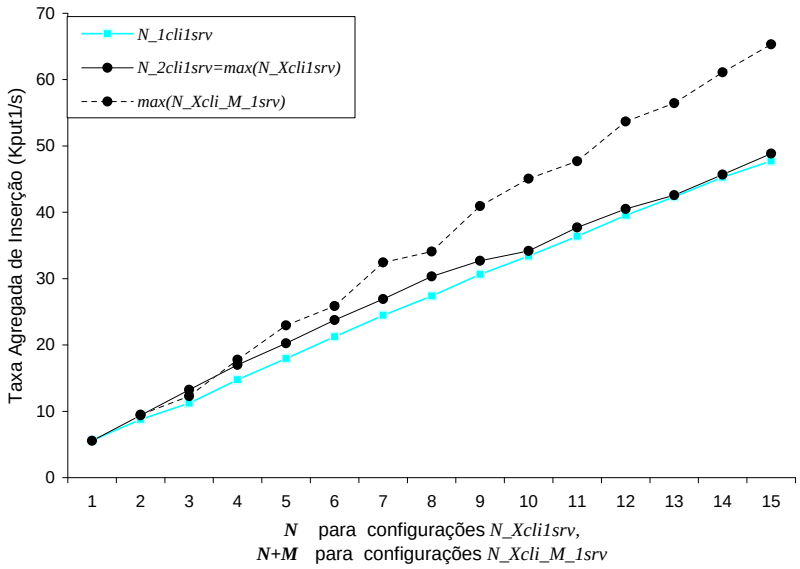


Figura 7.18: $\overline{\lambda}(put1)$ com a Tecnologia <`python-dict`,`ram`> e Configurações Partilhadas.

Como se observa, a distância entre as curvas *N_1cli1srv* e *N_2cli1srv* é reduzida e tende a atenuar-se com o aumento de N; o potencial para o aumento do desempenho através do aumento do número de clientes por nó é portanto bastante limitado, ao contrário da classe de configurações disjuntas, onde a execução de 6 clientes por nó ainda trouxe ganhos de desempenho. Adicionalmente, a escalabilidade da *taxa agregada de inserção* é de tipo linear, em função de N. A conclusão mais importante, todavia, decorre da comparação com a curva *max(N_Xcli_M_srv)*, representativa das configurações disjuntas, e que demonstra, claramente, que estas representam a melhor opção com a tecnologia <`python-dict`,`ram`>, uma vez que, para o mesmo número de nós envolvidos, o desempenho é em geral superior.

7.13.2.2 Avaliação com a Tecnologia <`domus-bsddb-btree`,`disk`>

Configurações Disjuntas e Partilhadas

As figuras 7.19 e 7.20 sintetizam os resultados da avaliação da tecnologia <`domus-bsddb-btree`,`disk`>, com configurações das classes *N_Xcli_M_1srv* (configurações disjuntas) e *N_Xcli1srv* (configurações partilhadas), respectivamente. O procedimento experimental de avaliação foi o mesmo descrito anteriormente, para a tecnologia <`python-dict`,`ram`>, e algumas das observações então registadas são também válidas no contexto presente.

Assim, comparativamente com a figura 7.14, também na figura 7.19 se observa que: i) as taxas crescem linearmente com $N + M$, até estabilizarem num patamar de saturação; ii) a curva formada pelas taxas de pico, *max(N_Xcli_M_1srv)*, escala de forma relativamente linear. Todavia, as taxas de pico são inferiores às obtidas com a tecnologia <`python-dict`,`ram`>, denotadas na figura 7.19 por *max(N_Xcli_M_1srv,<python-dict,ram>)*. Além disso, a maximização das taxas requer agora mais clientes Domus (X) para o mesmo

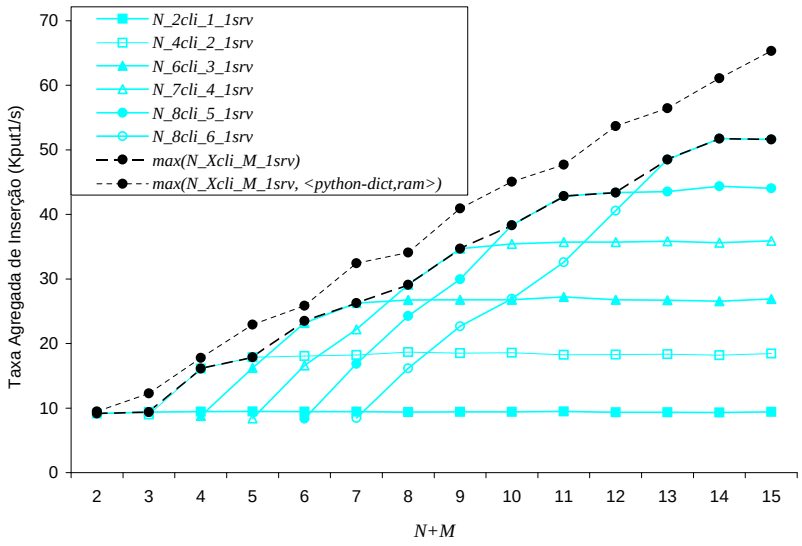


Figura 7.19: $\overline{\lambda}(put1)$ com a Tecnologia <`domus-bsddb-btree`,`disk`> e Confs. Disjuntas.

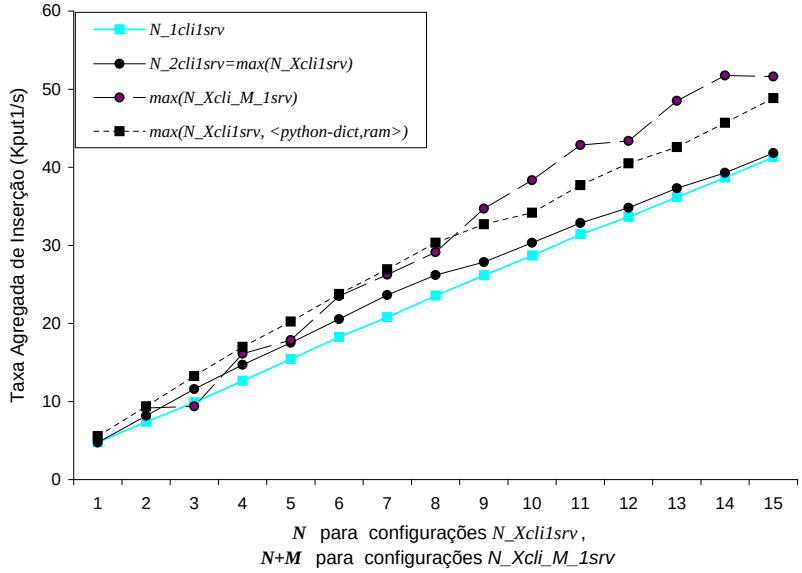


Figura 7.20: $\overline{\lambda}(put1)$ com a Tecnologia <`domus-bsddb-btree`,`disk`> e Confs. Partilhadas.

número de nós servidores (M); por exemplo, para a sub-classe *N_Xcli_3_1srv*, <`python-dict`,`ram`> necessita de $X = 4$ clientes por nó cliente para maximizar o desempenho, mas <`domus-bsddb-btree`,`disk`> necessita de $X = 6$ clientes; de facto, sendo a inserção em repositórios <`domus-bsddb-btree`,`disk`> mais lenta (rever secção 7.13.1.1), cada nó servidor irá demorar mais tempo para processar cada pedido, forçando os clientes Domus a maiores períodos de ociosidade[33], o que permite acomodar mais clientes Domus por nó.

Analogamente, a figura 7.20 exibe semelhanças com a figura 7.18: a sub-classe *N_2cli1srv*

[33]Pois o próximo pedido de inserção só é enviado depois de processado o anterior.

fornece as melhores taxas, exibe escalabilidade relativamente linear e o seu desempenho é inferior ao máximo obtido com configurações disjuntas (conforme se conclui comparando a curva *N_2cli1srv* com a curva *max(N_Xcli_M_1srv)* oriunda da figura 7.19). Adicionalmente, a comparação entre *N_2cli1srv* e *max(N_Xcli1srv,<python-dict,ram>)* comprova que, para configurações partilhadas, a tecnologia <`python-dict`,`ram`> é a mais atractiva.

Embora não se apresentem aqui valores, as *utilizações* de CPU e Interfaces de Rede dos nós clientes e servidores seguiram padrões de evolução semelhantes aos registados no estudo da tecnologia <`python-dict`,`ram`>. Notavelmente, a actividade E/S registada (em acessos ao Disco) revelou-se modesta, provavelmente pelo efeito da *cache* do sistema de ficheiros.

Em suma, para tecnologias <`python-dict`,`ram`> e <`domus-bsddb-btree`,`disk`>, as configurações que oferecem o melhor desempenho em condições de saturação são as *disjuntas*.

Comparação com o Bamboo

A avaliação da escalabilidade sob saturação termina comparando os resultados obtidos pela tecnologia <`domus-bsddb-btree`,`disk`>, com os exibidos pelo Bamboo [bam], uma plataforma de DHTs para ambientes P2P, relativamente bem conhecida. Esta plataforma é baseada na abordagem Pastry [RD01], mas inclui optimizações para elevadas taxas de junção/abandono às/das DHTs[34] em ambientes de largura de banda limitada. O Bamboo é codificado em Java e recorre à plataforma BerkeleyDB (com suporte transaccional activo) para o armazenamento dos registos em disco; estes são replicados (até quatro vezes[35]) e possuem um tempo de vida limitado, findo o qual são removidos. No Bamboo, os nós da DHT actualizam constantemente as rotas do grafo de localização distribuída, como forma de se adaptarem às mudanças assíncronas na composição do conjunto dos nós da DHT.

A avaliação do Bamboo realizou-se no mesmo *cluster* usado para avaliar o protótipo Domus, com clientes Bamboo expressamente codificados em Java, de papel similar aos clientes Domus em Python. Durante os testes, os clientes Bamboo inseriram pares de inteiros <*chave*, *dados*> numa DHT Bamboo, durante 210s, usando como *gateways* (pontos de entrada na DHT) serviços Bamboo aleatórios, de forma a balancear a carga de localização. A funcionalidade de selecção de vizinhos por proximidade (*proximity neighbour selection*) foi desligada, dado o conjunto de serviços Bamboo permanecer estático em cada teste.

A avaliação limitou-se à classe de configurações disjuntas, *N_Xcli_M_1srv*, tendo-se obtido os seguintes resultados: 1) a *taxa agregada de inserção*, $\overline{\lambda}(put1)$, revelou-se independente do número de clientes activos ($N \times X$); ii) à medida que o número de serviços (M) aumentou de 1 para 4, $\overline{\lambda}(put1)$ decresceu de $\approx 200\ put1/s$ para $\approx 107\ put1/s$ e, para $M \geq 5$, $\overline{\lambda}(put1)$, permaneceu constante, à volta de $\approx 93.5\ put1/s$ (resultados consistentes com a replicação quádrupla dos registos). Estes valores são inferiores aos obtidos com a tecnologia <`domus-bsddb-btree`,`disk`> em uma ordem de magnitude, exemplificando a inadequabilidade de abordagens P2P a ambientes *cluster*, mais rentabilizáveis por abordagens como a Domus.

[34] Fenómeno designado, na gíria própria, por *churn*.

[35] A replicação está profundamente embebida no código do Bamboo, não sendo prática a sua desactivação.

Capítulo 8

Discussão

O trabalho desenvolvido nesta dissertação consubstancia uma aproximação alternativa à exploração das capacidades de ambientes *cluster*, designadamente para armazenamento distribuído. Essa aproximação, baseada em Dicionários Distribuídos (DDs), oferece interfaces simples e familiares, dos quais os programadores podem tirar partido para agilizar o desenvolvimento de aplicações a executar em ambiente *cluster* (ou, mais genericamente, num ambiente distribuído). Notavelmente, essas aplicações não têm de ser programadas, de raíz, como aplicações paralelas/distribuídas: uma aplicação convencional (centralizada), pode ainda tirar partido dos recursos do *cluster* por via da interacção com um ou mais *serviços distribuídos* de DDs. Este paradigma de desenvolvimento é suficientemente poderoso para a construção de serviços sofisticados, desde que as operações básicas de acesso a dicionários (inserção, leitura, pesquisa e remoção, por chave única) sejam suficientes; operações de semântica mais elaborada (*e.g.*, envolvendo gamas de valores ou manipulações *in situ* dos registos) exigirão aproximações diferentes ou camadas adicionais de funcionalidade.

A contribuição principal desta dissertação é uma abordagem *integrada* à *co-operação* de múltiplas Tabelas de Hash Distribuídas (DHTs), em ambiente *cluster*. A perspectiva integradora da abordagem advém do facto de a mesma procurar responder a um leque alargado de questões, relativamente complexas, cobrindo tópicos como *particionamento* (incluindo *distribuição* e *posicionamento*), *localização*, *co-operação* e *balanceamento*. O protótipo desenvolvido, com base na especificação da arquitectura Domus demonstra, para a maior parte destas questões, a exequibilidade das soluções preconizadas. Tendo em conta os objectivos da dissertação, expressos no capítulo 1, a arquitectura Domus e o seu protótipo representam, em nosso entender, uma resposta qualificada a esses objectivos.

De seguida, passamos em revista as várias contribuições da dissertação e apresentamos as perspectivas de trabalho futuro. O confronto das nossas contribuições com outras do género, realizado ao longo da dissertação, é complementar da breve revisão que se segue.

8.1 Modelos de Distribuição

Os modelos de distribuição de uma DHT, descritos no capítulo 3, garantem uma qualidade de distribuição *óptima*. Todavia exigem, para o efeito, i) algum conhecimento global sobre a distribuição da DHT e ii) coordenação centralizada da redistribuição. Estes requisitos

tornam os modelos (no seu estágio actual de especificação) inapropriados a cenários distribuídos de larga escala e/ou com elevados tempos de comunicação (*e.g.*, cenários P2P).

A preocupação em garantir uma boa qualidade de distribuição é relevante no quadro da operação de mecanismos de balanceamento dinâmico, como os propostos na dissertação; a lógica subjacente é a de que, quanto melhor for a qualidade de distribuição, mais eficazes poderão ser esses mecanismos. Neste contexto, a qualidade de distribuição óptima, assegurada pelos nossos modelos, representa uma vantagem competitiva importante, face a outras aproximações à realização de DHTs, que exibem níveis inferiores desse parâmetro, como revelou a análise comparativa com o Hashing Consistente [KLL+97], na secção 3.8.

8.2 Modelos de Posicionamento e Localização

No capítulo 4 desenvolvemos modelos de *posicionamento* e de *localização* (distribuída) compatíveis com os modelos de *distribuição* do capítulo 3. As razões de fundo que motivaram a opção por localização de tipo distribuído foram i) a expectativa de, perante o dinamismo do meio de execução, as DHTs terem de se redistribuir frequentemente, e ii) a necessidade de gerir, em simultâneo, a localização de múltiplas DHTs; nestas circunstâncias, uma abordagem distribuída à localização (no espaço e no tempo) é mais escalável.

