Universidade do Minho
Escola de Engenharia
Departamento de Informática

Desconexão e Replicação em Aplicações Distribuídas

por

Albano Agostinho Gomes Alves

Dissertação apresentada à Universidade do Minho para
a obtenção do grau de Mestre em Informática

Orientador:
Prof. Francisco Soares Moura

Guimarães
Setembro de 1997

Aos meus pais e à minha irmã, pelo apoio que sempre me deram.

Agradecimentos

Ao meu orientador, Prof. Francisco Soares Moura, pelo tema de investigação proporcionado e pela constante preocupação na melhoria da qualidade desta dissertação.

A todos os colegas do Grupo de Sistemas Distribuídos, e em especial ao Carlos, pela ajuda e pelo bom ambiente de trabalho. Aos restantes colegas de mestrado, especialmente ao "grupinho" do Lab. 1, pelos bons momentos passados.

À Escola Superior de Tecnologia e de Gestão do Instituto Politécnico de Bragança, pelas facilidades concedidas enquanto elaborava este texto.

Ao Filipe, pelo incentivo, sem o qual provavelmente não me teria inscrito em mestrado.

À Guida, por todo o apoio e compreensão, e também pela ajuda nas correcções do português.

Resumo

O suporte à desconexão e a replicação são técnicas já bem conhecidas para aumentar a disponibilidade e a fiabilidade dos sistemas distribuídos. No entanto, até à data a prática comum tem sido a concretização destas ideias para casos pontuais, ou seja, cada programador implementa algoritmos deste tipo à sua maneira e de acordo com as características da aplicação em causa.

Nesta dissertação são apresentados uma metodologia para dotar uma aplicação já existente, desenvolvida segundo o modelo Cliente-Servidor, de suporte à desconexão e um mecanismo para replicação de servidores. O suporte à desconexão é conseguido com a introdução de um agente entre o cliente e o servidor. O agente é obtido por um tradutor que processa a especificação da interface anotada com primitivas de desconexão. Estas primitivas constituem uma extensão à linguagem de especificação de interfaces. A replicação de servidores é obtida com a intercepção dos pedidos (invocações remotas) dos clientes e seu posterior reenvio através de um serviço de comunicação em grupo.

O resultado deste trabalho é portanto o modelo Cliente-Agente-Servidores, para aplicação genérica a soluções desenvolvidas sem preocupações notórias no que respeita à disponibilidade e fiabilidade.

Abstract

Disconnected operation and replication are well known techniques to increase the availability and reliability of distributed systems. Nevertheless, until now the common practice has been the application of those ideas to specific cases, which means that each programmer implements replication and disconnection algorithms on his own way and according to the characteristics of the application under development.

This thesis extends the Client-Server model in order to handle disconnections and replication of servers. The support to disconnection is achieved with the introduction of an agent between the client and the server. The agent is generated by a parser that processes the interface specification annotated with disconnection primitives. These primitives form an extension to the usual interface specification language. The replication of servers is achieved by intercepting remote invocations and submitting these invocations to a group communication service.

The result of this work is a new model – Client-Agent-Servers – for generic use on applications developed without special cares in respect to availability and reliability.

Conteúdo

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Apresentação 1

1.2 Estrutura da Dissertação 3

1.3 Terminologia e Convenções 4

2 Enquadramento **5**

2.1 Programação de Sistemas Distribuídos 5

2.1.1 Introdução 6

2.1.2 Mecanismos Clássicos 7

2.1.3 Abstracções de Alto Nível 9

2.1.4 Sistemas Distribuídos Actuais 14

2.2 CORBA 16

2.2.1 Núcleo ORB 17

2.2.2 Linguagem para Definição de Interfaces 17

2.2.3 Interface de Invocação Dinâmica 19

2.2.4 Repositório de Interfaces 20

2.2.5 Adaptadores de Objectos 20

2.3 Suporte à Desconexão 21

2.3.1 O Problema da Desconexão 21

2.3.2 Modelo Cliente-Agente-Servidor 23

2.4 Replicação 24
2.4.1 Correcção dos Dados 25
2.4.2 Estratégias de Replicação 26
2.4.3 Lotus Notes 27
2.5 Comunicação em Grupo 28
2.5.1 Princípios Gerais 28
2.5.2 Uma Concretização: o GTS 32

3 Construção de Aplicações em ILU **37**
3.1 Sistema ILU 37
3.1.1 Conceitos Gerais 37
3.1.2 Definição de Interfaces 41
3.1.3 Utilização do ILU em C++ 44
3.2 Escolha do ILU 47
3.2.1 Motivos para Eleição 47
3.3 Exemplo de Utilização 48
3.3.1 Especificação da Interface 48
3.3.2 Construção do Servidor 51
3.3.3 Construção do Cliente 54

4 Anotações de Desconexão **57**
4.1 Modelo de Agentes 57
4.1.1 Funções do Agente 57
4.1.2 Localização do Agente 58
4.1.3 Integração em Aplicações já Existentes 60
4.2 Extensão da ISL do ILU 61
4.2.1 Especificação de Agentes 62
4.2.2 Anotações Propostas 63
4.3 Construção de Agentes 70
4.3.1 Processo de Obtenção de Agentes 71
4.3.2 Implementação do Tradutor 74

4.3.3 Editor de Anotações 75

5 Replicação de Objectos **77**

5.1 Modelo Proposto 78

5.1.1 Invocações Múltiplas 78

5.1.2 Difusão Selectiva de Pedidos ILU 81

5.2 Concretização: ILU Replicado sobre GTS 85

5.2.1 Intercepção de Pedidos 86

5.2.2 Serviço de Directoria 88

5.2.3 Gestor de Referências de Objectos ILU 91

5.2.4 Funcionamento em Larga-Escala 93

5.3 Difusão de Pedidos ILU 94

5.3.1 Adaptação do Sistema de Invocações Múltiplas 95

5.3.2 Movimentação de Informação 96

5.3.3 Integração no Gestor de Referências 97

6 Exemplo de Aplicação **99**

6.1 Reequacionamento do Problema 99

6.2 Replicação de Servidores 101

6.2.1 Instalação de Servidores 101

6.2.2 Criação de Grupos de Objectos 102

6.2.3 Instalação de Clientes 104

6.3 Operação Isolada 106

6.3.1 Anotação da Interface 106

6.3.2 Construção do Agente 110

6.3.3 Instalação de Agentes 114

6.4 Análise de Desempenho 115

7 Conclusões **121**

Bibliografia **129**

Lista de Figuras

3.1 Estrutura de um programa desenvolvido com base no ILU. 38
3.2 Diagrama Entidade-Relação para uma aplicação de facturação. 49

4.1 Modelo Cliente-Agente-Servidor. 59
4.2 Construção de agentes. 71
4.3 Editor de Anotações. 76

5.1 Modelo Cliente-Agente-Servidores. 83
5.2 ILU sobre GTS. 85
5.3 Interceptor. 86
5.4 Gestor de referências. 92
5.5 Serviço de directório distribuído. 94
5.6 Difusão de pedidos ILU. 95

6.1 Facturação distribuída. 100
6.2 Troca de mensagens via GTS. 115
6.3 Invocações ILU com e sem replicação. 116

Lista de Tabelas

4.1 Primitivas de desconexão e sua compatibilidade. 69

Capítulo 1

Introdução

1.1 Apresentação

Os sistemas distribuídos são hoje uma realidade, sendo indubitável a sua importância nos sistemas informáticos. Ouvimos constantemente a palavra distribuído, e é um facto que os grandes produtores de software estão a colocar no mercado versões distribuídas das suas soluções.

Porém, o desenvolvimento de aplicações deste tipo é um processo bastante complexo. Na verdade, faltam ainda algoritmos e modelos que solucionem totalmente os problemas inerentes à interligação de máquinas, o que coloca os programadores perante uma situação delicada: o código necessário para lidar com os pormenores de comunicação entre as entidades de um sistema distribuído representa uma fatia muito grande no total do código de uma aplicação. Esta situação agravou-se com a crescente utilização de soluções deste tipo, visto que as aplicações são cada vez mais complexas e o número e heterogeneidade dos sistemas envolvidos é cada vez maior. Assim, torna-se urgente disponibilizar mecanismos que facilitem a tarefa dos programadores e até dos utilizadores, por forma a vulgarizar a utilização de sistemas distribuídos.

Uma análise detalhada dos problemas adjacentes à programação de aplicações distribuídas revela que grande parte desses problemas advém da tentativa de alcançar dois objectivos fundamentais: maior disponibilidade[1] e maior fiabilidade[2]. Estes objectivos são cada vez mais importantes devido à crescente relevância dos sistemas distribuídos. De facto, a dependência das organizações em relação a soluções deste tipo implica um grande esforço, por parte dos programadores, para garantir o funcionamento contínuo das aplicações. Este requisito é posto em causa pelos inúmeros problemas que poderão surgir nos diversos

[1] Do inglês *availability*.
[2] Do inglês *reliability*.

componentes, desde a falta de conectividade entre máquinas até à falha de um servidor, por exemplo. Esta situação é agravada com a crescente utilização de sistemas portáteis, que nem sempre podem interactuar com componentes residentes na rede fixa, e com a utilização da Internet, onde não há garantia de comunicação permanente entre as várias partes. Para além desta garantia de disponibilidade, é imprescindível assegurar a correcta operação das aplicações ao longo do tempo, mesmo quando ocorrem falhas irremediáveis em determinadas partes do sistema, ou seja, é necessário tornar as aplicações fiáveis.

O Grupo de Sistemas Distribuídos, no seio do qual esta tese foi desenvolvida, tem vindo a trabalhar em computação nómada[3], mais especificamente no suporte à desconexão, e em replicação, que são técnicas bem conhecidas para garantir maior disponibilidade e maior fiabilidade nas aplicações distribuídas. No entanto, cada abordagem tem sido específica, "a feitio" para o caso em estudo.

Nesta tese acredita-se que o suporte à desconexão e a replicação são necessidades comuns de um grande número de aplicações distribuídas. Deste modo, há todo o interesse em automatizar a introdução destas abordagens em aplicações em desenvolvimento ou até em aplicações já existentes.

A linha orientadora desta tese é portanto a tentativa de oferecer suporte genérico à desconexão e replicação de aplicações distribuídas, sem esquecer que a maioria destas aplicações são desenvolvidas segundo o modelo Cliente-Servidor. O crescente sucesso da computação distribuída orientada ao objecto, na qual o Grupo de Sistemas Distribuídos tem especial interesse, é também um factor de grande importância para o desenvolvimento do sistema pretendido.

Neste trabalho apresenta-se, em primeiro lugar, uma concretização do modelo Cliente-Agente-Servidor, o qual constitui uma extensão ao tradicional modelo Cliente-Servidor, por forma a possibilitar a operação isolada de clientes (suporte à desconexão). O agente tem por finalidade substituir o servidor na impossibilidade de este ser contactado, exportando, portanto, a mesma funcionalidade que o servidor. A concretização proposta baseia-se na extensão de uma linguagem de especificação de interfaces. As anotações de desconexão, que constituem essa extensão, permitem especificar o comportamento dos objectos em caso de desconexão e têm como grande objectivo automatizar o processo de dotar uma aplicação de meios para suporte à desconexão. As anotações de desconexão, juntamente com a restante especificação da interface, são processadas por um tradutor que gera um agente em C++.

Em segundo lugar, é apresentado um modelo para replicação de objectos. Este modelo é baseado na intercepção de pedidos submetidos por um cliente a um servidor e no seu posterior reenvio através de um serviço de comunicação em grupo. A concretização deste modelo

[3]Do inglês *mobile computing*.

permite ainda a difusão[4] de invocações a objectos (entrega de pedidos a um conjunto indeterminado de servidores). Embora possua algumas limitações, a grande vantagem deste sistema de replicação é o facto de não ser necessário introduzir alterações no código de clientes e servidores já existentes, nem sequer ser necessário proceder à sua recompilação.

O resultado deste trabalho é, portanto, o modelo Cliente-Agente-Servidores, que foi concretizado tomando como base uma plataforma de computação distribuída orientada ao objecto que oferece o modelo Cliente-Servidor – o ILU.

1.2 Estrutura da Dissertação

Após a introdução, esta dissertação apresenta um capítulo de enquadramento, onde se expõem algumas metodologias para o desenvolvimento de aplicações distribuídas, para além de se tecerem algumas considerações sobre a inadequação de tais metodologias no contexto dos sistemas distribuídos actuais. Nesta sequência, surge uma sucinta exposição da especificação CORBA, introduzem-se alguns tópicos sobre replicação e caracterizam-se os mecanismos de comunicação em grupo, em especial o sistema de comunicação em grupo utilizado na concretização do sistema de replicação elaborado nesta tese.

Seguindo-se ao enquadramento, surge um capítulo que documenta em detalhe o sistema ILU, bem como a construção de clientes e servidores segundo esta metodologia. Neste capítulo há ainda a preocupação de justificar a escolha do ILU como suporte ao mecanismo de programação que se pretende oferecer.

No capítulo seguinte apresentam-se as anotações de desconexão. Depois de explicada a sintaxe das extensões propostas, assim como a integração de agentes em soluções desenvolvidas com base no ILU, são focados os pormenores da construção e funcionamento do tradutor de anotações de desconexão. São ainda evidenciados alguns aspectos relacionados com a anotação de interfaces, nomeadamente a utilização de um editor de anotações.

Segue-se um capítulo onde é proposto um modelo para replicação de objectos ILU. São também aqui apresentados os detalhes da concretização deste modelo, especialmente os pormenores relativos à interacção com o sistema ILU e à utilização do serviço de comunicação em grupo escolhido, no qual assenta todo o modelo.

No penúltimo capítulo é analisado um exemplo a fim de demonstrar a aplicabilidade e as vantagens das anotações de desconexão, bem como do mecanismo de replicação de objectos. São também apresentados os resultados relativos à análise de desempenho[5] da concretização do modelo Cliente-Agente-Servidores.

[4] Do inglês *broadcast*.
[5] Do inglês *performance*.

Por fim, são revistas algumas das contribuições deste trabalho, tal como possíveis melhoramentos e evoluções do mesmo.

1.3 Terminologia e Convenções

A escrita desta dissertação foi efectuada com a constante preocupação de utilizar terminologia exclusivamente portuguesa, apesar de, em muitos dos conceitos informáticos, serem bastante mais conhecidos os vocábulos estrangeiros (ingleses, na sua maioria). Assim, as traduções de termos informáticos ingleses, quando sujeitas a interpretações duvidosas, são acompanhadas de notas de rodapé aludindo ao termo original. Em determinados casos pontuais, algumas palavras são mantidas na sua forma original, por falta de uma tradução suficientemente esclarecedora e bem aceite. Tais palavras são apresentadas em itálico.

A introdução de siglas é feita pela apresentação da expressão por extenso, com letra maiúscula no início de cada palavra, seguida da sigla entre parêntesis. Uma excepção é feita quando, apesar da tradução para português da expressão em causa, se decide utilizar a sigla derivada do inglês devido à sua vulgarização. Neste caso, não são usadas letras maiúsculas no início de cada palavra e é usada uma nota de rodapé para evidenciar a expressão original (em inglês).

Os exemplos de codificação nas linguagens de programação são apresentados em inglês, o que permite uma melhor integração na sintaxe dessas linguagens invariavelmente de origem inglesa. A interface com o utilizador das diversas ferramentas produzidas neste trabalho é também em inglês, por forma a permitir uma maior divulgação destas. Todo o código ou extracto de código, bem como a invocação de comandos, são apresentados em letra de máquina e com espaçamento fixo.

Capítulo 2

Enquadramento

Neste capítulo faz-se uma apresentação da área onde se insere o modelo Cliente-Agente-Servidores, desenvolvido no âmbito deste trabalho. Começa-se por apresentar alguns tópicos relacionados com a programação de sistemas distribuídos, passando-se então à exposição mais detalhada de soluções adoptadas para a resolução de problemas específicos desta área.

Para além dos conceitos e terminologia necessários à compreensão dos capítulos que se seguem, são focados aspectos que constituem a motivação para o trabalho aqui apresentado. Essencialmente, pretende-se, com este capítulo, realçar a importância da programação de sistemas distribuídos e a necessidade de mecanismos mais flexíveis para o desenvolvimento de soluções deste tipo, tendo em consideração os recentes avanços tecnológicos.

2.1 Programação de Sistemas Distribuídos

O desenvolvimento de aplicações que tirem partido da interligação de múltiplos sistemas, com garantia de funcionamento independentemente do comportamento irregular dos vários componentes, é ainda considerado difícil, mesmo para os programadores mais experientes. Neste sentido, tem-se assistido a um grande esforço por parte dos investigadores, por forma a criar mecanismos que escondam a complexidade da distribuição de programas[1]. Fundamentalmente, procura-se conseguir transparência na distribuição, por forma a reduzir significativamente a barreira entre a programação centralizada e a programação distribuída.

[1]Por distribuição de programas entende-se o desenvolvimento de programas específicos para sistemas distribuídos.

2.1.1 Introdução

Até meados da década de oitenta era prática comum as organizações adquirirem o maior computador que pudessem já que, segundo a lei de Grosch [1], o poder computacional de uma máquina era proporcional ao quadrado do seu preço. Estas máquinas, de custo elevado, operavam de forma isolada, dado que poucas eram as empresas que podiam comprar mais do que uma máquina, com a agravante de praticamente não existirem mecanismos para a sua interligação.

O desenvolvimento de microprocessadores, cada vez mais poderosos e a preços extremamente baixos, juntamente com a invenção de tecnologias de rede de alto débito, veio alterar esta tendência. A partir deste momento torna-se económica e tecnologicamente viável ligar dezenas de microcomputadores através de uma rede local. No conjunto, estas máquinas acabam por custar menos que um sistema de grande porte e possuem uma capacidade de cálculo superior.

Desta forma, começa-se a assistir ao abandono de soluções centralizadas e aparecem os primeiros sistemas distribuídos, aos quais se podem apontar inúmeras vantagens, como por exemplo [1]: melhor relação preço/desempenho, maior disponibilidade, melhor rentabilização de periféricos, partilha mais eficiente e conveniente do software e hardware de cada máquina, concretização mais intuitiva de aplicações inerentemente distribuídas, etc. No entanto, estes sistemas requerem software radicalmente distinto do utilizado nos sistemas centralizados. Deste modo, à imagem dos sistemas operativos tradicionais, que visam esconder a complexidade de uma máquina em particular para além de gerir de forma eficiente os seus recursos [2], torna-se necessário criar mecanismos para facilitar o desenvolvimento de aplicações para sistemas que envolvam várias máquinas interligadas. Idealmente, seria desejável possuir um verdadeiro sistema operativo distribuído, mas devido à elevada complexidade dos problemas que se levantam, a prática comum é ir dotando os sistemas operativos clássicos de facilidades adicionais que permitam tirar partido da interligação dos sistemas.

No desenvolvimento de sistemas distribuídos devem, segundo [3], ser considerados os seguintes aspectos:

- falhas independentes – Devido à independência dos vários computadores interligados, quando um deles deixa de funcionar, os outros podem continuar operacionais. Neste caso, é desejável que o sistema como um todo continue em funcionamento, apesar da falha ocorrida numa das partes.

- comunicação não fiável – Na maior parte dos casos, as ligações entre os computadores não estão confinadas a um ambiente cuidadosamente controlado, pelo que não há garantias do seu funcionamento contínuo. Como tal, a comunicação entre dois sistemas

pode, em determinado momento, não ser possível, havendo ainda a possibilidade de se perder ou corromper parte das mensagens. Em suma, um computador não pode assumir que é capaz de comunicar claramente com outro, mesmo que ambos estejam a funcionar correctamente.

- comunicação insegura – As ligações entre os sistemas podem estar expostas a escutas não autorizadas ou a processos de modificação de mensagens. Esta situação tende a agravar-se com a expansão da Internet.

- comunicação dispendiosa – As ligações existentes entre os computadores de um sistema distribuído possuem uma baixa largura de banda, grande latência[2] e custos de transmissão elevados, quando comparadas com os canais de comunicação disponíveis entre processos em execução numa mesma máquina.

2.1.2 Mecanismos Clássicos

As linguagens de programação actuais foram, na sua maioria, desenvolvidas tendo em vista a programação de sistemas centralizados. Como tal, estas não colocam à disposição do programador instruções ou construtores que facilitem a comunicação entre módulos espalhados por diversas máquinas. Esta falha foi inicialmente colmatada com o aparecimento de uma biblioteca de *sockets*.

Os *sockets*, introduzidos no BSD UNIX [4], são uma abstracção do sistema operativo que permite operações de escrita e leitura sobre um canal de comunicação. Este canal liga duas entidades potencialmente residentes em máquinas distintas, sendo cada entidade identificada pelo endereço IP da máquina juntamente com um porto[3] da mesma. Os portos são necessários para multiplexar todas as mensagens que chegam à máquina, dado que numa máquina podem existir inúmeros processos a comunicar com o exterior, partilhando a mesma infra-estrutura de rede.

Em termos práticos, os *sockets* permitem a troca de sequências de bytes entre dois processos, quando conhecidas as suas localizações (endereço da máquina e porto), através de instruções específicas para envio e recepção desses bytes. Note-se que no código da entidade receptora terá que existir uma aceitação explícita dos dados que lhe são enviados, instrução `receive`, enquanto que o emissor, antes de invocar a função `send`, terá que transformar numa sequência de bytes a informação que deseja enviar.

A invocação de procedimentos remotos (RPC[4]) [5] surge como um grande avanço em relação à programação com *sockets*. Esta tecnologia possibilita que num programa se

[2] Do inglês *latency*.
[3] Do inglês *port*.
[4] Remote Procedure Call

efectuem invocações a procedimentos (ou funções) que serão executados em processos que executam em máquinas remotas. Neste paradigma, o desenvolvimento de aplicações distribuídas rege-se pela decomposição funcional, ou seja, os vários componentes de um sistema distribuído, espalhados por diversas máquinas de uma rede, são vistos como procedimentos ou funções que podem ser activados a partir de qualquer processo.

Sempre que um cliente invoca um procedimento remoto, são enviados ao servidor, para além do identificador do procedimento em causa, os parâmetros da invocação. Após a execução do código correspondente, é devolvido ao cliente o resultado da operação desencadeada. Basicamente, a programação com RPCs permite que num servidor se implementem alguns procedimentos, como se se tratasse de um programa normal, e que um cliente possa invocar esses procedimentos como se estes estivessem codificados localmente.

No desenvolvimento normal de uma aplicação, com RPCs, começa-se por especificar, numa linguagem própria, o conjunto de funções implementadas num determinado servidor, mais propriamente, o nome dessas funções, parâmetros de entrada e resultados. Esta especificação é processada por um tradutor que gera *stubs* para serem utilizados na compilação do servidor e dos clientes.

Os *stubs* contêm todo o código necessário para invocar operações remotas de forma transparente, para além de conterem os mecanismos necessários para activar determinada execução após a recepção de um pedido remoto, com o consequente envio do resultado. Na geração de *stubs* há ainda a considerar os seguintes aspectos [6]:

- heterogeneidade das máquinas – diferentes representações binárias dos vários tipos de dados obrigam à conversão para formatos comuns, usando-se na tecnologia RPC o XDR [7];
- passagem de parâmetros – a passagem de parâmetros por referência, por exemplo, requer um tratamento especial;
- estruturas de dados complexas – a passagem de estruturas de dados com apontadores para outras estruturas (uma lista ligada, por exemplo) obriga à utilização de mecanismos de empacotamento[5] e desempacotamento[6] mais complicados;
- falhas – a invocação de uma operação remota pode falhar independentemente da entidade invocadora.

Dado que os *stubs* podem ser gerados para várias linguagens de programação, pelo simples facto de que se pode desenvolver um tradutor para cada linguagem, os RPCs e o modelo Cliente-Servidor, por eles oferecido, são um mecanismo bastante flexível. De facto, uma

[5]Do inglês *marshalling*.
[6]Do inglês *unmarshalling*.

aplicação desenvolvida para um sistema centralizado pode ser facilmente transformada numa solução distribuída, reutilizando a quase totalidade do código escrito, independentemente da linguagem de programação utilizada.

2.1.3 Abstracções de Alto Nível

O desenvolvimento de aplicações distribuídas tem provado que a computação distribuída oferece realmente benefícios inegáveis, em relação à computação centralizada. No entanto, as ferramentas e as tecnologias convencionais usadas para este propósito não facilitam o suficiente a produção destas aplicações. Na verdade, mecanismos como os *sockets* e os RPCs não oferecem as abstracções desejadas, de modo a elevar a níveis aceitáveis a produtividade de um programador nesta área.

Linguagens para Programação Distribuída

Uma vez que as linguagens tradicionais não oferecem suporte à programação distribuída, a atitude mais vulgar tem sido a criação de bibliotecas de funções que escondam a complexidade e o baixo nível dos mecanismos de comunicação. Porém, estas bibliotecas nem sempre se integram adequadamente e com facilidade nas linguagens em que as aplicações são desenvolvidas. Por exemplo, o mecanismo RPC, disponível sob a forma de uma biblioteca de funções, quando usado num programa escrito em C++, não explora o paradigma da orientação ao objecto.

Assim, além de bibliotecas de funções para suportar a programação distribuída, houve quem propusesse extensões a linguagens já existentes, por forma a obter uma maior integração destas com as facilidades de suporte à programação distribuída. Nesta abordagem são criadas instruções, construtores e mecanismos de controlo que são interpretados directamente pelo compilador da linguagem. Estas extensões permitem converter aplicações dos sistemas centralizados, o que possibilita alguma reutilização de código. Porém, combinar num mesmo programa comportamentos sequenciais e paralelos[7] torna o código do programa difícil de compreender e impede que se imponha o rigor e automação necessários no desenvolvimento de soluções comerciais. No fundo, este modelo de programação não é suficientemente "elegante".

Para além das extensões a linguagens tradicionais, existem ainda algumas linguagens de programação desenvolvidas especificamente para a construção de aplicações em ambientes distribuídos, como é o caso, por exemplo, das linguagens Emerald [8] e ORCA [9]. Em [10] são comparadas algumas das linguagens de programação distribuída mais representativas,

[7] Num sistema distribuído, devido ao funcionamento independente das várias máquinas, os programas apresentam execuções paralelas.

que são baseadas em diferentes paradigmas: troca de mensagens, objectos concorrentes, programação lógica e programação funcional. Estas linguagens, apesar de disponibilizarem modelos de programação mais elaborados e mais fáceis de usar, dado que não estão obrigadas a manter a compatibilidade com construtores de uma linguagem de base, dificultam ainda mais a tarefa dos programadores. Na verdade, obrigam à aprendizagem de uma linguagem totalmente nova e não possibilitam a reutilização de componentes de software já existentes.

O desenvolvimento de aplicações distribuídas com linguagens específicas levanta ainda um problema, sem dúvida o mais importante: a impossibilidade de interligar componentes desenvolvidos em linguagens distintas. De facto, a heterogeneidade dos paradigmas usados nestas linguagens não permite a utilização de soluções híbridas, ou seja, com componentes implementados em várias linguagens. Note-se que a interligação de componentes é deveras importante se pensarmos na variedade de fornecedores de soluções informáticas e na impossível uniformização através da imposição de uma única linguagem de programação.

Posto isto, tem-se dado cada vez mais importância às linguagens para especificação de interfaces[8]. Uma interface representa a funcionalidade de um componente de software, o qual poderá ser um objecto, por exemplo. A especificação de uma interface é efectuada numa linguagem extremamente simples, com um conjunto reduzido de palavras chave, e é independente da implementação do componente em causa.

Basicamente, esta é a mesma abordagem que foi descrita para os RPCs. No entanto, possui um âmbito mais global, pois com o aparecimento de normas, como a descrita no subcapítulo 2.2, conseguem-se mecanismos para interligar componentes desenvolvidos segundo diferentes paradigmas de programação (nos RPCs impõe-se a decomposição funcional), para além de se oferecerem meios que possibilitam a localização transparente de componentes distribuídos.

Abordagem Orientada ao Objecto

Considerando o impacto que a tecnologia da orientação ao objecto teve no mundo das linguagens de programação, será lícito perguntar que contributos esta tecnologia pode dar no domínio da computação distribuída. A verdade é que as abordagens orientadas ao objecto podem trazer a esta área muitos dos benefícios que trouxeram à computação centralizada, nomeadamente, encapsulamento[9], reutilização, portabilidade e expansibilidade.

De facto, é ainda mais natural utilizar tecnologias orientadas ao objecto nos sistemas dis-

[8] Estas linguagens também são conhecidas por linguagens para descrição de interfaces ou linguagens para definição de interfaces. É até comum encontrarem-se siglas distintas, conforme a escolha da palavra especificação, definição ou descrição.

[9] Do inglês *encapsulation*.

tribuídos que nos sistemas centralizados. Nestes últimos, os programadores são levados a fazer melhoramentos, tendo em vista o desempenho final das aplicações, que prejudicam a abstracção e a modularidade trazidas pelos objectos. Num sistema distribuído, os diversos componentes interagem através da troca de mensagens que, fundamentalmente, corresponde ao mecanismo de invocação de métodos na programação orientada ao objecto[10].

Em [11] são identificados dois componentes básicos necessários para a implementação da interoperação orientada ao objecto: a concordância[11] de tipos e a correspondência[12] em objectos. O primeiro permite expressar as relações existentes entre os tipos de dados locais e os remotos. Estas relações podem ser de:

- equivalência – quando os tipos de dados podem ser migrados sem qualquer tratamento;
- tradução – quando os tipos de dados podem ser migrados após algumas transformações (caso do tipo `string`, que em algumas linguagens não é suportado directamente, havendo necessidade de o transformar num `array` de caracteres);
- concordância – quando são definidas associações[13] entre tipos de dados locais e tipos de dados remotos.

O segundo componente é responsável pela criação das classes que implementam a operacionalidade definida na concordância de tipos e pela instanciação e gestão dos objectos usados na interacção de aplicações.

Na computação distribuída orientada ao objecto (DOC[14]) há a considerar, segundo [12], três características principais. Em primeiro lugar, são introduzidos melhoramentos nas metodologias procedimentais do tipo do RPC semelhantes aos conseguidos nas linguagens de programação anteriores ao aparecimento da programação orientada ao objecto. Estes melhoramentos incluem:

- o encapsulamento – promovendo a separação entre interface e implementação;
- a herança de interfaces e tipos parametrizados – permitindo uma maior reutilização;
- tratamento de excepções baseado em objectos – que simplifica a lógica de um programa separando o código de tratamento de erros do processamento normal da aplicação.

[10] A invocação de um método de um objecto é, em alguma literatura, designada por envio de uma mensagem a esse método.
[11] Do inglês *matching*.
[12] Do inglês *mapping*.
[13] Do inglês *bindings*.
[14] Distributed Object Computing

Em segundo lugar, permite-se a cooperação de aplicações a um nível de abstracção mais elevado. O objectivo primordial da DOC é dotar os programadores de mecanismos que permitam utilizar técnicas familiares, como a invocação de métodos sobre objectos. Note-se que, na programação com *sockets*, por exemplo, o programador tem ao seu dispor um meio para troca de sequências de bytes, o que não permite uma programação fácil, portável, flexível ou segura.

Finalmente, oferece-se uma base para a construção de mecanismos de alto nível que facilitem a colaboração entre serviços de aplicações distribuídas. O suporte à invocação remota de métodos é o primeiro passo da DOC. Um número cada vez maior de aplicações requer mecanismos de colaboração mais sofisticados, como a migração de objectos, transacções, comunicação em grupo, etc.

Para terminar, refira-se como exemplo de plataformas para a computação distribuída orientada ao objecto o *Object Oriented Distributed Computing Environment* (OODCE) da OSF [13] e o *Distributed System Object Model* (DSOM) da IBM [14].

Memória Partilhada Distribuída

A memória partilhada é um dos mecanismos utilizados na comunicação entre processos de uma máquina. Nesta tecnologia, uma determinada zona de memória deixa de pertencer a um único processo para poder ser acedida por duas ou mais entidades. Obviamente, o acesso a esta memória deverá ser controlado de forma a evitar operações (leituras ou escritas) concorrentes; normalmente garante-se que durante uma actualização apenas um processo tenha acesso a essa posição de memória.

Com o aparecimento dos sistemas distribuídos surge a necessidade de alargar o conceito de memória partilhada a um conjunto de máquinas – memória partilhada distribuída (DSM[15]). Nesta situação ter-se-á um espaço de endereçamento espalhado por várias máquinas, cada uma com a sua memória particular, onde estarão armazenados dados de interesse para vários programas em execução nessas máquinas. Por outras palavras, existirá um estado global armazenado (parcialmente ou na sua totalidade) em vários locais, para o qual várias entidades espalhadas pela rede contribuem.