Os modelos de localização distribuída em causa são essencialmente adaptações ou reformulações, respectivamente, de grafos DeBruijn [dB46] e de grafos Chord [SMK+01], para o cenário peculiar dos nossos modelos de distribuição; estes geram partições *descontínuas* do contradomínio de uma função de hash, implicando que os vértices do grafo de localização de uma DHT sejam os seus *hashes*, em vez dos seus nós computacionais ou virtuais (como acontece em abordagens assentes em Hashing Consistente, que geram partições *contínuas*).

A principal contribuição do capítulo 4 são os *algoritmos de encaminhamento acelerado*; estes foram desenvolvidos com o objectivo de reduzir o custo da localização distribuída sobre grafos baseados em *hashes*, para níveis próximos do custo de localização sobre grafos baseados em nós computacionais (os quais garantem o custo mínimo com métodos convencionais de encaminhamento). Notavelmente, os algoritmos desenvolvidos permitem uma redução do custo de localização abaixo dos mínimos inicialmente projectados. Adicionalmente, os algoritmos gozam da mais valia de também poderem ser aplicáveis a *caches de localização* (ver secção 5.6.5.3) e grafos baseados em nós computacionais ou virtuais.

8.3 Arquitectura Domus de Co-Operação de DHTs

A *arquitectura Domus*, especificada no capítulo 5, representa o principal contributo teórico da dissertação, como *arquitectura de co-operação de DHTs*[1], resultante da síntese de um conjunto de outras contribuições: a arquitectura adopta os modelos de distribuição e localização definidos nos capítulos 3 e 4, e integra-os, de forma consistente, com mecanismos e modelos de suporte ao *armazenamento*, *heterogeneidade*, *gestão* e *balanceamento* de DHTs; a possibilidade de dissociação entre o endereçamento e armazenamento das DHTs é também assumida, atravessando o desenho de todos os componentes e mecanismos da arquitectura; essa dissociação procura contribuir para uma melhor rentabilização

[1] Vistas como "serviços de armazenamento distribuído de dicionários", em ambientes *cluster*.

dos recursos do *cluster*, ao permitir a assunção de papéis mais especializados pelos nós.

A aproximação seguida no tratamento das questões ligadas ao armazenamento é relativamente simples, procurando ser independente de pormenores de baixo nível; assim, parte-se do princípio de que é possível recorrer a um conjunto de *tecnologias de armazenamento* bem conhecidas, para operar *repositórios* segundo o paradigma dos dicionários; em geral, estes serão locais aos nós de armazenamento das DHTs, mas tais nós poderão ser meros pontos de acesso a dispositivos remotos, de tipo SAN ou NAS; a abordagem prosseguida é flexível, admitindo diversas granularidades e graus de partilha, na exploração dos repositórios, e facilitando o suporte a *tecnologias de armazenamento* inicialmente não previstas.

A *heterogeneidade* de DHTs assenta na possibilidade de, por via de certos parâmetros, moldar atributos de várias categorias, com influência na operação de uma DHT, incluindo: i) função de *hash* utilizada, ii) qualidade de distribuição pretendida, iii) restrições permanentes à distribuição, iv) política de evolução, v) algoritmo de localização distribuída, vi) tecnologia de armazenamento, vii) limiares para os mecanismos de balanceamento do Endereçamento e do Armazenamento, viii) estratégias de localização a utilizar pelas aplicações clientes, etc. A valoração não automática destes parâmetros/atributos é da responsabilidade dos programadores, obedecendo a requisitos aplicacionais específicos.

Para além das operações básicas de acesso a dicionários (inserção, consulta, remoção), a arquitectura prevê a disponibilização, por meio de uma biblioteca de funções, de um conjunto de operações de *gestão* de vários níveis de granularidade: i) criação/destruição e desactivação/reactivação de instâncias da arquitectura, ii) adição/remoção e desactivação/reactivação de serviços Domus específicos, iii) criação/destruição e desactivação/reactivação de DHTs específicas. Os parâmetros e operações previstos permitem assim uma gestão flexível da forma como as DHTs se (re)distribuem pelo *cluster*, sendo essa (re)distribuição sensível a condições impostas de forma automática ou administrativa.

Os mecanismos de *balanceamento* descritos ao longo do capítulo 6, permitem o *auto-ajustamento* de instâncias da arquitectura às condições dinâmicas do ambiente de execução.

8.4 Modelos de Balanceamento Dinâmico

As contribuições do capítulo 6 vão ao encontro de outra das motivações fundamentais da nossa investigação, apontadas no capítulo 1. Assim, é graças a essas contribuições que uma instância da arquitectura Domus deverá ser capaz de utilizar o *cluster* de forma eficaz, mesmo co-existindo com outras aplicações/serviços distribuídos (incluindo outras instâncias da arquitectura). A ideia subjacente é a de que num *cluster* em regime de exploração *partilhado* (ambiente alvo do nosso trabalho), e portanto sem mecanismos globais de regulação de carga, a operação de mecanismos de *auto-regulação* pelas aplicações/serviços conduzirá a uma exploração mais eficiente dos recursos do *cluster*. Neste contexto, é também importante realçar que esses mesmos mecanismos ainda serão úteis em *clusters* sob regime de exploração *em lotes*: uma vez aplicados aos nós que compõem a partição atribuída a um utilizador, contribuirão para a sua rentabilização, porventura com mais eficácia, pois é menor a probabilidade de nesses nós executarem aplicações de outros utilizadores.

O balanceamento dinâmico de carga baseia-se na operação de dois mecanismos que são,

por definição, ortogonais (um actua sobre o "posicionamento" de nós virtuais, e outro sobre o "número" de nós virtuais, das DHTs), mas que actuam cooperativamente, sob a coordenação de um serviço *supervisor*, com uma visão global do estado dos recursos do *cluster* e da utilização que deles fazem as várias DHTs instanciadas; a abordagem prosseguida para o balanceamento dinâmico é pois *global* e *centralizada*, mas com o benefício potencial de as decisões de balanceamento serem mais eficazes. Os mecanismos de balanceamento são baseados em modelos lineares, dependentes de certas métricas de caracterização de nós computacionais, serviços e DHTs; exploram também a dissociação entre as funções de endereçamento e armazenamento das DHTs, o que permite atribuir essas funções a nós diferentes, como forma de melhor rentabilizar os recursos do *cluster*. Em resumo, os mecanismos em causa prosseguem uma filosofia de "balanceamento ciente dos recursos", como forma de realização de "tabelas de hash distribuídas auto-adaptativas" num *cluster*.

8.5 Protótipo da Arquitectura Domus

O carácter modular da arquitectura Domus, e o recurso à linguagem Python (como linguagem de prototipagem rápida), agilizaram a codificação da plataforma pDomus. Os benefícios da adopção da linguagem/ambiente Python fizeram-se sentir a vários níveis: i) interacções com o sistema exógeno de monitorização Ganglia [DSKMC03], ii) interacções cliente-servidor (passagem de mensagens, atendimento de pedidos, gestão de múltiplos fios de execução, controle de concorrência, gestão da consistência, etc.), iii) interacções com o sistema de ficheiros e com o sistema operativo, iv) operações de suspensão/retoma da "execução" de componentes da arquitectura, e v) facilidade de integração na plataforma de suporte a novas *tecnologias de armazenamento* (incluindo tecnologias externas ao Python).

Adicionalmente, a avaliação do protótipo confirmou o posicionamento relativo esperado, em termos de desempenho, das várias *tecnologias de armazenamento* e *estratégias de localização* suportadas. A mesma avaliação demonstrou i) a boa escalabilidade do protótipo, em condições extremas (intensivas) de utilização, ii) a sua competitividade com plataformas de armazenamento de categorias alternativas (base de dados MySQL [mys]) e iii) a sua superioridade face a uma plataforma P2P popular (plataforma Bamboo [bam]). Por avaliar fica o impacto real da re-distribuição dinâmica de DHTs, uma vez que o protótipo pDomus realiza apenas (por enquanto), a distribuição (pesada) inicial, na sua criação.

8.6 Trabalho Futuro e Perspectivas

Embora a escolha da linguagem Python para a realização do protótipo apresente a possibilidade de compatibilidade multi-plataforma, existem razões que levam a considerar a hipótese de complementar/reforçar a funcionalidade/desempenho do protótipo recorrendo a outras linguagens/ambientes de programação. Assim, o facto de o Java continuar a ser uma espécie de língua-franca para aplicações distribuídas, determina a conveniência em desenvolver um interface nessa linguagem para acesso às bibliotecas do protótipo; de igual forma, o interesse em incrementar substancialmente o desempenho da plataforma Domus (incluindo a exploração de tecnologias de comunicação de alto desempenho, tipicamente acessíveis usando primitivas de baixo nível) poderá motivar a re-codificação de partes em C/C++, prosseguindo a perspectiva adoptada para certas plataformas de armazenamento.

Noutra dimensão, a exploração de facilidades de que o Python dispõe poderá permitir elevar o potencial de aplicabilidade da arquitectura/plataforma Domus para lá do armazenamento distribuído de dicionários. Refira-se, em particular, a possibilidade de injectar código Python em tempo de execução, nos serviços Domus, de forma a viabilizar o processamento paralelo/distribuído dos registos das DHTs, *à-medida* das aplicações clientes.

Durante o longo caminho que percorremos, verificou-se um reposicionamento da comunidade ligada aos temas de investigação que prosseguimos, em torno dos ambientes P2P e *Grid*, incluindo a aproximação/convergência entre os dois mundos [FI03]. Em parte, o nosso trabalho foi capaz de integrar e adaptar algumas destas novas contribuições, designadamente as relacionadas com localização distribuída em ambientes P2P. Procurando ir de encontro às tendências mais actuais de investigação e desenvolvimento em sistemas paralelos/distribuídos, será feito um esforço para dotar a plataforma Domus de capacidades de execução em ambientes *cluster* operados em lotes, seja de forma isolada ou em *Grid*.

Para a execução sob um regime de exploração em lotes é importante a capacidade de operar num universo fechado e transiente de nós. Neste contexto, será útil o facto de o protótipo permitir delimitar o universo de operação de uma instância da arquitectura Domus. Por outro lado, serão necessárias novas funcionalidades, como a terminação graciosa automática para um tempo de execução limitado e a compatibilização das facilidades de desactivação/reactivação de DHTs com a mudança do universo de nós entre execuções sucessivas, dado que cabe ao *gestor de tarefas* a escolha da partição do *cluster* a utilizar.

O suporte a ambientes *Grid* passa por enriquecer o protótipo Domus do *middleware* necessário para que possa ser acedido, remotamente, como mais um dos recursos de armazenamento do *cluster*. Por exemplo, no contexto da arquitectura EGEE [CER05], tal poderia corresponder à implementação do interface SRM (*Storage Resource Manager*). Como o armazenamento é sustentado num modelo baseado em ficheiros, o suporte do protótipo a ambientes *Grid* deverá ser precedido do suporte a um Sistema de Ficheiros Paralelo/Distribuído onde DHTs Domus são usadas para armazenar blocos. A realização desse sistema de ficheiros poderia explorar *toolkits* como o FUSE (*Filesystem in Userspace*) [fus], em conjugação com uma *camada de fragmentação* ao nível da biblioteca Domus.

A preocupação em compatibilizar os nossos modelos e plataformas com ambientes *cluster* operando em *Grid* demonstra uma intenção de aproximação da nossa investigação aos problemas mais actuais. Em particular, a *Grid* enquanto ambiente especialmente vocacionado para lidar com a heterogeneidade e a partilha de recursos é um domínio que julgamos especialmente propício ao desenvolvimento da nossa abordagem e modelos, desde que devidamente enquadrados e adaptados às dimensões de escala da computação global.

Mais recentemente a plataforma Domus foi seleccionada como base de trabalho para explorar um modelo de suporte à execução de algoritmos de *Ray Tracing*, baseados em memória global, no quadro do projecto IGIDE (*Iluminação Global Interactiva em Ambientes Dinâmicos*) [2]. Nesse contexto, prevê-se investigação no sentido de melhorar a plataforma Domus de forma a responder efectivamente a necessidades extremas de desempenho no acesso aos dados, através do recurso a mecanismos de comunicação de baixo nível, tais como comunicação unilateral (*one-sided*) por cima de tecnologias como 10G-Myrinet.

[2]Projecto PTDC/EIA/65965/2006, financiado pela Fundação para a Ciência e Tecnologia.

Bibliografia

[ABC+06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick. The Landscape of Parallel Computing Research: a View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[Abe01] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, volume Springer LNCS 2172, pages 179–194, 2001.

[ABKU94] Y. Azar, A.Z. Broder, A.R. Karlin, and E. Upfal. Balanced Allocations. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing (STOC '94)*, pages 593–602, 1994.

[ACP95] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 1995.

[ADH03] K. Aberer, A. Datta, and M. Hauswirth. The Quest for Balancing Peer Load in Structured Peer-to-Peer Systems. Technical report, Distributed Information Systems Laboratory, Ecole Polytechnique Federale de Lausanne, 2003.

[ADH05] K. Aberer, A. Datta, and M. Hauswirth. Multifaceted Simultaneous Load Balancing in DHT-based P2P systems: A new game with old balls and bins. *Self-* Properties in Complex Information Systems*, Springer LNCS 3460:373–391, 2005.

[AHPS02] K. Aberer, M. Hauswirth, M. Punceva, and R. Schmidt. Improving Data Access in P2P Systems. *IEEE Internet Computing*, 6:2–11, 2002.

[AVL62] G. Adelson-Velskii and E.M. Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk SSSR*, pages 263–266, 1962.

[azu] Azureus: Java BitTorrent Client. http://azureus.sourceforge.net/.

[bam] The Bamboo DHT. http://www.bamboo-dht.org/.

[BBK07] Brown, Buford, and Kolberg. Tork: A Variable-Hop Overlay for Heterogeneous Networks. In *Proceedings of the 5th IEEE International Conference on Pervarsive Computing and Communications Workshops (PerComW '07)*, 2007.

[BCM03] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[beo] The Beowulf Cluster Site. http://www.beowulf.org/.

[ber] Oracle Berkeley DB. http://www.oracle.com/technology/products/berkeley-db/index.html.

[BFP06] P. Balaji, W. Feng, and D.K. Panda. The Convergence of Ethernet and Ethernot: A 10-Gigabit Ethernet Perspective. Technical Report OSU-CUSRC-1/06-TR10, Ohio State University, 2006.

[BKK+03] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, 2003.

[BLS93] J-C. Bermond, Z. Liu, and M. Syska. Mean Eccentricities of de Bruijn Networks. Technical Report RR-2114, CNRS - Université de Nice-Sophia Antipolis, 1993.

[bsd] BerkeleyDB Python Bindings. http://pybsddb.sourceforge.net.

[BSNP95] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. Cryptographic Hash Functions: A Survey. Technical Report TR-95-09, Department of Computer Science, University of Wollongong, 1995.

[BSS00] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, Distributed Data Placement Strategies for Storage Area Networks (Extended Abstract). In *Procs. of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA '00)*, 2000.

[BSS02] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, Adaptive Placement Strategies for Non-Uniform Capacities. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*, pages 53–62, 2002.

[Bur02] T. Burkard. Herodotus: A Peer-to-Peer Web Archival System. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 2002.

[cdb] Constant Database Library. http://cr.yp.to/cdb.html.

[CDK+03] M. Castro, P. Druschel, A.M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-Bandwidth Content Distribution in a Cooperative Environment. In *Procs. of the 2nd Int. Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[cen] CentOS. http://www.centos.org/.

[CER05] CERN. EGEE Middleware Architecture. https://edms.cern.ch/file/594698/1.0/architecture.pdf, 2005.

[Cic80] R.J. Cichelli. Minimal Perfect Hash Functions made Simple. *Communications of the ACM*, 23(1):17–19, 1980.

[CJO+07] L Cheng, K. Jean, A. Ocampo, R.and Galis, P. Kersch, and R. Szabo. Secure Bootstrapping of Distributed Hash Tables in Dynamic Wireless Networks. In *IEEE International Conference on Communications (ICC '07)*, pages 1917–1922, 2007.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[CLRT00] P.H. Carns, W.B. Ligon, R.B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000.

[CNY03] K. Chen, L. Naamani, and K. Yehia. miChord: Decoupling Object Lookup from Storage in DHT-Based Overlays, 2003. http://www.loai-naamani.com/Academics/miChord.htm, 2003.

[COJ+06] K. Cheng, R. Ocampo, K. Jean, A. Galis, C. Simon, R. Szabo, P. Kersch, and R. Giaffreda. Towards Distributed Hash Tables (De)Composition in Ambient Networks. In *Procs. of the 17th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '06)*, LNCS 4269, pages 258–268. Springer, 2006.

[CRR+05] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker. A Case Study in Building Layered DHT Applications. In *Proceedings of the 2005 ACM SIGCOMM*, 2005.

[CRS03] A. Czumaj, C. Riley, and C. Scheideler. Perfectly Balanced Allocation. In *Proceedings of the 7th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '03)*, 2003.

[Dab01] F. Dabek. A Cooperative File System. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2001.

[dB46] N.G. de Bruijn. A Combinatorial Problem. In *Koninklijke Nederlandse Akademie Van Wetenschappen*, volume 49, pages 758–764, 1946.

[DBH+02] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan Data Management Services for Parallel Dynamic Applications. *IEEE Computing in Science & Engineering*, 4(2):90–96, 2002.

[Dev93] R. Devine. Design and implementation of DDH: a Distributed Dynamic Hashing Algorithm. In *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, pages 101–114, 1993.

[DKKM01] F. Dabek, M.F. Kaashoek, D. Karger, and R. Morris. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001.

[dox] Doxygen - Source Code Documentation Generator Tool. http://www.stack.nl/ dimitri/doxygen.

[DR01a] P. Druschel and A. Rowstron. PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HOTOS '01)*, 2001.

[DR01b] P. Druschel and A. Rowstron. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001.

[DSKMC03] Federico D. Sacerdoti, Mason J. Katz, Matthew L. Massie, and David E. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of the IEEE Cluster 2003 Conference*, 2003.

[ED88] R.J. Enbody and H.C. Du. Dynamic Hashing Schemes. *ACM Computing Surveys*, 20(20):85–113, 1988.

[emu] eMule-Project.net. http://www.emule-project.net/.

[Fai05] J. Faik. *A Model for Resource-Aware Load Balancing on Heterogeneous and Non-Dedicated Clusters*. PhD thesis, Rensselaer Polytechnic Institute, 2005.

[FFM04] M.J. Freedman, E. Freudenthal, and D. Mazières. Democratizing Content Publication with Coral. In *Proceedings of 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[FG03] P. Faigniaud and P. Gauron. An Overview of the Content Adressable Network D2B. In *Proceedings of Principles of Distributed Computing (PODC '03)*, 2003.

[FI03] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[FK03] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong. Extendible Hashing: a Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(315-344):315–344, 1979.

[FPJ+07] J. Falkner, M. Piatek, J.P. John, A. Krishnamurthy, and T. Anderson. Profiling a Million User DHT. In *Procs. of the 2007 Internet Measurement Conference*, 2007.

[F.T91] Leighton. F.T. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Academic Press / Morgan Kaufmann, 1991.

[fus] FUSE – Filesystem in Userspace. http://fuse.sourceforge.net/.

[GBHC00] S.D. Gribble, E.A. Brewer, J.M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI '00)*, 2000.

[GC97] R. C. Guimarães and J. A. S. Cabral. *Estatística*. McGrawHill, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GLR03] A. Gupta, B. Liskov, and R. Rodrigues. One Hop Lookups for Peer-to-Peer Overlays. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HOTOS '03)*, 2003.

[GLR04] A. Gupta, B. Liskov, and R. Rodrigues. Efficient Routing for Peer-to-Peer Overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[GLS+04] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Proceedings of the 23rd Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM '04)*, 2004.

[GM04] P. Ganesan and G.M. Manku. Optimal Routing in Chord. In *Proceedings of the 15th ACM Symposium on Distributed Algorithms (SODA '04)*, pages 169–178, 2004.

[gnu] Gnutella. http://www.gnutella.com.

[Gra81] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 144–154, 1981.

[GS78] L.J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In IEEE, editor, *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[GS05] P.B. Godfrey and I. Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05)*, volume 1, pages 596–606, 2005.

[Gun03] N. Gunther. UNIX Load Average Part 2: Not Your Average Average. http://www.teamquest.com/resources/gunther/display/7/index.htm, 2003.

[HBC97] V. Hilford, F.B. Bastani, and B. Cukic. EH* – Extendible Hashing in a Distributed Environment. In *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC '97)*, 1997.

[HCH+05] R. Huebsch, B. Chun, J.M. Hellerstein, B.T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A.R. Yumerefend. The Architecture of PIER: an Internet-Scale Query Processor. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, pages 28–43, 2005.

[HGRW06] T. Heer, S. Gotz, S. Rieche, and K. Wehrle. Adapting Distributed Hash Tables for Mobile Ad Hoc Networks. In *Proceedings of the 4th IEEE International Conference on Pervasive Computing and Communications (PERCOM '06)*, pages 173–178, 2006.

[Hin07] K. Hinsen. Parallel Scripting with Python. *IEEE Computing in Science and Engineering*, 9(6):82–89, 2007.

[HJS+03] N.J.A. Harvey, M.B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, pages 113–126, 2003.

[HKM+88] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), 1988.

[HM92] G. Havas and B.S. Majewski. Optimal Algorithms for Minimal Perfect Hashing. Technical Report TR-234, The University of Queensland, 1992.

[II81] I. Imase and M. Itoh. Design to Minimize Diameter on Building-Block Networks. *IEEE Transactions on Computers*, 30, 1981.

[ipe] Iperf - the tcp/udp bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf.

[IRD02] S. Iyer, A.I.T. Rowstron, and P. Druschel. Squirrel: A Decentralized Peer-ot-Peer Web Cache. In *Proceedings of Principles of Distributed Computing (PODC '02)*, 2002.

[Ive03] Damian Ivereigh. RB-Trees Library. http://libredblack.sourceforge.net, 2003.

[Kar97] J.S. Karlsson. A Scalable Data Structure for a Parallel Data Server. Licentiate thesis no. 609, Department of Computer Science and Information Science, Linkoping University, 1997.

[KBCC00] J. Kubiatowicz, D. Bindel, Y. Chen, and S. Czerwinski. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, 2000.

[KK03] M.F. Kaashoek and D.R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[KLL+97] D. Karger, E. Lehman, F. Leighton, D. Levine, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[KMXY03] A. Kumar, S. Merugu, J. Xu, and X. Yu. Ulysses: A Robust, Low-Diameter, Low-Latency Peer-to-Peer Network. In *Proceedings of the 2003 International Conference on Network Protocols*, 2003.

[Knu98] D.E. Knuth. *The Art of Computer Programming – Volume 3: Sorting and Searching.* Addison-Wesley, 2nd edition, 1998.

[KR04] D.R. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, 2004.

[KRRS04] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, 2004.

[KSB+99] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *Proceedings of the 8th International WWW Conference*, 1999.

[KTL96] R. Kruse, C.L. Tondo, and B. Leung. *Data Structures and Program Design in C*. Prentice Hall, 2nd edition, 1996.

[Lan04] H.P. Langtangen. *Python Scripting for Computational Science*. Springer, 2004.

[Lar78] P.A. Larson. Dynamic Hashing. *BIT*, 18(2):184–201, 1978.

[LCP+04] E.K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Survey and Tutorial*, 6(1), 2004.

[lin] LINPACK. http://www.netlib.org/linpack/.

[Lit80] W. Litwin. Linear Hashing: A new Tool for File and Table Addressing. In *Proceedings of the 6th Conference on Very Large Data Bases*, pages 212–223, 1980.

[LKO+00] M.L. Lee, M. Kitsuregawa, B.C. Ooi, K-T Tan, and A. Mondal. Towards Self-Tuning Data Placement in Parallel Database Systems. In *Proceedings of the 2000 ACM SIGMOD - International Conference on Management of Data*, pages 225–236, 2000.

[LKRG03] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routig Distances and FaultResilience. In *Proceedings of the 2003 ACM SIGCOMM*, 2003.

[LL04] Y. Li and Z. Lan. A Survey of Load Balancing in Grid Computing. In *Proceedings of the 1st International Symposium on Computational and Information Science (CIS '04)*, volume Springer LNCS 3314, pages 280–285, 2004.

[LNS93a] W. Litwin, M.-A. Neimat, and D.A. Schneider. LH*: Linear Hashing for Distributed Files. In *Proceedings of the 1993 ACM SIGMOD - International Conference on Management of Data*, pages 327–336, 1993.

[LNS93b] W. Litwin, M.-A. Neimat, and D.A. Schneider. LH*: Linear Hashing for Distributed Files. Technical Report HPL-93-21, HP Labs, 1993.

[LNS96] W. Litwin, M.-A. Neimat, and D.A. Schneider. LH*: A Scalable, Distributed Data Structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[Lus07] J. Luszczek, P. Dongarra. High Performance Development for High End Computing with Python Language Wrapper (PLW). *The International Journal of High Performance Computing Applications,*, 21(3):360–369, 2007.

[MA06] L.R. Monnerat and C.L. Amorim. D1HT: A Distributed One Hop Hash Table. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, 2006.

[Man04] G.S. Manku. *Dipsea: A Modular Distributed Hash Table*. PhD thesis, Stanford University, 2004.

[MH05] N. Matlof and F. Hsu. Introduction to Threads Programming in Python. University of California, May 2005.

[MHB90] B.J. McKenzie, R. Harries, and T. Bell. Selecting a Hashing Algorithm. *Software Practice and Experience*, 2(20):209–224, 1990.

[MKL+02] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Labs, 2002.

[MM02] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on P2P Systems (IPTPS '02)*, 2002.

[MNN01] R. Martin, K. Nagaraja, and T. Nguyen. Using Distributed Data Structures for Constructing Cluster-Based Services. In *Proceedings of the 1st Workshop on Evaluating and Architecting Systems dependabilitY (EASY '01)*, 2001.

[MNR02] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Procs. of Principles of Distributed Computing (PODC '02)*, 2002.

[Mor68] D.R. Morrison. PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MRS01] M. Mitzenmacher, A. Richa, and R. Sitaraman. *Handbook of Randomized Computing*, chapter The power of two random choices: A survey of the techniques and results., pages 255–312. Kluwer Academic Publishers, 2001.

[Mut02] Muthitachroen, A. and Morris, R. and Gil, T.M. and Chen, B. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[mys] MySQL Database. http://www.mysql.org.

[OBS99] M.A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference (FREENIX Track)*, 1999.

[osc] OSCAR – Open Source Cluster Application Resources. http://oscar.openclustergroup.org.

[OV99] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[Pea90] P.K. Pearson. Fast Hashing of Variable Length Text Strings. *Communications of the ACM*, 6(33):677, June 1990.

[Pea99] W.K. Preslan et all. A 64–bit, Shared Disk File System for Linux. In *Proceedings of the 7h NASA Goddard Conference on Mass Storage Systems and Technolohies (in cooperation with the Sixteenth IEEE Symposium on Mass Storage Systems)*, 1999.

[Pfa04] B. Pfaff. Performance Analysis of BSTs in System Software. In *Proceedings of 2004 ACM SIGMETRICS - International Conference on Measurement and Modeling of Computer Systems*, 2004.

[Pre99] B.R. Preiss. *Data Structure and Algorithms with Object-Oriented Design Patterns in Java*. Wiley, 1999.

[PRR97] C.G. Plaxton, R. Rajaraman, and A. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 311–320, 1997.

[psy] PSYCO. http://psyco.sourceforge.net/.

[Pug90] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

[pyt] Python Programming Language – Official Website. http://www.python.org/.

[Qui03] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

[RB04] R. Rodrigues and C. Blake. When Multi-Hop Peer-to-Peer Routing Matters. In *Procs. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, 2004.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large Peer–to–Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content–Addressable Network. In *Proceedings of the 2001 ACM SIGCOMM*, 2001.

[RGK+05] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of the 2005 ACM SIGCOMM*, 2005.

[rhe] RHEL – Red Hat Enterprise Linux. http://www.redhat.com.

[Rig04] A. Rigo. Representation-based Just-in-Time Specialization and the Psyco Prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 15–26, 2004.

[RLS+03] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Structured P2P Systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[roc] Rocks Clusters. http://www.rocksclusters.org.

[RPAE02] J. Rufino, A. Pina, A. Alves, and J. Exposto. Distributed Paged Hash Tables. In *Proceedings of the 5th International Meeting on High Performance Computing for Computational Science (VECPAR 2002) - Selected Papers and Invited Talks*, LNCS, pages 679–692. Springer, 2002.

[RPAE04a] J. Rufino, A. Pina, A. Alves, and J. Exposto. A Cluster oriented Model for Dynamically Balanced DHTs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '04)*, 2004.

[RPAE04b] J. Rufino, A. Pina, A. Alves, and J. Exposto. Toward a Dynamically Balanced Cluster oriented DHT. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN '04)*, 2004.

[RPAE05] J. Rufino, A. Pina, A. Alves, and J. Exposto. Domus - An Architecture for Cluster-oriented Distributed Hash Tables. In *Proceedings of the 6th International Conferenece on Parallel Processing and Applied Mathematics (PPAM '05)*, 2005.

[RPAE07a] J. Rufino, A. Pina, A. Alves, and J. Exposto. Full-Speed Scalability of the pDomus platform for DHTs. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN '07)*, 2007.