Assim, cada vez que um processo referencia um item de dados, se este não se encontrar localmente, terá que ser localizado e transferido. Por outro lado, sempre que se efectua qualquer actualização é necessário ter em consideração as eventuais cópias desse item espalhadas pelas diversas máquinas.

A programação com DSM, apesar de oferecer o modelo mais próximo da programação centralizada, não é ainda muito aliciante. De facto, os sistemas que suportam DSM não

[15]Distributed Shared Memory

estão ainda preparados para operar a larga escala e os algoritmos necessários são de tal forma complexos que não permitem os desempenhos desejados. Em [15] é explorado um sistema de memória distribuída baseada em objectos (o DiSOM), sendo detalhadamente analisados múltiplos aspectos da DSM, para além de serem apresentadas implementações de alguns algoritmos.

Computação Baseada em Documentos

Nos últimos anos tem-se verificado uma evolução da computação baseada em documentos. Este é o novo desafio que os sistemas operativos actuais têm explorado, principalmente os vocacionados para a utilização pessoal.

O OLE (*Object Linking and Embedding*) [16] é a resposta da Microsoft à necessidade de criar documentos de forma mais fácil e intuitiva. Esta tecnologia baseia-se em documentos compostos, que são documentos constituídos por vários tipos de conteúdo, todos a partilhar o mesmo ficheiro. Um documento destes pode conter praticamente qualquer tipo de informação: tabelas, gráficos, texto, vídeo, som, etc.

Num documento composto, o utilizador pode editar qualquer tipo de conteúdo, o que significa que vários editores têm de ser combinados, partilhando o mesmo documento. O processo de incluir um determinado tipo de informação num documento, sem que as várias partes percam a sua identidade ou as suas fronteiras, é chamado de *embedding*. No OLE existe ainda um outro mecanismo, chamado *linking*, que permite incluir partes através de referências, ou seja, sem que a informação fique explicitamente armazenada no documento.

As aplicações que manipulam documentos compostos têm que, de alguma forma, ter acesso ao código das aplicações que permitem editar os objectos inseridos nesses documentos. De facto, quando um processador de texto, por exemplo, permite inserir uma tabela de uma folha de cálculo, tem que ter acesso ao código dessa folha de cálculo, pois seria impraticável incluir no editor de texto rotinas para tratar cada tipo de objecto passível de ser incluído em documentos compostos.

O OLE é baseado no *Component Object Model* (COM), o qual é patenteado pela Microsoft e constitui basicamente um conjunto de regras que os programadores devem seguir na construção de componentes interligáveis [17]. O COM indica o conjunto de interfaces que devem ser fornecidas por um objecto e disponibiliza ainda algumas interfaces básicas que podem ser usadas pelos programadores. Dado que o COM não suporta herança, estas interfaces têm praticamente que ser copiadas – agregação [18].

O mecanismo de invocação de operações do OLE, versão 2.0, obriga a que todos os objectos se encontrem na mesma máquina. Esta limitação é ultrapassada com o *Network* OLE.

A IBM está a desenvolver um mecanismo similar ao OLE – o OpenDoc [19]. Este é

baseado no *System Object Model* (SOM), para o qual já existe uma extensão – o DSOM – que permite o acesso a objectos remotos e se encontra de acordo com a especificação CORBA exposta no subcapítulo 2.2.

2.1.4 Sistemas Distribuídos Actuais

A vulgarização dos sistemas distribuídos tem demonstrado que a maioria dos mecanismos existentes para desenvolver aplicações são inadequados ou, pelo menos, insuficientes. De facto, o alargamento do campo aplicacional destes sistemas, deixando de estar confinados a ambientes meramente académicos, obriga a repensar as metodologias disponíveis, por forma a contemplar novos requisitos.

Por outro lado, a evolução da indústria electrónica trouxe novos cenários ao mundo da computação distribuída, como é o caso da computação nómada. Com esta inovação, para além de se revelarem inadequadas algumas soluções utilizadas em redes de sistemas fixos, surge um novo conjunto de potenciais aplicações a desenvolver [20], que não faziam sentido na ausência de sistemas portáteis.

O ponto central do problema reside no facto de as tecnologias usadas no desenvolvimento de aplicações distribuídas, nomeadamente as mencionadas nas secções anteriores, não garantirem, na sua generalidade, a disponibilidade e fiabilidade desejadas, já para não falar em questões como o desempenho ou a segurança. Em suma, urge dotar os sistemas distribuídos de tolerância a faltas[16].

Ligado à crescente utilização de aplicações distribuídas surge ainda um outro problema: a necessidade de garantir a interoperabilidade do elevado número de componentes desenvolvidos pelos diferentes produtores de software. Por forma a solucionar este problema, têm vindo a ser criadas algumas normas que no futuro deverão caracterizar as aplicações desenvolvidas neste domínio.

Normalização

A programação orientada ao objecto tem hoje um papel fundamental no desenvolvimento de aplicações, acreditando-se que permite ou permitirá um aumento significativo da produtividade dos programadores. No entanto, a transição para esta nova tecnologia implica investimentos por parte das empresas, as quais necessitam de garantias efectivas dos benefícios daí provenientes [21]. De facto, nenhum produtor de software pode correr o risco de utilizar metodologias menos vantajosas que as dos demais concorrentes.

Por outro lado, quem adquire um determinado produto de software exige a sua correcta

[16] Do inglês *fault tolerance*.

operação quando integrado com outros produtos. Este requisito assume actualmente uma grande importância, dado que é prática comum adquirir produtos de variados fabricantes. Num sistema distribuído a necessidade de interoperação entre os vários componentes é facilmente perceptível.

Neste sentido, foi criado em 1989 o Object Management Group (OMG), com o intuito de estabelecer normas que permitissem a interoperação, a modularidade e a portabilidade de aplicações distribuídas orientadas ao objecto. Esta organização não produz qualquer tipo de software; apenas são criadas especificações, com base nas ideias e na tecnologia dos seus membros, que respondem a pedidos de informação e pedidos de propostas lançados pelo OMG. A importância de tais especificações reside no facto de que a maior parte das grandes empresas, que produzem e comercializam produtos na área da computação distribuída e orientada ao objecto, se encontra entre as centenas de companhias que compõem o OMG. Este dado é deveras interessante se pensarmos na interoperação de soluções de diferentes vendedores.

O seguimento de normas no desenvolvimento de componentes de software não é a solução para o problema da tolerância a faltas. No entanto, é desejável que qualquer solução para esse problema seja concordante com normas estabelecidas e aceites por um grande número de produtores de software, por forma a garantir a sua divulgação e aplicabilidade.

Desconexão

Num sistema distribuído, algumas máquinas poderão estar desconectadas da rede por determinados períodos de tempo. Isto acontece com frequência quando se utilizam sistemas móveis ou até quando certos segmentos da infra-estrutura de comunicação se encontram em manutenção. Nesta circunstância, não há envio ou recepção de mensagens, o que implica que um algoritmo distribuído, em execução nesse instante, seja suspenso até ao momento de reconexão ou gere excepções. As desconexões podem ser voluntárias ou involuntárias, mas no presente contexto apenas interessam as primeiras, usando-se o termo falha para designar o acontecimento involuntário [22].

Dada a natureza voluntária das desconexões, a máquina em causa poderá informar o restante sistema, com o qual interactua, da sua iminente desconexão, permitindo a execução de um protocolo especial. Este protocolo destina-se a assegurar que o computador desconectado possa operar num modo isolado durante um determinado período de tempo. Após um período de desconexão, e reposta a ligação com o resto do sistema, torna-se necessário executar um protocolo de reconexão, que se destina à actualização do estado global do sistema, com base nas evoluções de ambas as partes. Por exemplo, em [23] são apresentados

alguns pormenores relativos à desconexão e reconexão no Coda File System[17].

A capacidade de uma aplicação operar em modo isolado, isto é, desconectada dos restantes componentes do sistema distribuído, possibilita a sua utilização num número mais alargado de casos, resultando daí uma maior disponibilidade.

Replicação

Em determinadas situações, a operação isolada de alguns componentes de um sistema distribuído é completamente impossível. Nestas circunstâncias, a única forma de garantir a funcionalidade do sistema, perante a falha de algumas partes, é através da replicação de componentes. Numa aplicação desenvolvida segundo o modelo Cliente-Servidor, por exemplo, a solução passaria pela utilização de vários servidores replicados. Assim, perante a impossibilidade de contactar um servidor, o cliente teria ainda a hipótese de interagir com uma das demais réplicas.

A replicação não só possibilita aumentar a disponibilidade de uma aplicação, como também oferece um meio para melhorar a sua fiabilidade, eliminando anomalias devidas à deficiente operação de um determinado componente.

O facto de se manterem várias cópias de um determinado componente implica, na maior parte dos casos, a utilização de mecanismos de comunicação em grupo. Na verdade, a comunicação ponto-a-ponto torna-se ineficiente ou até inadequada quando coexistem mais que duas entidades em permanente comunicação.

2.2 CORBA

O OMG idealizou uma arquitectura de gestão de objectos (OMA[18]) que tenta definir, a um alto nível de abstracção, as várias facilidades necessárias à computação distribuída orientada ao objecto. O núcleo da OMA é o *Object Request Broker* (ORB) – um mecanismo que oferece transparência na localização, activação e comunicação de objectos. O OMG publicou a especificação CORBA (*Common Object Request Broker Architecture*) [24], que é uma descrição concreta das interfaces e serviços que devem ser prestados pelos ORBs. A partir desse momento, muitos membros começaram a vender produtos baseados nesta especificação ou, pelo menos, anunciaram intenções de o fazer a curto prazo.

Esta especificação é formada por cinco componentes principais [25]:

- núcleo ORB;

[17] Um sistema de ficheiros distribuído.
[18] Object Management Architecture

- linguagem para definição de interfaces;
- interface de invocação dinâmica;
- repositório de interfaces;
- adaptadores de objectos.

Em [26] é apresentada uma implementação da especificação CORBA – o Orbix, comercializado pela IONA Technologies Ltd.

2.2.1 Núcleo ORB

No modelo da OMA, os objectos fornecem serviços e os clientes lançam pedidos para esses serviços. A função do ORB é entregar pedidos aos objectos e devolver resultados aos clientes. Os serviços necessários para realizar esta tarefa são completamente transparentes para os clientes. Na verdade, estes não necessitam saber onde residem os objectos na rede, como comunicam, como estão implementados, como são armazenados ou como executam. O ORB precisa apenas de uma referência para o objecto, de forma a o conseguir identificar e localizar, podendo ainda proceder à sua activação, caso este não se encontre a executar. As referências dos objectos podem ser convertidas em cadeias de caracteres[19] através de um serviço do ORB. Os clientes podem armazenar estas cadeias de um modo persistente, de forma a manter associações entre objectos e aplicações.

A especificação CORBA propõe duas formas distintas para a emissão de pedidos por parte dos clientes:

- invocações estáticas, através de *stubs* específicos;
- invocações dinâmicas, através da Interface de Invocação Dinâmica.

Os objectos não têm conhecimento da forma como o pedido foi efectuado (invocação estática ou dinâmica), nem sobre a sua origem. A sua função é efectuar o serviço pedido e devolver os resultados para o ORB, que por sua vez os entregará ao cliente.

2.2.2 Linguagem para Definição de Interfaces

A referência para um objecto, apesar de o identificar univocamente num universo arbitrariamente vasto, não descreve, necessariamente, nada acerca da interface desse objecto. Para que uma aplicação possa fazer uso de um objecto, esta deve conhecer os serviços que estão disponíveis (interface do objecto). Em CORBA, as interfaces são descritas numa

[19] Do inglês *string*.

linguagem declarativa, com uma sintaxe próxima do C++, mas independente de qualquer linguagem de programação.

A linguagem para definição de interfaces (IDL[20]) oferece tipos de dados básicos (`short`, `long`, `float`, `double` e `boolean`), compostos (`struct` e `union`) e *template* (`sequence` e `string`). Estes tipos de dados são usados na declaração de operações, mais propriamente na definição de argumentos e resultados. Por sua vez, as operações são usadas na declaração de interfaces para definir os serviços oferecidos pelos objectos. Existe ainda o construtor `module` que permite aglomerar declarações de interfaces, definições de tipos e até outros módulos.

As interfaces constituem o conceito mais importante da IDL, dado que, para além de descreverem objectos, podem ser usadas como referências para objectos. Desta forma, as operações podem tomar como argumento, ou ter como resultado, objectos arbitrariamente complexos, usando-se como tipo (do argumento ou do resultado) o nome da interface que descreve esse objecto.

A IDL oferece ainda mecanismos para herança de interfaces, onde as interfaces derivadas herdam as operações e os tipos definidos nas interfaces de base. Usando a terminologia do C++ [27], esta herança pode ser caracterizada da seguinte forma:

- todas as interfaces de base são `public virtual`;
- todas as operações são `virtual`;
- as operações não podem ser redeclaradas nas interfaces derivadas;
- não existe a noção de herança de implementações.

O facto de a linguagem para definição de interfaces ser uma linguagem declarativa acentua a separação entre a interface e a implementação, o que é bastante importante no desenvolvimento de sistemas orientados ao objecto. Em C++, por exemplo, os conceitos de herança de interfaces e herança de implementações estão misturados. Uma classe C++ derivada contém sempre todas as estruturas de dados pertencentes às suas classes de base e, para propósitos de polimorfismo, só podem ser redefinidos os métodos explicitamente declarados como `virtual` na classe de base. Dado que a IDL não é uma linguagem de implementação, não são confundidos os dois tipos de herança.

Todas as interfaces descritas na IDL derivam implicitamente de uma interface de raiz denominada `Object`. Esta interface proporciona serviços comuns a todos os objectos, tais como duplicação, destruição e validação de referências de objectos. Muitos programadores que usam C++ discordam da utilização de hierarquias baseadas numa única raiz, visto que

[20]Interface Definition Language

estas originam classes com interfaces maiores. No entanto, com a herança de interfaces da IDL, o uso de uma interface de base comum faz todo o sentido, dado que todos os objectos são, por definição, objectos CORBA e portanto devem fornecer determinados serviços básicos.

Uma classe C++ representa aquilo que um objecto pode fazer e aquilo de que um objecto é feito, isto é, o comportamento e o estado do objecto. Na IDL não existe o conceito de estado; herdando determinada interface, um objecto compromete-se a suportar essa interface, mas não oferece garantias sobre a forma como a sua implementação desencadeará as várias operações. No entanto, nas implementações dos objectos, os programadores são livres de usar os mecanismos de herança disponibilizados pelas linguagens que utilizarem.

Os compiladores da IDL traduzem as descrições das interfaces para módulos em linguagens de programação específicas, de acordo com as correspondências apresentadas na especificação CORBA. Esta abordagem permite que todos os objectos tenham as suas interfaces descritas numa linguagem bem conhecida e por todos aceite, podendo a implementação destes ser feita em qualquer linguagem à escolha do programador.

2.2.3 Interface de Invocação Dinâmica

A compilação das declarações escritas na IDL origina *stubs*, os quais permitem aos clientes invocar operações sobre objectos conhecidos. No entanto, algumas aplicações devem ser capazes de efectuar chamadas a objectos sem a necessidade de se conhecerem as interfaces destes no momento em que são compiladas. Por exemplo, uma aplicação destinada a percorrer um repositório de interfaces de objectos com capacidades para se apresentarem num ambiente gráfico deverá ser capaz de descobrir as operações suportadas por cada objecto e em seguida invocar operações, de forma a que o utilizador possa interactuar com esses objectos. Neste caso, dada a natureza dinâmica de um repositório, tal aplicação nunca poderia ser compilada com os *stubs* correspondentes a todos os objectos nele referenciados; apenas se poderá exigir que a aplicação saiba interpretar a informação existente no repositório.

A interface de invocação dinâmica (DII[21]) é um *stub* genérico, que actua do lado do cliente, capaz de encaminhar qualquer pedido para qualquer objecto. Isto é conseguido interpretando os parâmetros dos pedidos e os identificadores das operações em tempo de execução[22]. Os clientes começam por obter informação sobre as operações de uma interface, seguindo-se a criação de um objecto específico para tratar invocações a objectos[23], o qual suporta operações para adicionar parâmetros, para invocar a operação por si

[21] Dynamic Invocation Interface
[22] Do inglês *run-time*.
[23] Este objecto é, na verdade, implementado como parte do ORB.

representada e para a recepção de respostas assíncronas.

Em [28] é questionada a obrigatoriedade do suporte à invocação dinâmica nos produtos concordantes com a especificação CORBA, defendendo o autor que seria preferível que os vendedores pudessem optar por fornecer ou não esta solução. Esta posição deve-se à complexidade deste serviço e à escassa informação que se depreende da especificação CORBA.

2.2.4 Repositório de Interfaces

Um outro serviço suportado pela interface `Object`, e portanto por todas as referências de objectos, é a operação que permite obter a descrição da interface de um objecto. Esta descrição é mantida num repositório de interfaces (IR[24]), o qual permite armazenar persistentemente as declarações de interfaces escritas na IDL. Os serviços oferecidos por um IR permitem navegar através da hierarquia de herança de um objecto, podendo-se obter a descrição de todas as operações que um objecto suporta.

Os repositórios de interfaces podem ser usados para variados propósitos: os visualizadores de interfaces podem percorrer a informação armazenada para ajudar os programadores na localização de componentes de software potencialmente reutilizáveis, os ORBs podem usar a informação relativa à descrição das operações para validar argumentos durante a execução, etc. A principal função do IR é disponibilizar informação sobre tipos, informação essa necessária para efectuar pedidos através do DII.

2.2.5 Adaptadores de Objectos

Um adaptador de objecto (OA[25]) fornece os meios pelos quais as possíveis implementações de objectos (um único programa servidor, um conjunto de *scripts*, aplicações desenvolvidas antes do aparecimento do CORBA, etc.) utilizam os serviços do ORB (geração de referências para objectos, invocação de métodos, segurança e activação/desactivação de objectos ou implementações). Dependendo do ORB em causa, o OA pode optar por recorrer aos serviços fornecidos pelo ORB ou por ser ele próprio a prestar esses serviços.

O *Basic Object Adapter* (BOA), incluído em todos os ORBs, disponibiliza serviços suficientemente flexíveis, por forma a suportar diferentes tipos de implementações de objectos. Estas implementações podem ser:

- persistentes – quando activadas por alguém que não o BOA;

[24] Interface Repository
[25] Object Adapter

- partilhadas – quando coexistem, no mesmo programa, múltiplas implementações de objectos;
- exclusivas – quando existe a implementação de um único objecto por programa;
- servidor por método – quando existe um programa por cada operação suportada pelo objecto.

2.3 Suporte à Desconexão

O modelo Cliente-Servidor é muito popular, desempenhando um papel importante em sistemas de gestão de bases de dados (SGBDs) [29], como acontece no Oracle, em linguagens de programação, como acontece no Delphi [30], e num grande número de aplicações que utilizam a tecnologia RPC. Neste modelo, um determinado módulo (o cliente) faz pedidos a outro (o servidor) aguardando a devolução de resultados ou uma indicação de que o pedido foi processado (no caso de a invocação não obrigar ao envio de informação do servidor para o cliente). Numa aplicação distribuída, o cliente e o servidor localizar-se-ão em máquinas distintas, porventura em redes geograficamente afastadas, o que implica alguma forma de comunicação entre os vários componentes.

2.3.1 O Problema da Desconexão

Num sistema real, a programação de aplicações distribuídas deverá contemplar situações de falta de interactividade, situações estas devidas a problemas relacionados com as infra-estruturas de comunicação ou com as várias máquinas que constituem o sistema. Este requisito torna-se cada vez mais pertinente devido às crescentes exigências em relação às soluções que envolvem a distribuição. De facto, quando se passa do domínio puramente académico para o domínio das soluções comerciais, deixa de ser aceitável que um determinado serviço permaneça suspenso, ou pura e simplesmente deixe de funcionar e implique a reinicialização de todo o sistema, sempre que o cliente fique isolado[26].

Por exemplo, na implementação do *Network File System* (NFS) levada a cabo pela Sun [31], a interacção entre o cliente e o servidor é efectuada através de RPCs. Cada vez que o servidor fica inacessível, o sistema onde executa o cliente fica bloqueado após a primeira tentativa de efectuar uma invocação, até que seja reposta a ligação com o servidor. Isto deve-se ao facto de o NFS ter sido implementado ao nível do sistema operativo, o que significa que um cliente fica à espera (bloqueado) no interior de uma chamada ao sistema

[26] Um cliente diz-se isolado se não puder contactar com o servidor, independentemente do motivo adjacente a esse facto.

operativo. Esta espera implica que a máquina onde se encontra o cliente não possa efectuar qualquer processamento.

Neste contexto, apareceram aplicações com tolerância a faltas, ou seja, programas que, apesar da adversidade das condições em que operam, criam no utilizador a ilusão de que tudo se encontra em perfeito funcionamento. Obviamente, estas soluções implicam uma sobrecarga, do ponto de vista da programação, para lidar com os problemas referidos; o próprio modelo de programação perde a sua principal vantagem – a simplicidade.

Com a crescente divulgação da computação nómada, a interoperação dos componentes das aplicações distribuídas torna-se mais complexa. O problema da desconexão, operação pela qual se isola um cliente de forma voluntária, não deve ser confundido com a situação até agora descrita. No entanto, a resolução deste problema é também válida para os casos de isolamento involuntário, pois basta considerar que se trata de uma desconexão onde não é efectuada nenhuma operação de preparação. Por outro lado, a comunicação entre os sistemas móveis possui características singulares que, em alguns casos, impedem a constante invocação de operações de um servidor remoto.

Do ponto de vista do programador, as tecnologias disponíveis para o desenvolvimento de aplicações distribuídas, nomeadamente para ambientes de computação nómada, devem tratar destes problemas, oferecendo um modelo de programação linear. No caso do modelo Cliente-Servidor, os programadores deverão ser capazes de desenvolver aplicações sem se preocuparem com processamentos extraordinários por cada invocação efectuada sobre um objecto remoto e conhecendo unicamente a interface dos serviços remotos de que vão fazer uso. Por outras palavras, para além de esconder do utilizador a complexidade de um sistema distribuído, será desejável criar mecanismos para que nem sequer os programadores necessitem de se familiarizar com os detalhes do funcionamento de um sistema deste tipo.

Neste seguimento surgiram alguns trabalhos, entre os quais se destacam o *Mobile* RPC (M-RPC) [32], os Stublets [33] e o Rover [34]. O M-RPC é uma adaptação do tradicional protocolo RPC para sistemas distribuídos com clientes instalados em máquinas móveis. Com o M-RPC é possível que um cliente troque de servidor (associação dinâmica), a comunicação entre o sistema móvel e a rede fixa é efectuada através do *Reliable Data Protocol* (RDP) – um protocolo de transporte que garante transferência fiável, usando comunicação sem fios[27] – e as invocações, em caso de insucesso, são repetidas automaticamente.

Os Stublets constituem um modelo para o desenvolvimento de aplicações, cujo objectivo é reduzir a utilização dos meios de comunicação dos sistemas móveis. Neste modelo as operações que implicam o acesso a informação remota são agrupadas em módulos especiais – *stublets*. Estes podem ser executados usando replicação, operações remotas ou delegação. A escolha da técnica a usar é efectuada em tempo de execução, com base na qualidade

[27] Do inglês *wireless communication*.

de serviço (QOS[28]) disponível e na semântica das operações, a qual é especificada numa linguagem própria, para cada *stublet*.

O Rover é constituído por um conjunto de funções que permitem mover dinamicamente objectos com uma interface bem definida, do servidor para o cliente e vice-versa, reduzindo significativamente os requisitos de comunicação entre estes. Estas funcionalidades são combinadas com RPCs pendentes[29], os quais permitem que as aplicações continuem a efectuar invocações não bloqueantes, mesmo quando o servidor está inacessível, sendo os pedidos e as respostas trocados logo que restabelecido o contacto com o servidor.

Todos os trabalhos desenvolvidos nesta área constituem avanços preciosos para a construção de uma plataforma de desenvolvimento suficientemente genérica. Porém, na maioria das aplicações distribuídas continuam a ser adoptadas soluções particulares. Em alguns casos adaptam-se serviços já existentes, que não contemplam o problema da desconexão. Este é o caso do *Mobile Integration of* NFS (MIo-NFS) [35], onde se criaram mecanismos para que os clientes NFS instalados em sistemas portáteis possam fazer réplicas de determinados ficheiros e consigam deste modo continuar o seu funcionamento normal, quando impossibilitados de interactuar com o servidor de NFS. Também no *Replicated* NFS (RepNFS) [36] se altera o sistema original, por forma a permitir a instalação de vários servidores NFS (servidores replicados), conseguindo-se assim que a indisponibilidade de um servidor não impeça o funcionamento do cliente.

2.3.2 Modelo Cliente-Agente-Servidor

As aplicações distribuídas são frequentemente baseadas no modelo Cliente-Servidor, que impõe uma clara distinção entre o fornecedor do serviço e o cliente. Na computação nómada, quando estas duas entidades não podem comunicar entre si (período de desconexão), é desejável que as aplicações continuem a funcionar. Por forma a atingir este objectivo, projectos como o CODA [37] e o D-NFS levaram à introdução da noção de agente. O agente actua como um pseudo-servidor do lado do cliente e como um pseudo-cliente do lado do servidor, daí resultando o modelo Cliente-Agente-Servidor (CAS).

Em [38] é apresentada uma metodologia de programação, baseada no modelo CAS usado no D-NFS, que suporta a operação de desconexão. Neste modelo, o agente é composto por um representante[30] do servidor, um despachante[31], funções de transição e um representante do cliente. O representante do servidor exporta a mesma interface que o servidor, mas pode operar no estado conectado ou desconectado. O despachante direcciona os pedidos para as

[28] Quality of Service
[29] Do inglês *queued* RPC.
[30] Do inglês *proxy*.
[31] Do inglês *dispatcher*.

funções apropriadas, baseando-se no seu estado. As funções de transição são o mecanismo usado pelo agente para trocar de um estado para outro. Finalmente, o representante do cliente permite que o agente comunique com o verdadeiro servidor (obviamente, o agente está localizado do lado do cliente).

Basicamente, esta metodologia resume-se no seguinte: por cada função `f()` exportada pelo servidor existirão duas funções – `f_connected()` e `f_disconnected()` – no agente, que serão invocadas conforme o estado deste. As acções desencadeadas numa transição terão que assegurar a correcta operação no novo estado.

Paralelamente a esta abordagem, e mais recentemente, tem vindo a ser dada grande importância aos mecanismos de acesso a informação distribuída que não usam a tradicional invocação de operações remotas. Em [39] é apresentada como solução a fragmentação de objectos. Distribuindo um objecto por várias máquinas (uma parte residente em cada máquina), pode-se explorar o facto de certas operações exigirem diferentes cargas comunicacionais entre as diversas partes. Decidir em quantas partes distintas se deve dividir um objecto e que partes devem residir no sistema móvel, se se pensar na computação nómada, é uma tarefa bastante complexa; o ideal seria que estas decisões fossem tomadas dinamicamente e em tempo de execução.

A fragmentação de um objecto só é possível se houver uma forma de descrever o comportamento desse objecto. De facto, o objecto não pode ser dividido em conjuntos arbitrários de métodos e estruturas de dados; é necessário ter em consideração as relações entre as diversas operações e as relações entre operações e dados.

A complexidade da fragmentação de objectos leva a crer que o modelo CAS é a abordagem mais apropriada para o suporte à desconexão, principalmente se se pensar na necessidade de uma concretização.

2.4 Replicação

Num sistema distribuído há frequentemente a necessidade de manter várias cópias – réplicas – dos mesmos dados em diversos sistemas. A replicação tem por finalidade aumentar:

- o desempenho – se se fizerem cópias de determinada informação perto dos locais onde habitualmente é necessária, evitam-se acessos remotos dispendiosos;
- a disponibilidade – ao armazenar cópias em pontos com probabilidades de falha independentes, aumenta-se a probabilidade de encontrar pelo menos uma cópia acessível;
- a fiabilidade – colocando em sistemas distintos várias cópias da informação con-

siderada importante, elimina-se o efeito catastrófico da perda de um sistema de armazenamento.

2.4.1 Correcção dos Dados

Nas aplicações que processem dados replicados é necessário garantir a correcção dos dados, ou seja, é necessário assegurar a coerência[32] das réplicas; todas as cópias do mesmo item lógico de dados devem concordar relativamente a um valor, num determinado momento. Quando a comunicação entre as máquinas onde residem cópias do mesmo item de dados não é possível, a coerência torna-se difícil de assegurar. De facto, se a falta de comunicação não for detectada por todos os pontos de replicação, podem ser executadas alterações de forma independente e descoordenada, ou seja, as réplicas podem evoluir de maneira diferente.

No planeamento de aplicações distribuídas com componentes replicados é obrigatório confrontar as necessidades de disponibilidade e de correcção da informação. Na verdade, estes dois objectivos são conflituosos, existindo uma dependência entre eles. A correcção pode ser conseguida pela suspensão da operação de todas as réplicas, excepto uma, e posterior envio de actualizações às restantes cópias. Por outro lado, a disponibilidade consegue-se facilmente ao permitir que todas as réplicas sejam actualizadas de forma independente e em simultâneo.

A actualização independente de réplicas levanta problemas em relação à coerência, pois as diversas actualizações podem originar resultados divergentes. Tais incorrecções podem ser aceitáveis em determinadas aplicações que possuam como requisito uma elevada disponibilidade.

Porém, certas aplicações impõem níveis elevados de correcção, o que impede, ou dificulta, a adopção de estratégias que favoreçam a disponibilidade. É o caso de uma aplicação bancária, na qual a informação relativa a uma conta deverá, a qualquer momento, ser a mesma em todas as agências e reflectir todas as operações desencadeadas, apesar de se poderem efectuar movimentos a partir de vários locais. De facto, esta é a única forma de evitar, por exemplo, que se efectuem dois levantamentos, em agências distintas, que na sua totalidade ultrapassem o valor inicial do saldo da conta.

A incoerência de réplicas surge quando ocorrem partições na rede ou quando as diversas operações obrigam a uma determinada ordem de execução e esta não é respeitada. Outro factor a considerar prende-se com a falha das máquinas onde residem réplicas, dado que, aquando da recuperação dessas máquinas, o estado das cópias em causa não irá reflectir as operações ocorridas desde o instante em que ocorreu a falha.

Relativamente às partições, facilmente se percebe que um subconjunto de réplicas, ao ficar

[32] Do inglês *consistency*.

isolado, passará a ter uma visão parcial do sistema, apercebendo-se apenas de uma parte dos acontecimentos.

No que respeita à ordem das operações, o que está em causa é a interdependência dessas operações. Assim, considerando novamente o exemplo da aplicação bancária, imagine-se que sobre uma conta eram despoletadas as seguintes operações: crédito de um valor monetário e vencimento de juros à taxa em vigor. É óbvio que estas actualizações levarão a resultados distintos, dependendo da ordem pela qual são efectuadas. No entanto, um problema mais sério ocorrerá se dois grupos de réplicas derem sequências diferentes às operações. Neste caso, ter-se-ia violado a coerência das cópias, apesar de a comunicação entre elas ser possível.

2.4.2 Estratégias de Replicação

As estratégias de replicação podem ser classificadas segundo duas dimensões ortogonais [40]:

- compromisso entre correcção e disponibilidade;
- tipo de informação usada para verificar a correcção das réplicas.

A primeira dimensão possui dois extremos: pessimista e optimista. Na replicação pessimista previne-se a incoerência limitando a disponibilidade. Parte-se do princípio que o funcionamento normal do sistema levará garantidamente a incoerências de difícil resolução, sendo portanto necessário impedir que várias réplicas sejam alteradas de forma independente. Basicamente, nesta estratégia permite-se que apenas uma cópia seja alterada num determinado instante.

Quando se opta por uma estratégia optimista, espera-se que as operações desencadeadas sobre cada uma das réplicas não interfiram entre si. Desta forma, será possível, após um período de incoerência das réplicas, chegar a um consenso no que respeita ao seu estado. Note-se que, se porventura aquele pressuposto não se verificar, será uma tarefa extremamente complexa determinar um novo estado no momento em que seja reposta a ligação entre as várias cópias.

Para a segunda dimensão têm-se como extremos as abordagens sintáctica e semântica. A abordagem sintáctica usa como critério de correcção uma propriedade denominada seriação a uma única cópia[33]. Sucintamente, esta propriedade diz que a execução concorrente de um conjunto de operações pelas várias réplicas deverá ser equivalente, em termos de resultado final, à execução sequencial dessas operações numa única cópia, sem replicação.