[RPAE07b] J. Rufino, A. Pina, A. Alves, and J. Exposto. pDomus: a Prototype for Cluster-oriented Distributed Hash Tables. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP '07)*, 2007.

[RS98] M. Raab and A. Steger. Balls into Bins — A Simple and Tight Analysis. In *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '98)*, pages 159–170, 1998.

[RS04] C. Riley and C. Scheideler. A Distributed Hash Table for Computational Grids. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '04)*, 2004.

[SAB+05] L.G.A. Sung, N. Ahmed, R. Blanco, H. Li, M. Soliman, and D. Hadaller. A Survey of Data Management in Peer-to-Peer Systems. Technical report, School of Computer Science, University of Waterloo, 2005.

[SAZ+02] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proceedings of the 2002 ACM SIGCOMM*, 2002.

[SBE07] M. Steiner, E.W. Biersack, and T. Ennajjary. Actively Monitoring Peers in KAD. In *Procs. of the 6th Int. Workshop on Peer-to-Peer Systems (IPTPS'07)*, 2007.

[SBR04] C. Schindelhauer, S. Bottcher, and F. Ramming. The Design of Pamanet - the Paderborn Mobile Had-Hoc Network. In *Proceedings of the 2nd International Workshop on Mobility Management and Wireless Access Protocols (MobiWac '04)*, pages 119–121. ACM Press, 2004.

[sci] SciPy – Scientific Tools for Python. http://www.scipy.org/.

[Sed98] R. Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching*. Addison-Wesley Professional, 3rd edition, 1998.

[SEP+97] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart. The Global File System: A File System for Shared Disk Storage. *IEEE Transactions on Parallel and Distributed Systems*, 1997.

[SGK+85] S. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the USENIX 1985 Summer Conference*, 1985.

[sha95] Secure Hash Standard. NIST, U.S. Dept. of Commerce, National Technical Information Service FIPS 180-1, 1995.

[SHK95] B.A. Shirazi, A.R. Hyrson, and K.M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. Wiley-IEEE Computer Society Press, 1995.

[Sie79] H.J. Siegel. Interconnection Networks for SIMD Machines. *Computer*, 6(12), 1979.

[Slo04] J. Sloan. *High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI*. O'Reilly Media, Inc, 1st edition, 2004.

[SMK+01] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balkrishnan. Chord: A Scalable Peer–to–Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM*, pages 149–160, 2001.

[SP01] S. Sinha and M. Parashar. Adaptive Runtime Partitioning of AMR Applications on Heterogeneous Clusters. In *Proceedings of the 3rd IEEE International Conference on Cluster Computing*, pages 435–442. IEEE, 2001.

[SPW90] C. Severance, S. Pramanik, and P. Wolberg. Distributed Linear Hashing and Parallel Projection im Main Memory Databases. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 674–682, 1990.

[SS05] C. Schindelhauer and G. Schomaker. Weighted Distributed Hash Tables. In *Procs. of the 17th ACM Symp. on Parallel Algorithms and Architectures (SPAA '05)*, 2005.

[SSLM03] A. Singh, M. Srivatsa, L. Liu, and T. Miller. Apoidea: A Decentralized Peer-to-Peer Architecture for Crawling the World Wide Web. In *Proceedings of the ACM SIGIR Workshop on Distributed Information Retrieval*, 2003.

[SW05] R. Steinmetz and K. Wehrle, editors. *Peer-to-Peer Systems and Applications*. Number 3485 in LNCS. Springer, 2005.

[swi] SWIG. http://www.swig.org.

[sys] Sysstat. http://perso.orange.fr/sebastien.godard/.

[TFF05] J. Teresco, J. Faik, and J. Flaherty. Resource-Aware Scientific Computation on a Heterogeneous Cluster. *IEEE Computing in Science and Engineering*, 7(2):40–50, 2005.

[TTZ06] D. Talia, P. Trunfio, and J. Zeng. Peer-to-Peer Models for Resource Discovery in Large-scale Grids: A Scalable Architecture. In *Procs. of the 7th International Conference on High Performance Computing in Computational Science (VECPAR 2006) - Selected Papers and Invited Talks*, volume LNCS 4395, pages 66–79. Springer, 2006.

[Uzg91] R.C. Uzgalis. General Hash Functions. Technical Report TR 91-01, University of Hong Kong, 1991.

[VBW94] R. Vingralek, Y. Breitbart, and G. Weikum. Distributed File Organization with Scalable Cost/Performance. In *Proceedings of the 1994 ACM SIGMOD - International Conference on Management of Data*, 1994.

[VBW98] R. Vingralek, Y. Breitbart, and G. Weikum. Snowball: Scalable Storage on Networks of Workstations with Balanced Load. *Distributed and Parallel Databases*, 6(2):117–156, 1998.

[vmw] VMware: Virtualization, Virtual Machine & Virtual Server Consolidation. http://www.vmware.com.

[WPP+04] L. Wang, K.S. Park, R. Pang, V.S. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proceedings of the USENIX 2004 Annual Technical Conference*, 2004.

[WSH99] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5–6):757–768, 1999.

[XL97] C-Z. Xu and F.C.M. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, 1997.

[Xu03] J. Xu. On The Fundamental Tradeoffs between Routing Table Size and Network Diameter in Peer-to-Peer Networks. In *Proceedings of INFOCOM 2003. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, 2003.

[ZKJ01] B. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, Computer Science Division, University of California, Berkeley, 2001.

[ZLP96] M.J. Zaki, W. Li, and S. Parthasarathy. Customized Dynamic Load Balancing for a Network of Workstations. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC-5 '96)*. 282, 1996.

[ZYKG04] D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos. Information Retrieval Techniques for Peer-to-Peer Networks. *IEEE Computing in Science and Engineering*, 6(4):20–26, 2004.

Apêndice A

Publicações Principais (Resumos)

0. José Rufino, António Pina, Albano Alves and José Exposto. Distributed Paged Hash Tables. 5^{th} International Meeting on High Performance Computing for Computational Science (VECPAR'02) - Selected Papers and Invited Talks, LNCS 2565, Springer, pp. 679-692, Porto, 2002;

Abstract

In this paper we present the design and implementation of DPH, a storage layer for cluster environments. DPH is a Distributed Data Structure (DDS) based on the distribution of a paged hash table. It combines main memory with file system resources across the cluster in order to implement a distributed *dictionary* that can be used for the storage of very large data sets with key based addressing techniques. The DPH storage layer is supported by a collection of cluster–aware utilities and services. Access to the DPH interface is provided by a user–level API. A preliminary performance evaluation shows promising results.

1. José Rufino, António Pina, Albano Alves and José Exposto. Toward a dynamically balanced cluster oriented DHT. IASTED Int. Conference on Parallel and Distributed Computing and Networks (PDCN'04), Acta Press, pp. 48-55, Innsbruck, 2004;

Abstract

In this paper, we present a model for a cluster oriented Distributed Hash Table (DHT). It introduces *software nodes*, *virtual nodes* and *partitions* as high level entities that, in conjunction with the definition of a certain number of invariants, provide for the balancement of a DHT across a set of heterogeneous cluster nodes. The model has the following major features: **a)** the share of the hash table handled by each cluster node is a function of its *enrollment level* in the DHT; **b)** the enrollment level of a cluster node in the DHT may change dynamically; **c)** cluster nodes are allowed to dynamically join or leave the DHT. A preliminary evaluation proved that the quality of the balancement of partitions of the hash table across the cluster,

measured by the standard deviation with relation to the ideal average, surpass the one achieved by using another well known approach.

2. José Rufino, António Pina, Albano Alves and José Exposto. A cluster oriented model for dynamically balanced DHTs. 18^{th} International Parallel & Distributed Processing Symposium (IPDPS'04), IEEE, Santa Fe, 2004;

Abstract

In this paper, we refine previous work on a model for a Distributed Hash Table (DHT) with support to dynamic balancement across a set of heterogeneous cluster nodes. We present new high-level entities, invariants and algorithms developed to increase the level of parallelism and globally reduce memory utilization. In opposition to a global distribution mechanism, that relies on complete knowledge about the current distribution of the hash table, we adopt a local approach, based on the division of the DHT into separated regions, that possess only partial knowledge of the global hash table. Simulation results confirm the hypothesis that the increasing of parallelism has as counterpart the degradation of the quality of the balancement achieved with the global approach. However, when compared with Consistent Hashing and our global approach, the same results clarify the relative merits of the extension, showing that, when properly parameterized, the model is still competitive, both in terms of the quality of the distribution and scalability.

3. José Rufino, António Pina, Albano Alves and José Exposto. Domus - An Architecture for Cluster-oriented Distributed Hash Tables. 6^{th} International Conference on Parallel Processing and Applied Mathematics (PPAM'05) - Revised Selected Papers, LNCS 3911, Springer, pp. 296-303, Poznan, 2005;

Abstract

This paper presents a high level description of Domus, an architecture for cluster-oriented Distributed Hash Tables. As a data management layer, Domus supports the concurrent execution of multiple and heterogeneous DHTs, that may be simultaneously accessed by different distributed/parallel client applications. At system level, a load balancement mechanism allows for the (re)distribution of each DHT over cluster nodes, based on the monitoring of their resources, including CPUs, memory, storage and network. Two basic units of balancement are supported: *vnodes*, a coarse-grain unit, and *partitions*, a fine-grain unit. The design also takes advantage of the strict separation of object *lookup* and *storage*, at each cluster node, and for each DHT. Lookup follows a distributed strategy that benefits from the joint analysis of multiple partition-specific routing information, to shorten routing paths. Storage is accomplished through different kinds of data repositories, according to the specificity and requirements of each DHT.

4. José Rufino, António Pina, Albano Alves and José Exposto. pDomus: a prototype for Cluster-oriented Distributed Hash Tables. 15^{th} Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2007), IEEE Computer Society, pp. 97-104, Naples, 2007;

Abstract

The Domus architecture for Distributed Hash Tables (DHTs) is specially designed to support the concurrent deployment of multiple and heterogeneous DHTs, in a dynamic shared-all cluster environment. The execution model is compatible with the simultaneous access of several distributed/parallel client applications to the same or different running DHTs. Support to distributed routing and storage is dynamically configurable per node, as a function of applications requirements, node base resources and the overall cluster communication, memory and storage usage. pDomus is a prototype of Domus that creates an environment where to evaluate the model embedded concepts and planned features. In this paper, we present a series of experiments conduced to obtain figures of merit i) for the performance of basic dictionary operations, and ii) for the storage overhead resulting from several storage technologies. We also formulate a ranking formula that takes into account access patterns of clients to DHTs, to objectively select the most adequate storage technology, as a valuable metric for a wide range of application scenarios. Finally, we also evaluate client applications and services scalability, for a select dictionary operation. Results of the overall evaluation are promising and a motivation for further work.

5. José Rufino, António Pina, Albano Alves and José Exposto. Full-Speed Scalability of the pDomus Platform for DHTs. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN'07), ACTA Press, pp. 69-76, Innsbruck, 2007;

Abstract

Domus is an architecture for Distributed Hash Tables (DHTs) tailored to a shared-all cluster environment. Domus DHTs build on a (dynamic) set of cluster nodes; each node may perform routing and/or storage tasks, for one or more DHTs, as a function of the node base (static) resources and of its (dynamic) state. Domus DHTs also benefit from a rich set of user-level attributes and operations. pDomus is a prototype of Domus that creates an environment where to evaluate the architecture concepts and features. In this paper, we present a set of experiments conduced to obtain figures of merit on the scalability of a specific DHT operation, with several lookup methods and storage technologies. The evaluation also involves a comparison with a database and a P2P-oriented DHT platform. The results are promising, and a motivation for further work.

§ As publicações 5, 4, 3, 1, e 0 encontram-se indexadas nas bases de dados do Institute for Scientific Information (ISI Web of Knowledge). As contribuições da publicação 0 [RPAE02] não integram a dissertação mas fazem parte de investigação preliminar.

Apêndice B

Conceitos Básicos de Hashing

Os conceitos apresentados são aprofundáveis pela consulta de obras de referência da área de Algoritmia e Estruturas de Dados [KTL96, Knu98, Sed98, Pre99, CLRS01]. No caso do Hashing Dinâmico, para além dos artigos em que as abordagens foram originalmente propostas [Lar78, FNPS79, Lit80], existem também estudos comparativos [ED88]. A notação usada é consistente com a da dissertação; é também uniforme entre abordagens diferentes.

B.1 Funções de Hash

Uma função de hash $f : K \mapsto H$ faz corresponder chaves k de um conjunto K, a hashes h de um conjunto H. Em termos computacionais, f processa a sequência de bits de uma chave (cujo número de bits pode variar) e produz outra sequência de bits (o hash da chave). Sendo $\mathcal{L}_f$ o número de bits do hash[1], então $\#H = 2^{\mathcal{L}_f}$ e, na base 10, $H = \{0, 1, ..., 2^{\mathcal{L}_f} - 1\}$. Com Hashing Estático, $\mathcal{L}_f$ é fixo; com esquemas de Hashing Dinâmico, $\mathcal{L}_f$ pode variar.

Por norma, o recurso ao hashing ocorre em cenários onde $\#K >> \#H$, *i.e.*, quando o número de chaves é muito superior ao de hashes; nestas condições, as funções de hash são não injectivas[2], havendo necessariamente chaves diferentes associadas a um mesmo hash.

B.1.1 Funções de Hash Perfeitas

Por vezes, é possível definir *funções de hash perfeitas*, intrinsecamente injectivas. Para tal, é necessário conhecer *a priori* a totalidade das chaves K. Uma variante são as *funções de hash perfeitas mínimas* [Cic80, HM92], que garantem que todos os hashes têm uma (e uma só) correspondência inversa (*i.e.*, uma chave correspondente). A construção de tabelas de símbolos pelos compiladores ou de tabelas de localização de ficheiros em dispositivos do tipo *read-only* representam situações onde são aplicáveis funções de hash deste tipo.

[1] Ou melhor, o número actualmente relevante de bits, de entre os que são geráveis por f.

[2] Uma função f diz-se injectiva se e só se, quaisquer que sejam x e y pertencentes ao domínio da função, x é diferente de y implica que $f(x)$ é diferente de $f(y)$.

B.1.2 Eficácia das Funções de Hash

A eficácia de uma função de hash mede-se basicamente por duas métricas: i) qualidade da dispersão de K sobre H e ii) tempo de computação de um hash. A qualidade será tanto maior quanto mais uniforme for o número de ocorrências de cada hash, ou seja, se a função de hash associar um número semelhante de chaves a cada hash; este requisito é mais facilmente realizável por *funções de hash específicas*, que optimizam a dispersão para um certo tipo de dados (sequências de caracteres, inteiros, reais, etc.), do que por *funções de hash genéricas*, que se esperam igualmente eficazes para vários tipos de chaves; com efeito, a definição de funções de hash genéricas é, por definição, mais difícil e complexa, embora já exista alguma investigação consolidada em torno do tema [MHB90, Uzg91, Pea90].

O segundo critério de eficácia requer que as rotinas de computação de um hash sejam eficientes. Para o efeito, é comum a codificação de funções de hash em linguagens de baixo nível e a exploração de facilidades específicas das arquitecturas/processadores alvo; neste contexto, o conhecimento do tipo de dados das chaves poderá também ser relevante.

B.1.3 Funções de Hash Criptográficas

Importa ainda registar a distinção entre funções de hash *não-criptográficas* e *criptográficas* [BSNP95] sendo que, no contexto desta tese, é suficiente o recurso a funções de hash não-criptográficas. Basicamente, as funções criptográficas (também conhecidas por *funções de hash unidirecionais*[3]), gozam de propriedades específicas, como: a) dado um hash é difícil obter (por inversão) uma chave correspondente, b) dada uma chave e o seu hash, é difícil descobrir outra chave com hash igual, c) é difícil arquitectar duas chaves com hash igual.

B.2 Tabelas de Hash

Uma *tabela de hash* é uma estrutura de dados vectorial (*i.e.*, um array unidimensional) em que o acesso a cada entrada é precedido da execução de uma função de hash associada à tabela. Tipicamente, a função de hash é aplicada à componente *chave* de um registo do tipo $< chave, dados >$ e o hash resultante actua como índice de uma entrada da tabela, associada ao registo. Cada entrada da tabela poderá comportar um ou mais registos. Dada uma tabela de hash T_f, cujo acesso é governado pela função de hash $f : K \mapsto H$, T_f terá $\#H = 2^{\mathcal{L}_f}$ entradas distintas, cujos índices são dados pelo intervalo $\{0, 1, ..., \#H - 1\}$.

A possibilidade de acesso directo a cada entrada constitui a propriedade mais atractiva das tabelas de hash. Assim, é frequente o recurso a estas estruturas de dados quando que se exige um acesso rápido a um registo qualquer, de entre um conjunto de registos (*e.g.*, realização de tabelas em bases de dados, tabelas de símbolos em compiladores, etc.).

[3]Do inglês *one-way hash functions*.

B.2.1 Tratamento de Colisões

Um dos principais problemas associados às tabelas de hash são as *colisões*, que degradam o desempenho do acesso. As colisões ocorrem quando a função de hash usada não é injectiva, traduzindo-se na associação de dois ou mais registos diferentes à mesma entrada da tabela. Na perspectiva das abordagens ao tratamento de colisões, as tabelas de hash podem classificar-se de i) *abertas* – cada entrada comporta um número fixo de registos ou ii) *fechadas* – cada entrada comporta um número virtualmente ilimitado de registos, pelo uso de estruturas de dados dinâmicas (listas ligadas[4], árvores, skip-lists [Pug90], etc.).

O tratamento de colisões numa tabela de hash aberta socorre-se de técnicas diversas. À partida, cada entrada da tabela pode ser vista como um *contentor* (usualmente denominado de *bucket* ou *bin*) com certa capacidade (fixa) de registos. Esgotada a capacidade do contentor, recorre-se a contentores especiais (*overflow buckets*) ou a técnicas de *probing/rehashing* para encontrar uma entrada com capacidade de armazenamento disponível.

Com *probing linear*, percorre-se a tabela sequencialmente, a partir da entrada ocupada, em busca de uma entrada livre. Na tentativa de minimizar fenómenos de *clustering* (formação de sequência de entradas ocupadas que intercalam com sequências de entradas livres), é comum o recurso a variantes desta técnica, incluindo o uso de a) deslocamentos aleatórios (com semente determinística, dedutível da chave), b) deslocamentos quadráticos ($x + 1$, $x + 4$, $x + 9$, ...), c) deslocamentos baseados em número primos ($x + 1$, $x + 3$, $x + 5$, ...), etc. Com *rehashing não-linear*[5], geram-se hashes sucessivos (com a função de hash principal/primária, ou com funções secundárias) até se encontrar uma entrada disponível.

Quando o número de chaves/registos é superior ao número de hashes, uma tabela de hash fechada representa a opção adequada, beneficiando da flexibilidade que decorre do uso de estruturas de dados dinâmicas para o tratamento de colisões. Ainda assim, importa referir a sobrecarga introduzida, em termos de consumo de RAM, pelo suporte às estruturas de dados dinâmicas (apontadores, etc.); essa sobrecarga será mais ou menos relevante consoante a dimensão dos registos de dados a salvaguardar. Se a tabela de hash for à partida sobre-dimensionada (em geral viável apenas em casos particulares), *i.e.*, se o numero de entradas/hashes for superior ao número previsto de chaves/registos, então uma tabela de hash aberta poderá ser suficiente; neste caso, mesmo que ocorram colisões, o facto de existir um número suficientemente grande de entradas disponíveis, permite rapidamente encontrar uma entrada livre para acomodar um registo; por outro lado, é preciso levar em conta o desperdício potencial de RAM, representado pelas entradas livres remanescentes.

B.2.2 Desempenho do Acesso

O desempenho do acesso a uma tabela de hash depende directamente da eficácia da função de hash usada. Neste contexto, para além do cálculo eficiente de cada hash em particular[6], acaba por ser mais importante que a função de hash produza uma dispersão tendencial-

[4]No caso particular do recurso a listas ligadas, a abordagem toma a designação de *(separate-)chaining*.
[5]Estratégia também designada, por vezes, de *hashing secundário* ou *hashing duplo* [Sed98].
[6]Operação que representa, em geral, uma fracção modesta do tempo global de acesso a um registo.

mente uniforme do conjunto das chaves sobre o conjunto dos hashes, de forma a garantir um número de registos semelhante, por cada entrada, e um tempo médio de acesso reduzido, por registo, independentemente da estratégia usada para o tratamento de colisões.

B.3 Hashing Estático e Hashing Dinâmico

O Hashing Estático é a abordagem convencional ao Hashing, na qual é fixo o número de bits (relevantes) gerados pela função de hash; consequentemente, é fixo o número de entradas da Tabela de Hash associada à função. Com Hashing Dinâmico, o número de bits (relevantes) gerados pela função de hash é variável, o que permite ajustar o número de entradas da Tabela de Hash associada, de forma a reduzir a probabilidade de colisões e/ou dar resposta ao esgotamento da capacidade de armazenamento das entradas.

B.3.1 Hashing Interno e Hashing Externo

Em geral, o Hashing Estático é *interno*, ou seja, está associado à exploração de Tabelas de Hash em memória principal (RAM). Por outro lado, o Hashing Dinâmico é usualmente *externo*, ou seja, recorre a memória secundária (Disco) como meio de armazenamento.

De facto, embora a concretização de Hashing Dinâmico em RAM seja perfeitamente viável[7], as abordagens desse tipo foram originalmente concebidos para a realização de dicionários em Disco; estes, organizam-se em unidades de armazenamento de capacidade fixa, designados por *contentores* (*buckets*), de forma que, i) um contentor é realizável por um ou mais blocos do Disco e ii) a função/tabela de hash servirá para agilizar a sua localização.

B.3.2 Hashing Dinâmico com Directoria e sem Directoria

Os esquemas de Hashing Dinâmico distinguem–se, essencialmente, pela forma como a tabela e os contentores evoluem, à medida que os registos vão sendo inseridos/removidos. Adicionalmente, tendo em conta o mecanismo usado para a localização dos contentores, os esquemas de Hashing Dinâmico classificam-se em i) *com directoria* ou ii) *sem directoria*.

Nos esquemas *com directoria*, a própria Tabela de Hash é também designada de *directoria*, dado que cada entrada contém uma referência para o contentor respectivo em Disco. A utilização do termo *directoria* deve-se ao facto de a Tabela de Hash intermediar o acesso a objectos (contentores) em memória secundária, pelo que o papel da Tabela de Hash é semelhante ao desempenhado por uma directoria num sistema de ficheiros tradicional.

Nos esquemas *sem directoria* dispensa-se a referida directoria: a utilização de várias funções de hash, assistida por certas estruturas auxiliares, permite gerir a expansão/contracção do dicionário (um contentor de cada vez) e localizar facilmente qualquer contentor.

[7] Caso da abordagem DLH [SPW90] de Hashing Linear sobre memória partilhada, ou da abordagem LH*LH [Kar97] de Hashing Linear Distribuído sobre uma máquina paralela.

B.4 Hashing Dinâmico Centralizado

De seguida, apresenta-se uma descrição sumária de abordagens de referência ao Hashing Dinâmico Centralizado [ED88], incluindo esquemas *com directoria* e *sem directoria*. A apresentação destas abordagens é essencial para uma boa percepção do salto evolutivo que representam as respectivas versões distribuídas, apresentadas e comparadas na secção 2.4.

B.4.1 Hashing Dinâmico de Larson (DH)

B.4.1.1 Árvores de Prefixos

O Hashing Dinâmico de Larson [Lar78] é uma abordagem *com directoria*, na qual a directoria se organiza como uma *árvore de prefixos* (*prefix tree*), também conhecida por *trie* [Mor68]. Numa *trie*, um percurso da *raiz* até uma *folha* corresponde a uma sequência de símbolos de um determinado alfabeto. Por exemplo, na secção a) da figura B.1 pode-se observar uma *trie* para o alfabeto binário, com apenas três sequências possíveis, a partir da raiz R: 00, 01 e 1; cada sequência pode ser vista como um *prefixo* de outra, mais longa, de forma que, todas as sequências com o mesmo prefixo podem ser agrupadas e referenciadas a partir da mesma folha (nó terminal); por exemplo, as sequências **00**011, **00**110 e **00**101 partilham o prefixo **00**, sendo todas "acessíveis" a partir da folha F da *trie*.

B.4.1.2 Mecanismo Básico

Basicamente, as folhas da *trie* (directoria) referenciam contentores de registos, de capacidade limitada. Quando um contentor b encher (situação de *overflow*), será necessário criar outro contentor, b', e repartir com ele os registos (evento de *split*); o processo comporta o aumento da *profundidade* (local) l_b do contentor b e, eventualmente, da *profundidade* (global) l da *trie* (os termos *profundidade* ou *altura* dizem respeito a um certo número de níveis ou bits). Quando a função de hash f utilizada produz hashes de $\mathcal{L}_f$ bits, tem-se $l_b \leq l \leq \mathcal{L}_f$.

A profundidade da *trie* evolui então dinamicamente, podendo não só aumentar, em resultado da divisão (*split*) de contentores, como também diminuir, após a fusão (*merge*) de contentores. Uma fusão é viável sempre que o conteúdo de dois contentores vizinhos (*i.e.*, descendentes directos da mesma folha da *trie*) puder ser acomodado por um só contentor.

Se a função de hash produzir hashes distribuidos uniformemente pelo seu contra-domínio, a *trie* será balanceada, pelo que i) a profundidade de qualquer ramo será semelhante, ii) assim como a taxa de ocupação dos contentores e iii) o esforço médio de acesso a qualquer registo; uma medida deste esforço é o número de níveis da *trie* que é necessário descer até ao contentor apropriado; no limite, esse número corresponde ao número de bits do hash (ou seja, a *altura/profundidade máxima* da *trie* é definida pelo número de bits do hash).

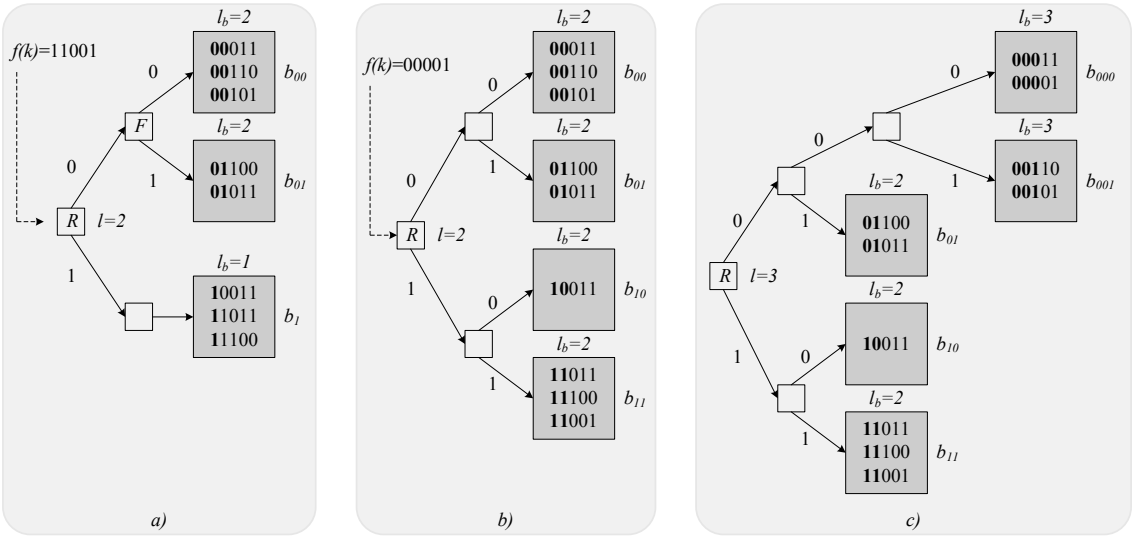


Figura B.1: Exemplo ilustrativo da aplicação de Hashing Dinâmico de Larson.

B.4.1.3 Exemplo Ilustrativo

A figura B.1 serve de apoio à descrição de um exemplo ilustrativo do mecanismo anterior.

Assim, na figura B.1, i) assume-se que a função de hash (f) produz hashes de 5 bits ($\mathcal{L}_f = 5$), ii) a profundidade global (l) surge ao lado da raiz e as locais (l_b) no topo dos contentores (representados a cinzento), iii) a designação de cada contentor ($b_{...}$) aparece à sua direita, baseada na sequência dos bits definida pelo trajecto mais directo, da raiz da *trie*, até ao contentor, iv) a sequência de bits anterior determina quais os registos armazenáveis por um contentor (apenas os registos para os quais a sequência é um prefixo do hash da chave), v) a capacidade dos contentores é, neste caso, limitada a três registos[8].

Na figura B.1.a), a *trie* referencia três contentores, dos quais dois já esgotaram a sua capacidade (b_{00} e b_1) e outro ainda tem capacidade para mais um registo (b_{01}). A transição do cenário a) para o b) é despoletada pela inserção de um registo para o qual o hash da chave é a sequência 11001. À partida, b_1 seria o contentor apropriado. Como está cheio, é necessário criar um novo contentor e repartir com ele os registos. Do processo resulta, a substituição de b_1 pelos contentores b_{10} e b_{11}, sendo o novo registo acomodado por b_{11}.

A transição do cenário b) para o c) prossegue uma lógica semelhante, mas em que agora é necessário criar mais um nó da *trie*, aumentando a profundidade do ramo em causa. Assim, durante a inserção do registo para o qual o hash da chave é a sequência 00001, verifica-se que o contentor apropriado (b_{00}) tem a sua capacidade esgotada, pelo que é necessário criar outro e repartir com ele os registos; do processo resulta a substituição do contentor b_{00} pelos contentores b_{000} e b_{001}, sendo o novo registo acomodado por b_{000}.