[33] Do inglês *one-copy serializability*.

No outro extremo, a abordagem semântica usa tanto a semântica das operações como a semântica dos dados para definir correcção.

Deste modo, as diferentes estratégias de replicação podem ser classificadas como: pessimistas-sintácticas, optimistas-sintácticas, pessimistas-semânticas ou optimistas-semânticas. Na prática, as diferentes estratégias podem ser combinadas, ou seja, o sistema pode começar por usar uma estratégia optimista e, com o decorrer do tempo, trocar para uma estratégia pessimista se, por exemplo, se verificarem demasiados conflitos. Por outro lado, podem utilizar-se em simultâneo diferentes estratégias conforme os itens de dados. Estas duas formas de combinar estratégias de replicação são denominadas, respectivamente, por vertical e horizontal.

2.4.3 Lotus Notes

O Lotus Notes 4.1 [41] (desenvolvido pela Lotus) tem por finalidade criar um ambiente integrado para projectos em grupo. Outras ferramentas do mesmo género são o Microsoft Exchange (da Microsoft) e o GroupWise (da Novel), embora se encontrem bastante menos desenvolvidos (na melhor das hipóteses, comparáveis a versões anteriores do Lotus Notes).

O Lotus Notes introduz o poder e a conveniência da gestão de bases de dados no armazenamento de informação não estruturada. Cada entrada numa base de dados Lotus Notes é um documento com alguns campos de informação juntamente com o conteúdo do documento, guardado em formato RTF (*Rich Text Format*). Os utilizadores podem consultar vistas[34] que mostram informação relativa aos documentos e podem configurar novas vistas para ordenar, filtrar ou agrupar documentos. Toda a informação relativa a utilizadores, bases de dados e conexões entre sistemas é mantida num servidor, numa base de dados especial – o livro de nomes e endereços[35]. A troca de informação entre utilizadores e máquinas é efectuada através de correio electrónico.

A replicação faz do Lotus Notes uma ferramenta única, já que os demais concorrentes não tratam este aspecto. No Lotus Notes a mesma base de dados pode existir em diferentes locais, podendo os utilizadores não só consultar mas também adicionar, apagar ou modificar documentos. O sistema encarrega-se de sincronizar todas as alterações.

Para a partilha de informação a nível de uma empresa, o Lotus Notes oferece replicação entre servidores. O livro de nomes e endereços de cada servidor inclui documentos de conexão especificando os servidores com quem replicar, o esquema de replicação e a forma de entrega de informação via correio electrónico. As empresas podem também replicar informação com clientes ou outras empresas associadas.

[34] Do inglês *views*.
[35] Do inglês *name and address book*.

A replicação oferecida pelo Lotus Notes facilita a operação dos utilizadores que se conectam via modem ou que necessitam de bases de dados (ou parte delas) para trabalhar em viagens. Os utilizadores podem replicar qualquer base de dados, seleccionando a informação que necessitam para trabalhar isoladamente. Uma vez desconectados, os utilizadores podem criar ou alterar documentos, com a garantia que no momento da reconexão (com o servidor Lotus) o sistema enviará as mensagens necessárias para todas as réplicas da base de dados em causa.

2.5 Comunicação em Grupo

Um pressuposto da programação com *sockets* ou RPC é a comunicação envolver unicamente duas entidades – o cliente e o servidor. No entanto, certas aplicações podem tirar partido, ou mesmo necessitar, da interligação de mais que dois processos. Seria portanto de grande utilidade um mecanismo que enviasse uma mensagem para vários destinatários, numa única operação[36]. Este é o papel da comunicação em grupo.

A comunicação em grupo ou difusão selectiva[37] é de extrema importância na implementação de soluções que usam replicação. Nesta dissertação em particular, a evolução coerente das várias réplicas de um determinado objecto é conseguida através de um sistema de difusão selectiva (descrito no final deste subcapítulo).

2.5.1 Princípios Gerais

Um grupo é um conjunto de processos com uma característica particular: quando se envia uma mensagem para um grupo, todos os membros a recebem. A qualquer momento, podem criar-se ou destruir-se grupos, bem como ser-lhes adicionados ou retirados elementos. Um processo pode ser simultaneamente membro de vários grupos.

O objectivo da comunicação em grupo é permitir aos processos lidar com conjuntos de processos, como se se tratasse de uma única abstracção, sem necessidade de se conhecer os vários elementos (membros).

Em [1] são identificados alguns aspectos importantes no funcionamento dos serviços de comunicação em grupo, os quais são apresentados de seguida.

[36] Com *sockets* ou RPC seria necessário enviar, explicitamente, uma mensagem por cada destinatário.
[37] Do inglês *multicast*.

Grupos Fechados/Abertos

Os sistemas que suportam comunicação em grupo são divididos em duas categorias, dependendo de quem pode enviar mensagens para quem. Se apenas os membros de um grupo podem enviar mensagens para esse grupo, então está-se na presença de um grupo fechado. Se, pelo contrário, um processo exterior ao grupo puder enviar mensagens para esse grupo, então diz-se que o grupo é aberto.

Os grupos fechados são usados, tipicamente, para processamento paralelo. Nesta situação, os processos que constituem o grupo têm um objectivo comum e trocam informação entre si, usando os mecanismos de comunicação em grupo, por forma a alcançarem esse objectivo, mas não interagem com o exterior.

Por outro lado, quando se pretende replicar servidores, criando-se para tal um grupo, é importante que os clientes, que lhe são exteriores, possam enviar pedidos para esse mesmo grupo. Note-se que, neste caso, os vários servidores também usam a comunicação em grupo para gerir a sua operação conjunta.

Grupos Equitativos/Hierárquicos

A distinção entre grupos fechados e abertos tem a ver com os possíveis remetentes das mensagens destinadas a um grupo. Uma outra distinção importante está relacionada com a estrutura interna dos grupos. Em alguns grupos, todos os processos são tratados de forma igual; todas as decisões são tomadas colectivamente. Noutros grupos, pode existir alguma forma de hierarquia. Por exemplo, um processo poderá assumir o papel de coordenador.

Cada uma destas organizações tem as suas vantagens e desvantagens. No primeiro caso – grupos equitativos[38] – não existe um ponto singular de falha[39]; se um membro do grupo falhar, o grupo torna-se mais reduzido, mas continua operacional. Uma desvantagem destes grupos é a complexidade do processo de tomada de decisões; todos os membros têm que colaborar para se decidir seja o que for.

Nos grupos hierárquicos, a perda do coordenador impede a operação de todo o grupo, pelo que são necessárias estratégias de eleição de um novo coordenador.

Gestão de Grupos

Num sistema com suporte para comunicação em grupo são necessários mecanismos para criar grupos, destruir grupos, entrar para um grupo e abandonar um grupo. Uma possível solução seria ter um servidor de grupos, ao qual seriam enviados todos estes pedidos. No

[38] Do inglês *peer groups*.
[39] Do inglês *single point of failure*.

entanto, esta abordagem sofre o mesmo problema que as técnicas centralizadas: um ponto singular de falha.

Uma solução para este problema é a gestão distribuída de membros dos grupos. Assim, num grupo aberto, um processo exterior ao grupo envia uma mensagem para todos os membros do grupo (para o grupo em si), anunciando a sua intenção de pertencer ao mesmo. Num grupo fechado, algo similar será necessário; com efeito, até um grupo fechado terá que ser aberto no que respeita à operação de entrada. Para abandonar o grupo, um membro terá que enviar uma mensagem informativa aos restantes membros.

Apesar da aparente simplicidade desta estratégia, existem alguns aspectos que requerem tratamento especial. Em primeiro lugar, se um membro falhar, significará que abandona o grupo. No entanto, não haverá nenhuma mensagem para alertar os outros membros deste facto; estes terão que descobrir a falha ocorrida através da constatação de que o membro em causa deixa de responder às várias mensagens enviadas.

Em segundo lugar, as operações de entrada e saída do grupo deverão ser síncronas. Isto quer dizer que a partir do momento em que um processo se junta a um grupo e passa a fazer parte deste, deverá receber todas as mensagens enviadas a esse grupo. Analogamente, logo que um membro decida abandonar um grupo, não deverá receber mais nenhuma mensagem destinada ao grupo e os restantes membros não deverão receber nenhuma mensagem por si enviada.

Finalmente, a falha de vários membros de um grupo poderá acarretar alguns problemas. De facto, o grupo poderá deixar de funcionar ao ficar demasiadamente reduzido. Nestes casos, é obrigatório recorrer a mecanismos de reconstrução de grupos.

Endereçamento de Grupos

Para enviar uma mensagem para um grupo, um processo deve possuir algum meio para especificar de que grupo se trata. Por outras palavras, é necessário um método para identificar os vários grupos constituídos num sistema. Uma possibilidade é atribuir a cada grupo um endereço único, de forma a que um processo possa indicar esse endereço como destino de uma determinada mensagem. Outra solução será obrigar o emissor a especificar os vários destinatários da mensagem, ou seja, cada membro do grupo terá que manter uma lista de todos os membros para conseguir endereçar uma mensagem.

Envio e Recepção de Mensagens

Idealmente, a comunicação ponto-a-ponto e a comunicação em grupo deveriam ser uniformizadas num conjunto único de instruções. No entanto, adaptar a filosofia do modelo

RPC, mecanismo usual na interligação de entidades ponto-a-ponto, à comunicação em grupo e abandonar as tradicionais instruções `send` e `receive` é uma tarefa difícil. A principal dificuldade deve-se às inúmeras respostas que seriam devolvidas por cada pedido efectuado ao grupo e à incapacidade de o cliente lidar com mais que uma resposta.

Deste modo, uma abordagem comum é abandonar o modelo pedido-resposta subjacente ao RPC e efectuar invocações explícitas para o envio e recepção de mensagens.

Atomicidade

Uma característica da comunicação em grupo é a propriedade de tudo-ou-nada. De facto, a maioria dos sistemas de comunicação em grupo garante que qualquer mensagem destinada ao grupo seja recebida ou por todos os seus membros ou por nenhum; não se admitem situações em que alguns membros recebem a mensagem e outros não. Esta propriedade é denominada por atomicidade ou difusão atómica.

A atomicidade é desejada porque facilita muito a programação de sistemas distribuídos. Cada vez que um processo envia uma mensagem para um grupo não precisa de contemplar a possibilidade de algum não receber essa mensagem; a partir do momento em que a mensagem é dada como entregue, garantidamente, todos a receberam.

A implementação da difusão atómica não é tão simples quanto a sua explicação. É necessário que todos os membros do grupo acusem a recepção de mensagens e, se se pretender tolerância a faltas, são necessários algoritmos bastante complexos.

Ordenação de Mensagens

Para tornar a comunicação em grupo fácil de perceber e de usar, são necessárias duas propriedades: a atomicidade e a ordenação de mensagens. Para entender esta última, considere-se o caso em que são enviadas, num curto espaço de tempo, duas mensagens para o mesmo grupo. Se dois membros do grupo receberem estas mensagens, cada um por uma ordem diferente, pode acontecer que as operações desencadeadas pela recepção dessas mensagens levem a estados incoerentes desses membros.

Assim, seria desejável que todas as mensagens fossem entregues instantaneamente e exactamente pela ordem em que são geradas[40], ou seja, que fosse utilizada uma ordenação temporal e global. Dado que este comportamento é bastante difícil de se conseguir, opta-se geralmente por algoritmos que garantem uma ordenação um pouco mais fraca: impõe-se apenas que todos os membros de um grupo recebam as mensagens pela mesma ordem

[40] As tecnologias de comunicação actuais não garantem, por si só, que duas mensagens sejam entregues exactamente pela mesma ordem em que são enviadas; nem sequer garantem que dois receptores vejam essas mensagens pela mesma ordem.

– ordenação total. Em alguns casos basta até que as mensagens sejam ordenadas pelos vários elementos do grupo segundo o seu efeito – ordenação causal. Na realidade, é pouco importante determinar, ou até mesmo impossível descobrir, qual de entre duas mensagens foi a primeira a ser enviada, se tudo tiver ocorrido num curto espaço de tempo.

A ordenação de mensagens, até aqui referida, diz respeito a mensagens enviadas para um determinado grupo. No entanto, um processo pode pertencer a vários grupos, ou seja, num sistema real existirão grupos não disjuntos. Nestes casos, apesar da ordenação de mensagens no interior de um grupo, nada se garante acerca da ordem entre mensagens destinadas a grupos distintos. Assim, pode acontecer que dois processos pertencentes simultaneamente a dois grupos recebam duas mensagens, uma destinada a cada grupo, por ordem diferente.

A implementação da ordenação de mensagens entre grupos é ainda mais complicada que a ordenação de mensagens no interior de um grupo.

Escalabilidade

Muitos algoritmos usados na comunicação em grupo têm um comportamento inaceitável quando os grupos contêm um elevado número de membros ou quando se criam demasiados grupos. Isto deve-se, por exemplo, à complexidade dos algoritmos utilizados e ao recurso a componentes centralizados.

Um outro problema tem a ver com a extensão de um grupo a um conjunto de processos espalhados por várias redes locais (LANs[41]), uma vez que alguns algoritmos fazem uso de determinadas propriedades da comunicação nestas redes, por exemplo, a impossibilidade de num determinado instante existirem duas mensagens em trânsito.

Quando o mecanismo de comunicação em grupo tem estes problemas, diz-se que não possui um boa escalabilidade[42].

2.5.2 Uma Concretização: o GTS

O *Generic Multicast Transport Service* (GTS) [42] é um sistema de difusão selectiva que garante a entrega fiável de mensagens com preservação de ordem (ordem total).

No GTS, as mensagens são entregues aos destinatários logo que estes se encontrem disponíveis, mas nada se impõe sobre o tempo de vida das mensagens. Assim, se num determinado instante um processo não se encontrar em condições de aceitar mensagens, toda a informação a si destinada é armazenada pelo sistema de comunicação para entrega poste-

[41] Local Area Networks
[42] Do inglês *scalability*.

rior. Esta abordagem permite que o GTS nunca exclua elementos de um grupo, a não ser que um determinado elemento o peça explicitamente. Noutras tecnologias de comunicação em grupo, sempre que um processo deixa de responder a mensagens é considerado faltoso, sendo portanto excluído. Este tipo de decisão dificulta a constituição de grupos onde alguns dos elementos possam ficar inacessíveis por períodos de tempo indeterminados.

Por outro lado, no GTS, os grupos não estão confinados a uma LAN; nada impede que os elementos de um grupo estejam espalhados por várias LANs com atrasos de comunicação ponto-a-ponto arbitrariamente elevados. Além disso, o código do GTS é suficientemente flexível para se poder proceder à sua compilação e instalação em variadas arquitecturas e sistemas operativos, o que aumenta significativamente o seu campo aplicacional.

Estas duas características, sem esquecer o facto de se tratar de um sistema de uso livre (sem encargos), fazem do GTS um sistema preferível, quando se pretende uma elevada portabilidade ou quando se está na presença de máquinas que operam de forma desconectada. Refira-se que tecnologias de comunicação em grupo como o xAMp [43], por exemplo, não possuem nenhuma destas características.

Funcionamento

O GTS distingue entre dois tipos de entidades:

- servidores, que implementam *spool* de mensagens (papel de sequenciador) e comunicação ponto-a-ponto;
- aplicações finais, que são programas que usam o GTS para enviar e receber informação, recorrendo para tal a um servidor.

Um servidor GTS juntamente com as aplicações que fazem uso de si constituem um *cluster*, que na generalidade dos casos se encontra confinado a uma LAN. Sempre que uma aplicação de um *cluster* pretende enviar uma mensagem para uma aplicação pertencente a outro *cluster* tem que submeter essa mensagem ao seu sequenciador, o qual a fará chegar ao sequenciador do outro *cluster*, que por sua vez a entregará à aplicação destino.

Quando se utilizam sistemas portáteis, ou outro tipo de sistemas que necessitem de operar de forma isolada, torna-se necessário instalar um sequenciador para cada máquina. Desta forma, consegue-se que todas as mensagens permaneçam armazenadas (pendentes) até que haja conectividade. Refira-se ainda que a falha de um servidor é perfeitamente recuperável, dado que toda a informação relativa aos grupos criados e as próprias mensagens pendentes são armazenadas persistentemente.

O GTS suporta um vasto conjunto de protocolos para a transferência de mensagens ponto-a-ponto (TCP, IP, AppleTalk, e-mail, UUCP, etc.), oferecendo aos programado-

res uma interface independente destes protocolos e com garantias de difusão selectiva fiável em qualquer circunstância. Isto levou a que o esquema de endereçamento escolhido fosse o mais simples e flexível possível, tendo-se optado pela utilização de localizadores uniformes de recursos (URLs)[43]. Os URLs do GTS obedecem à seguinte estrutura: `protocol://cluster:server:localAddress/ticket`.

A primeira parte do URL – `protocol` – define o protocolo usado para entregar a mensagem no seu destino, isto é, a forma de fazer chegar a mensagem ao sequenciador com o qual o processo detentor de tal endereço interactua. A segunda parte – `cluster:server` – contém o endereço do *cluster* destino, sendo normalmente formado pelo nome da máquina onde se encontra o sequenciador desse *cluster* (`domínio:máquina`). O `localAddress` é um identificador específico do protocolo utilizado pelo sequenciador GTS, por exemplo, o número de um porto TCP, uma conta de correia de electrónico, etc. O `ticket` (parte local do URL) identifica a aplicação ou grupo de aplicações a que se destina a mensagem após ter sido atingido o sequenciador onde o endereço está registado.

Deste modo, qualquer processo que faça uso do GTS terá que possuir um endereço (um URL), estando para tal registado num sequenciador. Os grupos também possuem um URL registado num determinado sequenciador, embora os seus elementos possam pertencer a *clusters* variados, com sequenciadores distintos. Fundamentalmente, a informação de registo de um grupo é a lista dos elementos (URLs) que o constituem. A parte local dos URLs dos grupos é obrigatoriamente numérica (número sequencial que o GTS atribui quando se cria um novo grupo) e alguns dos seus elementos podem eventualmente ser outros grupos.

Quando um sequenciador recebe uma mensagem, através da análise do URL do destino, determina se tem que a enviar para outro *cluster* ou não. Se se tratar de um URL registado localmente e se corresponder a um grupo, então é obtida a lista de membros e passa-se ao envio de uma cópia para aqueles que se encontram registados noutros sequenciadores. As mensagens destinadas a processos locais são armazenadas até que surjam pedidos explícitos de recepção.

Utilização

O sistema GTS disponibiliza três classes C++ para se desenvolverem aplicações que usem o serviço de difusão selectiva:

- `Message`, que possibilita compor mensagens para posterior envio e permite interpretar uma mensagem recebida (saber qual o originador, a informação, etc.);

[43] Uniform Resource Locators

- `TcpStreamAd`, necessária para a aplicação se conectar a um sequenciador;
- `SimpleApi`, que permite o envio e recepção de mensagens (métodos `send` e `receive`), para além da gestão de grupos.

O envio de uma mensagem GTS corresponde na prática à entrega dessa mensagem ao sequenciador (acção de submeter a mensagem). No caso de se utilizar como protocolo o TCP, por exemplo, submeter um mensagem (destinada a um grupo ou a um único processo) significará enviá-la ao sequenciador através de um *socket*. Após a entrega da mensagem ao sequenciador, é recebida uma confirmação de que esta foi aceite pelo sistema.

Por outro lado, a recepção de mensagens implica inquirir o sequenciador, isto é, a operação `receive` na realidade contacta o sequenciador para averiguar se existem mensagens ou não. Por forma a tornar este mecanismo mais eficiente, criou-se a possibilidade de escolher um comportamento bloqueante ou não, através de um argumento desta operação.

Capítulo 3

Construção de Aplicações em ILU

Neste capítulo apresenta-se a plataforma ILU, que serviu de base ao trabalho desenvolvido nesta tese. Após serem descritos o modo de funcionamento e as potencialidades deste sistema, expõem-se os motivos que levaram à sua escolha. Por fim, é apresentado um exemplo de utilização do ILU, na construção de aplicações distribuídas, sendo focados os aspectos mais relevantes de todo o processo de desenvolvimento.

3.1 Sistema ILU

3.1.1 Conceitos Gerais

A plataforma *Inter-Language Unification* (ILU), desenvolvida pela Xerox e detalhadamente documentada em [44], trata essencialmente das interfaces entre módulos de software. Um módulo engloba uma determinada parte lógica de um programa, com uma elevada coesão interna e pouca interacção com outras partes. O ILU oferece meios para escrever uma interface orientada ao objecto para um determinado módulo. Esta interface é processada por várias ferramentas do sistema ILU, de modo a que se consiga um uso sistemático do módulo.

Os módulos podem ser partes de um programa, escritas na mesma linguagem ou não, ou partes de diferentes programas em execução em diferentes máquinas. Um módulo pode até ser um sistema distribuído implementado por vários programas em máquinas distintas. Pode ainda ter-se um módulo a ser usado simultaneamente por dois programas. O ILU proporciona todas as traduções e todas as tarefas de comunicação necessárias para que se possam usar todos estes tipos de módulos num único programa.

Dado que um dos requisitos no planeamento do ILU foi a utilização de normas já existentes, em vez de se criar qualquer coisa completamente nova, a plataforma ILU pode ser usada

para implementar serviços e clientes para protocolos como o RPC ou o Xerox Courier [45]. O ILU pode também ser utilizado para escrever serviços e clientes concordantes com a especificação CORBA.

Funcionamento

A abordagem usada no ILU é semelhante à dos sistemas baseados em RPC e à das muitas implementações da especificação CORBA; descrevem-se interfaces numa linguagem de especificação independente (neutra) de qualquer linguagem de programação. Nestas interfaces são descritos os tipos de dados e as excepções necessárias e definem-se métodos, nos objectos, para indicar a funcionalidade que se pretende exportar. As ferramentas que processam a descrição da interface produzem *stubs* para linguagens de programação específicas.

O código dos *stubs* é ligado[1] ao código da aplicação, a algum código que contém o suporte do ILU para a linguagem na qual a aplicação foi escrita e à biblioteca do núcleo[2] ILU que se encontra escrita em ANSI C. O esquema da figura 3.1 mostra as diferentes partes de um cliente ou servidor desenvolvido com base no ILU.

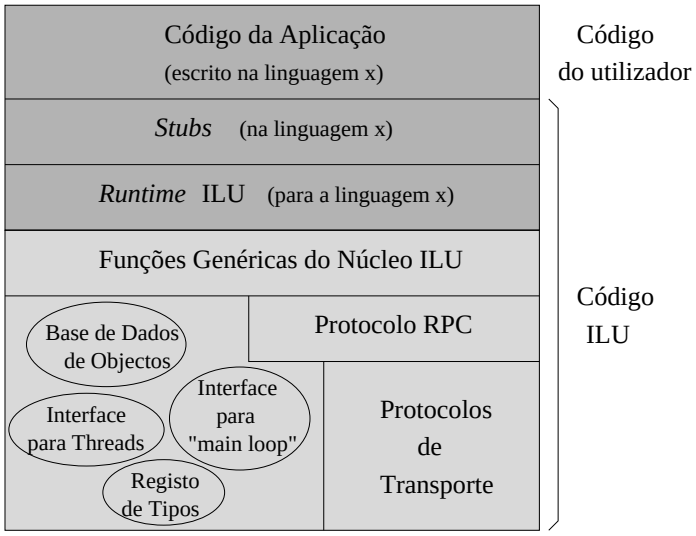

Figura 3.1: Estrutura de um programa desenvolvido com base no ILU.

[1] Do inglês *linked*.
[2] Do inglês *kernel*.

Objectos

O mecanismo principal de encapsulamento do ILU é o tipo de dados `object`, ou seja, o ILU é orientado ao objecto. Toda a funcionalidade de um módulo é exportada na forma de métodos que podem ser invocados numa instância de um objecto.

No que respeita a uma instância de um objecto ILU em particular, um módulo é designado por servidor se implementar os métodos desse objecto e é designado por cliente se invocar, mas não implementar, esses métodos. Um módulo pode portanto ser cliente de um objecto e servidor de outro. Os objectos podem ser passados como parâmetros de métodos e também podem ser devolvidos como resultados de invocações desses métodos.

Para um dado objecto ILU existirão vários objectos de uma particular linguagem de programação (a linguagem usada no desenvolvimento de determinada aplicação). Destes objectos, um é designado por objecto real[3] e implementa o objecto ILU em causa, fazendo portanto parte do servidor. Os métodos deste objecto são escritos pelo programador da aplicação, não sendo portanto gerados pelo ILU. Os restantes objectos são designados por objectos substitutos[4] e são usados pelos módulos clientes, quando o servidor se encontra noutro programa (e eventualmente numa outra máquina) ou utiliza uma representação distinta para os dados (módulos escritos em linguagens distintas, por exemplo).

O modelo de objectos oferecido pelo ILU dispõe de herança múltipla de interfaces. Assim, qualquer objecto pode ser especificado recorrendo a especificações já existentes. Note-se que o ILU apenas trata das interfaces; a implementação dos objectos poderá, ou não, ser efectuada em linguagens orientadas ao objecto, e no caso de o ser poderão usar-se todos os mecanismos de herança oferecidos por essa linguagem. Os subtipos (objectos especificados por herança) deverão disponibilizar todos os métodos descritos nos supertipos, juntamente com alguns métodos novos.

Instanciação de Objectos

Um cliente, para poder invocar métodos de um objecto ILU, deverá obter um objecto numa linguagem de programação específica (objecto substituto). Isto pode ser conseguido de duas formas:

- o cliente recebe, como resultado da invocação de um método de um objecto ILU, uma instância do objecto em causa;
- a partir de informação de endereçamento ou de designação[5] relativa ao objecto, são

[3] Do inglês *true object*.
[4] Do inglês *surrogate objects*.
[5] Do inglês *naming*.

usados mecanismos estandardizados para a sua instanciação.

A informação de endereçamento denomina-se *String Binding Handle* (SBH), e a biblioteca de *runtime* do ILU inclui procedimentos para obter um objecto na linguagem desejada com base no SBH.

A criação de qualquer instância de um objecto substituto implica alguma comunicação com o módulo servidor, com a finalidade de se estabelecer uma associação com o objecto real.

O processo de criação de uma instância de um objecto ILU pode ainda envolver um serviço de nomes, o qual permite aos servidores espalhados pela rede registarem instâncias (de objectos reais). O servidor regista a informação de designação e o correspondente SBH, enquanto os *stubs* do cliente se encarregam de consultar o serviço de nomes, para obter um SBH a partir de uma designação (nome).

O SBH constitui a representação em formato de cadeia de caracteres para uma dada referência de um objecto ILU. A conversão entre SBHs e referências de objectos é conseguida mediante a utilização de rotinas do sistema ILU[6]. Um SBH é composto por várias partes:

- o identificador do servidor (SID[7]), que identifica especificamente o servidor que implementa o objecto;
- o identificador da instância (IH[8]), que identifica o objecto no contexto de um determinado servidor;
- o identificador do tipo mais específico (MSTID[9]), que corresponde ao tipo de dados que garante a funcionalidade mínima para o objecto;
- a informação de contacto, que especifica uma ou mais formas de o cliente comunicar com o objecto.

Na hierarquia de tipos do ILU qualquer subtipo oferece a funcionalidade dos seus supertipos. Deste modo, ao instanciar um determinado objecto poder-se-á obter uma instância de um objecto com uma interface mais completa que a desejada. O objecto do tipo mais específico será aquele que não oferece nenhuma funcionalidade extra e o SBH contém esta informação no MSTID.

O par constituído pelo SID e pelo IH é conhecido como identificador do objecto (OID[10]).

[6] Note-se que este é um dos serviços do ORB da especificação CORBA.
[7] Server IDentifier
[8] Instance Handle
[9] Most Specific Type IDentifier
[10] Object Identifier

Conexões

Quando usado na construção de sistemas distribuídos, o ILU não tem a noção de conexão. Isto significa que o módulo invocado não possui qualquer referência do invocador e não é capaz de interagir com este, a não ser na devolução de respostas. O cliente pode identificar-se através de credenciais enviadas nos seus pedidos, mas isto não identifica o local de origem desses pedidos. Os protocolos que necessitem desta informação devem enviá-la explicitamente, como parâmetros dos métodos.

Serviço de Nomes Simplificado

O ILU dispõe de um serviço de nomes opcional e de grande simplicidade. Este mecanismo permite que um módulo publique um objecto, de forma a que outro módulo o possa importar conhecendo unicamente o seu OID. A interface deste mecanismo é formada por três operações: `Publish`, `Withdraw` e `Lookup`. A operação `Withdraw` é a inversa da operação `Publish`, ou seja, permite anular o registo criado na publicação de um objecto. A operação `Lookup` permite pesquisar os registos correspondentes a objectos publicados.

A implementação do serviço de nomes simplificado, fornecida com a versão actual do ILU (`2.0 alpha 8`), utiliza um directório de um sistema de ficheiros para armazenar informação relativa aos objectos publicados. Num sistema distribuído, este serviço de nomes implica a partilha, entre as várias máquinas que constituem o sistema, do directório referido, requerendo portanto um sistema de partilha de ficheiros.

3.1.2 Definição de Interfaces

As interfaces dos módulos podem ser definidas tanto na IDL apresentada na especificação CORBA como na linguagem de especificação de interfaces (ISL[11]) do ILU.

Uma interface definida através da ISL é composta por quatro tipos de instruções: cabeçalho da interface, declarações de tipos de dados, declarações de excepções e declarações de constantes.

Todos os identificadores usados na ISL são alfanuméricos, iniciando-se por uma letra e sem dois hífenes consecutivos. O ILU não distingue entre letras maiúsculas e minúsculas, mas preserva a integridade dos identificadores quando gera código para uma linguagem em particular. Estes identificadores são usados para os nomes dos tipos de dados, das excepções e das constantes.

Os nomes de quaisquer tipos de dados, excepções ou constantes são formados por duas partes – o identificador da interface e um identificador local (ao contexto da interface) –

[11] Interface Specification Language

separadas por um ponto, e pela ordem mencionada. Se o identificador da interface for omitido, o nome será interpretado no contexto da interface actual.

Cabeçalho da Interface

No cabeçalho da interface são indicados o nome da interface e uma lista de interfaces a incluir.

O nome da interface é usado em várias produções específicas de uma determinada linguagem de programação (aquando da geração de código pelas ferramentas do sistema ILU) para criar espaços de nomes, nos quais são declarados os tipos, as excepções e as constantes definidas na interface em causa.

Uma interface pode opcionalmente importar outras interfaces, com a finalidade de mencionar tipos de dados, excepções e constantes definidos nessas interfaces. Este é o motivo pelo qual se usam nomes compostos, em que a primeira parte corresponde ao identificador da interface. Em interfaces distintas podem ser usados identificadores iguais, para os vários tipos de dados, excepções e constantes, o que impede a combinação de várias interfaces, a não ser que se utilize informação adicional em cada identificador.

Declaração de Tipos de Dados

A declaração de tipos de dados na ISL do ILU obedece aos mesmos princípios que qualquer outra linguagem de programação. Assim, são disponibilizados tipos de dados básicos (`INTEGER`, `SHORT INTEGER`, `LONG INTEGER`, `CARDINAL`, `SHORT CARDINAL`, `LONG CARDINAL`, `BYTE`, `BOOLEAN`, `REAL`, `SHORT REAL`, `LONG REAL`, `CHARACTER`, `SHORT CHARACTER` e `NULL`) e construtores (`ARRAY`, `SEQUENCE`, `RECORD`, `UNION`, `OPTIONAL`, `ENUMERATION` e `OBJECT`).

Os vários tipos de dados básicos e os vários construtores foram seleccionados por analogia com linguagens de programação existentes. No entanto, o construtor `OPTIONAL` requer alguma atenção. Este construtor permite definir tipos de dados cujas variáveis assumem valores de um determinado tipo indicado na declaração, ou do tipo `NULL`.

Declaração de Objectos

O construtor `OBJECT` é o tipo de dados mais importante do ILU. De facto, este é o construtor que permite especificar os componentes – objectos – de uma aplicação distribuída[12].

Na declaração de um objecto interessam principalmente dois aspectos: os mecanismos de herança e a declaração de métodos. Na ISL, o construtor `OBJECT` permite indicar uma

[12]Embora o ILU não se destine exclusivamente ao desenvolvimento de aplicações distribuídas, como já foi referido, no contexto deste trabalho apenas interessa esse tipo de utilização.

lista de supertipos. Um objecto assim definido herda os métodos de todos os supertipos indicados.