[8]Nos contentores são apenas representados os hashes correspondentes às chaves dos registos aí inseridos.

B.4.2 Hashing Extensível de Fagin (EH)

B.4.2.1 Mecanismo Básico

O Hashing Extensível de Fagin [FNPS79] é também uma abordagem *com directoria*, mas em que a directoria é colapsada numa tabela (ver figura B.2): cada entrada da tabela, directamente indexada por uma subsequência de bits do hash, referencia um contentor de registos, de capacidade limitada. Tal como na abordagem de Larson, quando um contentor esgota a sua capacidade, é necessário criar outro e repartir os registos com ele; todavia, quando isso acontece, pode ser necessário duplicar o número de entradas da directoria.

Num dado instante (antes da l'ésima expansão), a directoria comporta 2^l entradas, sendo l a *profundidade global* da directoria (com $l \leq \mathcal{L}_f$). Nesse contexto, é usada a função f_l para indexar a directoria; tal função gera hashes de l bits, limitados ao intervalo $\{0, 1, ..., 2^l - 1\}$.

Adicionalmente, definem–se *profundidades locais* para cada contentor b, denotadas por l_b (com $l_b \leq l$ para qualquer b); essas profundidades correspondem ao número de bits que é efectivamente usado para distinguir o hash das chaves/registos guardadas no contentor (ou seja, na prática, f_{l_b} é a função de hash usada no contexto de b). Desta forma, é admissível que várias entradas da directoria apontem para o mesmo contentor: basta que a profundidade local do contentor seja inferior à profundidade global. Por exemplo, se $l = 3$ e $l_b = 2$, então haverá $2^{l-l_b} = 2^1$ entradas da directoria que apontam para b; análogamente, se $l_b = 1$, então haverá $2^{l-l_b} = 2^4$ entradas da directoria apontando para b.

Se o contentor b com profundidade local l_b esgota a capacidade após uma colisão, então:

1. se $l_b = l$, é necessário expandir a directoria; mais ainda, se f_{l+1} não for suficiente para evitar a colisão, será necessária repetir a expansão, até se eliminar a colisão;

2. se $l_b < l$, a profundidade local aumenta uma unidade e cria–se um contentor vizinho (*buddy*) com essa profundidade; se a função f_{l_b+1} for suficiente para evitar a colisão, repartem-se as chaves/registos entre os dois contentores; se a colisão persistir, repete–se o processo até que, no limite, $l_b = l$, sendo necessário expandir a directoria.

Se a directoria se expandir, apenas a profundidade local dos contentores envolvidos no processo será actualizada, permanecendo inalterada a dos restantes. Naturalmente, quando a directoria se expande, aumenta o número de entradas que apontam para os contentores.

Note-se que os recursos consumidos pela directoria não crescem de forma suave/linear, mas sim de forma exponencial (por duplicações sucessivas). Em consequência, poderão ser frequentes muitas entradas redundantes, ou seja, entradas que apontam para um contentor partilhado; esta situação ocorre imediatamente a seguir a uma duplicação e persiste enquanto a profundidade local da maior parte dos contentores permanecer inferior à global.

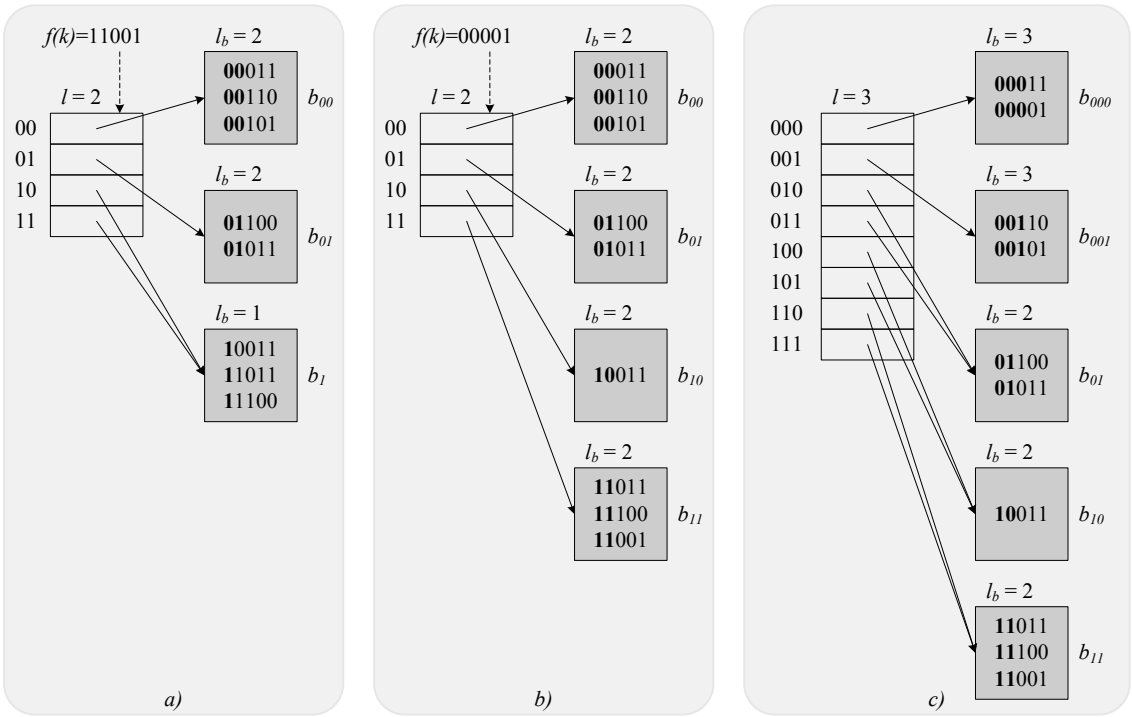


Figura B.2: Exemplo ilustrativo da aplicação de Hashing Extensível.

B.4.2.2 Exemplo Ilustrativo

A figura B.2 ilustra a aplicação do mecanismo descrito anteriormente numa situação que corresponde à transposição do cenário da figura B.1, usado no contexto do Hashing Dinâmico de Larson. Desta forma, é possível verificar a equivalência das duas abordagens.

Assim, na figura B.2, 1) a função de hash (f) produz hashes de 5 bits ($\mathcal{L}_f = 5$), 2) a profundidade global (l) surge no topo da directoria e as locais (l_b) no topo dos contentores (representados a cinzento), 3) a designação de cada contentor ($b_{...}$) aparece à sua direita, baseada numa sequência específica dos bits esquerdos do hash, em número dado pela profundidade local, 4) a sequência de bits anterior determina quais os registos armazenáveis por um contentor (apenas os registos cujo hash da chave contém, na parte esquerda, aquela sequência de bits), 5) a capacidade dos contentores é limitada de novo a três registos.

O ponto de partida, dado pelo cenário a), é caracterizado por uma profundidade global $l = 2$, ou seja, a directoria comporta $2^2 = 4$ entradas; existem ainda três contentores (b_{00}, b_{01} e b_1); destes, dois já esgotaram a sua capacidade (b_{00} e b_1) e outro ainda tem capacidade para mais um registo (b_{01}); adicionalmente, a profundidade local dos contentores b_{00} e b_{01} é já a máxima possível (2), tendo em conta a restrição $l_b \leq l$, enquanto que a profundidade local do contentor b_1 é apenas de 1; precisamente, sendo $l_{b_1} = 1 < l = 2$, tal significa que há $2^{l-l_{b_1}} = 2^{2-1} = 2$ entradas da directoria que apontam para b_1 (mais especificamente, b_1 concentra os registos cujo hash da chave assume, no bit mais à esquerda, o valor **1**).

A transição do cenário a) para o b) é despoletada pela inserção de um registo para o qual

o hash da chave é a sequência 11001. À partida, b_1 seria o contentor apropriado. Como está cheio, é necessário criar um novo contentor e repartir com ele os registos. Do processo resultam b_{10} e b_{11}, com profundidade local 2, sendo o novo registo acomodado por b_{10}.

Finalmente, a transição do cenário b) para o c) representa uma situação em que é necessário duplicar a directoria. Assim, na tentativa de inserir um registo para o qual o hash da chave é a sequência 00001, chega-se à conclusão que o contentor apropriado (b_{00}) tem a sua capacidade esgotada; todavia, como a profundidade local de b_{00} já atingiu o máximo ($l_{b_{00}}) = 2 = l$), é necessário duplicar a directoria, em paralelo com a criação de um novo contentor; em resultado do processo, a profundidade global aumenta uma unidade ($l = 3$) e surgem os contentores b_{000} e b_{001}, que repartem entre si os registos do contentor b_{00} (sendo b_{000} responsável por acomodar o registo que despoletou todo o processo). Os restantes contentores permanecem na profundidade anterior, sendo apenas de registar o facto de, agora, mais entradas (o dobro) da directoria apontarem para cada um desses contentores.

B.4.3 Hashing Linear de Litwin (LH)

B.4.3.1 Mecanismo Básico

O Hashing Linear de Litwin [Lit80] é a abordagem mais representativa de um esquema de Hashing Dinâmico *sem directoria*. O objectivo é duplo: 1) acesso mais eficiente aos registos ao evitar a consulta prévia de uma directoria; 2) evitar o armazenamento da directoria.

A ideia base é a de que um dicionário é visto como uma sequência de contentores (em termos lógicos[9]), que cresce/decresce linearmente, um contentor de cada vez, por acrescento/remoção, sempre a partir do fim da sequência. Esta particularidade dispensa um índice/directoria dos contentores, permitindo ainda um acesso directo aos mesmos, mas impõe i) a utilização simultânea de duas funções de hash, ii) restrições à sub-divisão (*spliting*) de contentores em sobrecarga e iii) a necessidade de algumas estruturas auxiliares.

Em termos formais, o Hashing Linear pode ser descrito com o auxílio da seguinte notação.

Seja l o nível de subdivisão (*splitlevel*) actual dos contentores[10] e B o conjunto dos contentores b_j, com $2^l \leq \#B < 2^{l+1}$ e $j \in \{0, ..., 2^{l+1} - 1\}$. Por exemplo, na figura B.3.g), tem-se $l = 1$, $B = \{b_{00}, b_1, b_{10}\}$ (com b_j expresso em binário) e $2^1 = 2 \leq \#B = 3 < 2^{1+1} = 4$.

Seja ainda um índice s, com $s \in \{0, ..., 2^l - 1\}$, que define qual o próximo contentor a subdividir. Neste contexto: 1) os contentores $B' = \{b_j : j \in \{0, ..., s - 1\}\}$ já foram subdivididos, 2) os contentores $B'' = \{b_j : j \in \{s, ..., 2^l - 1\}\}$ ainda não foram subdivididos e 3) os contentores $B''' = \{b_j : j \in \{2^l, ..., \#B - 1\}\}$ resultaram da subdivisão dos contentores B'. Por exemplo, na figura B.3.g), verifica-se que $B' = \{b_{00}\}$, $B'' = \{b_1\}$ e $B''' = \{b_{10}\}$.

Num dado instante, apenas é admissível a subdivisão do contentor b_s. A subdivisão dos contentores vai-se processando, de forma linear, à medida que o índice s é incrementado. A subdivisão de b_s apenas ocorre se for ultrapassado um factor de carga (*load factor*) pré-

[9] Uma vez que o sistema de ficheiros pode não garantir contiguidade dos blocos associados ao dicionário.

[10] Em que l corresponde, de certa forma, à altura da *trie* usada pela abordagem DH de Larson ou, equivalentemente, à profundidade global da directoria usada na abordagem EH de Fagin.

definido, correspondente a uma certa taxa de ocupação (que pode ser inferior a 100%). A eventual ultrapassagem dessa taxa em outros contentores não despoleta a subdivisão desses contentores, uma vez que são forçados a aguardar a sua vez. Um contentor em sobrecarga poderá ter necessidade de alijar registos em excesso para contentores especiais.

Adicionalmente, é necessário operar, em simultâneo, com duas funções de hash: uma função f_l para aceder aos contentores que ainda não foram subdivididos, e uma função f_{l+1} para aceder aos contentores que sofreram ou resultaram de uma subdivisão. Estas funções correspondem, respectivamente, à utilização de um sufixo (ou prefixo, c.f. a convenção) de l ou $l+1$ bits (sendo l o nível de subdivisão que introduzimos previamente), extraídos a partir do hash gerado por uma função f suficientemente prolífica (*i.e.*, $\mathcal{L}_f \geq l+1, \forall l$).

Note-se que, apesar da função f_l gerar hashes de l bits no intervalo $\{0, ..., 2^l - 1\}$, apenas serão relevantes os hashes do subconjunto $\{s, ..., 2^l - 1\}$, correspondentes aos contentores que ainda não foram subdivididos. Análogamente, para a função f_{l+1}, que gera hashes no intervalo $\{0, ..., 2^{l+1} - 1\}$, apenas serão relevantes 1) os hashes do subconjunto $\{0, ..., s-1\}$, associados a contentores já subdivididos e 2) os hashes do subconjunto $\{2^l, ..., \#B - 1\}$, relativos a contentores resultantes de uma subdivisão; mais ainda, assumindo a utilização de sufixos em vez de prefixos, $f_{l+1} = f_l$ na situação 1), e $f_{l+1} = f_l + 2^l$ na situação 2).

Para aceder a um registo com chave k, começa-se por gerar $f_l(k)$; se $f_l(k) < s$, então o contentor do registo já foi subdividido e $f_{l+1}(k)$ fornece o índice pretendido. Quando $s = 0$, ainda nenhum contentor foi subdividido, para o nível de subdivisão l actual, pelo que é suficiente utilizar f_l. Quando $s = 2^l$, então todos os contentores do nível l foram subdivididos, havendo agora o dobro dos contentores que havia quando se iniciou o nível l; nestas condições, progride-se para o próximo nível de subdivisão (*i.e.*, $l = l+1$ e $s = 0$).

B.4.3.2 Exemplo Ilustrativo

A figura B.3 demonstra a aplicação de Hashing Linear com base no mecanismo e notação introduzidos na secção anterior. O exemplo em causa não tem relação com os explorados na demonstração dos outros esquemas de hashing dinâmico. Adicionalmente, neste exemplo, a indexação dos contentores baseia-se no sufixo de l bits do hash, em vez do prefixo.

A figura representa uma série de cenários encadeados (de a) a h)), partilhando algumas definições e convenções: 1) a função de hash (f) produz hashes de 5 bits ($\mathcal{L}_f = 5$), 2) l e s aparecem representados para cada cenário, 3) os contentores são representados a cinzento e a sua designação (b_j) recorre a índices expressos em binário, 4) a sequência de bits correspondente ao índice de um contentor determina quais os registos nele armazenáveis (apenas os registos cujo hash da chave contém, na parte direita, aquela sequência de bits), 5) a capacidade dos contentores é agora limitada a dois registos, pelo que qualquer excesso tem de ser acomodado por contentores de sobrecarga, representados sem cor de fundo, 6) assume-se que a subdivisão de um contentor só ocorre se o mesmo estiver em sobrecarga[11].