Na declaração de métodos (uma lista) indicam-se, para cada método, o nome, uma lista de parâmetros (argumentos), o tipo de dados do resultado devolvido numa invocação e o conjunto de excepções que podem ser assinaladas durante o processo de invocação. A declaração de cada argumento consta de um nome – o nome do argumento – e de um tipo de dados, para além de uma direcção (parâmetro de entrada, de saída ou de entrada/saída). Note-se que os restantes tipos de dados (para além do tipo `OBJECT`) são usados apenas na declaração dos métodos, mais precisamente, para indicar o tipo de dados dos parâmetros e do resultado. Obviamente, também são usados na declaração de tipos compostos, através de construtores, mas esta será uma utilidade intermédia; o objectivo final será sempre definir tipos de dados para possibilitar a declaração de métodos de objectos.

Um método pode, opcionalmente, ser etiquetado com os qualificadores `FUNCTIONAL` e `ASYNCHRONOUS`. No primeiro caso, indica-se que o método é idempotente[13] para um determinado valor dos seus argumentos. Isto significa que o método irá devolver sempre o mesmo resultado para uma determinada avaliação[14] dos seus parâmetros, o que permite o uso de caches do lado do cliente.

Com o qualificador `ASYNCHRONOUS` consegue-se que a invocação do método visado termine antes da execução do código correspondente no objecto real, ou seja, o objecto substituto encarrega-se de passar o controlo para a entidade invocadora (cliente) logo após o envio dos argumentos do método. Estes métodos não podem devolver nenhum resultado. A passagem de informação entre o servidor e o cliente terá que ser efectuada com a invocação de métodos, ou seja, o servidor e o cliente deverão trocar de papéis.

Declaração de Excepções

Em ILU, as excepções são ocasionadas pelos métodos e permitem que situações de erro sejam assinaladas à entidade invocadora. Isto significa que, na invocação de métodos de objectos remotos, o cliente não só pode receber resultados associados à execução das operações inerentes a esses métodos como também poderá receber alguma indicação relativamente a erros que impossibilitaram a execução com sucesso dessas operações. Note-se que um método poderá ocasionar excepções distintas, conforme os diferentes tipos de erro que podem ocorrer na tentativa da sua execução.

As excepções, para além de um nome que as identifica univocamente no contexto da interface, possuem um tipo de dados associado. Deste modo, para além de se poder

[13] Os efeitos laterais de uma invocação são exactamente os mesmos que os resultantes de muitas invocações.

[14] Do inglês *evaluation*.

assinalar a ocorrência de determinado erro, podem-se ainda indicar alguns parâmetros relativos a esse erro (motivo, explicação adicional, etc.).

O processo de geração de excepções pode ser desencadeado pelo código do ILU ou pelo código do utilizador (ver figura 3.1). No primeiro caso, as excepções destinam-se a indicar a impossibilidade de activar o método remoto (problemas de comunicação com a máquina onde se encontra a implementação do objecto, por exemplo).

No segundo caso, apesar de o método iniciar a sua execução, é detectado que determinadas operações não podem ser desencadeadas. Por exemplo, um método que receba como parâmetros dois números inteiros e, durante a sua execução, necessite de determinar o quociente entre eles, poderá ocasionar uma excepção, no caso de o divisor possuir o valor zero.

Declaração de Constantes

A especificação de uma interface pode conter declarações de constantes, as quais podem ser utilizadas nos servidores e nos clientes. Este é um mecanismo de declarar constantes de forma independente, em relação às linguagens de programação utilizadas no desenvolvimento de clientes e servidores.

As constantes da ISL do ILU podem ser dos tipos `INTEGER`, `CARDINAL`, `BYTE` e `ilu.CString`. O tipo de dados `ilu.CString` é, na verdade, um tipo composto definido na interface `ilu` (daí o nome composto começado por `ilu`), que é importada de forma automática em todas as interfaces que se definam.

3.1.3 Utilização do ILU em C++

Conforme referido anteriormente, o ILU disponibiliza ferramentas para processar as definições de interfaces, tendo em vista uma determinada linguagem de programação. De entre as linguagens suportadas pelo ILU, ou seja, no universo de linguagens para as quais é possível gerar *stubs*, com a finalidade de implementar ou invocar métodos, o C++ tem especial importância, dada a sua grande popularidade[15] e visto que o paradigma da orientação ao objecto tem grandes vantagens no desenvolvimento de aplicações distribuídas.

No desenvolvimento de uma aplicação distribuída, numa determinada linguagem de programação e utilizando o ILU para interligação dos vários componentes, é necessário conhecer:

- a correspondência entre a ISL do ILU e a linguagem em causa;

[15] Para além de existirem vários compiladores para C++, pode encontrar-se uma vasta gama de produtos comerciais desenvolvidos nesta linguagem.

- os procedimentos necessários para utilizar um determinado módulo;

- a forma de implementar um servidor de maneira a que os clientes possam utilizar os seus serviços.

Correspondência entre a ISL e o C++

Segundo a Xerox, a utilização do ILU com o C++ será eventualmente compatível com a especificação CORBA, ou seja, a criação de *stubs* e a construção de nomes (ou identificadores) deverão ser concordantes com as correspondências para C++ especificadas pelo OMG[16].

Os identificadores da ISL são transformados em identificadores do C++ através da substituição dos hífenes por caracteres de sublinhado. Os nomes dos tipos, excepções e constantes são construídos a partir do nome da interface e dos nomes destes, usando-se, respectivamente, os infixos "`_T_`", "`_E_`" e "`_C_`". Por exemplo, o tipo de dados designado por `cabecalho-factura`, definido na interface `facturacao`, cujo nome composto seria, na ISL, `facturacao.cabecalho-factura`, é transformado em `facturacao_T_cabecalho_factura`.

Na conversão de tipos de dados apenas o tipo `SEQUENCE` requer atenção especial, dado que os restantes possuem correspondência directa, ou quase directa, para tipos de dados do C++. O tipo `SEQUENCE` é transformado numa classe com métodos para inserir, remover e processar elementos, os quais são armazenados numa lista ligada.

O tipo `OBJECT` oferecido pelo ILU torna-se uma classe do C++. Esta classe é subclasse de uma pré-definida: `iluObject`.

As excepções são suportadas adicionando um argumento a cada método. Assim, para além dos argumentos especificados na interface, ter-se-á um outro, que por sinal é o primeiro argumento do método, no qual serão assinaladas possíveis excepções. Refira-se que este argumento será obrigatoriamente um argumento de saída, dado que o objectivo é passar informação do servidor para o cliente. O cliente deverá testar este parâmetro no final de cada invocação, por forma a averiguar se ocorreu algum erro. Só assim poderão ser usados com garantia os eventuais parâmetros de saída desse método, bem como o resultado devolvido.

As constantes declaradas numa interface ILU são implementadas com recurso à directiva `#define` do C++.

[16] Alguns componentes do ILU encontram-se ainda em desenvolvimento.

Utilização de Módulos em C++

Um cliente desenvolvido em C++ pode obter uma instância de um objecto ILU de três formas:

- a partir do SBH, recorrendo à função `ILUCreateFromSBH`, a qual é gerada para todas as subclasses da classe `iluObject` declarada nos *stubs*;
- através da função `Lookup` da classe `iluObject`, que pesquisa o serviço de nomes simplificado, com base num OID;
- recebendo uma instância directamente como resultado ou como parâmetro de saída de um método de um outro objecto.

Implementação de Módulos em C++

Para cada objecto declarado numa interface ILU, as ferramentas do ILU geram uma classe C++, conforme mencionado anteriormente. Para implementar o objecto real é necessário derivar uma subclasse e reescrever os seus métodos, em especial os métodos declarados na interface. Esta classe será subclasse de uma gerada pelo ILU, correspondente ao objecto declarado na interface, que é, por sua vez, subclasse da classe pré-definida `iluObject`.

As instâncias dos objectos reais são disponibilizadas aos módulos clientes (exportadas) pelos servidores núcleo[17]. Estes permitem o acesso aos objectos através de portos que, fundamentalmente, tratam dos pormenores de comunicação (protocolos de transporte, endereços de máquinas, etc.).

Na criação de instâncias de objectos reais é obrigatório especificar um servidor núcleo e um identificador de instância. Isto faz-se reescrevendo os métodos `ILUGetServer` e `ILUGetInstanceHandle`, que são herdados da classe `iluObject`.

O servidor núcleo é representado, em C++, pela classe `iluServer`, cujo construtor aceita um identificador como parâmetro. O método `iluServer::AddPort` permite adicionar portos a um servidor núcleo, através dos quais este será contactado.

Note-se que o SID, pertencente ao OID de um objecto ILU, é o identificador do servidor núcleo que disponibiliza esse objecto.

Para permitir que os clientes de um módulo encontrem os objectos por este exportados, é necessário efectuar o registo desses objectos. Desde a versão 1.6, e até à versão 2.0, o ILU suporta um mecanismo experimental, que é o serviço de nomes simplificado. Em futuras versões este mecanismo poderá ser alterado, mas a sua funcionalidade deverá no

[17] Do inglês *kernel servers*.

entanto manter-se. Em C++, o processo de registo de um objecto real é desempenhado pelo método `ILUPublish`, herdado da classe `iluObject`.

3.2 Escolha do ILU

Tal como já foi referido, o ILU serviu de base ao trabalho aqui apresentado. A justificação para a sua escolha foi intencionalmente deixada para depois da sua apresentação, para melhor realçar alguns dos aspectos tidos em consideração.

Tendo como objectivo disponibilizar uma plataforma (conjunto de ferramentas) para a construção de aplicações distribuídas, com suporte para desconexão e replicação, e reconhecendo a impossibilidade de desenvolver de raiz uma solução deste tipo, optou-se por adicionar alguns mecanismos a uma tecnologia já existente e bem implantada. Consequentemente, houve necessidade de seleccionar uma tecnologia que servisse de ponto de partida para a plataforma pretendida.

3.2.1 Motivos para Eleição

Os motivos que levaram à escolha do ILU foram os seguintes:

- a possibilidade de usar livremente[18] o ILU, sendo este distribuído com todo o código fonte, o que permite eventuais alterações para incluir comportamentos desejados;
- as abstracções oferecidas, nomeadamente a orientação ao objecto e a especificação de interfaces;
- a documentação disponível, tanto do ponto de vista da utilização como do ponto de vista da sua concepção, onde se destaca uma lista de distribuição de correio electrónico – `ILU.parc@xerox.com` – na qual se discutem os mais variados problemas relacionados com o sistema ILU;
- a possibilidade de compilar e usar as suas ferramentas numa vasta gama de sistemas operativos (Linux, SunOS, OSF OS, NT, Win95, etc.);
- o suporte a variadas linguagens de programação (Common Lisp, Python, ANSI C, C++, Modula-3, etc.);
- a razoável interoperação entre aplicações produzidas com base no ILU e aplicações desenvolvidas segundo outras tecnologias, devido à utilização de normas, nomeadamente a concordância, mesmo que parcial, com a especificação CORBA;

[18] Obviamente, a adopção de soluções comerciais, para além dos custos inerentes, dificultaria a divulgação e a distribuição do sistema final.

- a escolha do ILU para projectos de relevância internacional, como é o caso de: Digital Libraries Project (Stanford University), GeoScope (Universal Spatial Data Access Consortium), ILU and Java (O/SPACE), etc.;
- a experiência na utilização do ILU por parte do Grupo de Sistemas Distribuídos da Universidade do Minho (GSD), no seio do qual este trabalho é desenvolvido.

3.3 Exemplo de Utilização

O ILU possibilita o desenvolvimento de aplicações distribuídas segundo o modelo Cliente-Servidor, mediante a utilização de abordagens orientadas ao objecto. Deste modo, qualquer aplicação a desenvolver deverá ser pensada em termos de objectos, mantidos num servidor, e operações – métodos – que esses objectos disponibilizam e que os clientes podem invocar.

Assim, depois de identificada a funcionalidade que um servidor deve exportar, começa-se por especificar uma ou mais interfaces. Note-se que um servidor poderá ser acedido por diferentes tipos de clientes, cada um usando uma interface distinta. Seguidamente, implementa-se o servidor, nomeadamente os objectos e métodos que permitem suportar as várias operações indicadas nas interfaces. Finalmente, podem ser desenvolvidos clientes, conhecendo apenas as especificações das interfaces disponibilizadas pelo servidor.

Posto isto, passar-se-á ao desenvolvimento de uma solução para o seguinte caso de estudo:

> Uma empresa possui um armazém e vários locais de venda (lojas) espalhados por uma cidade. Pretende-se que toda a informação relativa aos produtos e à comercialização destes se encontre num servidor de bases de dados localizado no armazém e que as vendas de cada loja actualizem o estado desse servidor com a maior prontidão possível.

Poder-se-á conceber uma solução integrada de facturação, ou seja, uma aplicação baseada numa base de dados relacional, capaz de controlar as quantidades de cada produto, as facturas correspondentes às vendas e os dados relativos aos clientes. O respectivo modelo de dados está representado na figura 3.2 e constitui uma aproximação simplificada para este problema, apenas para efeito de demonstração dos conceitos adjacentes à plataforma ILU.

3.3.1 Especificação da Interface

A especificação da interface de um servidor pressupõe, antes de mais, o conhecimento do funcionamento deste. Note-se que, mesmo nas situações em que a descrição da interface

é escrita antes de se desenvolver o serviço em causa, terá que existir uma planificação das tarefas a desempenhar pelo servidor e da forma como estas serão suportadas.

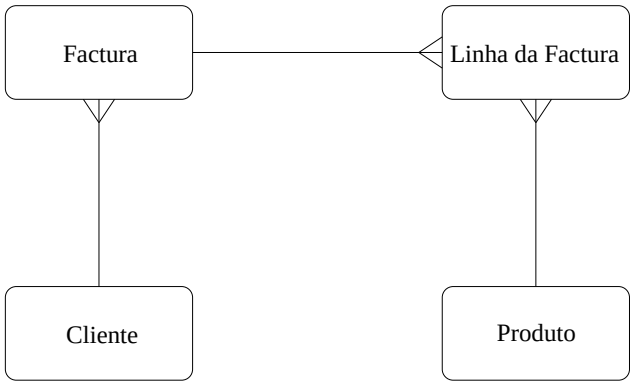

Figura 3.2: Diagrama Entidade-Relação para uma aplicação de facturação.

Apresenta-se a seguir uma possível especificação em ILU da interface que deve ser oferecida pelo servidor aos clientes instalados nos locais de venda. Fundamentalmente, trata-se de identificar as operações que o servidor deverá permitir desencadear sobre o modelo de dados apresentado e os objectos usados para agrupar tais operações.

```
INTERFACE facturacao;

TYPE codigo-produto = ARRAY OF 14 SHORT CHARACTER;
TYPE valor-quantidade = SHORT CARDINAL;
TYPE valor-monetario = SHORT REAL;
TYPE codigo-cliente = ARRAY OF 6 SHORT CHARACTER;
TYPE numero-contribuinte = CARDINAL;
TYPE numero-factura = RECORD
  loja : BYTE,
  sequencia : CARDINAL
END;
TYPE data-dma = RECORD
  dia : BYTE,
  mes : BYTE,
  ano : SHORT CARDINAL
END;
TYPE cabecalho-factura = RECORD
  numero : numero-factura,
  cliente : codigo-cliente,
```

```
  data : data-dma,
  total : valor-monetario
END;
TYPE linha-factura = RECORD
  produto : codigo-produto,
  quantidade : valor-quantidade,
  preco-unitario : valor-monetario
END;
TYPE linhas-factura = SHORT SEQUENCE OF linha-factura;

EXCEPTION produto-inexistente;
EXCEPTION stock-insuficiente;
EXCEPTION cliente-inexistente;
EXCEPTION plafond-insuficiente;
EXCEPTION factura-inexistente;

TYPE produtos = OBJECT
  METHODS
    stock (codigo : codigo-produto) : valor-quantidade
      RAISES produto-inexistente END,
    preco (codigo : codigo-produto) : valor-monetario
      RAISES produto-inexistente END,
    saida-stock (codigo : codigo-produto, quantidade : valor-quantidade)
      RAISES produto-inexistente, stock-insuficiente END
  END;

TYPE clientes = OBJECT
  METHODS
    contribuinte (codigo : codigo-cliente) : numero-contribuinte
      RAISES cliente-inexistente END,
    plafond (codigo : codigo-cliente) : valor-monetario
      RAISES cliente-inexistente END,
    gasta-plafond (codigo : codigo-cliente, valor : valor-monetario)
      RAISES cliente-inexistente, plafond-insuficiente END,
    repoe-plafond (codigo : codigo-cliente, valor : valor-monetario)
      RAISES cliente-inexistente END
  END;
```

```
TYPE facturas = OBJECT
  METHODS
    cria-cabecalho (cabecalho : cabecalho-factura),
    cria-linhas (numero : numero-factura, linhas : linhas-factura)
      RAISES factura-inexistente END,
    paga-factura (numero : numero-factura)
      RAISES factura-inexistente END
  END;
```

Na descrição desta interface podem-se identificar:

- o cabeçalho da interface, que contém apenas o nome – `facturacao`;
- declarações de tipos de dados, os quais são utilizados na declaração de métodos, nomeadamente na definição de argumentos e valores de retorno;
- declarações de excepções destinadas a especificar possíveis anomalias a assinalar aquando da invocação de determinados métodos;
- definição de objectos e respectivos métodos – a interface do servidor propriamente dita.

Os objectos declarados – `produtos`, `clientes` e `facturas` – têm por finalidade permitir a interacção dos clientes com a base de dados do servidor, para o modelo proposto (figura 3.2). Refira-se que esta decomposição não pode ser considerada completamente orientada ao objecto. De facto, não existirá a instanciação de objectos; o servidor terá uma única instância de cada objecto e os vários clientes invocarão métodos sobre cada uma dessas instâncias. No entanto, esta é a abordagem mais comum quando se trata de aplicações Cliente-Servidor, onde o servidor é responsável por armazenar persistentemente determinado estado.

A forma como todos os métodos serão usados pelos clientes, por fim a processar facturas, ficará clara nas exposições relativas à construção do servidor e do cliente.

3.3.2 Construção do Servidor

O servidor deverá disponibilizar instâncias dos objectos reais através de um ou mais servidores núcleo. Para isso é necessário instanciar um objecto da classe `iluServer` – classe que representa o servidor núcleo – e adicionar-lhe um porto, para que este possa ser contactado. Isto efectua-se, em C++, com o seguinte código:

```
class iluServer server ("srv-facturacao", NULL);
server.AddPort (NULL, NULL, ilu_TRUE);
```

Observe-se a atribuição do SID – `srv-facturacao` – ao servidor núcleo, no momento da sua criação. Este identificador deverá designar univocamente o servidor em causa.

Seguidamente, será necessário instanciar os objectos que serão disponibilizados através da publicação dos seus identificadores. Tal como referido na secção 3.1.3, a implementação dos objectos reais faz-se derivando subclasses das classes geradas pelas ferramentas do ILU, para cada objecto especificado na interface. Assim, ter-se-á o seguinte código:

```
class facturacao_T_produtos_impl produtos ("obj-produtos", &server);
if (!produtos.ILUPublish())
  {
    cerr << "Erro na publicação do objecto obj-produtos.\n";
    exit (1);
  }
cout << produtos.ILUStringBindingHandle() << "\n";
```

Observe-se a indicação do IR do objecto – `obj-produtos` – e do servidor núcleo a utilizar[19]. O método `ILUPublish` permite registar o objecto no serviço de nomes simplificado, enquanto que o método `ILUStringBindingHandle` possibilita descobrir o SBH do objecto.

Finalmente, resta colocar o servidor núcleo em funcionamento, mediante a invocação do método `Run`:

```
iluServer::Run();
```

A partir do momento desta invocação, o programa servidor ficará indefinidamente à espera de pedidos de clientes. De referir que a utilização de mais do que um servidor núcleo obriga ao recurso à programação concorrente.

Na implementação dos objectos reais é necessário reescrever alguns métodos, para além daqueles especificados na interface. Assim, para o objecto `produtos`, por exemplo, a subclasse a derivar será a seguinte:

```
class facturacao_T_produtos_impl : public facturacao_T_produtos
{
public:
```

[19] Num mesmo servidor, poder-se-ão instanciar vários servidores núcleo, por forma a que cada objecto utilize um em particular.

```
  facturacao_T_produtos_impl (char *instance_handle, iluServer *server);

  virtual char * ILUGetInstanceHandle ();
  virtual iluServer * ILUGetServer ();

  virtual facturacao_T_valor_quantidade
  stock (facturacaoStatus *status, facturacao_T_codigo_produto codigo);
  virtual facturacao_T_valor_monetario
  preco (facturacaoStatus *status, facturacao_T_codigo_produto codigo);
  virtual void
  saida_stock (facturacaoStatus *status,
               facturacao_T_codigo_produto codigo,
               facturacao_T_valor_quantidade quantidade);
private:
  char *ourInstanceHandle;
  iluServer *ourServer;
};
```

Os métodos `ILUGetInstanceHandle` e `ILUGetServer` destinam-se a devolver, respectivamente, o IH do objecto e o seu servidor núcleo, os quais são argumentos do construtor da classe. Estes métodos são usados pelo código do ILU no momento do registo do objecto.

Os métodos `stock`, `preco` e `saida_stock` correspondem aos indicados na interface e a sua implementação terá como fim a interacção com a base de dados. O primeiro argumento, presente em todos os métodos sem que seja explicitamente declarado na interface, destina-se à indicação de excepções. Para o método `stock`, por exemplo, no caso de não existir o produto correspondente ao código recebido como parâmetro, será indicada a excepção `produto_inexistente`, declarada na interface. A codificação do método em causa será a seguinte:

```
facturacao_T_valor_quantidade
facturacao_T_produtos_impl::stock (facturacaoStatus *status,
                                   facturacao_T_codigo_produto codigo)
{
  static facturacao_T_valor_quantidade valor_stock;

  if (!existe_produto (codigo))
    status->returnCode = facturacao_E_produto_inexistente;
  else
    valor_stock = valor_stock_produto (codigo);
```

```
  return (valor_stock);
}
```

Refira-se ainda que a compilação do código do servidor requer alguns ficheiros produzidos pelo *stubber* do ILU (neste caso o programa `c++stubber`). Estes ficheiros são:

- `facturacao.hh`, que contém as definições das classes para os tipos e operações definidos na interface;
- `facturacao.cc`, que contém código genérico para o suporte à operação de clientes e servidores e código destinado à invocação de métodos remotos (o qual é usado apenas nos clientes);
- `facturacao-server-stubs.cc`, que contém código destinado exclusivamente aos servidores, para recepção de invocações e processamento das respectivas respostas.

3.3.3 Construção do Cliente

Com base na descrição da interface apresentada na secção 3.3.1, e depois de terem sido focados os principais aspectos do desenvolvimento de um servidor, expõem-se agora os passos mais relevantes da construção de um cliente.

Para um determinado serviço, e entenda-se serviço como o conjunto de funcionalidades exportado por um servidor, é possível desenvolver uma infinidade de clientes, correspondendo cada um a uma interface distinta com o utilizador[20]. Cada cliente poderá ainda desencadear um tipo de interacção particular com o servidor. Por outras palavras, diferentes implementações de um dado cliente poderão combinar de forma distinta as primitivas para interacção com o servidor, as quais se encontram especificadas na sua interface, dependendo do algoritmo utilizado.

No caso em análise, os programas instalados nas lojas, independentemente da sua apresentação e interacção com o operador, deverão invocar as operações disponibilizadas pelo servidor, por forma a reflectirem as alterações associadas ao processamento de vendas. Assim, para além das triviais operações de consulta – quantidade existente em armazém e preço de um determinado produto e número de contribuinte e limite de crédito de um cliente – serão invocadas operações de actualização – `saida-stock` e `gasta-plafond` – por cada produto vendido, ou seja, por cada linha da factura[21]. No final do processamento de uma venda, haverá lugar à inserção de uma factura na base de dados do servidor, através

[20] Não confundir interface de um serviço com interface de um programa com o utilizador.

[21] Estas operações deverão ser invocadas no momento em que o operador introduz os dados relativos a cada linha da factura para impedir que, para um determinado produto, sejam vendidas quantidades inexistentes ou sejam processadas vendas, para um dado cliente, de valor superior ao seu limite de crédito.

das operações `cria-cabecalho` e `cria-linhas`. A operação `repoe-plafond` será usada no momento em que o cliente pagar uma determinada factura, assumindo que podem ser efectuadas vendas a crédito.

A fim de utilizar as operações oferecidas pelo servidor, o cliente deverá, em primeiro lugar, obter instâncias dos objectos que constam da interface desse servidor.

Dado que, na implementação do servidor, os objectos foram registados no serviço de nomes simplificado, ter-se-á o seguinte código para instanciar o substituto que interactua com o objecto produtos do servidor:

```
facturacao_T_produtos *produtos =
  (facturacao_T_produtos *) iluObject::Lookup ("srv-facturacao",
              "obj-produtos", facturacao_T_produtos::ILUClassRecord);
if (produtos == NULL)
{
  cerr << "Objecto srv-facturacao/obj-produtos desconhecido.\n";
  exit (1);
}
```

O método `Lookup` pesquisa o serviço de nomes com base no SID e IH indicados, devolvendo um objecto substituto capaz de interactuar com a implementação residente no servidor. O terceiro argumento deste método destina-se a indicar o tipo do objecto pretendido. Visto que no sistema ILU os subtipos implementam a funcionalidade dos supertipos, este processo de pesquisa poderá devolver um objecto com uma interface mais rica do que a pretendida. Assim, o tipo assinalado no método `Lookup` servirá para estabelecer o ponto da hierarquia de tipos a partir do qual são satisfeitas as necessidades do cliente.

A invocação de métodos remotos faz-se de forma natural, através do correspondente objecto substituto, como se este se tratasse de um apontador para um objecto vulgar implementado localmente. A única diferença reside no facto de ser exigido um argumento adicional destinado à indicação de excepções[22]. Para obter o preço de um determinado produto, por exemplo, utiliza-se o seguinte código:

```
facturacaoStatus status;
facturacao_T_codigo_produto codigo;
facturacao_T_valor_monetario preco;

...
```

[22]Este argumento pode ainda ser usado para passar algum contexto para o servidor.

```
preco = produtos->preco (&status, codigo);
if (status.returnCode != NULL)
  {
    /* servidor inacessivel ou produto inexistente */
    ...
  }
```

De referir ainda que os tipos de dados usados na declaração de variáveis, nomeadamente `facturacaoStatus`, `facturacao_T_codigo_produto` e `facturacao_T_valor_monetario`, são gerados automaticamente pelo *stubber* do ILU, a partir da descrição da interface.

Capítulo 4

Anotações de Desconexão

Neste capítulo apresenta-se a primeira contribuição deste trabalho – as anotações de desconexão. Em primeiro lugar, introduz-se o modelo de agentes, o qual possibilita a operação desconectada de clientes. A seguir surgem as anotações à linguagem de definição de interfaces do ILU, que têm por fim a construção de agentes de forma sistemática e automática. Finalmente, são focados os pormenores da implementação do tradutor das anotações.

4.1 Modelo de Agentes

A reestruturação de serviços clássicos, desenvolvidos sem a preocupação da tolerância a faltas, levou à divulgação do modelo Cliente-Agente-Servidor, já apresentado na secção 2.3.2. De facto, esta abordagem, para além do sucesso obtido na adaptação de soluções já existentes tendo em vista a computação nómada, cria boas perspectivas para a construção de novas aplicações, dado que pode ser vista como uma extensão ao modelo Cliente-Servidor, que é bastante bem aceite pela generalidade dos programadores.

4.1.1 Funções do Agente

A utilização do modelo CAS pode passar, por exemplo, pela alteração do código do cliente, de modo a suprimir algumas invocações em determinados casos e assim permitir alguma funcionalidade adicional quando o servidor não se encontrar acessível. Se se tratar de uma aplicação nova (construída de raiz), ao introduzir o código normal para invocação de métodos remotos, será acrescentado código para se efectuarem processamentos adicionais, resultando algo semelhante à adaptação de uma aplicação já existente. Este procedimento tem por objectivo adicionar ao cliente um agente, que permitirá suportar períodos de desconexão ou faltas dos servidores ou dos meios de comunicação. Note-se que o agente

não é mais que um conjunto de porções de código que acompanham as normais instruções de invocação de métodos remotos, pelo que as suas tarefas basicamente são:

- averiguar se determinada invocação pode ou não ser efectuada, podendo para tal recorrer a informação relativa ao estado do cliente (desconectado ou não), a dados respeitantes ao estado da rede ou dos servidores, caso isso seja possível, ou, no pior dos casos, ao erro devolvido na tentativa de efectuar essa invocação, concluindo portanto que tal invocação não deveria ter sido efectuada;

- ponderar a necessidade de se efectuar certas invocações, mesmo quando possíveis, de maneira a reduzir ao mínimo a utilização dos meios de comunicação;

- processar os resultados de uma invocação remota efectuada com sucesso, de modo a "aprender" a substituir o servidor quando necessário e possível;

- desencadear processamentos locais que visem obter resultados semelhantes aos da execução de um método de um objecto remoto, tendo como base os resultados obtidos em situações anteriores;

- na iminência de uma desconexão, e dado que o acto de desconexão é voluntário, podendo existir um aviso prévio, despoletar um conjunto de operações que interactuem com o servidor e permitam suportar o período de desconexão que se avizinha (uma espécie de "aprendizagem" forçada);

- depois de restabelecida a conectividade com o servidor, efectuar as devidas interacções com este, por forma a reflectir aquelas operações, desempenhadas pelo agente durante o período de inacessibilidade do servidor, que levam à alteração do estado do servidor.

4.1.2 Localização do Agente

A inserção do agente no código do cliente apresenta algumas desvantagens. Na verdade, esta operação nem sempre é possível, pois envolve o acesso ao código fonte dos programas. Portanto, esta abordagem não se adequa ao processo de conversão de aplicações desenvolvidas segundo o modelo Cliente-Servidor.

Uma outra solução seria incluir o código do agente nos *stubs*, já que estes contêm código respeitante ao processo de invocação de todas as operações remotas, mas, mesmo assim, ainda seria necessário possuir o código objecto do cliente. No entanto, dado que os *stubs* constituem um módulo bem delimitado e são gerados de forma automática e disciplinada (a partir da especificação de uma interface), poder-se-iam adoptar técnicas para automatizar

este processo. Note-se que a alteração do código fonte de uma aplicação, mesmo tratando-se de pequenos acréscimos, na maior parte dos casos, só é possível com o auxílio do programador dessa aplicação.

No caso do ILU, visto que o código fonte de todas as ferramentas se encontra disponível, seria possível alterar o *stubber* – programa que gera os *stubs* a partir de uma especificação escrita na ISL – de maneira a conseguir uma ferramenta capaz de produzir *stubs* com suporte para desconexão incluído. Esta abordagem foi utilizada em [46], onde foi alterado o gerador de *stubs* do RPC da Sun, por forma a interactuar com um sistema de comunicação em grupo. Em [47] é apresentado um mecanismo de inserção automática de testes de desempenho em aplicações distribuídas, baseado na alteração do compilador da IDL usada no *Distributed Computing Environment* (DCE[1]). Poder-se-ia ainda pensar numa nova ferramenta capaz de introduzir alterações nos *stubs* gerados pela via normal, evitando-se assim a modificação do *stubber*.

Esta solução é atractiva do ponto de vista do resultado final, uma vez que permite acrescentar aos *stubs* tarefas relacionadas com a tolerância a faltas, tratando já estes de pormenores relativos à comunicação entre sistemas. No entanto, apresenta uma grande desvantagem: obriga a conhecer detalhadamente os pormenores de funcionamento e implementação do *stubber* ou, na segunda vertente, impõe um estudo aprofundado sobre a estrutura dos *stubs* produzidos, por forma a se poderem introduzir modificações. Apesar de toda a documentação disponível sobre o ILU, esta informação não é fácil de conseguir, havendo sempre necessidade de efectuar alguma re-engenharia, a fim de compreender determinados aspectos.

Deste modo, neste trabalho opta-se por separar fisicamente o agente do cliente, ou seja, considerar o agente como um outro processo, com o qual o cliente interage. A figura 4.1 esquematiza o modelo Cliente-Agente-Servidor, com o agente como entidade independente.

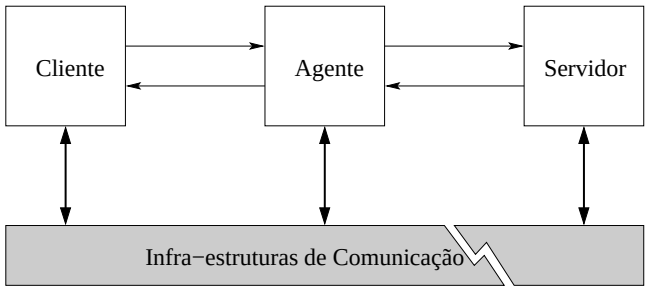

Figura 4.1: Modelo Cliente-Agente-Servidor.

[1]O DCE oferece uma plataforma para desenvolvimento de aplicações distribuídas baseada em RPCs.

4.1.3 Integração em Aplicações já Existentes

Nesta abordagem, o agente age como servidor, do ponto de vista do cliente, exportando precisamente a mesma interface que o verdadeiro servidor. Por outras palavras, o cliente deixa de invocar métodos de objectos implementados pelo servidor e passa a invocar métodos de objectos disponibilizados pelo agente. Esta troca só é possível se o cliente permitir indicar qual o servidor a utilizar ou se houver a possibilidade de alterar o registo do servidor (alterar os identificadores do servidor, para que o agente possa ocupar o seu lugar). A maior parte dos clientes permite indicar no momento de arranque (através de parâmetros na linha de comando, por exemplo) a identificação dos servidores a usar (o SID e eventualmente o IH, no caso de aplicações desenvolvidas com base no ILU). Na alteração do registo do servidor, o único entrave que se levanta prende-se com a operação de possíveis clientes pelo processo normal, ou seja, clientes que não pretendem usar os serviços de um agente e que apenas conhecem os antigos identificadores do servidor.

Os objectos exportados pelo agente interagem com os seus homólogos, exportados pelo servidor, ou seja, os objectos do agente são clientes dos serviços disponibilizados pelo servidor e desempenham as tarefas anteriormente mencionadas. Note-se que a execução de qualquer método do servidor passará pela execução de um método correspondente no agente; todas as operações executadas pelo servidor são desencadeadas com um nível de indirecção. Obviamente, este procedimento origina uma penalização no desempenho das aplicações, mas, por outro lado, é conseguida uma maior disponibilidade para essas mesmas aplicações.

O agente invoca métodos do servidor apenas nos casos indispensáveis, dado que a interacção entre estes nem sempre é possível ou desejável. As operações que o servidor executa, a pedido do agente, são aquelas que o cliente desencadearia no caso de não existir o agente, mas de uma forma controlada, ou seja, sem que exista obrigatoriamente uma relação directa entre os métodos invocados pelo cliente e os invocados pelo agente, tendo em vista a minimização da informação transferida e o suporte a desconexões. Eventualmente, o agente poderá invocar operações que o cliente não invocava, com o objectivo de oferecer um melhor suporte à desconexão. Isto significaria que o cliente não usava toda a funcionalidade oferecida pelo servidor, para o seu funcionamento normal (sem tolerância a faltas). Daí que a figura 4.1 indique interfaces distintas (I_1 e I_2) entre o cliente e o agente e entre o agente e o servidor.

Nesta particularização do modelo CAS, o agente é um programa servidor e cliente em simultâneo, que poderá estar instalado na mesma máquina que o cliente ou numa outra qualquer. O cliente interage com o agente através dos mecanismos oferecidos pelo ILU, tal como se tratasse da interacção com um servidor vulgar.

Embora o agente possa ser instalado num ponto qualquer da rede, dever-se-á ter em consideração que a inoperação do agente não deverá ter consequências no funcionamento do sistema, caso contrário ter-se-ia introduzido um novo ponto de falha, em vez de se solucionar o problema existente. Por outras palavras, o agente poderá ficar inoperacional ou inacessível apenas quando o cliente não estiver em funcionamento, ou pelo menos deverá estar acessível mais vezes que o servidor. Portanto, o agente deve ser instalado na mesma máquina que o cliente ou numa máquina com menor probabilidade de falha ou de isolamento que a do servidor, justificando-se esta segunda opção sempre que o processamento do agente seja tal que necessite de recursos não disponíveis no sistema onde se encontra o cliente. Os sistemas portáteis, por exemplo, devido às limitações que possuem, poderão não ter capacidades suficientes para desempenhar algumas tarefas necessárias a um determinado agente.

O modelo apresentado possibilita a utilização de clientes já existentes, sem se efectuarem quaisquer alterações. Isto abre boas perspectivas para a transformação de serviços ou aplicações desenvolvidos mediante o modelo Cliente-Servidor e sem a preocupação de tratar o problema da desconexão ou eventual inacessibilidade do servidor. Em relação à construção de novas aplicações, este modelo oferece uma boa modularidade devido à separação do agente, o qual pode até ser desactivado e activado sempre que se queira. Assim, as aplicações construídas segundo este modelo podem também funcionar segundo o modelo Cliente-Servidor.

Note-se que na abordagem aqui efectuada nunca é colocada a possibilidade de modificação dos servidores. Na verdade, o problema reside no processo de acesso às funcionalidades exportadas pelo servidor e não na forma como estas são implementadas ou na insuficiência do leque de funcionalidades que cada servidor oferece.

De referir ainda que, embora este modelo tenha como seguimento a concretização para o sistema ILU, poder-se-iam aplicar as mesmas ideias a outros mecanismos de comunicação Cliente-Servidor, como é o caso do RPC da Sun.

4.2 Extensão da ISL do ILU

Adoptado o modelo CAS como meio para garantir uma maior disponibilidade das aplicações distribuídas, nomeadamente na presença de sistemas móveis, e estabelecido o modo de operação do agente, é agora necessário criar mecanismos para obter agentes de forma sistemática e o mais automatizada possível.

A geração automática de código, mesmo que parcial, é sempre vantajosa, pois elimina possíveis pontos de introdução de erros e evita a escrita de determinadas rotinas que, apesar da sua simplicidade, constituem uma tarefa demasiado repetitiva e pouco atractiva

para o programador.

4.2.1 Especificação de Agentes

Até ao momento não são conhecidas estratégias para geração automática de agentes. Existem, como já foi referido, soluções para casos pontuais codificadas manualmente.

No entanto, pode ser identificado um conjunto de operações que são usadas na generalidade dos casos, ou seja, certas tarefas de um agente não são específicas de nenhuma aplicação em particular. Estas operações são até tão gerais que a sua codificação numa determinada linguagem de programação pode ser usada em várias aplicações, sem alterações relevantes, desde que se usem algoritmos devidamente parametrizados.

Basicamente, a função de um agente é re-implementar a funcionalidade de um determinado servidor. Por outras palavras, para cada objecto exportado pelo servidor, e por cada método de cada objecto, o agente executará algumas operações adicionais, para além de invocar as correspondentes operações no servidor. Dado que a funcionalidade de um servidor é especificada num ficheiro apropriado[2], recorrendo-se, no caso do ILU, à ISL, e visto que algumas das acções desempenhadas pelo agente, dada a sua generalidade, podem ser descritas de uma forma sintética, pode-se criar uma metodologia de anotação de interfaces. Estas anotações são um conjunto de instruções a acrescentar à ISL e terão como finalidade especificar um agente para o serviço correspondente à interface em causa. Dado que as anotações são usadas para especificar agentes e que estes são responsáveis por garantir o suporte à desconexão, as anotações serão denominadas por anotações de desconexão.

Por definição, a descrição da interface de um módulo deve conter, pura e simplesmente, uma descrição das funcionalidades oferecidas por esse módulo, isto é, o nome dos objectos que podem ser instanciados numa aplicação cliente e a lista de métodos, juntamente com os respectivos tipos de dados dos argumentos e dos resultados, que podem ser despoletados remotamente. No entanto, as instruções intercaladas na especificação de uma interface com o intuito de descrever o agente – anotações de desconexão – não especificam funcionalidade. Na verdade, as anotações de desconexão especificam a forma como essa funcionalidade será oferecida, ou seja, indicam pormenores de implementação, o que contraria o objectivo primordial das linguagens de especificação de interfaces (ver secção 2.2.2).

Porém, o processo de anotação de uma interface não deve ser entendido como uma adulteração da descrição das funcionalidades de um determinado serviço. Este processo cons-

[2]Em ILU, uma interface usada para descrever a funcionalidade de um módulo corresponde, normalmente, a um ficheiro. Porém, nada impede que várias interfaces sejam descritas através da ISL num mesmo ficheiro, havendo também a possibilidade de se importar uma interface quando se define outra (vários ficheiros para uma só interface). Um servidor pode ainda implementar vários módulos.

titui um passo intermédio no desenvolvimento de agentes.

Quando um programador analisa a especificação da interface de um serviço e desenvolve um cliente ou um servidor para esse serviço, poder-se-á dizer que se está a efectuar uma transformação dessa especificação. Este processo não envolve a alteração do ficheiro que contém a especificação da interface, mas poder-se-á concluir que, com base nessa especificação e mais algum código escrito em outros ficheiros, se obtém uma aplicação distribuída.

Em [48] é apresentada uma ferramenta para geração automática de documentação para uma interface, que utiliza uma abordagem semelhante à das anotações. Esta ferramenta – `idldoc` – processa a descrição da interface escrita na IDL da especificação CORBA juntamente com algumas primitivas extra IDL, por sinal escritas em forma de comentário, e gera documentação em HTML (*Hypertext Markup Language*). Fundamentalmente, trata-se de uma extensão à IDL para escrever linhas de comentário com informação estruturada.

No desenvolvimento de um agente, o resultado final é obtido a partir da especificação da interface e de algumas considerações, sobre a forma de implementar as operações nela descritas, que são as anotações de desconexão. É claro que se poderia manter a separação entre a descrição da interface e a descrição do modo de operação do agente, usando ficheiros distintos. No entanto, descrever o modo como determinada funcionalidade é conseguida sem descrever essa funcionalidade é algo difícil de se conceber. Assim, a especificação do agente deverá também conter a descrição da interface, pelo que se optou por propor algumas extensões à ISL do ILU.

4.2.2 Anotações Propostas

As primitivas aqui apresentadas destinam-se ao processamento automático, por parte de um tradutor, visando a codificação de determinadas tarefas do agente, sem intervenção do programador. Por forma a conseguir uma maior flexibilidade no uso das anotações de desconexão e um campo de aplicação mais abrangente, certas primitivas permitem apenas a criação de esqueletos que deverão ser completados pelo programador, por forma a incluir alguns tratamentos específicos.

Para uma melhor integração com as restantes primitivas da ISL e para possibilitar uma melhor divulgação das anotações de desconexão, os identificadores escolhidos para as primitivas a acrescentar à ISL derivam do nome em inglês da acção que representam. De referir ainda que estes identificadores são algo extensos pelo facto de se pretender uma maior significância com base unicamente no nome da primitiva.

Descrição das Anotações

`BEFORE DISCONNECTION` Permite indicar, para cada objecto, um conjunto de acções a executar imediatamente antes de uma desconexão. Tais acções terão por objectivo transferir alguma informação do servidor para o agente, para que este possa servir os pedidos do cliente durante o período de desconexão. Basicamente, serão invocados métodos do servidor e serão armazenados os resultados devolvidos.

As desconexões involuntárias não podem ser tratadas por este mecanismo. No entanto, faltas devidas à indisponibilidade do servidor podem ser contempladas, desde que este envie um pré-aviso.

Se a descrição da interface de um objecto indicasse explicitamente métodos destinados à obtenção do estado desse objecto ou até alguma semântica relativa à implementação do objecto, a parte do agente destinada à preparação de uma desconexão poderia ser codificada na sua totalidade de forma automática. No entanto, apesar de um servidor poder exportar métodos para transferir parte, ou a totalidade, do seu estado, não existe ao nível da descrição de interfaces nenhuma forma de identificar exclusivamente estes métodos. Como tal, esta primitiva visa apenas criar esqueletos para as operações em causa.

A operação de desconexão é entendida no contexto de um objecto, e não por cada método, porque cada objecto possui um estado próprio que o agente tentará obter, sendo usado por todos os métodos desse objecto.

`ON RECONNECTION` Primitiva complementar da anterior que, de forma análoga, permite indicar um conjunto de operações que serão despoletadas automaticamente no momento em que é restabelecido o contacto com o servidor.

Note-se que, por forma a suportar um período de desconexão, as acções indicadas na primitiva anterior poderão criar réplicas de determinados itens de dados. No sentido mais lato, poder-se-á dizer que o agente mantém réplicas dos objectos implementados no servidor. Isto obrigará, eventualmente, a um tratamento especial no momento da reconexão. Todo este processamento será codificado pelo programador, tendo em conta que apenas podem ser usadas as operações do servidor especificadas na sua interface.

Como reconexão é entendido o acto de cessação de uma desconexão (acto voluntário e devidamente indicado pelo utilizador) ou o término de um período de incapacidade para contactar o servidor. Neste último caso, o agente descobrirá, por si só, e conforme explicado na primitiva que se segue, que a ligação está restabelecida.

`ALTERNATIVE` Permite indicar, para cada método, uma alternativa local (ao agente), que aceitará os mesmos parâmetros e devolverá o mesmo tipo de resultado. Isto significa que o método alternativo terá um protótipo de acordo com a definição do método substituído, ou seja, o mesmo número e a mesma ordem para os parâmetros e os mesmos tipos de dados para os parâmetros e o resultado. Esta alternativa será executada quando o objecto remoto não se encontrar disponível, isto é, a partir do momento em que o agente é notificado que se vai iniciar um período de desconexão e até ao instante da reconexão. Obviamente, o agente terá que manter uma variável de estado, para determinar se num dado instante decorre um período de desconexão ou não.

Em condições de funcionamento normal de todo o sistema, o agente apenas serve como intermediário, no que respeita à invocação de métodos remotos. Desta forma, quando o agente, para determinado pedido do cliente, invoca a operação homóloga do servidor e esta invocação não é bem sucedida, por falta temporária do servidor ou dos meios de comunicação, também é executada a operação alternativa indicada nesta primitiva. Neste caso, o agente tentará o contacto com o servidor sempre que o cliente efectuar uma invocação, servindo este procedimento também para detectar a reposição da ligação. Note-se que, no caso de uma desconexão, o agente só retoma o processo de interacção com o servidor a partir do momento em que surge a indicação voluntária de fim de desconexão.

A execução de uma alternativa local recorrerá eventualmente a dados obtidos com as acções executadas antes da desconexão. No caso de não existir um processo de desconexão, não existirão os dados provenientes dessas acções. Ter-se-á, portanto, uma situação equivalente à não especificação de acções para o acto de desconexão (ausência da primitiva `BEFORE DISCONNECTION`). Os mecanismos oferecidos por algumas das primitivas que ainda se seguem também poderão dar origem a informação útil para a execução de uma alternativa. Estas operações alternativas poderão produzir outros dados ou alterações que serão usados pelas acções executadas no momento de reconexão.

A ausência desta primitiva leva a que o cliente receba uma indicação de falha (uma excepção), no caso de não ser possível invocar o método do servidor e desde que não exista nenhuma outra forma de o agente substituir as funções do servidor. A descrição da primitiva que se segue esclarecerá melhor este ponto.

`STORE` Permite indicar, para cada método, a forma de guardar os resultados das últimas invocações, bem como a forma de devolver valores anteriormente armazenados.

As operações remotas que o agente desencadeia no servidor, e que são efectuadas com sucesso, e os resultados delas provenientes são a base dos mecanismos de "aprendizagem" usados pelo agente. Um destes mecanismos corresponde a guardar os resultados obtidos nas várias invocações efectuadas até um dado momento.

A ISL do ILU já permite etiquetar cada método com o qualificador `FUNCTIONAL`, para efeitos de utilização de caches no cliente. No entanto, não é oferecido nenhum controlo sobre o modo de operação dessas caches. Além disso, não existe nenhum mecanismo para preenchimento automático das caches, nem é possível efectuar qualquer tipo de processamento com os valores nelas armazenados. As operações alternativas, por exemplo, poderão beneficiar da consulta e manipulação dos resultados mantidos numa cache.

Com as parametrizações oferecidas nesta primitiva poder-se-á controlar:

- o tamanho da estrutura de armazenamento, em termos de número de invocações de métodos com consequente armazenamento do resultado;
- o modo de inserção na estrutura de armazenamento depois de atingido o seu limite, isto é, se se pretende desperdiçar os resultados das invocações posteriores a esta ocorrência, se se pretende retirar os resultados das primeiras invocações, para dar lugar aos novos, ou se se pretende a execução de um determinado método que determinará o resultado a retirar, para dar lugar ao novo;
- a forma de fazer corresponder os resultados à avaliação dos parâmetros das invocações, ou seja, se os resultados devem ser guardados sem a correspondente avaliação dos parâmetros usados, se devem ser guardados pares formados pela avaliação dos parâmetros da invocação e pelo respectivo resultado ou se para cada avaliação dos parâmetros devem ser guardados os resultados de várias invocações;
- a forma de usar os valores previamente armazenados, ou seja, se os resultados armazenados devem ser devolvidos ao cliente apenas quando não puder ser executada a operação no servidor ou sempre e se, aquando de uma invocação do cliente, deve ser devolvido o último ou o primeiro resultado para a avaliação de parâmetros em causa ou deve ser executado um determinado método para determinar qual o valor a devolver.

Em relação à terceira parametrização, refira-se que, no caso de serem guardados pares formados pela avaliação dos parâmetros da invocação e pelo respectivo resultado, será sempre assumido o resultado da última invocação, para uma determinada avaliação dos parâmetros de invocação. Note-se que o modo de inserção dirá respeito à estratégia de retirar pares da estrutura de armazenamento (para se inserirem novos) e não ao modo de determinar qual o resultado a armazenar para uma dada avaliação dos parâmetros. No caso de se guardarem resultados de várias invocações para cada avaliação dos parâmetros, o tamanho escolhido para a estrutura de armazenamento ditará o número de resultados a guardar para cada avaliação dos parâmetros, o que equivale a terem-se várias estruturas de armazenamento. O modo de inserção dirá respeito à forma de guardar os resultados em cada uma dessas estruturas.

Em relação à última parametrização convém esclarecer que:

- a ausência desta parametrização significará que os resultados armazenados nunca serão automaticamente devolvidos nos pedidos do cliente, ou seja, a estrutura de armazenamento não funcionará como cache;
- aquando de uma invocação do cliente, se a estrutura de armazenamento funcionar como cache e se tiver sido indicado que os resultados devem ser sempre devolvidos, então, se existir um resultado que satisfaça a avaliação dos parâmetros da invocação[3], este é devolvido, caso contrário passa-se ao processamento indicado para a primitiva `ALTERNATIVE`;
- no caso semelhante ao anterior, mas com indicação de que os resultados devem ser devolvidos só quando o servidor não estiver disponível, é tentado, em primeiro lugar, o contacto com o servidor[4], de seguida é procurado um resultado na estrutura de armazenamento e só em último recurso se passa à operação alternativa.

`ON SUCCESS` Permite indicar, para cada método, um conjunto de operações a executar após uma invocação com sucesso.

Dado que, em alguns casos, o armazenamento dos resultados das invocações não é por si só suficiente, dá-se a possibilidade de efectuar algum processamento extra sobre esses resultados, através da activação automática de um método que engloba as operações em causa, com a passagem do resultado devolvido pelo servidor e dos parâmetros utilizados na invocação.

A coexistência desta primitiva com a primitiva `STORE` é perfeitamente pacífica.

`TRY` Permite indicar, para cada método, o número de tentativas que devem ser efectuadas, a fim de se conseguir interagir com o servidor remoto. Assim, se momentaneamente não for possível efectuar uma determinada invocação com sucesso, o agente, de forma automática, encarregar-se-á de tentar a invocação numa fase posterior. Do ponto de vista do cliente, apenas existirá a percepção de que a invocação em causa foi um pouco mais demorada.

Além do número de tentativas, pode ainda ser especificado um tempo máximo para se efectuarem essas tentativas ou um intervalo de tempo entre cada tentativa. No primeiro caso, o agente efectuará uma invocação do método remoto (do servidor) e logo que receber a indicação de insucesso (uma excepção) recomeçará uma nova tentativa. O tempo decorrido entre duas tentativas consecutivas dependerá do desempenho do agente e do tempo de

[3]Note-se que a estrutura de armazenamento poderá não conter nenhum resultado para a avaliação dos parâmetros da invocação que se está a processar.

[4]Se se estiver perante um período de desconexão, esta etapa não se efectuará.

resposta dos mecanismos de invocação do ILU. O agente deixará de tentar o contacto com o servidor depois de esgotadas as tentativas possíveis ou quando findo o tempo máximo para a execução dessas tentativas.

Note-se que as operações disponibilizadas pelo agente, para o caso de ser impossível contactar o servidor, só serão desencadeadas após o insucesso da última tentativa, e durante um período de desconexão esta primitiva nunca terá efeito.

`WAIT` Permite indicar, para cada método, quanto tempo o agente deve esperar antes de processar um determinado pedido. Assim, a partir do momento em que o agente recebe um pedido de um cliente, é activado um temporizador. Entretanto, poderão chegar novos pedidos (do mesmo cliente ou de outros). Findo o tempo especificado nesta primitiva, e que começou a ser contabilizado no momento da primeira invocação, são processados os vários pedidos (invocações) recebidos até então.

Fundamentalmente, este mecanismo permite que as trocas de informação entre o agente e o servidor ocorram entre intervalos de tempo maiores e de forma mais concentrada. Isto poderá reduzir custos de comunicação e, eventualmente, possibilitará ultrapassar situações de inacessibilidade do servidor.

Nesta primitiva há ainda a possibilidade de indicar quantas invocações podem ser recebidas durante o tempo de espera e se se devem ignorar invocações, consecutivas ou não, com a mesma avaliação dos parâmetros. Com a primeira indicação permite-se que o agente passe ao processamento dos pedidos, mesmo que não se tenha esgotado o tempo especificado. A última parametrização poderá reduzir significativamente o número de invocações efectuadas pelo agente e, consequentemente, o tráfego entre este e o servidor.

Se no momento de se efectuarem as invocações, correspondentes aos pedidos pendentes, se verificar que não é possível contactar o servidor, então o processamento dessas invocações ficará suspenso até à chegada de um novo pedido ou até ao término de um novo período de espera. Refira-se que, no momento em que é iniciado o processamento dos pedidos pendentes, o temporizador associado é colocado a zero.

Os mecanismos disponibilizados por esta primitiva destinam-se exclusivamente a métodos que nunca devolvam qualquer resultado. Assim, esta primitiva jamais coexistirá com as primitivas `STORE` e `PREFETCH`. Por outro lado, devido ao adiamento contínuo das invocações pendentes no caso de não ser possível o contacto com o servidor, as primitivas `TRY` e `ALTERNATIVE` também não podem ser usadas em conjunção com esta.

`PREFETCH` Permite indicar, para cada método, se devem ser efectuadas invocações de forma automática, por forma a suportar futuras faltas. Trata-se de um meio de o agente invocar, por sua iniciativa, métodos do servidor. Os resultados destas invocações serão even-

tualmente processados pelos mecanismos oferecidos pelas primitivas `STORE` e `ON SUCCESS`.

Por outras palavras, este procedimento oferece uma fonte de informação adicional, para o agente construir um estado interno que permita substituir as funções do servidor. Para além das invocações desencadeadas por este mecanismo apenas existem as invocações iniciadas pelo cliente e aquelas que constam das operações de preparação de uma desconexão (primitiva `BEFORE DISCONNECTION`).

Nesta primitiva é permitido parametrizar o intervalo de tempo entre invocações consecutivas, para além de se poder limitar opcionalmente o número de invocações a efectuar.

Quando o método em causa possui argumentos de entrada, são utilizados os parâmetros armazenados através da primitiva `STORE`, de uma forma rotativa. Se a estrutura de armazenamento se encontrar vazia ou se apenas se encontrarem armazenados resultados, sem os parâmetros de invocação correspondentes, então este mecanismo não terá qualquer efeito.

Refira-se que as invocações desencadeadas por este mecanismo não são tratadas pelas operações correspondentes às primitivas `TRY`, `WAIT` e `ALTERNATIVE`, e que este é suspenso em caso de desconexão.

Para concluir, apresenta-se a seguir uma tabela com todas as primitivas de desconexão e as respectivas compatibilidades, ou seja, a forma como as várias primitivas podem ser combinadas entre si.

Tabela 4.1: Primitivas de desconexão e sua compatibilidade.

Primitiva		Compatibilidade (coexistência)							
Designação	Abrev.	BD	OR	A	SR	OS	T	W	P
BEFORE DISCONNECTION	BD	–	√	√	√	√	√	√	√
ON RECONNECTION	OR	√	–	√	√	√	√	√	√
ALTERNATIVE	A	√	√	–	√	√	√		√
STORE ... RETURN ...	SR	√	√	√	–	√	√		√
ON SUCCESS	OS	√	√	√	√	–	√		√
TRY	T	√	√	√	√	√	–		√
WAIT	W	√	√					–	
PREFETCH	P	√	√	√	√*	√*	√		–

* Uma das primitivas terá que estar presente.

Sintaxe das Anotações

A gramática da ISL aumentada com as anotações de desconexão é apresentada a seguir, tendo-se usado a notação *Extended Backus-Naur Form* (EBNF).

```
object_definition
  : ...
    ["BEFORE" "DISCONNECTION" method_name]
    ["ON" "RECONNECTION" method_name]
  ;
method
  : ...
    ["ALTERNATIVE" method_name]
    ["STORE"
     ("LAST" | "FIRST" | "USING" method_name) number "RESULTS"
     ("IGNORING" | "INDEXED" "BY" |
      "FOR" "EACH") "EVALUATION" "OF" "PARAMETERS"
     ["RETURN" ("LAST" | "FIRST" | "USING" method_name)
      ("WHEN" "DISCONNECTED" | "ALWAYS")]]
    ["ON" "SUCCESS" method_name]
    ["TRY" number "TIMES"
     [("FOR" number "SECONDS") |
      ("WAITING" number "SECONDS")]]
    ["WAIT" number "SECONDS" ["FOR" number "CALLS"]
     ["DISCARDING" ["CONSECUTIVE"] "EQUAL" "CALLS"]]
    ["PREFETCH" [number] "RESULTS" "WAITING" number "SECONDS"]
```

Neste excerto são apresentadas apenas duas produções – definição de um objecto e definição de um método – visto que foram as únicas alteradas em relação à gramática original da ISL. De referir ainda que as reticências representam todas as regras que se podem encontrar nessa mesma gramática, para cada uma das produções, e que se mantêm sem alteração.

O símbolo terminal `method_name` é um método que englobará um determinado conjunto de operações, enquanto que `number` diz respeito a um número inteiro.

4.3 Construção de Agentes

Conforme já foi referido, as anotações apresentadas destinam-se à automatização da construção de agentes. Nas secções que se seguem clarifica-se todo o processo de desenvolvimento de um agente, bem como a implementação do tradutor das anotações de desconexão.

4.3.1 Processo de Obtenção de Agentes

A figura 4.2 esquematiza, de forma simplificada, o processo de construção de um agente.

Por forma a abreviar a exposição, considere-se que num dado ficheiro se encontra a especificação de uma única interface e que esse ficheiro possui o mesmo nome que a interface (excluindo o sufixo `.isl`, usado no ficheiro). Assim, partindo da especificação de uma interface, mantida por exemplo no ficheiro `xyz.isl` e escrita na ISL do ILU, começa-se por anotar as várias declarações que a constituem. Este processo implica que, para cada objecto e para cada método de cada objecto, se utilizem as primitivas descritas na secção 4.2.2, segundo a sintaxe correspondente. Deste processo resultará um novo ficheiro com a especificação do agente. Este ficheiro poder-se-á designar por `xyz.isl++`, visto que contém declarações escritas na linguagem ISL, juntamente com anotações de desconexão, as quais constituem uma extensão a esta linguagem. O nome deste ficheiro não é usado em qualquer processamento, pelo que poderá ser escolhido livremente, por quem efectua as anotações.

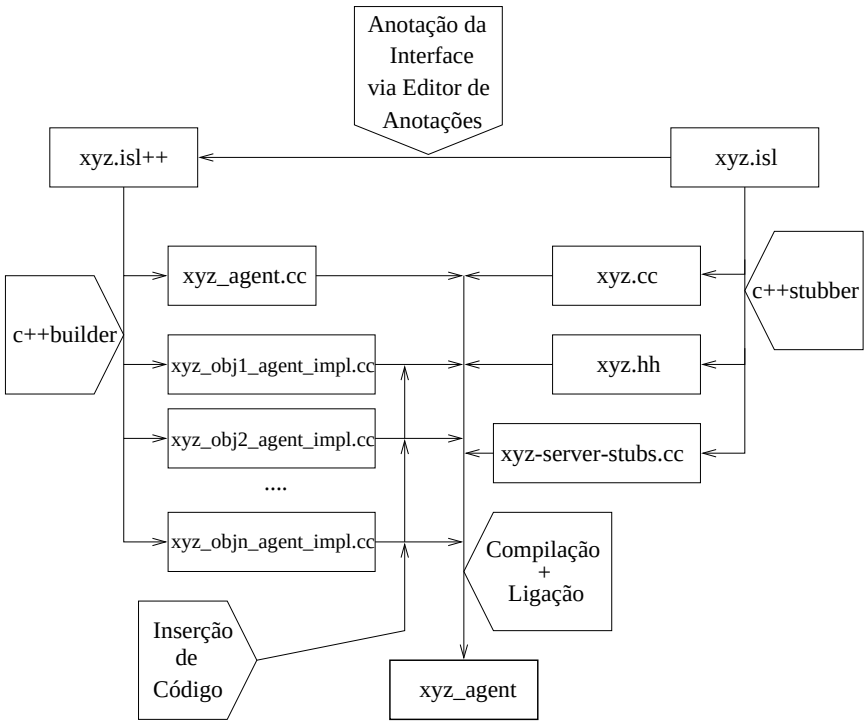

Figura 4.2: Construção de agentes.

A especificação do agente (ficheiro `xyz.isl++`) é processada por um tradutor desenvolvido no âmbito deste trabalho. Tal tradutor – `c++builder` – é responsável por gerar o código para o agente em causa. De referir que a estratégia adoptada foi a de utilizar um agente

por cada interface do ILU. Assim, se num único ficheiro se encontrarem declaradas várias interfaces, o que é permitido pela ISL do ILU, serão produzidos vários agentes (um para cada interface).

O tradutor produz código em C++ disperso por vários ficheiros (módulos). Um dos módulos – `xyz_agent.cc` – contém o código necessário para registar os vários objectos exportados pelo agente, para além do código usado para instanciar, quando possível, os objectos exportados pelo servidor, que correspondem aos objectos implementados no agente. O nome deste módulo é determinado pelo nome da interface, que neste exemplo se considerou igual ao nome do ficheiro que contém a especificação. Note-se que o agente é simultaneamente servidor e cliente para cada objecto descrito na interface.

Um conjunto de módulos irá conter as implementações específicas de cada objecto. Assim, para cada objecto é criado um ficheiro – `xyz_obj?_agent_impl.cc` – onde `obj?` se refere ao nome do objecto em causa (obtido a partir da especificação). Neste ficheiro encontra-se a classe C++ que implementa o objecto real no agente (ver secção 3.1.3). Para além dos métodos que o agente exporta, para o objecto em causa, nesta classe encontram-se os esqueletos dos métodos indicados em algumas das anotações de desconexão. Nos métodos que o agente exporta, aqueles para os quais o programador pode escrever anotações, encontra-se todo o código necessário para interagir com o servidor e para desencadear os mecanismos locais que permitem devolver resultados ao cliente, segundo as anotações escritas, mesmo quando o servidor se encontra inacessível.

A classe codificada pelo tradutor contém ainda a declaração de todas as estruturas de dados que constituem parte do estado do objecto e que são necessárias para implementar a funcionalidade das primitivas `STORE` e `WAIT`. Esta última requer que seja mantido um tampão[5] com os parâmetros de todos os pedidos pendentes.

As primitivas `PREFETCH` e `WAIT`, que requerem mecanismos de temporização e activação automática[6], obrigam à criação de métodos específicos para desempenhar as operações inerentes a cada uma delas. Estes métodos, um por cada anotação efectuada com uma destas primitivas, são activados pelos mecanismos de temporização oferecidos pelo sistema operativo. Motivos de ordem técnica relacionados com estes mecanismos levam a que estes métodos sejam, na realidade, funções que estão autorizadas a manipular o estado do objecto – *friends*, na terminologia C++.

Depois de escritas as rotinas específicas em cada um dos módulos usados para implementar os vários objectos reais do agente, rotinas para as quais o tradutor apenas constrói esqueletos, pode-se passar à compilação do agente. Dado que o agente é servidor e cliente

[5] Do inglês *buffer*.