O ponto de partida do exemplo é dado pelo cenário a), com apenas um contentor b, denotado sem índice; este contentor acomodará quaisquer registos (não sendo ainda necessário tomar em consideração o hash das suas chaves), enquanto a sua capacidade não se esgotar.

[11] Ou seja, se já tiver havido necessidade de recorrer a um contentor de sobrecarga.

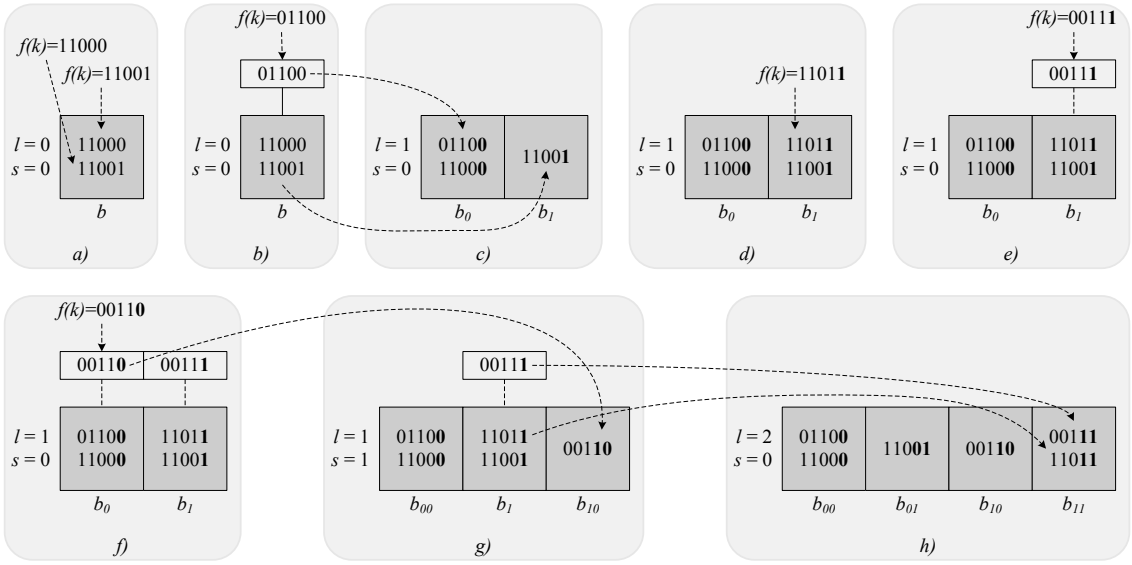


Figura B.3: Exemplo ilustrativo da aplicação de Hashing Linear.

No cenário b), a inserção de um novo registo, associado ao hash 01100, esgota a capacidade do contentor b e obriga à utilização (temporária) de um contentor de sobrecarga (*overflow bucket*); como $s = 0$, o índice implícito de b é 0 (já que é o único contentor regular) e a carga máxima de b foi ultrapassada, então efectua-se a subdivisão de b, a qual inclui: 1) renomeação de b para b_0, 2) criação do novo contentor b_1, 3) repartição dos registos de b_0 e do seu contentor de sobrecarga, com b_1, baseada no sufixo de um ($l + 1 = 1$) bit dos hashes associados, 4) incremento de s, que passa a 1. Após a subdivisão, e dado que todos os contentores do nível $l = 0$ foram subdivididos, incrementa-se l, que passa a 1, e re-inicializa-se s, que passa a 0. A configuração resultante é representada pelo cenário c).

No cenário d), o contentor b_1 esgota a sua capacidade, após a inserção de um registo associado ao hash 11011 e, no cenário e), é inclusivamente necessário criar um contentor de sobrecarga, para acomodar um registo associado ao hash 00111. Todavia, dado que $s = 0$, o contentor b_1 não se pode ainda subdividir, sendo obrigado a aguardar a sua vez.

No cenário f), a inserção de um registo associado ao hash 00110 coloca b_0 em sobrecarga, despoletando a sua subdivisão, dado que $s = 0$; dessa subdivisão resulta a configuração dada pelo cenário g), após: 1) renomeação de b_0 para b_{00}, 2) criação do novo contentor b_{10}, 3) repartição dos registos de b_{00} e do seu contentor de sobrecarga, com b_{10}, baseada no sufixo de dois ($l + 1 = 2$) bits dos hashes associados, 4) incremento de s, que passa a 1.

O incremente de s para 1 faz com que, no cenário g), haja condições para a subdivisão de b_1, em sobrecarga desde o cenário e), e aguardando a sua vez de subdivisão desde então. Em consequência da subdivisão de b_1 surge o cenário h), após: 1) renomeação de b_1 para b_{01}, 2) criação do novo contentor b_{11}, 3) repartição dos registos de b_{01} e do seu contentor de sobrecarga, com b_{11}, baseada no sufixo de dois ($l + 1 = 2$) bits dos hashes associados, 4) incremento de s, que passa a 2 (sendo agora $s = 2^l = 2^1$). Uma vez que todos os contentores do nível $l = 1$ foram subdivididos, incrementa-se l para 2, e repõe-se s a 0.

Apêndice C

Conceitos Básicos de Grafos

C.1 Definição de Grafo

Um grafo $G(V, A)$ é uma estrutura matemática definida por i) um conjunto V de *vértices* (ou *nodos*), e por ii) um conjunto A de *arestas* (ou *arcos*). A notação $V(G)$ e $A(G)$ permite referenciar vértices e arestas de diferentes grafos G, sem qualquer risco de ambiguidade.

$|V(G)|$ denota o número de vértices do grafo $G(V, A)$. Com base nas convenções da Teoria de Conjuntos, seguidas na dissertação, $\#V(G)$ é uma notação válida alternativa. Correspondentemente, $|A(G)|$ ou $\#A(G)$ denotam o número de arestas de $G(V, A)$.

Cada vértice $v \in V(G)$ é um identificador único. Uma aresta $a \in A(G)$ é um par de vértices (v', v'') com $v', v'' \in V(G)$. Dada uma aresta $a = (v', v'')$ diz-se que v' e v'' são *incidentes* a a; reciprocamente, diz-se também que a é *incidente* aos vértices v' e v''.

Dois vértices são *adjacentes* se incidem sobre a mesma aresta, *i.e.*, se existe uma aresta que os liga; reciprocamente, duas arestas são *adjacentes* se incidem sobre um vértice comum.

O conjunto dos *vizinhos* de um vértice v é composto por todos os vértices adjacentes a v.

C.2 Grafos Simples

Um *laço* é uma aresta em que as terminações são dadas pelo mesmo vértice. Um grafo é *simples* se não tem laços e existe, no máximo, uma aresta entre quaisquer dois vértices.

C.3 Grafos Regulares

Um grafo é *k-regular* quando todos os seus vértices têm grau k (ou seja, têm k vizinhos).

O *grau de um vértice* é o número de vértices seus vizinhos ou, equivalentemente, o número de arestas incidentes ao vértice[1]. O *grau de um grafo* é o maior grau dos seus vértices.

[1] Um laço conta duas vezes para o grau do vértice que o inicia e termina.

C.4 Grafos Conexos

Um grafo é *conexo* se for possível definir um *caminho* entre qualquer par dos seus vértices.

Um *caminho* é uma sequência de vértices em que, para cada vértice, existe uma aresta para o vértice seguinte. Um *caminho simples* é aquele em que nenhum dos vértices do caminho se repete. O *comprimento do caminho* é o número de arestas que o caminho usa.

A *distância* entre dois vértices corresponde ao comprimento do caminho mais curto entre eles. A *distância* pode ser diferente da que separa os dois vértices num espaço Euclidiano.

O *diâmetro* de um grafo é a máxima distância entre qualquer par dos seus vértices.

C.4.1 Distâncias Médias

A distância média de um vértice $v \in V(G)$ a todos os outros é dada pela expressão:

$$\overline{d}(v, V(G)) = \frac{\sum_{v' \in V(G)} d(v, v')}{\#V(G)} \tag{C.1}$$

Por vezes, não se considera neste cálculo a distância de um vértice a ele próprio:

$$\overline{d}(v, V(G)) = \frac{\sum_{v' \in V(G) \setminus \{v\}} d(v, v')}{\#V(G) - 1} \tag{C.2}$$

A média dos valores $\overline{d}(v, V(G))$ define a "distância média entre qualquer par de vértices":

$$\overline{d}(G) = \frac{\sum_{v \in V} \overline{d}(v, V)}{\#V} \tag{C.3}$$

A expressão anterior é genérica, comportando o cálculo exaustivo de todos os valores $\overline{d}(v, V(G))$. Para muitos grafos, $\overline{d}(G)$ é conhecida, ou então conhecem-se aproximações.

C.5 Grafos Direccionados ou Orientados (Digrafos)

Quando as arestas têm uma *direcção* associada, o grafo diz-se *direccionado*, chamando-se de *digrafo*. Neste contexto, o termo *arco* é usado preferentemente, em vez do termo *aresta*.

Dado um arco (v', v''), o seu vértice de *origem* é v' e o seu vértice de *destino* é v''. Diz-se ainda que v'' é *sucessor* de v' ou, equivalentemente, que v' é *antecessor/predecessor* de v''.

O *grau de um vértice* é agora dado pela soma dos seus *grau de partida* e *grau de chegada*. O *grau de partida* (*chegada*) de v' é o número de arcos que parte de (chegam a) v'. O *grau de partida* (*chegada*) de um *grafo* é o maior *grau de partida* (*chegada*) dos seus vértices.

Um digrafo é *k-regular* se *grau de partida*=*grau de chegada*=k, para todos os seus vértices.

Num digrafo, a *distância* de v' a v'' não é necessariamente igual à *distância* de v'' a v'.

Apêndice D

Demonstrações de Propriedades

D.1 Demonstração da Propriedade (4.28)

Seja $h_{|\mathcal{L}|}$ uma entrada de nível $\mathcal{L}$, na base 10. Em resultado da aplicação da fórmula 4.20:

$$\begin{aligned} desc(h_{|\mathcal{L}|}, 0) &= h \\ desc(h_{|\mathcal{L}|}, 1) &= h + 2^{\mathcal{L}} \end{aligned}$$

Para cada descendente de $h_{|\mathcal{L}|}$, no nível $\mathcal{L}+1$, é então possível determinar as sucessoras, no grafo $G_B(\mathcal{L}+1)$, aplicando a fórmula 4.8:

$$\begin{array}{lll} suc[desc(h_{|\mathcal{L}|}, 0), 0] & & = 2h \ mod \ 2^{\mathcal{L}+1} \\ suc[desc(h_{|\mathcal{L}|}, 0), 1] & & = (2h+1) \ mod \ 2^{\mathcal{L}+1} \\ suc[desc(h_{|\mathcal{L}|}, 1), 0] & = [2(h+2^{\mathcal{L}})] \ mod \ 2^{\mathcal{L}+1} & = (2h+2^{\mathcal{L}+1}) \ mod \ 2^{\mathcal{L}+1} \\ suc[desc(h_{|\mathcal{L}|}, 1), 1] & = [2(h+2^{\mathcal{L}})+1] \ mod \ 2^{\mathcal{L}+1} & = [(2h+1)+2^{\mathcal{L}+1}] \ mod \ 2^{\mathcal{L}+1} \end{array}$$

Ora, uma vez que, de uma forma genérica, "$(x + k.y) \ mod \ y = x \ mod \ y$", então deduz-se

$$suc[desc(h_{|\mathcal{L}|}, 1), 0]) = suc[desc(h_{|\mathcal{L}|}, 0), 0] = 2h \ mod \ 2^{\mathcal{L}+1} \quad \text{(D.1)}$$

$$suc[desc(h_{|\mathcal{L}|}, 1), 1] = suc[desc(h_{|\mathcal{L}|}, 0), 1] = (2h+1) \ mod \ 2^{\mathcal{L}+1} \quad \text{(D.2)}$$

Definidos os "sucessores dos descendentes de h" (podendo-se constatar que esses sucessores são comuns aos descendentes), é preciso ainda demonstrar que "os sucessores dos descendentes de h" são um subconjunto dos "descendentes dos sucessores de h". Assim, começamos por recordar que, pela aplicação da equivalência 4.8, as sucessoras de $h_{|\mathcal{L}|}$ são

$$
\begin{aligned}
suc(h_{|\mathcal{L}|}, 0) &= 2h \ mod \ 2^{\mathcal{L}} \\
suc(h_{|\mathcal{L}|}, 1) &= (2h+1) \ mod \ 2^{\mathcal{L}}
\end{aligned}
$$

Depois, definimos as descendentes destas sucessoras, recorrendo novamente à fórmula 4.20:

$$
\begin{aligned}
desc[suc(h_{|\mathcal{L}|}, 0), 0] &= 2h \ mod \ 2^{\mathcal{L}} && \text{(D.3)} \\
desc[suc(h_{|\mathcal{L}|}, 0), 1] &= (2h \ mod \ 2^{\mathcal{L}}) + 2^{\mathcal{L}} && \text{(D.4)} \\
desc[suc(h_{|\mathcal{L}|}, 1), 0] &= (2h+1) \ mod \ 2^{\mathcal{L}} && \text{(D.5)} \\
desc[suc(h_{|\mathcal{L}|}, 1), 1] &= [(2h+1) \ mod \ 2^{\mathcal{L}}] + 2^{\mathcal{L}} && \text{(D.6)}
\end{aligned}
$$

Ora, tendo em conta que h é uma entrada de nível $\mathcal{L}$, ou seja, $h \in \{0, 1, ..., 2^{\mathcal{L}} - 1\}$, então

$$
2h \ mod \ 2^{\mathcal{L}+1} \ \text{(D.1)} = \begin{cases} 2h \ mod \ 2^{\mathcal{L}} & \text{(D.3)} \quad \textbf{se} \quad 2h \in \{0, ..., 2^{\mathcal{L}} - 1\} \\ (2h \ mod \ 2^{\mathcal{L}}) + 2^{\mathcal{L}} & \text{(D.4)} \quad \textbf{se} \quad 2h \in \{2^{\mathcal{L}}, ..., 2^{\mathcal{L}+1} - 1\} \end{cases}
$$

$$
(2h+1) \ mod \ 2^{\mathcal{L}+1} \ \text{(D.2)} = \\
\begin{cases} (2h+1) \ mod \ 2^{\mathcal{L}} & \text{(D.5)} \quad \textbf{se} \quad (2h+1) \in \{0, ..., 2^{\mathcal{L}} - 1\} \\ \left[(2h+1) \ mod \ 2^{\mathcal{L}}\right] + 2^{\mathcal{L}} & \text{(D.6)} \quad \textbf{se} \quad (2h+1) \in \{2^{\mathcal{L}}, ..., 2^{\mathcal{L}+1} - 1\} \end{cases}
$$

Dito de outra forma, $\text{(D.1)} \in \{\text{(D.3)}, \text{(D.4)}\}$, assim como $\text{(D.2)} \in \{\text{(D.5)}, \text{(D.6)}\}$, donde

$$
\{\text{(D.1)}, \text{(D.2)}\} \subseteq \{\text{(D.3)}, \text{(D.4)}, \text{(D.5)}, \text{(D.6)}\}
$$

o que, em linguagem informal quer dizer, precisamente que "os sucessores dos descendentes" são um subconjunto dos "descendentes dos sucessores", como queríamos demonstrar.