[6] Na primitiva `PREFETCH` é despoletada periodicamente uma invocação, enquanto que na primitiva é `WAIT`, findo um determinado período de tempo, processam-se os pedidos pendentes.

de objectos ILU, haverá necessidade de ligar os vários módulos que o constituem com o código produzido pelo *stubber* do ILU. Visto que o tradutor das anotações de desconexão produz código C++, ter-se-á que recorrer à ferramenta `c++stubber` do ILU, a qual produz *stubs* para C++. Os *stubbers* do ILU produzem módulos para serem usados tanto no cliente como no servidor e produzem um módulo que se destina apenas aos servidores. Na construção do agente serão usados todos os módulos, pelo facto de este ser cliente e servidor simultaneamente.

O C++ foi a linguagem escolhida para codificar os agentes pelos motivos já referidos na secção 3.1.3. Esta escolha condiciona os programadores na escrita das rotinas específicas mencionadas no parágrafo anterior. No entanto, nada impede que sejam desenvolvidas variantes do tradutor de anotações, por forma a obter agentes codificados em outras linguagens. Note-se que as anotações de desconexão são independentes de qualquer linguagem que se use na sua implementação. Refira-se também que a linguagem utilizada para implementar os clientes e os servidores não depende da codificação do agente.

Se a especificação do agente tivesse informação relativa às estruturas de dados usadas para manter os estados dos objectos e alguma informação sobre a manipulação dos dados por cada operação disponibilizada pelo objecto (semântica das operações), poder-se-iam obter concretizações bastante mais completas. Isto significa que não seria exigido ao programador que codificasse manualmente parte do agente, pois o tradutor seria capaz de retirar da especificação a informação necessária para produzir um agente na sua totalidade.

Por exemplo, a linguagem de prototipagem CAMILA [49] oferece mecanismos para descrever matematicamente o comportamento de componentes complexos. Os construtores desta linguagem permitem que especificações relativamente reduzidas (em termos de código escrito) traduzam estruturas de dados e processamentos de uma certa complexidade, cuja implementação, numa linguagem de programação vulgar, levaria à escrita de programas muito mais extensos. As especificações escritas em CAMILA podem até ser executadas, por forma a produzirem os resultados pretendidos. Mais ainda, os componentes executados desta forma podem interagir com programas escritos na linguagem C e vice-versa [50].

Isto leva a pensar que a especificação de objectos remotos poderia ser efectuada através de uma linguagem de descrição de interfaces juntamente com uma linguagem como o CAMILA. No entanto, o CAMILA é uma linguagem de prototipagem, o que implica que o eventual protótipo obtido para um determinado agente teria que, de alguma forma, ser concretizado numa linguagem de programação particular, e a automatização desta tarefa não é nada fácil. De facto, não é razoável manter-se um protótipo em operação devido aos recursos exigidos para o efeito e devido ao baixo desempenho assim conseguido. Um protótipo serve apenas para validar o planeamento de determinada operacionalidade.

4.3.2 Implementação do Tradutor

O tradutor das anotações de desconexão – `c++builder` – foi construído com base no PCCTS, o qual oferece uma plataforma de desenvolvimento semelhante à oferecida pelo Lex e pelo Yacc [51], em conjunto.

O PCCTS (versão 1.33) [52] é um gerador de analisadores, que possui três características que o distinguem dos demais:

- leitura antecipada[7] de vários símbolos, que faz com que o analisador possa tomar decisões em casos de ambiguidade;
- predicados semânticos, que permitem indicar, nas várias produções da gramática, condições que guiam o analisador nas suas decisões;
- predicados sintácticos, que permitem o reconhecimento condicional de uma sequência de símbolos (retrocesso[8] selectivo).

O analisador/tradutor desenvolvido tem como base a gramática incluída no manual do ILU, a qual foi acrescentada com as construções sintácticas apresentadas na secção 4.2.2. A gramática da ISL disponível na documentação do ILU, embora incompleta e mesmo com alguns erros, está escrita na notação *Backus-Naur Form* (BNF). Isto facilita a sua introdução no ANTLR, o qual aceita gramáticas escritas em formato EBNF.

O ANTLR é a ferramenta do PCCTS que permite construir analisadores recursivos descendentes, em C ou C++, a partir de gramáticas LL(k)[9]. Neste caso, optou-se por escrever as acções semânticas das várias produções da gramática em C++ e usar o ANTLR na sua variante C++, pelo simples facto de esta linguagem ter já sido escolhida para implementar os agentes.

As tarefas desempenhas pelo analisador, no processamento da especificação de um agente, são as seguintes:

- análise sintáctica das construções da ISL do ILU, dado que as anotações de desconexão só fazem sentido no contexto da declaração de uma interface ILU;
- análise semântica de algumas construções da ISL do ILU, nomeadamente das declarações de tipos de dados e das declarações de argumentos e resultados dos métodos, por forma a obter informação necessária à codificação das classes que implementam os objectos reais e à codificação das invocações de métodos do servidor;

[7] Do inglês *lookahead*.
[8] Do inglês *backtracking*.
[9] Uma gramática diz-se LL(k) se a leitura antecipada de k símbolos a tornar numa gramática não ambígua.

- análise sintáctica e semântica das anotações de desconexão.

Resumidamente, estas tarefas têm por finalidade, numa primeira fase, construir uma estrutura de dados com a descrição da interface processada. Para cada tipo de dados declarado é obtido o tipo ILU correspondente (`RECORD`, `ARRAY`, `INTEGER`, etc.) e para cada objecto são armazenados os parâmetros das eventuais primitivas de desconexão (`BEFORE DISCONNECTION` ou `ON RECONNECTION`) e informação relativa a todos os métodos. Para cada método, para além do seu nome, são armazenados os parâmetros das eventuais primitivas de desconexão (`ALTERNATIVE`, `STORE`, `ON SUCCESS`, `TRY`, `WAIT` e `PREFETCH`) e os nomes e tipos de dados dos argumentos e resultados desse método.

Numa segunda fase passa-se à geração do código do agente, tomando como base a estrutura de dados previamente construída.

4.3.3 Editor de Anotações

Com o objectivo de facilitar o processo de escrita das anotações, foi desenvolvido um editor de anotações – `annotator`. O anotador, construído em Tcl/Tk [53], permite visualizar graficamente a lista dos objectos e métodos que compõem uma interface e, através de botões, oferece a possibilidade de adicionar informação para suporte à desconexão.

A figura 4.3 mostra o aspecto desta ferramenta, que pode ser usada para editar especificações de interfaces ainda sem anotações ou especificações previamente anotadas.

A informação necessária para a operação do anotador – lista de interfaces, de objectos e de métodos – é obtida recorrendo ao tradutor de anotações, o qual permite obter relatórios sobre especificações de interfaces (anotadas ou não), mediante uma indicação especial (parâmetro na linha de comando).

Para cada anotação de desconexão existe, no anotador, um botão correspondente, que estará activo ou não, conforme o contexto das anotações (ao nível do objecto ou ao nível do método). No exemplo da figura 4.3, os botões correspondentes às primitivas `BEFORE DISCONNECTION` e `ON RECONNECTION` estão desactivados, porque estas anotações só se aplicam a objectos e, no caso representado, está-se a anotar um método.

Os botões permitem adicionar, alterar e remover anotações de um objecto ou de um método. Os parâmetros de uma anotação são pedidos, ou é dada a possibilidade de se efectuar a sua alteração, no momento em que se carrega no botão correspondente.

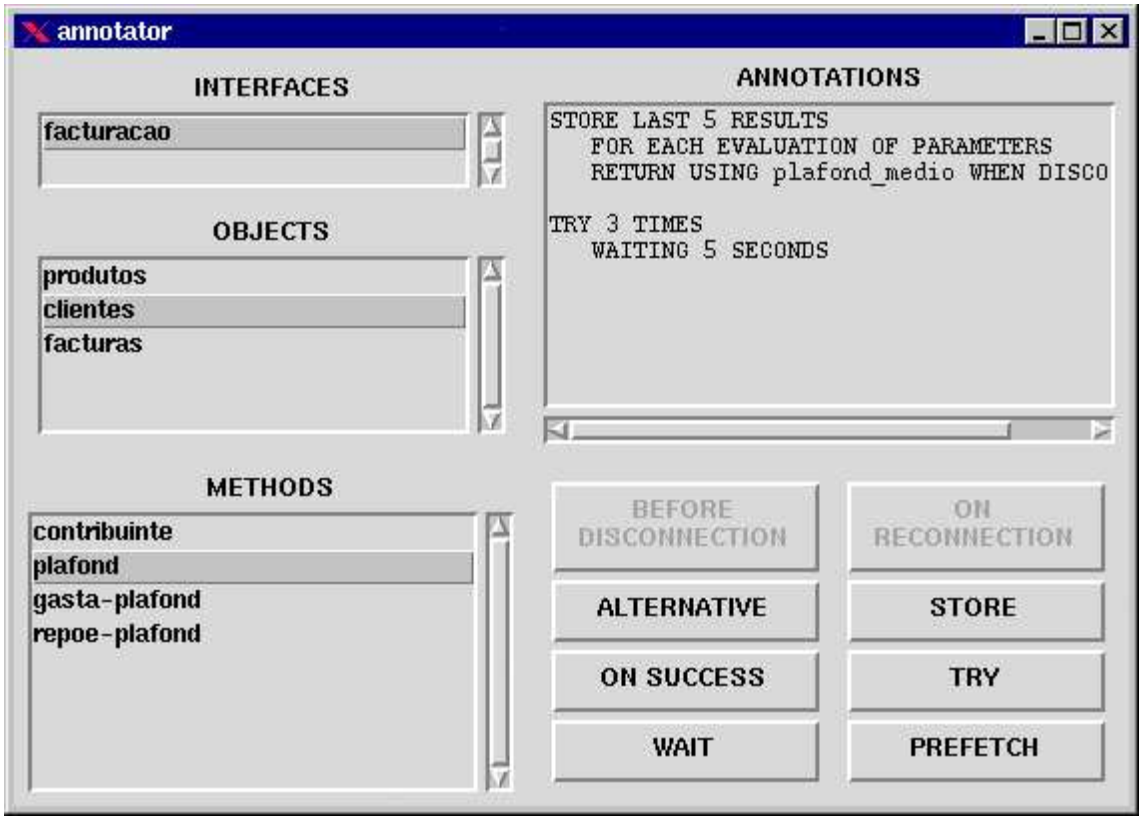

Figura 4.3: Editor de Anotações.

Capítulo 5

Replicação de Objectos

Com os mecanismos descritos na capítulo 4 espera-se que um número significativo de aplicações passe a tolerar a operação de desconexão, isto é, que os clientes de certas aplicações possam funcionar de forma isolada, sem a necessidade de contactar o servidor durante determinados períodos de tempo.

No entanto, nos casos em que o serviço tem obrigatoriamente que se encontrar disponível, a solução para aumentar a tolerância a faltas passa pela replicação de servidores. Note-se que as caches mantidas no agente (modelo CAS) já são uma forma de replicação, mas apenas permitem obter parte da funcionalidade do serviço[1].

Na replicação de servidores faz todo o sentido usar um modelo análogo ao modelo Cliente-Servidor, ou mesmo ao modelo CAS, como base para o encaminhamento dos pedidos de um cliente para um grupo de servidores.

Neste capítulo é apresentado um modelo para a invocação de objectos ILU replicados. O objectivo principal deste modelo é tornear as deficiências do mecanismo de invocação de métodos disponibilizado no ILU. Como cada invocação visa um único método de um determinado servidor, o modelo proposto baseia-se na intercepção das invocações do ILU e na utilização de um serviço de comunicação em grupo.

A concretização deste modelo, também aqui apresentada, tem por base o GTS e oferece um mecanismo simples para a replicação de servidores desenvolvidos com base no ILU.

No final do capítulo são tecidas algumas considerações sobre difusão de informação em aplicações que utilizam objectos ILU e sobre a adequação à tarefa de difusão do modelo aqui apresentado.

[1] As caches mantidas nos agentes do modelo CAS replicam apenas parte do estado dos objectos.

5.1 Modelo Proposto

5.1.1 Invocações Múltiplas

A invocação de procedimentos remotos, ou de acordo com os paradigmas actuais, a invocação de métodos sobre objectos distribuídos, segue o tradicional modelo Cliente-Servidor. Assim, as ferramentas para desenvolvimento de aplicações distribuídas baseadas neste modelo permitem que um cliente invoque uma única operação sobre uma determinada concretização remota (objecto ou procedimento). No entanto, certas aplicações podem beneficiar da execução da mesma operação em vários servidores e em simultâneo, como acontece, por exemplo, na replicação de servidores. Neste caso, convém que uma única instrução num cliente (a invocação da operação remota) desencadeie execuções em várias máquinas. No que respeita ao ILU, a invocação de um método de um objecto remoto deverá activar a execução desse método em mais do que uma implementação do objecto em causa.

Soluções Existentes

Recentemente surgiram alguns trabalhos nesta área, dos quais se destacam o Electra e o *Generic Remote Invocation Protocol* (GRIP).

O Electra [54] adiciona ao CORBA a abstracção de grupo de objectos aliada à difusão selectiva fiável. Fundamentalmente, trata-se de um ORB construído de acordo com a especificação CORBA e caracterizado pelo conceito de grupo.

O sistema Electra foi desenvolvido para operar sobre plataformas do género do Horus [55] ou do Isis [56]. Estas plataformas baseiam-se em modelos que asseguram que decisões relacionadas tomadas por processos distintos num sistema distribuído são mutuamente coerentes e que as acções são correctas temporalmente, mesmo quando os vários componentes comunicam assincronamente. Estes modelos permitem que uma aplicação distribuída tenha um comportamento previsível apesar do mau funcionamento de alguns componentes e da natureza mutatória destes (entrada e saída de processos).

Presentemente, o Electra é constituído por um compilador da IDL do CORBA, um DII, uma interface para o ORB desenvolvido[2], um BOA e adaptadores de objectos para o Horus e para o Isis.

O GRIP [57] tem como objectivo manter a transparência do esquema de replicação de objectos. Este protocolo é independente do protocolo do serviço de replicação e não coloca restrições ao modelo de coerência das réplicas.

[2]Esta interface permite o acesso às operações que manipulam grupos de objectos, para além das operações normais de um ORB.

O modelo de interacção oferecido pelo GRIP tem como base um serviço de comunicação em grupo com ordenação causal. Neste modelo, um cliente interage com um servidor através de um protocolo específico – *o Remote Access Protocol* (RAP). Simplificadamente, do lado do cliente, este protocolo trata de obter a lista dos servidores disponíveis num dado momento, efectuando de seguida sucessivas invocações até que um dos servidores devolva uma resposta. Do lado do servidor, este protocolo encarrega-se de entregar o pedido recebido de um cliente, devolvendo o resultado associado, no final da execução da operação em causa. Um cliente pode ainda receber, através do RAP, informação sobre o servidor mais aconselhável para uma determinada invocação.

Por forma a garantir uma semântica do tipo quando-muito-uma-vez[3], os vários servidores executam opcionalmente um outro protocolo – *o Protocol for Repeated Invocation DEtection* (PRIDE). Note-se que, devido à fiabilidade das ligações ponto-a-ponto, cada réplica executará quando muito uma vez cada pedido efectuado por um cliente. No entanto, dado o funcionamento do RAP, um cliente pode invocar a mesma operação sobre vários servidores, e entre aqueles que recebem o pedido poderão existir vários a executar a operação em causa.

A coerência das réplicas é garantida através da criação de um grupo – grupo de servidores – onde é trocada informação relacionada com a actualização individual de cada réplica. O protocolo usado entre as várias réplicas é totalmente independente do GRIP.

Invocações Transparentes aos Clientes e Servidores

A invocação simultânea de múltiplas concretizações de um objecto não só tem interesse na replicação de serviços como também pode ser de grande valor no desenvolvimento de aplicações cooperativas, onde colaboram mais que duas entidades, e de aplicações com necessidades de difusão de informação, onde uma entidade divulga determinada informação para um conjunto de potenciais receptores.

No caso da replicação, o objectivo primordial é a tolerância a faltas, que também constituiu a motivação dos projectos Electra e GRIP. Para os outros casos, não se impõe requisitos tão fortes no que respeita à sincronização de réplicas. Aliás, o protocolo usado pelos servidores para garantir a coerência das réplicas deverá ser independente do mecanismo das invocações e em alguns casos poderá até haver interesse em replicar objectos cuja concretização original não inclui código para sincronização entre réplicas. Este pressuposto permitirá desenvolver um mecanismo de invocação múltipla mais simples, do ponto de vista da sua concepção e até no que respeita à sua utilização.

Note-se que, na difusão de informação e nas aplicações cooperativas, os vários receptores

[3]Do inglês *at-most-once*.

de uma mensagem – uma invocação no modelo Cliente-Servidor – serão vários servidores que implementam um determinado objecto. Deste modo, tais receptores serão também réplicas do objecto em causa. Fundamentalmente, esta abordagem permite uma forma de comunicação em grupo, onde os vários membros trocam informação através de invocações de métodos, as quais constituem abstracções para uma interacção a um nível mais elevado que a troca de mensagens oferecida pelos serviços de comunicação em grupo.

Por outro lado, nem o Electra nem o GRIP permitem invocações múltiplas de forma transparente aos clientes, ou seja, o despoletar de múltiplas execuções remotas com uma única invocação efectuada pelo cliente não é totalmente transparente para este. De facto, no Electra existem operações específicas para interagir com o ORB, as quais são desencadeadas pelo cliente, enquanto que no GRIP, o cliente é obrigado a executar um protocolo especial – o RAP. No que respeita aos servidores, embora estes não tenham conhecimento dos restantes destinatários de um determinado pedido recebido, a não ser que o protocolo que usam para garantir a coerência de réplicas a isso obrigue, impõe-se que a sua concretização tenha em consideração o sistema usado para suportar as invocações múltiplas. Deste modo, e contrariamente ao sistema aqui desenvolvido, é colocada de parte a hipótese de construir serviços replicados tendo como base aplicações já existentes e desenvolvidas segundo o modelo Cliente-Servidor.

Partindo do mecanismo de invocação de métodos remotos oferecido pelo ILU – invocações 1:1[4] – interessa portanto desenvolver um sistema que faça chegar uma cópia de um pedido de um cliente a vários servidores – invocações l:n – tendo em consideração os seguintes requisitos:

- deverá ser possível construir um sistema com servidores replicados, tomando como base um cliente e um servidor desenvolvidos para operar segundo o tradicional.modelo Cliente-Servidor, sem que seja necessário alterar os programas (cliente e servidor) já existentes;

- deverá ser possível definir grupos de objectos ILU e constituir identificadores que permitam tratar um conjunto de objectos como uma entidade única;

- as implementações dos vários objectos que constituem um grupo deverão receber uma cópia idêntica do pedido efectuado pelo cliente;

- a execução, por parte de um servidor, das operações associadas a uma determinada invocação deverá ser independente do número de servidores (objectos) que constituem o grupo visado por tal invocação;

[4]Um cliente invoca um método em um servidor.

- as múltiplas respostas a um determinado pedido, uma por cada réplica, deverão ser filtradas de maneira a que ao cliente apenas chegue uma resposta[5];
- o mecanismo de entrega de pedidos aos vários elementos de um grupo deverá assegurar a recepção do pedido por todos estes e deverá garantir uma determinada ordem – total ou, pelo menos, causal – na entrega dos pedidos, por forma a possibilitar alguma coerência no estado das réplicas, mesmo quando estas não executam nenhum protocolo para o efeito;
- todo o sistema de suporte às invocações múltiplas deverá ser independente do código dos clientes e dos servidores e não deverá ser necessário introduzir quaisquer alterações aos *stubs* gerados pelas ferramentas do ILU.

De referir que o sistema a desenvolver, com base nestes requisitos, não terá em consideração o problema da duplicação desnecessária de interacções [58, 59]. Isto significa que, se um determinado pedido de um cliente chegar a vários servidores (uma cópia a cada) e se estes servidores usarem serviços de uma terceira entidade, replicada ou não, esta última receberá indicação para efectuar várias vezes a mesma operação. Por outras palavras, dado que não existe qualquer tipo de coordenação entre os servidores, cada um efectuará invocações de forma independente. Se as operações associadas a estas invocações não forem idempotentes, então um único pedido de um cliente despoletará várias execuções, num mesmo servidor, que poderão criar incoerências. Este é um problema que fica em aberto no mecanismo de replicação de objectos aqui apresentado.

5.1.2 Difusão Selectiva de Pedidos ILU

Tal como nas anotações de desconexão, e pelos motivos já referidos, o ILU servirá de base ao modelo de invocação de objectos replicados, mas agora com um pretexto adicional: a integração do sistema de replicação de objectos com o sistema de suporte à desconexão.

Mecanismo de Invocação do ILU

As aplicações distribuídas desenvolvidas com base no ILU obrigam o servidor a registar objectos num serviço de directório, de modo a que um cliente seja capaz de os localizar. O processo de registo de um objecto consiste basicamente na associação do seu identificador a um endereço IP, juntamente com um porto. Neste porto, e na máquina identificada pelo endereço em causa, encontrar-se-á à escuta o processo que executa o código correspondente à implementação desse objecto (servidor ILU).

[5]No modelo Cliente-Servidor, um cliente espera uma única resposta por cada invocação efectuada.

Quando um cliente interroga o serviço de directório sobre um determinado objecto, obtém como resultado o endereço IP e o porto da máquina onde se encontra a executar o processo com a concretização desse objecto. Nesse momento é criado um *socket*, através do qual se efectuarão todas as trocas de informação, entre o cliente e o servidor, que dizem respeito à invocação de métodos por parte do cliente. Note-se que um servidor pode disponibilizar vários objectos criando um único *socket* para escuta. Neste caso, os *stubs* do cliente e do servidor encarregam-se de distinguir entre objectos que se encontrem associados ao mesmo ponto de comunicação.

Aquando da criação do referido *socket*, efectua-se a seguinte interacção entre o cliente e o servidor:

- envio, por parte do cliente, de um pacote de *ping*, cujo conteúdo é independente do objecto, do servidor e do cliente em causa;

- resposta do servidor com um pacote de aceitação[6], também independente das entidades envolvidas[7].

Depois de estabelecida a conexão entre o cliente e o servidor, para um dado objecto, todas as invocações de métodos referentes a esse objecto são efectuadas através do *socket* criado para o efeito. A invocação de um método de um objecto remoto, depois de estabelecida a ligação, compreende as seguintes fases:

- envio, por parte de cliente, de um pacote com os nomes do objecto e do método, juntamente com os parâmetros para a invocação deste, mas sem nenhuma informação relativa ao servidor ou ao cliente;

- resposta do servidor, após a execução do método, com os valores de retorno deste (métodos que não retornam quaisquer dados envolvem sempre o envio de um pacote para indicar a conclusão da sua execução).

Em suma, podemos dizer que o ILU não utiliza nenhum mecanismo de certificação ou de segurança, o que possibilita a manipulação das mensagens trocadas entre clientes e servidores. Em particular, poder-se-ão interceptar as invocações efectuadas pelos clientes, estratégia esta que servirá de base ao modelo proposto.

[6]Do inglês *acknowledge*.

[7]Na verdade, estes pacotes estão relacionados por um contador, ou seja, o número de sequência atribuído pelo cliente ao pacote de *ping*, e refira-se que o cliente numera todos os pacotes que envia, é devolvido na resposta do servidor.

Modelo de Interceptores

Dada a natureza do mecanismo de invocação do ILU, é perfeitamente viável inserir uma entidade entre o cliente e o servidor capaz de manipular os pedidos desencadeados pelos clientes. Desta forma, poder-se-á fazer chegar um pacote de um cliente (pacote de *ping* ou pacote de invocação) a vários servidores, mediante a utilização de um serviço de comunicação em grupo, sem que estes se apercebam de que se trata da mesma invocação e sem que o cliente detecte que o seu pedido foi multiplicado. Do lado dos servidores tudo se passa como se o cliente tivesse efectuado várias invocações, ou seja, tivesse invocado explicitamente a mesma operação em cada um dos servidores. Por outro lado, as várias respostas (uma por cada servidor disponível) poderão ser processadas, fazendo chegar apenas uma ao cliente.

O sistema de invocações múltiplas obtido, e representado na figura 5.1, é constituído pelos seguintes elementos:

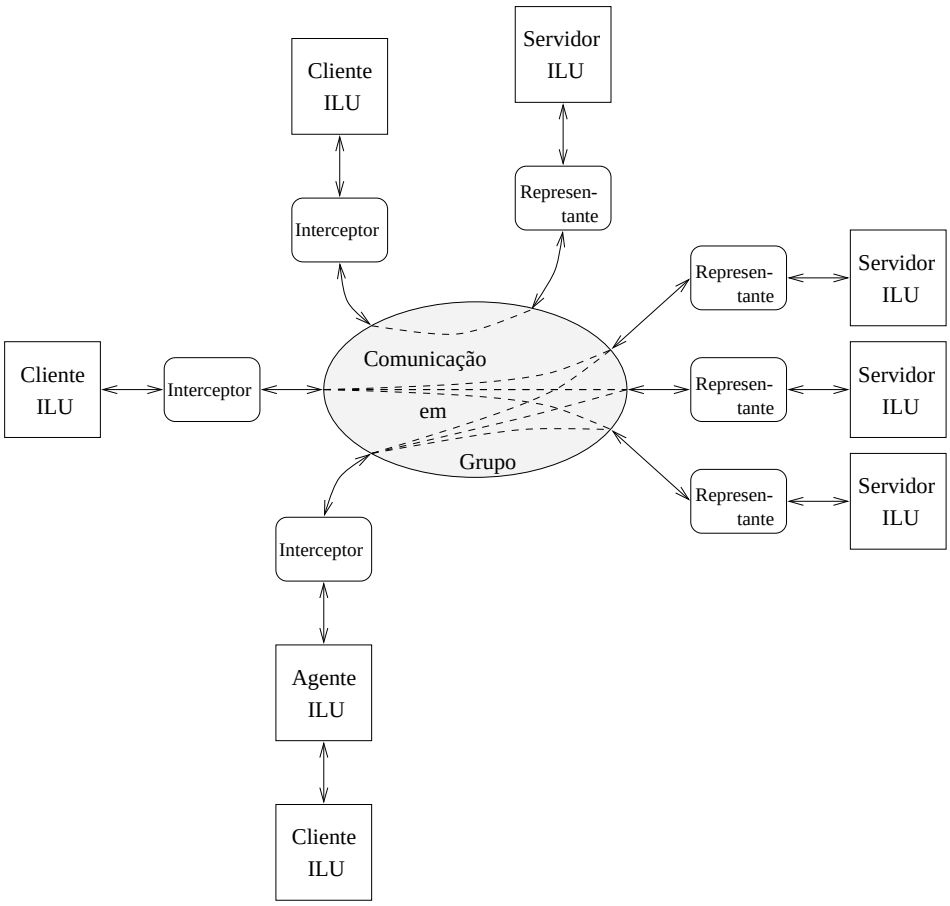

Figura 5.1: Modelo Cliente-Agente-Servidores.

- interceptor – Intercepta a comunicação entre o cliente e o servidor e envia cada pacote do cliente para um conjunto de servidores – grupo de objectos – fazendo uso de um serviço de comunicação em grupo (serviço de difusão selectiva de mensagens). À medida que os vários servidores vão respondendo com pacotes de aceitação ou de resultados, conforme o pacote enviado pelo cliente, o interceptor tem ainda a seu cargo a tarefa de recolher todas essas respostas, entregando apenas uma ao cliente em causa.

- serviço de difusão selectiva – Faz chegar um determinado pacote, enviado por um cliente, a um grupo de servidores. Pressupõe a criação de grupos de objectos e a atribuição de um identificador a cada grupo.

  Este serviço será responsável por garantir uma evolução coerente no estado das várias réplicas, conforme já referido.

- representantes – Recebem pedidos através do serviço de difusão selectiva e entregam-os ao respectivo servidor, devolvendo de igual forma os resultados associados à operação executada.

  Os representantes são necessários, neste modelo, devido ao facto de os servidores, por si só, não serem destinatários válidos de um sistema de difusão selectiva genérico, quer se trate de um grupo de servidores, quer se trate de um servidor em particular.

Integração com o Modelo CAS

A descrição até aqui efectuada, relativa ao modelo de interceptores, evidencia a adaptação de aplicações desenvolvidas segundo o modelo Cliente-Servidor. No entanto, nada é dito em relação ao modelo CAS apresentado no capítulo 4. Note-se que, com a integração destas duas abordagens, ter-se-ia um modelo de programação com suporte para desconexão e com a possibilidade de replicar servidores.

Dado que o agente do modelo CAS age como cliente na sua interacção com o servidor, o sistema de intercepção de pedidos ILU pode ser colocado entre estas duas entidades (agente e servidor). De referir que a hipótese de interceptar pedidos do cliente e proceder ao seu encaminhamento para vários agentes não tem o mínimo interesse, pois o agente é uma entidade bastante "próxima" do cliente, sendo sempre possível a comunicação entre estes.

As invocações múltiplas permitem a migração para o modelo Cliente-Servidores. Em conjunção com a noção de Agente tem-se a arquitectura Cliente-Agente-Servidores (CASs), onde o sistema de invocações múltiplas é colocado entre o agente e as réplicas dos servidores. Na figura 5.1 encontra-se representada a interacção de um cliente com um agente, cujos pacotes são processados por um interceptor que os faz chegar a três réplicas.

5.2 Concretização: ILU Replicado sobre GTS

A concretização do modelo apresentado teve como base o GTS, o qual possui uma característica única que é a preservação da ordem das mensagens, com tolerância a faltas, sobre diversos protocolos, incluindo correio electrónico. Esta característica, a facilidade de instalação em diversos sistemas operativos (incluindo o Linux) e os poucos recursos necessários para a sua operação fazem do GTS um sistema preferível relativamente ao xAMp ou ao Isis, apesar de estes serem bastante mais robustos.

Uma visão global do sistema de replicação é apresentada na figura 5.2.

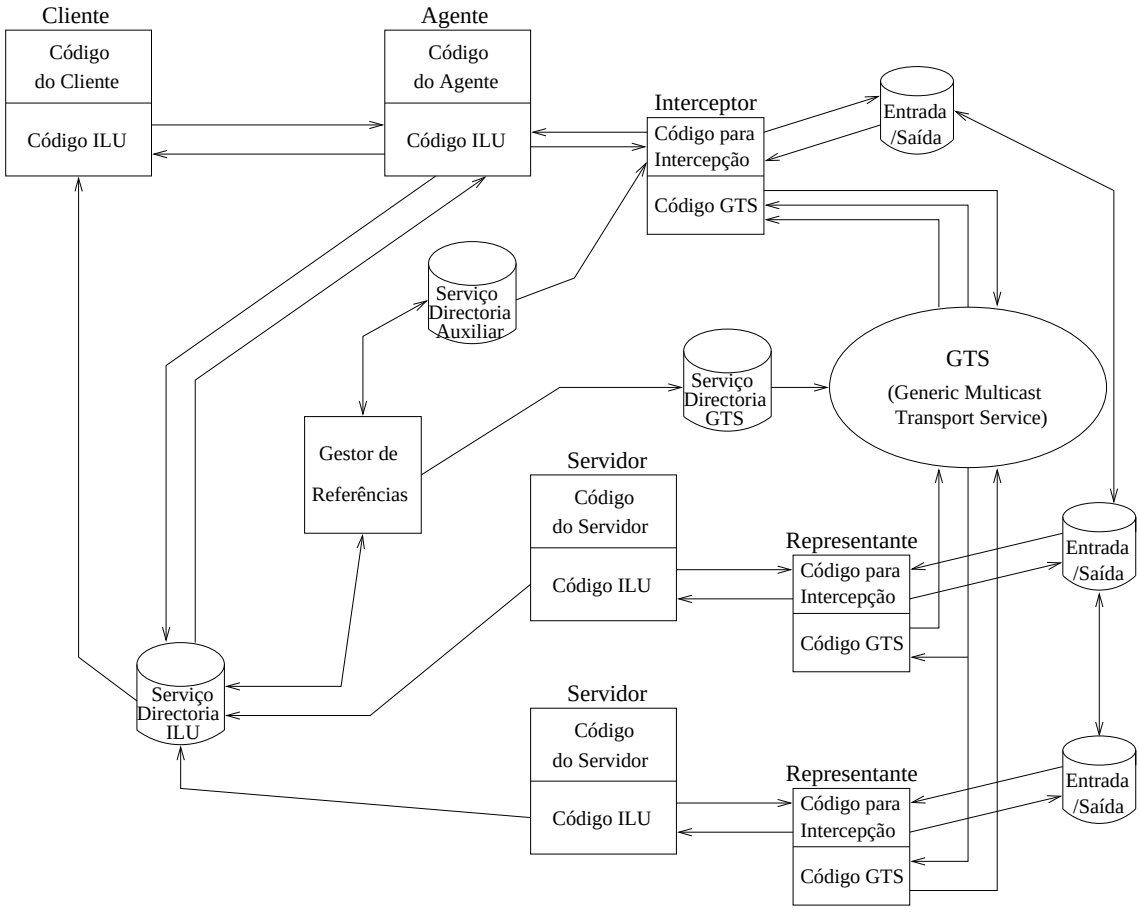

Figura 5.2: ILU sobre GTS.

A interacção entre os vários constituintes do sistema e particularmente o funcionamento do gestor de referências e das áreas de armazenamento de entrada/saída serão alvo de exposição detalhada nas secções que se seguem.

5.2.1 Intercepção de Pedidos

A intercepção de pedidos, com a respectiva devolução de respostas, é efectuada por dois *daemons* – `ILUintercepter` e `ILUproxy` – desenvolvidos em C++. A escolha desta linguagem deve-se ao facto de o GTS ser oferecido como uma classe C++ com métodos para criação de grupos e envio e recepção de mensagens.

O interceptor, que se encontra esquematizado na figura 5.3, inicia o seu processamento criando dois fios de execução[8]: um para escutar pedidos ILU num determinado *socket* e outro para receber mensagens do serviço de difusão selectiva, mensagens estas que encapsulam respostas a pedidos ILU[9].

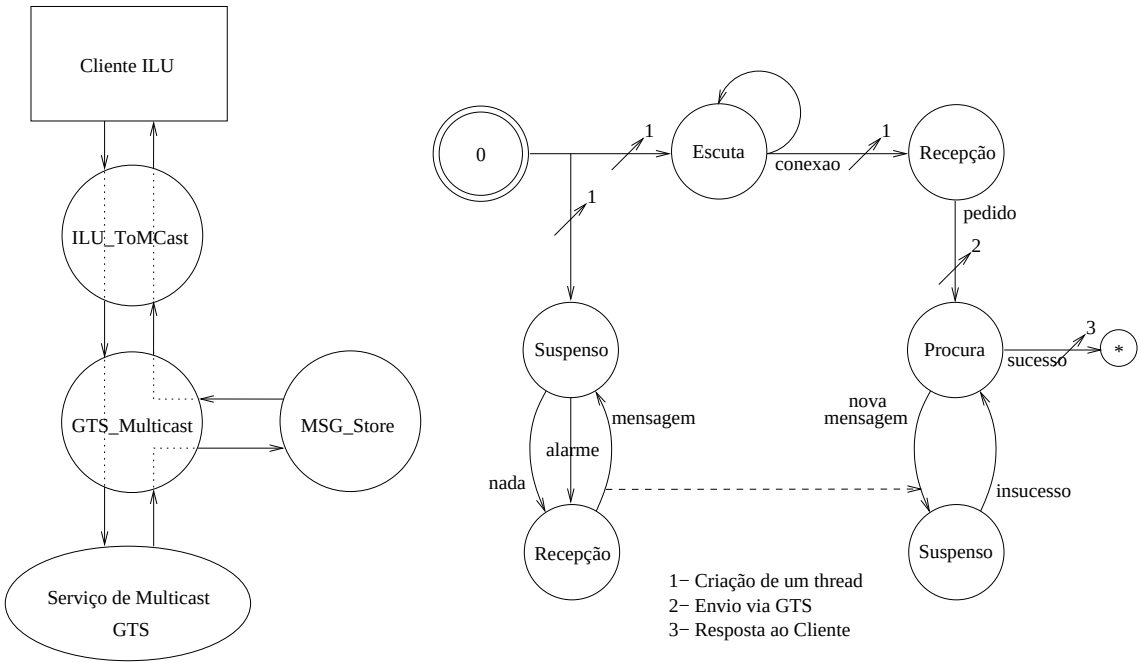

Figura 5.3: Interceptor.

Quando é recebida a conexão de um cliente, é criado um novo fio de execução para processar os pedidos subsequentes. Este processamento implica a composição de uma mensagem GTS para cada pedido, com a atribuição de um número de sequência – encapsulamento do pedido ILU. Esta mensagem é enviada para um conjunto de representantes, o qual constitui um determinado grupo de objectos, seguindo-se um estado de espera até à chegada da primeira resposta. As respostas dos restantes representantes são ignoradas nesta imple-

[8]Do inglês *threads*.

[9]O termo "pedido ILU" é aqui usado para designar qualquer pacote (*ping* ou invocação) que um cliente ILU envie para o servidor.

mentação em concreto, mas poder-se-iam desenvolver estratégias de votação, ou outras, que tirassem partido da recepção das várias respostas.

Em paralelo com este processamento, são recebidas quaisquer mensagens (via GTS) com respostas a pedidos até então efectuados. Estas são armazenadas mediante o seu número de sequência, e a recepção de mensagens com o mesmo número significa que houve mais do que uma resposta ao mesmo pedido, ou seja, vários servidores a processar o mesmo pedido. Neste caso, é armazenada apenas a primeira resposta, independentemente da sua comparação com outras.

Note-se que a recepção das respostas aos pedidos submetidos é totalmente assíncrona. Isto deve-se ao facto de existirem vários fios de execução a submeter mensagens ao serviço de difusão selectiva, correspondentes a pedidos de vários clientes, mensagens essas que desencadearão processamentos de duração variável, e à partida desconhecida, em vários servidores. Desta forma, torna-se necessário identificar univocamente cada uma das mensagens, de maneira a relacionar determinada resposta com o respectivo pedido. De referir que um determinado interceptor possuirá um endereço GTS, por forma a receber as respostas enviadas pelos representantes, mas este endereço é independente do pedido ILU a ser processado num determinado instante.

A espera pela resposta a um pedido, anteriormente mencionada, resulta na consulta do sistema de armazenamento de mensagens; caso não seja encontrada a mensagem em causa (resposta a um determinado pedido), o fio de execução responsável pelo processamento desse pedido é suspenso até à chegada de novas mensagens, iniciando-se então uma nova procura. O fio de execução encarregue da recepção de mensagens provenientes do serviço de difusão selectiva faz a difusão de um sinal para todos aqueles que se encontram bloqueados na variável de condição que indica a chegada de novas mensagens. O diagrama de estados da figura 5.3 demonstra de forma mais clara todo este processamento.

Tendo em vista uma maior portabilidade, o interceptor e o representante foram desenvolvidos de forma a permitir, através de uma opção de compilação, o uso de dois mecanismos de suporte a fios de execução distintos: fios de execução da Sun [60] e fios de execução Posix [61].

A opção pela utilização de fios de execução no desenvolvimento destes dois *daemons* levantou alguns problemas. De facto, o GTS não foi desenvolvido de forma a suportar fios de execução, obrigando à introdução de alguma sincronização entre determinadas primitivas. Na verdade, o GTS usa um único *socket* para interactuar com um sequenciador, e qualquer primitiva resulta no envio e recepção de um pacote para essa entidade. Desta forma, torna-se impossível efectuar duas operações `send` concorrentes, ou mesmo efectuar um `send` concorrentemente com um `receive`. A consequência mais grave desta deficiência

foi a impossibilidade do uso de primitivas bloqueantes[10]. Assim, tornou-se obrigatório o sistema de sondagem[11] para a recepção de mensagens.

O representante tem uma arquitectura relativamente mais simples do que o interceptor, bastando-lhe criar um fio de execução para processar cada pedido recebido (através do GTS). Este processamento implica a criação de um *socket*, para comunicação com o servidor ao qual o representante está associado, o envio do pedido para o servidor e a devolução da resposta ao interceptor. A mensagem de resposta é etiquetada com o número de sequência do pedido correspondente.

5.2.2 Serviço de Directoria

Os clientes ILU interrogam um serviço de directório por forma a encontrar os objectos desejados. Este serviço, na sua variante mais simples, é constituído por um directório partilhado por NFS. Neste directório é criado um ficheiro por cada objecto registado. Este ficheiro, cujo nome é obtido a partir de uma transformação – um resumo[12] – do SBH do objecto, contém, para além do SBH, o endereço IP e o porto que possibilitam a comunicação com o objecto representado.

Por forma a integrar o sistema de intercepção de pedidos com a operação normal de clientes e servidores ILU, e tirando partido deste serviço de directório de grande simplicidade, foram criados mecanismos para:

- definir grupos de objectos;
- activar interceptores;
- activar representantes.

Coexistência de Réplicas

A activação de várias réplicas de um objecto torna-se algo problemática devido ao mecanismo de registo de objectos do ILU. Na verdade, não é possível registar dois objectos com o mesmo identificador (SBH), dado que o último objecto registado é aquele que prevalece, devido à sobreposição dos ficheiros de registo; objectos com o mesmo SBH implicam ficheiros de registo com o mesmo nome. Deste modo, ao registar pela segunda vez um objecto, os clientes deixam de poder interagir com a implementação registada anteriormente; passa a ser "visível" apenas o novo objecto.

[10] O GTS disponibiliza métodos para recepção de mensagens de forma bloqueante ou não.
[11] Do inglês *polling*.
[12] Do inglês *digest*.

Assim, impõe-se a atribuição de identificadores distintos para as réplicas de um determinado objecto. No entanto, a maior parte dos programas que registam objectos ILU não permitem a livre escolha destes identificadores, ou seja, executando pela segunda vez um determinado servidor, este irá registar os mesmos objectos, com os mesmos identificadores, sobrepondo os registos do primeiro.

Por forma a ultrapassar esta limitação, é instalado um *daemon* – `refgen` – que detecta a criação de novos ficheiros de registo (inclusive aqueles que se sobrepõem), analisa o seu conteúdo e adiciona a um ficheiro de referências de objectos (`obj-refs`) uma entrada correspondente ao identificador e à localização do objecto desse registo. Este ficheiro de referências é mantido no mesmo directório que o ILU usa para implementar o seu serviço de directório. As entradas deste ficheiro possuem o formato `SID.IH@IP.port`, onde `SID` e `IH` constituem o OID do objecto e `IP` e `port` indicam a localização do servidor núcleo para esse objecto.

Desta forma, embora o mecanismo de pesquisa do serviço de directório do ILU apenas encontre o último objecto registado, um serviço auxiliar poderá encontrar os anteriores, através da consulta do ficheiro de referências.

Refira-se ainda que o problema da coexistência de réplicas não é específico do ILU. De facto, trata-se de um problema genérico de qualquer sistema que registe objectos ou serviços e que não ofereça por si só suporte à replicação. No entanto, no que respeita à detecção de registos de novos objectos, a solução encontrada (o `refgen`) é particular ao mecanismo de registo de objectos usado actualmente pelo ILU.

Instalação de Representantes

Por cada objecto passível de ser replicado é instalado um representante. Refira-se que seria possível construir representantes mais complexos, com capacidade de interagir com vários objectos ou mesmo com diferentes servidores. No entanto, por razões de simplicidade, quer do modelo quer da concretização destes representantes, optou-se pelo uso de um representante por cada objecto.

Na instalação de representantes, é consultado o ficheiro de referências de objectos (que mantém apontadores para todos os objectos, mesmo aqueles cujo registo foi sobreposto), obtendo-se o endereço IP e o porto a contactar por cada pedido recebido. Cada representante regista o seu identificador num ficheiro de referências de representantes (`prx_refs`), tal como sucede com as referências de objectos.

O representante é identificado por um URL do sistema GTS, ou seja, um endereço do serviço de difusão selectiva. Note-se que os representantes são contactados pelos interceptores através deste serviço de comunicação. Como parte local do URL de um representante

é usado o próprio identificador do objecto, o qual se encontra no ficheiro de referências.

Criação de Grupos de Réplicas

Depois de instalados representantes para os objectos a replicar, impõe-se a criação de identificadores para grupos de objectos, de maneira a poder endereçar um pedido a um conjunto de réplicas.

Um grupo de réplicas é um conjunto de representantes identificado também por um URL do sistema GTS. Visto que o GTS obriga à utilização de números para as partes locais dos URLs respeitantes a grupos, os identificadores de conjuntos de réplicas serão menos representativos que os usados no caso anterior para os representantes.

A criação de um grupo no GTS equivale à criação de um ficheiro com os URLs dos seus elementos. Este ficheiro tem como nome a parte local do URL do grupo que representa, e o serviço de directório do GTS é composto basicamente por ficheiros deste tipo. Neste caso, ter-se-á um conjunto de ficheiros representando grupos de réplicas, ou seja, cada ficheiro será constituído por um conjunto de URLs respeitantes a representantes de objectos ILU.

A criação de grupos de réplicas passa portanto pela consulta do ficheiro de referências de representantes (integrado no serviço de directório do ILU), por forma a identificar possíveis elementos para o grupo a constituir, seguindo-se a criação de um ficheiro com os URLs dos representantes que formarão o novo grupo (integrado no serviço de directório do GTS). A atribuição do identificador do grupo resume-se ao incremento de um contador que dará origem à parte local do URL a utilizar.

Dado que o GTS poderá ser utilizado para diversos fins, existirão no seu serviço de directório ficheiros relativos a grupos que não os criados para suportar a replicação de objectos ILU. Por este motivo, e de forma a evidenciar melhor os grupos de réplicas de objectos ILU, é mantido um ficheiro com referências de grupos e com os respectivos identificadores dos membros de cada grupo (`grp_refs`), de forma análoga aos ficheiros de referências para objectos e referências para representantes, ou seja, integrado no serviço de directório do ILU.

Instalação de Interceptares

Os interceptores, instalados do lado do cliente para cada objecto que se encontre replicado, endereçam um pedido ILU a um conjunto de réplicas, utilizando o URL do respectivo grupo. Assim, a instalação de interceptores necessita consultar o ficheiro de referências de grupos de objectos, de forma a descobrir o URL a utilizar. De referir que, devido ao facto de os URLs dos grupos usarem obrigatoriamente partes locais numéricas, só o conhecimento

dos seus constituintes permite concluir sobre os objectos a que dizem respeito.

Por outro lado, o cliente ILU terá que passar a contactar o interceptor em vez do servidor original. Para tal, aquando da activação do interceptor, é alterado o registo do objecto em causa. Dada a natureza do mecanismo de registo de objectos do ILU, basta alterar o endereço IP e o porto que permitem chegar ao processo com a concretização desse objecto, fazendo reflectir a localização do interceptor. Esta alteração é efectuada no ficheiro que contém o registo do objecto, e o nome desse ficheiro é obtido a partir do ficheiro de referências de objectos; associado a cada entrada deste ficheiro encontra-se o nome do ficheiro (resumo do SBH do objecto) que o ILU usa para registar esse objecto.

Deste modo, sempre que um cliente ILU consulta o serviço de directório com o intuito de encontrar a concretização de um determinado objecto, recebe como resposta a localização do interceptor, passando daí em diante a interagir com este, como se se tratasse do verdadeiro servidor.

5.2.3 Gestor de Referências de Objectos ILU

Os mecanismos que suportam a integração do sistema de intercepção de pedidos ILU na operação normal de clientes e servidores foram integrados numa única aplicação: um gestor de referências de objectos ILU. Esta aplicação – `objmanager` – foi desenvolvida em Tcl/Tk, permitindo assim a gestão de objectos ILU de uma forma simples e eficaz. A figura 5.4 mostra o aspecto desta aplicação.

O gestor de referências permite visualizar os objectos registados, mediante a consulta do ficheiro de referências que é actualizado periodicamente pelo *daemon* encarregue de detectar novos registos. Para cada objecto, pode ser instalado um representante ou ser efectuado o redireccionamento do registo desse objecto (activação de um interceptor). Na activação de interceptores, para além das tarefas mencionadas na secção anterior, é guardada informação num ficheiro específico (`interceptions`), que permite averiguar qual o interceptor a ser utilizado para um determinado objecto (endereço IP, porto e identificador do processo) e qual o grupo de objectos para o qual esse interceptor envia os pedidos recebidos. Na instalação de representantes, também é guardada informação adicional associada ao identificador de cada representante, nomeadamente o endereço IP e o identificador do processo, por forma a localizar os processos que implementam os vários representantes.

Quando se tenta redireccionar um determinado objecto pela segunda vez, o que significaria activar um segundo interceptor, tal facto é detectado através do ficheiro de intercepções e é dada a possibilidade de repor o registo original. Desta forma, é sempre possível fazer com que os clientes voltem a contactar os objectos directamente, sem recorrer ao sistema

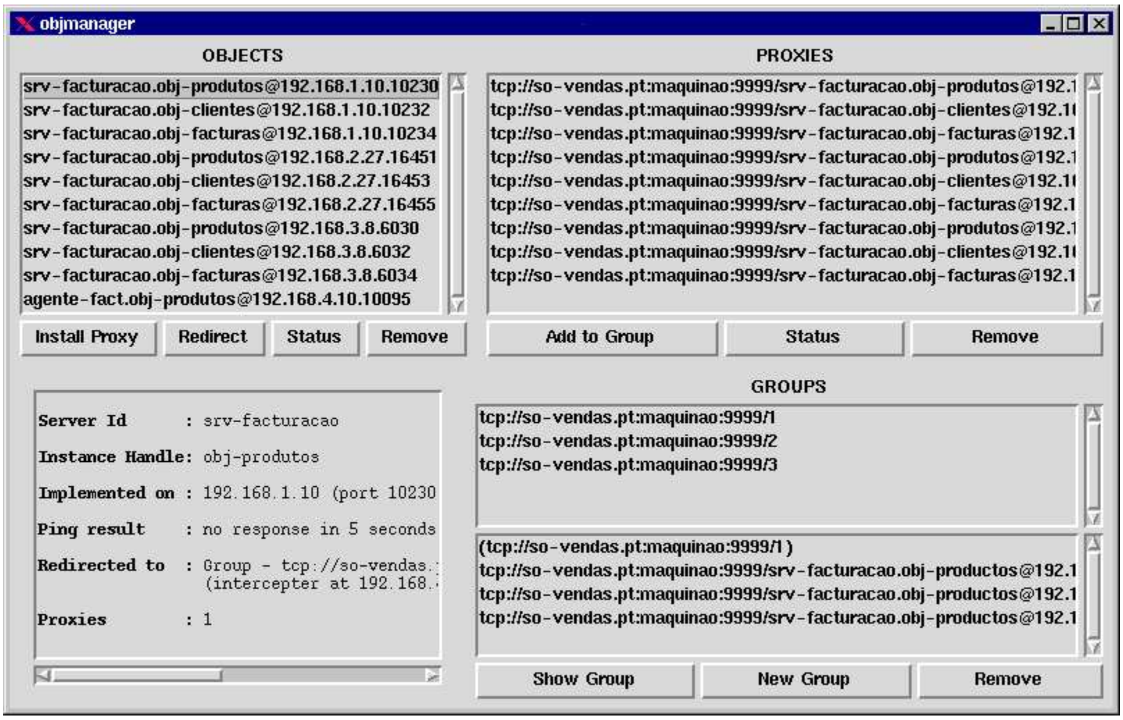

Figura 5.4: Gestor de referências.

de intercepção de pedidos. Isto é conseguido visto que para cada objecto é armazenado o endereço IP e o porto do seu servidor núcleo no momento em que é detectada a operação de registo. Esta reposição implica ainda desinstalar o interceptor anteriormente activado, o que é conseguido através do identificador do processo armazenado no momento da activação.

Para cada objecto, há hipótese de visualizar alguma informação relativa ao seu estado. Assim, pode-se averiguar se o servidor núcleo de um objecto se encontra operacional ou não, através do envio de um pacote de *ping* do ILU, pode-se descobrir se o registo de um objecto aponta para a implementação desse objecto ou para um determinado grupo de representantes, pode-se identificar qual o interceptor utilizado no redireccionamento de um objecto e se se encontra operacional ou não (os interceptores também dispõe de mecanismos para responder a mensagens de *ping*) e pode-se ver o número de representantes instalados para um dado objecto.

Os representantes podem ser adicionados a grupos previamente criados e, tal como nos objectos, pode-se visualizar alguma informação de estado sobre cada representante, nomeadamente se este se encontra operacional ou não (tal como os interceptores, os representantes também respondem a mensagens de *ping*), a sua localização e o sequenciador GTS

utilizado.

Existe ainda a possibilidade de remover grupos, representantes e registos de objectos. Na remoção de um representante, também é efectuada a sua desinstalação, mediante o identificador do processo armazenado no ficheiro de referências de representantes.

5.2.4 Funcionamento em Larga-Escala

Até ao momento, o sistema de invocações múltiplas foi apresentado tendo como pressupostos que todo o universo de objectos se encontra disponível através de um único serviço de directório (que não passa de um directório partilhado por NFS) e que o sistema de difusão selectiva – o GTS – actua como uma entidade centralizada, com um único sequenciador, havendo conhecimento de todos os grupos de réplicas criados. No entanto, nem sempre é possível ou desejável que tal aconteça. De facto, determinados factores contribuem para tal, nomeadamente as partições a nível de rede, a operação em regime de desconexão de determinados sistemas (computadores portáteis, por exemplo), o custo elevado que algumas ligações apresentam e que impede a conexão permanente, razões administrativas que impossibilitam a partilha de ficheiros entre determinados sistemas, etc.

A figura 5.5 apresenta uma situação de um sistema de invocações múltiplas amplamente distribuído. Neste caso, podem-se observar três zonas, cada uma com o seu serviço de directório ILU independente e com um sequenciador GTS exclusivo, ao qual se encontra associado um serviço de directório próprio. Uma zona representa um conjunto de máquinas com total conectividade entre si, onde residem clientes e servidores de objectos ILU. Entre as várias zonas, a interacção será mais limitada, pelos motivos apontados no parágrafo anterior.

Em cada zona são registados objectos ILU, são instalados representantes para os objectos locais com replicação e são criados grupos a partir de representantes locais. A criação de grupos constituídos por réplicas dispersas por várias zonas é efectuada na zona de um dos representantes, sendo necessário coligir os ficheiros de referências de representantes das várias zonas no local onde o grupo é criado. Na instalação de interceptores, é necessário efectuar o mesmo tipo de operação para as referências de grupos e para os registos dos objectos.

Neste contexto, os interceptores apenas têm efeito na zona onde são instalados, ou seja, apenas os pedidos dos clientes dessa zona serão interceptados. Isto deve-se ao facto de o processo de instalação de interceptores alterar o registo dos objectos apenas no serviço de directório local. Desta forma, definindo zonas de forma apropriada, podem-se ter diferentes clientes a usarem o mesmo objecto com replicação ou sem replicação. Note-se que se podem ainda constituir grupos não disjuntos de réplicas, para além de se poderem

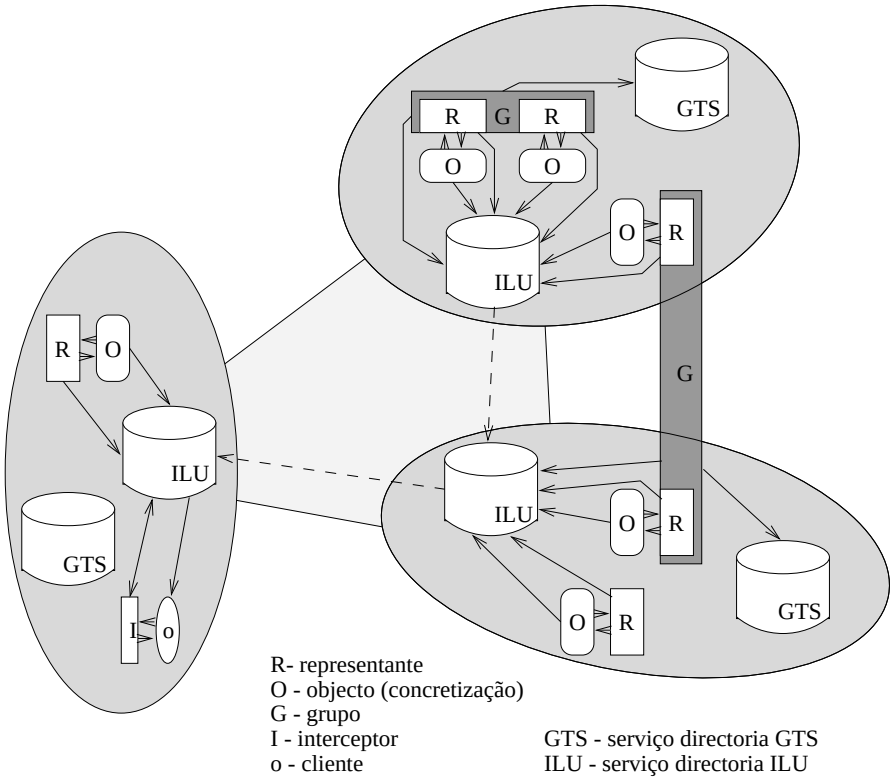

Figura 5.5: Serviço de directório distribuído.

instalar vários representantes para o mesmo objecto. Assim, obtém-se um vasto leque de configurações para a interacção entre clientes e servidores de objectos ILU.

5.3 Difusão de Pedidos ILU

A comunicação em grupo abre grandes perspectivas para o aparecimento de aplicações distribuídas cooperativas ou aplicações com componentes replicados. No entanto, determinadas soluções necessitam, ou podem beneficiar, de sistemas de difusão de informação. Neste caso, e tomando como base o modelo de distribuição de objectos do ILU, uma aplicação invoca operações sobre objectos remotos com a finalidade de divulgar determinada informação, e sem esperar qualquer tipo de reposta.

A particularidade deste tipo de comunicação reside no desconhecimento dos destinatários de determinado pedido. De facto, esta é a característica particular dos mecanismos de difusão; qualquer potencial receptor deverá ser capaz de receber e interpretar os pedidos efectuados.

5.3.1 Adaptação do Sistema de Invocações Múltiplas

Tendo em vista a difusão de pedidos ILU, o interceptor e o representante foram alterados, passando a existir as variantes `DILUintercepter` e `DILUproxy`, de maneira a não ser necessária a utilização do serviço de difusão selectiva GTS. Na realidade, este é o verdadeiro obstáculo à difusão de informação no sistema até aqui apresentado, pois a generalidade dos mecanismos de comunicação em grupo, e em particular o GTS, obrigam à definição do conjunto de receptores antes do envio de uma mensagem.

Dado que se pretende um serviço de distribuição onde cada cliente faz pedidos destinados a uma determinada gama de objectos com determinadas características, sem que seja necessário enumerar esse conjunto de objectos, surge a necessidade de criar um novo conceito de grupo e um mecanismo de transporte independente dos destinatários.

O sistema proposto, representado na figura 5.6, baseia-se no armazenamento persistente dos pedidos dos clientes, juntamente com a identificação do objecto a que se destinam. Fundamentalmente, trata-se de um sistema de armazenamento e reenvio[13], onde os pedidos são transferidos entre áreas de armazenamento, daí resultando a sua difusão.

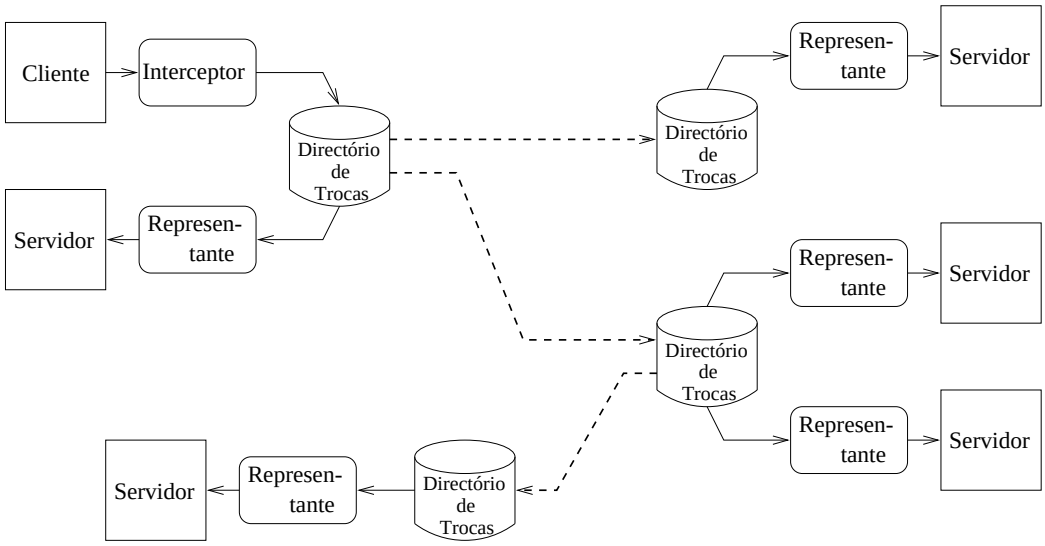

Figura 5.6: Difusão de pedidos ILU.

Por cada invocação de um cliente ILU, é criado um ficheiro num determinado directório, ficheiro esse que conterá toda a informação (sequência de bytes) que o cliente enviaria para o servidor numa situação normal. Os vários pedidos armazenados num dado disco duro – pedidos ILU "congelados" – poderão ser copiados ou transferidos para outra máquina,

[13]Do inglês *store and forward*.

a fim de serem entregues a servidores ILU aí instalados ou para serem armazenados e posteriormente copiados para um terceiro sistema. O processo de cópia de pedidos ILU entre sistemas de ficheiros poderá repetir-se indefinidamente e no momento em que os utilizadores assim o desejarem.

Este sistema pode ser comparado ao mecanismo de difusão das News [62], no qual um determinado servidor recebe registos para disponibilizar a leitores locais ou para transferir para um outro servidor, servindo neste último caso como intermediário. Em [63] é apresentado um modelo para invocações ILU que envolvam portáteis – *indirect calls* – o qual utiliza uma abordagem semelhante à aqui adoptada.

A concretização deste sistema implicou as seguintes alterações ao sistema de difusão selectiva de pedidos ILU:

- Os interceptores passam a gravar num directório apropriado – directório de trocas ou de entrada/saída – os pedidos recebidos de clientes ILU, em vez de os submeterem ao serviço de difusão selectiva.

- Os interceptores devolvem de imediato uma resposta aos clientes, pois neste tipo de invocações o tempo de resposta dos servidores é indeterminado. Aliás, a resposta à difusão de informação serviria apenas para indicar a recepção de um pedido e/ou a conclusão de determinada execução por parte de um servidor, já que nunca existe a devolução de quaisquer resultados. Assim, os clientes devem interpretar a resposta à invocação como uma garantia da entrega do pedido ao sistema encarregue de efectuar a sua difusão.

- Os representantes lêem o directório de trocas, processando cada novo pedido a eles destinado.

- A transferência de pedidos entre directórios de trocas de sistemas distintos é efectuada via `ftp` ou através de outro qualquer sistema para cópia de ficheiros remotos.

5.3.2 Movimentação de Informação

Nesta solução, a transferência de informação entre clientes e servidores baseia-se na utilização de directórios de trocas. Dado que poderá existir uma grande variedade de objectos para difusão de informação, terá que existir alguma forma de identificar os vários pedidos que se encontram num directório, de maneira a que os representantes possam obter apenas os pedidos que lhes dizem respeito.

Na movimentação de pedidos entre directórios de trocas, que no fundo é o mecanismo que permite fazer chegar os pedidos aos representantes espalhados por uma rede, também

é necessário algum meio para distinguir pedidos oriundos de locais diferentes. Note-se que, quando um cliente efectua uma invocação e esta é interceptada, o pedido em causa é escrito pelo interceptor num determinado directório, mas depois poderá ser copiado para outros directórios, já que esta é a forma de fazer evoluir o processo de divulgação desse pedido. Num ambiente de computação nómada, por exemplo, pode-se pensar no computador portátil como um meio de transporte de informação, já que ao ser conectado em diferentes pontos da rede poderá efectuar trocas de pedidos entre sistemas.

Posto isto, adoptaram-se as seguintes medidas, por forma a controlar a informação transferida entre interceptores e representantes:

- a cada directório de trocas é atribuído um identificador único;
- em cada directório de trocas é mantido um ficheiro com uma entrada por cada pedido ILU aí armazenado – identificação dos pedidos;
- cada pedido é identificado pelo OID do objecto a que se destina, juntamente com o nome do ficheiro onde se encontra o pacote ILU correspondente a esse pedido;
- o nome do ficheiro usado para guardar um pedido é obtido no momento em que o interceptor procede ao seu armazenamento, com base no identificador do directório de trocas e num contador local a esse directório, que é incrementado por cada pedido aí armazenado.