D.2 Demonstração da Propriedade (4.29)

Seja $h_{|\mathcal{L}|}$ uma entrada de nível $\mathcal{L}$, na base 10. Em resultado da aplicação da fórmula 4.25:

$$\begin{aligned} desc(h_{|\mathcal{L}|}, 0) &= 2h \\ desc(h_{|\mathcal{L}|}, 1) &= 2h + 1 \end{aligned}$$

Para cada descendente de $h_{|\mathcal{L}|}$, no nível $\mathcal{L}$+1, o conjunto das sucessoras, no grafo $G_C(\mathcal{L}+1)$, é determinado pela fórmula 4.12 como:

$$\begin{aligned} Suc\left[desc(h_{|\mathcal{L}|}, 0)\right] &= \{(2h + 2^0)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad (2h + 2^1)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad \ldots \\ &\quad (2h + 2^{\mathcal{L}})\ mod\ 2^{\mathcal{L}+1}\} \end{aligned}$$

$$\begin{aligned} Suc\left[desc(h_{|\mathcal{L}|}, 1)\right] &= \{((2h + 1) + 2^0)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad ((2h + 1) + 2^1)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad \ldots \\ &\quad ((2h + 1) + 2^{\mathcal{L}})\ mod\ 2^{\mathcal{L}+1}\} \end{aligned}$$

Sendo $Suc(Desc(h)) = Suc\left[desc(h_{|\mathcal{L}|}, 0)\right] \cup Suc\left[desc(h_{|\mathcal{L}|}, 1)\right]$ então, eliminando repetições

$$\begin{aligned} Suc(Desc(h)) &= \{(2h + 1)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad (2h + 2^1)\ mod\ 2^{\mathcal{L}+1}, ((2h + 1) + 2^1)\ mod\ 2^{\mathcal{L}+1} \\ &\quad (2h + 2^2)\ mod\ 2^{\mathcal{L}+1}, ((2h + 1) + 2^2)\ mod\ 2^{\mathcal{L}+1} \\ &\quad \ldots \\ &\quad (2h + 2^{\mathcal{L}})\ mod\ 2^{\mathcal{L}+1}, ((2h + 1) + 2^{\mathcal{L}})\ mod\ 2^{\mathcal{L}+1}\} \\ &= \{desc(h_{|\mathcal{L}|}, 1), \\ &\quad (2h + 2^1)\ mod\ 2^{\mathcal{L}+1}, ((2h + 2^1) + 1)\ mod\ 2^{\mathcal{L}+1} \\ &\quad (2h + 2^2)\ mod\ 2^{\mathcal{L}+1}, ((2h + 2^2) + 1)\ mod\ 2^{\mathcal{L}+1} \\ &\quad \ldots \\ &\quad (2h + 2^{\mathcal{L}})\ mod\ 2^{\mathcal{L}+1}, ((2h + 2^{\mathcal{L}}) + 1)\ mod\ 2^{\mathcal{L}+1}\} \end{aligned}$$

, formulação que permite concluir que um dos "sucessores dos descendentes de h" coincide com um dos "descendentes de h" (o descendente $desc(h_{|\mathcal{L}|}, 1)$) e que o resto dos

"descendentes dos sucessores de h" se organizam em pares, separados de uma unidade. Para se demonstrar (4.29), é necessário determinar os "descendentes dos sucessores de h", $Desc(Suc(h))$; para o efeito, comecemos por exprimir $Suc(h)$, recorrendo à fórmula 4.12:

$$\begin{aligned} Suc(h) &= \{(h+2^0)\ mod\ 2^{\mathcal{L}}, \\ &\quad (h+2^1)\ mod\ 2^{\mathcal{L}}, \\ &\quad \ldots \\ &\quad (h+2^{\mathcal{L}-1})\ mod\ 2^{\mathcal{L}}\} \end{aligned}$$

Donde, os "descendentes dos sucessores de h", recorrendo novamente à fórmula 4.25, são:

$$\begin{aligned} Desc(Suc(h)) &= \{2(h+2^0)\ mod\ 2^{\mathcal{L}+1}, (2(h+2^0)+1)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad 2(h+2^1)\ mod\ 2^{\mathcal{L}+1}, (2(h+2^1)+1)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad \ldots \\ &\quad 2(h+2^{\mathcal{L}-1})\ mod\ 2^{\mathcal{L}+1}, (2(h+2^{\mathcal{L}-1})+1)\ mod\ 2^{\mathcal{L}+1}\} \\ &= \{(2h+2^1)\ mod\ 2^{\mathcal{L}+1}, ((2h+1)+2^1)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad (2h+2^2)\ mod\ 2^{\mathcal{L}+1}, ((2h+1)+2^2)\ mod\ 2^{\mathcal{L}+1}, \\ &\quad \ldots \\ &\quad (2h+2^{\mathcal{L}-1})\ mod\ 2^{\mathcal{L}+1}, ((2h+1)+2^{\mathcal{L}})\ mod\ 2^{\mathcal{L}+1}\} \end{aligned}$$

Ora, comparando a formulação de $Desc(Suc(h))$ com a de $Suc(Desc(h))$, conclui-se que

$$Desc(Suc(h)) = Suc(Desc(h)) \setminus \{desc(h_{|\mathcal{L}|}, 1)\} \tag{D.7}$$

ou, equivalentemente

$$Suc(Desc(h)) = Desc(Suc(h)) \cup \{desc(h_{|\mathcal{L}|}, 1)\} \tag{D.8}$$

o que implica (4.29), dado que $\{desc(h_{|\mathcal{L}|}, 1)\} \subseteq Desc(h)$ $\therefore$

Apêndice E

Exemplos de Código

E.1 Utilização da Classe `cDomusUsrProxy`

O exemplo seguinte ilustra situações encadeadas, em que se recorre a métodos da classe `cDomusUsrProxy` (rever secção 7.8.1) para a gestão de um cluster Domus e seus serviços:

```
#!/usr/bin/python
import sys
from domus_libusr import *
from domus_libsys import *

#############################################################################
# criar um proxy de um cluster Domus (proxy cDomus)
_CDOMUS_USRPROXY=cDomusUsrProxy(cluster_id="myClusterDomus", \
                                cluster_supervisor_id=("192.168.96.254",7571))

# criar o cluster Domus referenciado pelo proxy (i.e., arrancar o supervisor)
_ret=_CDOMUS_USRPROXY.cluster_create()
if _ret[0]!=0: print STRERROR[ret[0]], STRERROR[ret[1]]; sys.exit(0)

# acrescentar um serviço ao cluster Domus (i.e., arrancar o serviço)
_ret=_CDOMUS_USRPROXY.service_add(srvAddress=("192.168.96.1",8379))
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# quebrar a associação do proxy cDomus ao cluster Domus (que continua activo)
_CDOMUS_USRPROXY.cluster_close()

#############################################################################
# criar um novo proxy cDomus, para o cluster Domus "myClusterDomus"
_CDOMUS_USRPROXY=cDomusUsrProxy(cluster_id="myClusterDomus", \
                                cluster_supervisor_id=("192.168.96.254",7571))

# verificar que o cluster "myClusterDomus" existe e está no estado activo
_ret=_CDOMUS_USRPROXY.cluster_ping()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)
```

```
# associar o novo proxy cDomus ao cluster "myClusterDomus"
_ret=_CDOMUS_USRPROXY.cluster_open()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# verificar se um certo serviço Domus existe e está no estado activo
_ret=_CDOMUS_USRPROXY.service_ping(srvAddress="192.168.96.1")
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# acrescentar um novo serviço ao cluster Domus (i.e., arrancar esse serviço)
_ret=_CDOMUS_USRPROXY.service_add(srvAddress=("192.168.96.2",8379))
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# desactivar o cluster Domus, que passará ao estado inactivo
_ret=_CDOMUS_USRPROXY.cluster_shutdown()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

##############################################################################
# criar um novo proxy cDomus, para o cluster Domus "myClusterDomus"
_CDOMUS_USRPROXY=cDomusUsrProxy(cluster_id="myClusterDomus", \
                                cluster_supervisor_id=("192.168.96.254",7571))

# reactivar o cluster Domus
_ret=_CDOMUS_USRPROXY.cluster_restart()
if _ret[0]!=0: print STRERROR[ret[0]], STRERROR[ret[1]]; sys.exit(0)

# obter a lista dos serviços Domus do cluster Domus
_ret=_CDOMUS_USRPROXY.cluster_attget(attName="attr_cluster_services")
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)
else: _attValue=_ret[1]; print _attValue
```

E.2 Utilização da Classe `dDomusUsrProxy`

O exemplo seguinte ilustra outro conjunto de situações encadeadas, aplicadas ao cluster Domus criado no exemplo anterior (que não chega a ser destruído), e em que se recorre a métodos da classe `dDomusUsrProxy` (rever secção 7.8.1) com o objectivo de operar DHTs:

```
# . . . (continuação do exemplo anterior)
##############################################################################
# criar um proxy de uma DHT Domus (proxy dDomus) associado a um proxy cDomus
_DDOMUS_USRPROXY=dDomusUsrProxy(dht_id="myDhtDomus", \
                                cluster_proxy=_CDOMUS_USRPROXY)

# criar a DHT referenciada pelo proxy dDomus
_ret=_DDOMUS_USRPROXY.dht_create()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# quebrar a associação do proxy dDomus à DHT (que continuará no estado activo)
_DDOMUS_USRPROXY.dht_close()
```

```
#############################################################################
# criar um novo proxy dDomus, para a DHT "myDhtDomus1"
_DDOMUS_USRPROXY=dDomusUsrProxy(dht_id="myDhtDomus1", \
                                cluster_proxy=_CDOMUS_USRPROXY)

# verificar que a DHT "myDhtDomus1" existe e está no estado activo
_ret=_DDOMUS_USRPROXY.dht_ping()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# associar o novo proxy dDomus à DHT "myDhtDomus1"
_ret=_DDOMUS_USRPROXY.dht_open()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# obter a tabela de distribuição do armazenamento da DHT "myDhtDomus1"
_ret=_DDOMUS_USRPROXY.dht_attget(attName="attr_dht_tda")
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)
else: _attValue=_ret[1]; print _attValue

# desactivar a DHT "myDhtDomus1" (que passará ao estado inactivo)
_ret=_DDOMUS_USRPROXY.dht_shutdown()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

#############################################################################
# criar um novo proxy dDomus, para uma outra DHT, do mesmo cluster Domus
_DDOMUS_USRPROXY=dDomusUsrProxy(dht_id="myDhtDomus2", \
                                cluster_proxy=_CDOMUS_USRPROXY)

# definir explicitamente o meio de armazenamento da DHT "myDhtDomus2"
_DDOMUS_USRPROXY.dht_attset(attName="attr_dht_ma", attValue="disco")

# criar a DHT "myDhtDomus2" referenciada pelo proxy dDomus
_ret=_DDOMUS_USRPROXY.dht_create()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# inserir (sobrepondo, se existir) um registo na DHT "myDhtDomus2";
# ( _key e _data são quaisquer objectos Python serializáveis )
_ret=_DDOMUS_USRPROXY.dht_record_put(_key = ..., _data = ...)
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# recuperar um registo da DHT "myDhtDomus2", indexado por certa chave _key
_ret=_DDOMUS_USRPROXY.dht_record_get(_key = ...)
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)
else: _data=ret[1]; print _data

# remover um registo da DHT "myDhtDomus2", indexado por certa chave _key
_ret=_DDOMUS_USRPROXY.dht_record_del(_key = ...)
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# verificar a existência de um presumível registo na DHT "myDhtDomus2"
_ret=_DDOMUS_USRPROXY.dht_record_probe(_key = ...)
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)
```

```
# determinar a localização de um presumível registo da DHT "myDhtDomus2"
_ret=_DDOMUS_USRPROXY.dht_record_lookup(_key = ...)
if _ret[0]!=0: print STRERROR[ret[0]], STRERROR[ret[1]]; sys.exit(0)
else:
   _partition=ret[1][0]; _routing_service=ret[1][1]; _storage_service=ret[1][2]
   print _partition, _routing_service, _storage_service

# suspender o cluster Domus (o que implica desactivar a DHT "myDhtDomus2")
_ret=_CDOMUS_USRPROXY.cluster_shutdown()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

###############################################################################
# criar um novo proxy cDomus, para o cluster Domus "myClusterDomus"
_CDOMUS_USRPROXY=cDomusUsrProxy(cluster_id="myClusterDomus", \
                                cluster_supervisor_id=("192.168.96.254",7571))

# reactivar o cluster "myClusterDomus"
#  - a DHT "myDhtDomus1" continuará inactiva
#  - a DHT "myDhtDomus2" será reactivada
_ret=_CDOMUS_USRPROXY.cluster_restart()
if _ret[0]!=0: print STRERROR[ret[0]], STRERROR[ret[1]]; sys.exit(0)

# criar um novo proxy dDomus, para a DHT "myDhtDomus1"
_DDOMUS_USRPROXY=dDomusUsrProxy(dht_id="myDhtDomus1", \
                                cluster_proxy=_CDOMUS_USRPROXY)

# verificar que a DHT "myDhtDomus1", mesmo inactiva, pertence ao cluster Domus
_ret=_DDOMUS_USRPROXY.dht_probe()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# neste ponto, seria possível destruir a DHT "myDhtDomus1" sem a reactivar
#_ret=_DDOMUS_USRPROXY.dht_destroy()
#if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# reactivar a DHT "myDhtDomus1"
_ret=_DDOMUS_USRPROXY.dht_restart()
if _ret[0]!=0: print STRERROR[ret[0]]; sys.exit(0)

# determinar a localização de um hash na DHT "myDhtDomus1"
_ret=_DDOMUS_USRPROXY.dht_lookup(_hash = 0x1234ABCD)
if _ret[0]!=0: print STRERROR[ret[0]], STRERROR[ret[1]]; sys.exit(0)
else:
    _partition=ret[0]; _routing_service=ret[1]; _storage_service=ret[1]
    print _partition, _routing_service, _storage_service

# destruir o cluster Domus (destruindo todas as DHTs, activas e inactivas)
_ret=_CDOMUS_USRPROXY.cluster_destroy()
if _ret[0]!=0: print STRERROR[ret[0]], STRERROR[ret[1]]; sys.exit(0)
```