Desta forma, consegue-se identificar univocamente cada pedido, independentemente de este já ter sido copiado ou não para um directório de trocas distinto daquele em que foi guardado pelo interceptor. De referir que a transferência de um pedido entre directórios obriga, para além da transferência do ficheiro que contém o pacote ILU, à actualização do ficheiro que mantém a identificação dos pedidos. Os nomes dos ficheiros e as identificações dos pedidos não são modificados no processo de transferência.

A unicidade dos identificadores dos pedidos permite garantir que os representantes não processarão mais que uma vez cada pedido e que esses pedidos não sejam duplicados num determinado directório. Note-se que um determinado pedido pode chegar a um directório de várias formas e podem até ocorrer ciclos no processo de transferência de um pedido.

5.3.3 Integração no Gestor de Referências

O gestor de referências anteriormente apresentado é também usado para activar representantes e interceptores para o sistema de difusão. Na verdade, todo o sistema se comporta da mesma forma, com excepção dos identificadores dos representantes. Estes identificadores deixam de ser URLs do sistema GTS e passam a ter o formato `broadcast:io-dir:obj-ref`,

onde `io-dir` designa o directório de trocas a consultar e `obj-ref` diz respeito ao identificador do objecto. O prefixo `broadcast` destina-se a identificar o representante como pertencente ao sistema de difusão.

Refira-se também que os representantes destinados à difusão não podem ser usadas na constituição de grupos. Na realidade, na difusão de pedidos não se consideram grupos de objectos. Os destinatários de um determinado pedido serão todos os representantes instalados para concretizações do objecto em causa.

Na instalação de representantes, o `objmanager` passa a perguntar se o objecto vai ser usado para difusão de informação ou não, por forma a decidir qual a variante do representante que deve ser usada. Através do identificador usado para o representante, a activação de um interceptor pode efectuar-se de forma inequívoca.

Capítulo 6

Exemplo de Aplicação

A apresentação das anotações de desconexão e do mecanismo de replicação de objectos ILU foi efectuada sem recurso a exemplos concretos de aplicação; a sua aplicabilidade ficou, em certa medida, subentendida na descrição do seu funcionamento.

Neste capítulo é explorado como exemplo de aplicação o problema da facturação, cuja abordagem inicial, no que respeita ao desenvolvimento da aplicação distribuída mediante a utilização do sistema ILU, já foi efectuada na secção 3.3. Este exemplo destina-se a realçar, de forma generalizada, a utilidade das anotações de desconexão em conjunção com o mecanismo de suporte à replicação de objectos (servidores).

6.1 Reequacionamento do Problema

Aquando da exposição do processo de construção da aplicação distribuída de facturação, ficou patente que o objectivo era a interligação das lojas ao armazém central. No entanto, nada foi referido sobre as infra-estruturas de comunicação utilizadas para tal interligação, nem sobre as imposições (requisitos) da organização, em termos de disponibilidade de informação. De facto, o comportamento da aplicação em situações de falta de conectividade é até certo ponto desconhecido.

Na verdade, na solução apresentada assumiu-se que todos os intervenientes operam em condições óptimas por um período indefinido de tempo. No entanto, tal pressuposto não é aceitável perante um caso real. Por outro lado, exigências de acordo com as tecnologias mais recentes (computação nómada, por exemplo) tornam a solução proposta de certo modo inadequada.

Considere-se então o seguinte reequacionamento do problema da interligação dos locais de venda ao armazém, ou seja, os novos requisitos do sistema de facturação:

- os pontos de venda poderão ficar, embora temporariamente, isolados em relação ao armazém, devido a problemas da infra-estrutura de comunicação;
- alguns pontos de venda poderão ser sistemas móveis – vendedores viajantes munidos de computadores portáteis;
- para além do armazém até então considerado, passarão a existir um outro armazém e um escritório (sede da empresa);
- apesar das faltas de comunicação, a operação dos pontos de venda deverá ser sempre possível, dentro de certos limites;
- os servidores existentes nos dois armazéns e no escritório deverão ser réplicas fiéis, reflectindo, qualquer um deles, os movimentos de todos os pontos de venda.

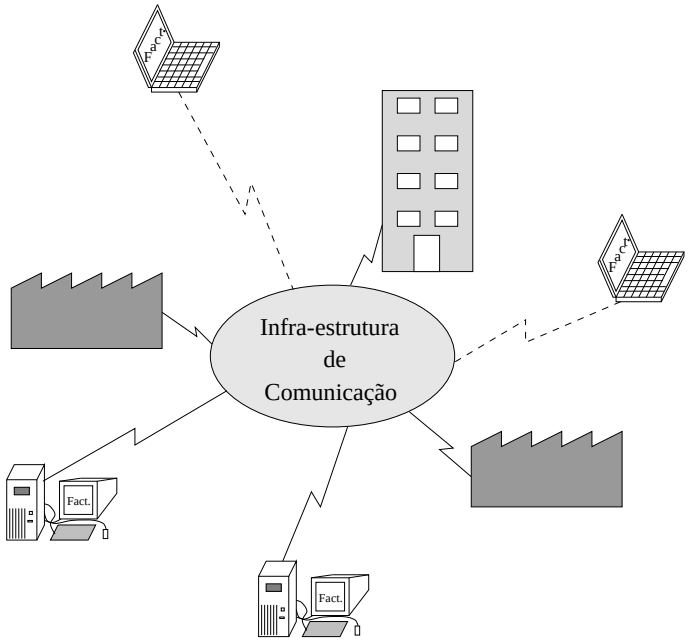

Figura 6.1: Facturação distribuída.

A figura 6.1 mostra a configuração do novo sistema de facturação. Neste esquema encontram-se representados os sistemas portáteis, que através de ligações "frágeis" (comunicação sem fios) comunicam com os servidores, e os sistemas fixos das lojas, os quais terão ligações algo mais fiáveis, mas não isentas de probabilidade de falha. Os servidores, instalados nos armazéns e no escritório, para além de oferecerem informação a eventuais clientes locais, possibilitam, no seu conjunto, um aumento da disponibilidade da informação para os clientes dos vários pontos de venda, dado que na impossibilidade de contactar um

determinado servidor poderá ser utilizado um outro qualquer. Obviamente, a utilização de servidores replicados terá ainda a vantagem de garantir uma maior fiabilidade à aplicação.

Fundamentalmente, está em causa a necessidade de aumentar a disponibilidade e a fiabilidade do sistema informático da organização, o qual foi desenvolvido sem a preocupação da tolerância a faltas. Na prática, qualquer aplicação desenvolvida segundo as tradicionais metodologias Cliente-Servidor não é por si só suficientemente flexível para resolver os problemas levantados no momento em que são considerados requisitos do género dos apontados anteriormente. Uma solução para estes casos seria, por exemplo, a utilização de aplicações comerciais do género do Lotus Notes. No entanto, o facto de esta ferramenta ser orientada ao documento dificulta o tratamento de bases de dados relacionais, nas quais se baseia a maioria dos sistemas de informação das empresas. Por outro lado, o mecanismo de resolução de conflitos na reconciliação de réplicas é praticamente manual, no caso do Lotus Notes, apesar de poderem ser utilizadas estratégias de replicação optimistas.

Recorrendo aos mecanismos de geração de agentes e replicação de objectos desenvolvidos no âmbito desta dissertação, a migração de aplicações construídas de acordo como o modelo Cliente-Servidor para o modelo Cliente-Agente-Servidores garante o cumprimento dos requisitos mencionados, sem a necessidade de se efectuarem alterações nos clientes ou no servidor.

6.2 Replicação de Servidores

6.2.1 Instalação de Servidores

Conforme evidenciado no capítulo 5, o mecanismo de replicação de objectos ILU não requer qualquer alteração dos servidores desenvolvidos para a operação sem replicação. Assim, serão utilizados os servidores obtidos segundo o processo apresentado na secção 3.3, os quais poderiam até já se encontrar em funcionamento segundo o tradicional modelo Cliente-Servidor.

No arranque de cada um dos servidores, um em cada armazém e um terceiro no escritório, ocorrerá o registo dos objectos implementados por estes. Tal operação obrigará à partilha de um directório por NFS, por forma a tornar visível a toda a organização a totalidade dos objectos registados. Esta imposição, que em princípio será eliminada em futuras versões do sistema ILU com o aparecimento de um novo serviço de nomes, é facilmente ultrapassável, dado que podem ser efectuadas cópias locais de tal directório. De facto, as esporádicas actualizações dos ficheiros mantidos neste directório[1] minimizam o impacto

[1] Apenas há alteração ou criação de ficheiros no momento em que se registam novos objectos, e neste caso não existe criação dinâmica de objectos; os objectos `produtos`, `clientes` e `facturas` são criados e publicados uma única vez, a não ser que haja necessidade de reiniciar algum servidor.

de se manterem cópias locais (caches).

Após a activação dos servidores, e devido à actuação do *daemon* `refgen`, são criadas as seguintes entradas no ficheiro de referências de objectos:

```
srv-facturacao.obj-produtos@192.168.1.10.10230 (f6775b36)
srv-facturacao.obj-clientes@192.168.1.10.10232 (a56c78d4)
srv-facturacao.obj-facturas@192.168.1.10.10234 (97bge30c)
srv-facturacao.obj-produtos@192.168.2.27.16451 (f6775b36)
srv-facturacao.obj-clientes@192.168.2.27.16453 (a56c78d4)
srv-facturacao.obj-facturas@192.168.2.27.16455 (97bge30c)
srv-facturacao.obj-produtos@192.168.3.8.6030 (f6775b36)
srv-facturacao.obj-clientes@192.168.3.8.6032 (a56c78d4)
srv-facturacao.obj-facturas@192.168.3.8.6034 (97bge30c)
```

Observe-se a existência de três entradas por cada objecto, correspondentes às três réplicas do servidor. O endereço IP e o porto, que aparecem a seguir ao SID e ao IR do objecto, identificam a localização do objecto, enquanto que o nome entre parêntesis designa o ficheiro de registo usado pelo ILU. De referir que os endereços IP aqui apresentados foram escolhidos a título de exemplo, não correspondendo a nenhuma configuração de máquinas existente. No entanto, será perfeitamente normal que uma organização deste tipo possua um rede IP em cada armazém e no escritório, ligadas entre si por linhas alugadas, por exemplo.

As sucessivas operações de registo dos mesmos objectos, por parte dos vários servidores, leva a que apenas os últimos a serem publicados sejam visíveis, ou seja, os objectos do servidor instalado na máquina `192.168.3.8` são aqueles que os clientes ILU contactam após a consulta do serviço de nomes.

6.2.2 Criação de Grupos de Objectos

Depois de instalados os três servidores, e de acordo com o mecanismo desenvolvido para a replicação de objectos ILU, torna-se necessário criar grupos de objectos. Para tal, começa-se por instalar um representante para cada objecto. Esta operação pode ser desencadeada através do `objmanager`. No final, para além da activação dos *daemons* correspondentes aos representantes, obter-se-ão as seguintes entradas no ficheiro de referências de representantes[2]:

[2]Devido à extensão das linhas deste ficheiro, foi necessário "partir" cada uma das entradas. Assim, aquelas linhas cujo início se dá mais à esquerda constituem a continuação das entradas iniciadas nas linhas imediatamente anteriores.

```
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-produtos@192.168.1.1
0.10230 192.168.1.10 2620
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-clientes@192.168.1.1
0.10232 192.168.1.10 2625
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-facturas@192.168.1.1
0.10234 192.168.1.10 2633
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-produtos@192.168.2.2
7.16451 192.168.2.27 8745
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-clientes@192.168.2.2
7.16453 192.168.2.27 8756
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-facturas@192.168.2.2
7.16455 192.168.2.27 8760
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-produtos@192.168.3.8
.6030 192.168.3.8 456
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-clientes@192.168.3.8
.6032 192.168.3.8 459
  tcp://so-vendas.pt:maquinao:9999/srv-facturacao.obj-facturas@192.168.3.8
.6034 192.168.3.8 461
```

As várias entradas deste ficheiro correspondem aos URLs dos representantes instalados (endereços do sistema GTS que permitem contactar cada um desses representantes), seguidos de um endereço IP e de um identificador de processo. Estes últimos destinam-se a localizar o *daemon* correspondente. Cada representante deverá ser instalado na máquina onde reside o servidor que implementa o objecto correspondente, por forma a evitar a falta de comunicação entre estes.

A partir dos representantes instalados podem ser criados grupos. Neste caso existem três objectos distintos replicados em três servidores, isto é, três réplicas de cada objecto. Dado que cada objecto possui o seu representante, serão estabelecidos três grupos, um para cada conjunto de representantes do mesmo objecto. Da operação de criação destes grupos resultará o ficheiro de referências a seguir apresentado[3]:

```
  tcp://so-vendas.pt:maquinao:9999/1
  tcp://so-vendas.pt:maquinao:9999/1 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-produtos@192.168.1.10.10230
  tcp://so-vendas.pt:maquinao:9999/1 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-produtos@192.168.2.27.16451
  tcp://so-vendas.pt:maquinao:9999/1 : tcp://so-vendas.pt:maquinao:9999/sr
```

[3]Tal como no caso do ficheiro de referências de representantes, tornou-se necessário dividir as entradas respeitantes aos membros de cada grupo em duas linhas.

```
v-facturacao.obj-produtos@192.168.3.8.6030
  tcp://so-vendas.pt:maquinao:9999/2
  tcp://so-vendas.pt:maquinao:9999/2 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-clientes@192.168.1.10.10232
  tcp://so-vendas.pt:maquinao:9999/2 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-clientes@192.168.2.27.16453
  tcp://so-vendas.pt:maquinao:9999/2 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-clientes@192.168.3.8.6032
  tcp://so-vendas.pt:maquinao:9999/3
  tcp://so-vendas.pt:maquinao:9999/3 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-facturas@192.168.1.10.10234
  tcp://so-vendas.pt:maquinao:9999/3 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-facturas@192.168.2.27.16455
  tcp://so-vendas.pt:maquinao:9999/3 : tcp://so-vendas.pt:maquinao:9999/sr
v-facturacao.obj-facturas@192.168.3.8.6034
```

Neste ficheiro podem ser observados os seguintes grupos de objectos:

- grupo 1 (`tcp://so-vendas.pt:maquinao:9999/1`) – engloba todos os objectos cujo IH é `obj-produtos`;
- grupo 2 (`tcp://so-vendas.pt:maquinao:9999/2`) – engloba todos os objectos cujo IH é `obj-clientes`;
- grupo 3 (`tcp://so-vendas.pt:maquinao:9999/3`) – engloba todos os objectos cujo IH é `obj-facturas`.

Estes grupos permitem que um pedido ILU efectuado sobre um determinado objecto remoto seja entregue a um conjunto de representantes, os quais interactuam com as várias réplicas do objecto em causa. Visto que a entrega de pedidos se efectua através do sistema GTS, antes ainda da activação de representantes, impõe-se a instalação de um ou mais sequenciadores GTS. O prefixo presente em todos os URLs apresentados – `tcp://so-vendas.pt:maquinao:9999` – identifica a localização do sequenciador utilizado, nomeadamente o nome da máquina (`maquinao.so-vendas.pt`[4]) e o porto (`9999`).

6.2.3 Instalação de Clientes

A instalação de representantes e sequente criação de grupos não interfere com o serviço de nomes do ILU. Como tal, qualquer pedido de um determinado cliente ILU continuará

[4]O nome da máquina e o respectivo domínio foram escolhidos a título de exemplo, não correspondendo a algum sistema conhecido.

a chegar a um único objecto – o objecto correspondente à última réplica a ser registada. É então necessário instalar interceptores e alterar o registo de cada um dos três objectos, para que os clientes passem a contactar grupos de objectos replicados.

Com as sucessivas instalações de servidores (dos armazéns e do escritório), o directório que o ILU utiliza para implementar o serviço de directório incluirá três ficheiros de registo, um para cada objecto[5]. Se se optar por efectuar cópias locais deste directório em cada máquina ou rede local onde se instalem clientes, os interceptores podem ser activados localmente. Note-se que a instalação de mais que um interceptor para um mesmo objecto, em várias máquinas que partilham o directório do serviço de nomes ILU, é completamente inútil. Na verdade, as sucessivas actualizações do ficheiro de registo do objecto em causa, correspondentes à instalação dos vários interceptores, resultariam na activação de um único interceptor, o último a ser instalado, devido à sobreposição das actualizações referidas.

Posto isto, depois de instalados interceptores numa máquina de uma determinada loja, na qual existirá um cliente ILU, serão criadas as seguintes entradas no ficheiro de intercepções[6]:

```
  f6775b36 -> 192.168.4.10.1029 148 -> tcp://so-vendas.pt:maquinao:9999/1
(Group)
  a56c78d4 -> 192.168.4.10.1031 149 -> tcp://so-vendas.pt:maquinao:9999/2
(Group)
  97bge30c -> 192.168.4.10.1033 152 -> tcp://so-vendas.pt:maquinao:9999/3
(Group)
```

Neste ficheiro pode-se observar, para cada objecto, o nome do ficheiro de registo criado pelo servidor aquando da publicação desse objecto, a identificação do interceptor a utilizar (endereço IP, porto e identificador do processo) e o endereço (URL) do grupo de representantes ao qual o interceptor entregará cópias dos pedidos recebidos de um cliente. O qualificador `(Group)`, presente no final de cada entrada deste ficheiro, destina-se a indicar que o URL especificado diz respeito a um grupo, já que um interceptor também pode ser usado para enviar pedidos para um único representante (qualificador `(proxy)`), no lugar de um grupo.

O endereço IP e o porto respeitantes ao interceptor são usados para actualizar o ficheiro de registo do objecto. Assim, no caso do objecto designado pelo SID `srv-facturacao` e pelo IH `obj-produtos`, a informação de registo passará a ser seguinte:

```
  ilu:srv-facturaca%bj-produtos;ilu%3AlO-PBAXwc-GG4J-jUz6CV4JloGG;sunrpc_2
```

[5]Conforme já referido, as sucessivas operações de registo de um dado objecto resultam na sobreposição do ficheiro utilizado para o efeito.

[6]As três entradas deste ficheiro também foram divididas em duas linhas, devido à sua extensão.

```
_0x61a78_2571064954@sunrpcrm=tcp_192.168.4.10_1029
```

Imediatamente antes da instalação do interceptor, o endereço IP e o porto deste registo seriam `192.168.3.8` e `6030`, reflectindo a localização do último objecto registado. Depois de instalado o interceptor, o registo é redireccionado para este.

6.3 Operação Isolada

Até ao momento, foi descrito o processo para obtenção do sistema de facturação replicado, ou seja, com servidores instalados em vários locais. Esta solução permite aumentar a disponibilidade e a fiabilidade de todo o sistema. No entanto, para o caso de se terem clientes isolados em relação às infra-estruturas de comunicação onde residem os servidores, o que poderá ocorrer em especial para os vendedores munidos de computadores portáteis, a operação dos pontos de venda será completamente suspensa. Por forma a solucionar este problema, impõe-se dotar os clientes de alguma autonomia, no que respeita ao desempenho de determinadas tarefas. Esse é o papel dos agentes gerados a partir das anotações apresentadas no capítulo 4.

6.3.1 Anotação da Interface

Mediante a utilização do editor de anotações, a especificação da interface desenvolvida na secção 3.3, mais precisamente as declarações de objectos e respectivos métodos, passará a incluir descrições relativas ao comportamento dos clientes em caso de desconexão. A seguir apresenta-se o fragmento da descrição da interface correspondente às declarações dos objectos `produtos`, `clientes` e `facturas` anotadas com as primitivas de desconexão[7]. As parametrizações relativas às várias primitivas foram, neste caso, escolhidas com base no bom senso. No entanto, factores como a dimensão da memória ou requisitos de desempenho da aplicação em causa poderão ditar outros valores para estas parametrizações.

```
TYPE produtos = OBJECT
  ON RECONNECTION efectua_saidas_stock
  METHODS
    stock (codigo :  codigo-produto) :  valor-quantidade
      STORE LAST 100 RESULTS
        INDEXED BY EVALUATION OF PARAMETERS
        RETURN LAST WHEN DISCONNECTED
```

[7] As linhas não sublinhadas correspondem ao código vindo da especificação original da interface, ou seja, ao código isento de anotações de desconexão.

```
    TRY 3 TIMES WAITING 5 SECONDS
    PREFETCH RESULTS WAITING 5 SECONDS
    RAISES produto-inexistente END,
  preco (codigo :  codigo-produto) :  valor-monetario
    STORE LAST 100 RESULTS
      INDEXED BY EVALUATION OF PARAMETERS
      RETURN LAST WHEN DISCONNECTED
    TRY 3 TIMES WAITING 5 SECONDS
    RAISES produto-inexistente END,
  saida-stock (codigo :  codigo-produto, quantidade :  valor-quantidade)
    ON SUCCESS decrementa_stock_local
    ALTERNATIVE armazena_saida_stock
    RAISES produto-inexistente, stock-insuficiente END
 END;

TYPE clientes = OBJECT
 BEFORE DISCONNECTION traz_ultimos_plafonds
 ON RECONNECTION efectua_gastos_plafond
 METHODS
  contribuinte (codigo :  codigo-cliente) :  numero-contribuinte
    STORE LAST 100 RESULTS
      INDEXED BY EVALUATION OF PARAMETERS
      RETURN LAST ALWAYS
    TRY 3 TIMES WAITING 5 SECONDS
    RAISES cliente-inexistente END,
  plafond (codigo :  codigo-cliente) :  valor-monetario
    STORE LAST 5 RESULTS
      FOR EACH EVALUATION OF PARAMETERS
    RETURN USING plafond_medio WHEN DISCONNECTED
    TRY 3 TIMES WAITING 5 SECONDS
    RAISES cliente-inexistente END,
  gasta-plafond (codigo :  codigo-cliente, valor :  valor-monetario)
    ON SUCCESS decrementa_plafond_local
    ALTERNATIVE armazena_gasto_plafond
    RAISES cliente-inexistente, plafond-insuficiente END,
  repoe-plafond (codigo :  codigo-cliente, valor :  valor-monetario)
    WAIT 3600 SECONDS
    ON SUCCESS incrementa_plafond_local
```

```
    RAISES cliente-inexistente END
  END;

TYPE facturas = OBJECT
  METHODS
    cria-cabecalho (cabecalho :  cabecalho-factura)
      WAIT 900 SECONDS,
    cria-linhas (numero :  numero-factura, linhas :  linhas-factura)
      WAIT 900 SECONDS
      RAISES factura-inexistente END,
    paga-factura (numero :  numero-factura)
      WAIT 900 SECONDS
      RAISES factura-inexistente END
  END;
```

As anotações adicionadas à descrição da interface original tem por finalidade permitir que o cliente desencadeie invocações remotas sem a necessidade de o servidor se encontrar contactável. Assim, para cada um dos objectos, ter-se-á um conjunto de operações desempenhadas pelo agente que são destinadas a aumentar a disponibilidade dos serviços requeridos pelos clientes.

Objecto produtos

No que respeita ao objecto produtos, sempre que for invocado o método `stock` ou o método `preco`, os resultados devolvidos pelo servidor serão armazenados[8]. Por forma a evitar o crescimento desmesurado da estrutura de dados destinada a guardar tais resultados, as primitivas `STORE` utilizadas nos dois métodos incluem um limite de `100` resultados. Desta forma, apenas serão guardados cem pares `<código de produto, valor em stock>` e cem pares `<código de produto, preço>`, correspondentes às últimas invocações. Com base nestas duas caches, o cliente poderá obter determinadas respostas sem a necessidade de se contactar o servidor. Neste caso, foi indicado que só serão devolvidos valores previamente armazenados se o servidor se encontrar incontactável.

Para estes dois métodos, foi ainda especificado que em caso de insucesso na tentativa de contactar o servidor deverão ser efectuadas até três tentativas, com intervalos de cinco segundos.

No método `stock`, optou-se pela utilização da primitiva `PREFETCH`, com o intuito de actu-

[8]Desde que o servidor se encontre disponível e responda ao pedido efectuado.

alizar periodicamente os valores armazenados. Note-se que os valores das existências dos produtos mudarão com uma frequência bastante mais elevada que os respectivos preços, facto que levaria a uma maior probabilidade de se usarem valores desactualizados, no caso de uma desconexão.

Relativamente à operação `saida-stock`, dado que diz respeito a uma actualização no servidor, recorreu-se à anotação `ALTERNATIVE`, a qual possibilita indicar uma operação a desencadear localmente, para o caso de não se poder invocar o método remoto. Neste exemplo, a alternativa local – `armazena_saida_stock` – deverá testar o valor da existência armazenado localmente e, no caso de este ser superior ao valor pretendido para a saída, ocupar-se-á de decrementar esse valor e armazenar os parâmetros da invocação, para processamento posterior. No caso de não ser necessário recorrer à alternativa local (servidor contactável), se existir uma entrada na cache para o produto em causa, também se efectuará o devido acerto, mediante a invocação automática do método `decrementa_stock_local`.

As invocações pendentes do método `saida-stock`, armazenadas durante um período de desconexão pelo mecanismo descrito, serão processadas no momento da reconexão pela operação `efectua_saidas_stock`.

Objecto clientes

No objecto `clientes`, o método `contribuinte` foi anotado de forma equivalente à descrita para o método `preco` do objecto `produtos`. No entanto, existe uma diferença subtil que respeita à devolução de resultados previamente guardados. De facto, dado que o número de contribuinte do cliente constitui informação estática, são sempre utilizados os valores existentes na cache, independentemente de o servidor se encontrar contactável ou não.

Para a operação `plafond`, optou-se por um esquema de armazenamento de resultados de invocações algo diferente; para cada cliente são guardados os valores correspondentes às cinco últimas consultas efectuadas com sucesso e, no caso de ser impossível o contacto com o servidor, é devolvida a média dos limites de crédito armazenados. Deste modo, existirá uma espécie de historial para o cliente, optando-se pela utilização do valor médio do seu limite de crédito no passado recente.

No que respeita ao método `gasta-plafond`, especificou-se um comportamento semelhante ao do método `saida-stock` do objecto `clientes`, embora com duas ligeiras diferenças:

- no teste do valor local do limite de crédito, para decidir se a operação pode ser desencadeada ou não, deverá ser usado o valor médio, em conformidade com o esquema de devolução de resultados mencionado anteriormente;
- na actualização do valor do limite de crédito local, dever-se-á ter em consideração

que para cada cliente poderão existir até cinco valores armazenados, devendo-se portanto decrementar o último valor obtido.

A operação `efectua_gastos_plafond` destina-se a processar, no momento da reconexão, todas as invocações pendentes.

Por outro lado, foi especificado que deverá ser despoletada de forma automática a operação `traz_ultimos_plafonds`, sempre que se avizinha um período de desconexão. Esta operação tem por objectivo obter informação mais actualizada relativamente ao limite de crédito dos clientes, por forma a garantir decisões mais correctas nos processamentos efectuados em modo desconectado.

Por fim, para o método `repoe_plafond`, é indicado que o contacto com o servidor só deve ser tentado de hora em hora, ou seja, as invocações ocorridas durante este período de tempo ficam pendentes, sendo processadas no final. No caso de o servidor se encontrar incontactável no momento de processar as invocações, será tentado o contacto sempre que for efectuada um nova invocação ou no final de um outro período de uma hora. Note-se que este método, para além de não constituir uma operação crítica do ponto de vista da empresa[9], não devolve nenhum resultado, pelo que as suas invocações podem ser mantidas pendentes mais facilmente.

Após a reposição do limite de crédito no servidor, isto é, sempre que uma invocação ao método `repoe_plafond` é executada com sucesso, procede-se à actualização conveniente do valor do limite de crédito mantido localmente.

Objecto facturas

Os métodos do objecto `facturas`, à imagem do método `repoe_plafond` do objecto `clientes`, foram anotados com a primitiva `WAIT`. Assim, o lançamento das facturas propriamente dito só será efectuado de 15 em 15 minutos, caso o servidor se encontre disponível.

6.3.2 Construção do Agente

A funcionalidade do cliente em períodos de desconexão é especificada na interface anotada e é obtida através de um agente. Este é gerado pelo tradutor `c++builder`, com base nas primitivas de desconexão.

Os ficheiros produzidos pelo tradutor de anotações, para o caso em estudo, serão:

- `facturacao_agent.cc`;

[9] No pior dos casos, o cliente receberá a informação de que não possui crédito suficiente para a compra que deseja efectuar, apesar de já ter procedido ao pagamento das facturas em dívida.

- `facturacao_produtos_agent_impl.cc`;
- `facturacao_clientes_agent_impl.cc`;
- `facturacao_facturas_agent_impl.cc`.

Nos módulos respeitantes à implementação dos objectos `produtos` e `clientes` haverá necessidade de codificar alguns métodos, nomeadamente os métodos referenciados em algumas anotações, para os quais o tradutor não pode gerar código automaticamente.

Dado que os dois objectos terão implementações similares para os métodos referidos, será exemplificada apenas a codificação dos métodos do objecto `produtos`. Assim, os métodos a codificar são: `efectua_saidas_stock`, `decrementa_stock_local` e `armazena_saida_stock`. Começando pelo último, será necessário declarar uma fila de espera na classe gerada para o objecto em causa, com a finalidade de armazenar as várias saídas de produtos, cabendo a este método o papel de introduzir elementos nessa fila. No que respeita ao teste e eventual actualização do valor da existência do produto mantido localmente, será necessário conhecer a estrutura de dados usada pelo agente para implementar a cache de resultados. O código gerado pelo tradutor, respeitante à primitiva `STORE`, utiliza um objecto da classe `storage`, a qual disponibiliza métodos para inserir, remover, consultar e alterar pares do tipo `<argumentos, resultados>`. Uma implementação possível para este método será[10]:

```
class facturacao_T_produtos_AgentImpl :  public facturacao_T_produtos
{
...
private:
  queue saidas_stock_pendentes;
...
}

void
facturacao_T_produtos_AgentImpl::
  armazena_saida_stock (facturacaoStatus *status,
                        facturacao_T_codigo_produto codigo,
                        facturacao_T_valor_quantidade quantidade)
{
  struct
    {
```

[10] As linhas sublinhadas correspondem ao código gerado pelo `c++builder` – definição da classe referente ao objecto `produtos` e esqueleto do método alternativo à invocação remota. As reticências sublinhadas são usadas para omitir código não relevante para esta exposição, e que também é gerado pelo tradutor.

```
    facturacao_T_codigo_produto codigo;
    facturacao_T_valor_quantidade quantidade;
  }
 saida_stock;
 facturacao_T_valor_quantidade *valor_stock;

 if (this->stock_storage.get (&codigo, valor_stock))
  if (*valor_stock >= quantidade)
   {
    *valor_stock -= quantidade;
    this->stock_storage.set (&codigo, valor_stock);
    saida_stock.codigo = codigo;
    saida_stock.quantidade = quantidade;
    this->saidas_stock_pendentes.enqueue (&saida_stock,
                                          sizeof (saida_stock));
   }
  else
   status->returnCode = facturacao_E_stock_insuficiente;
 else
  status->returnCode = facturacao_E_produto_inexistente;
}
```

As operações de saída de produtos armazenadas na fila de espera são processadas posteriormente pelo método `efectua_saidas_stock`, isto é, logo que seja possível contactar o servidor serão efectuadas as invocações remotas previamente suspensas. Completando o esqueleto gerado pelo tradutor ter-se-á, por exemplo, a seguinte implementação para esse método:

```
void
facturacao_T_produtos_AgentImpl::efectua_saidas_stock ()
{
 struct
  {
   facturacao_T_codigo_produto codigo;
   facturacao_T_valor_quantidade quantidade;
  }
 saida_stock;
 facturacaoStatus status;
```

```
  bool servidor_ok=TRUE;

  while (servidor_ok &&
        this->saidas_stock_pendentes.dequeue (&saida_stock,
                                              sizeof (saida_stock)))
    {
      facturacao_produtos_remote_inst->saida_stock (&status,
                                                    saida_stock.codigo,
                                                    saida_stock.quantidade);
      if (status.returnCode != NULL)
        {
          servidor_OK = FALSE;
          this->saidas_stock_pendentes.enqueue (&saida_stock,
                                                sizeof (saida_stock));
        }
    }
}
```

Quanto ao método `decrementa_stock_local`, e depois de referido o modo de interacção com a cache utilizada para armazenar os resultados das invocações do método `stock`, poder-se-á conceber a seguinte implementação:

```
void
facturacao_T_produtos_AgentImpl::decrementa_stock_local (
                          facturacao_T_codigo_produto codigo,
                          facturacao_T_valor_quantidade quantidade)
{
  facturacao_T_valor_quantidade *valor_stock;

  if (this->stock_storage.get (&codigo, valor_stock))
    {
      *valor_stock -= quantidade;
      this->stock_storage.set (&codigo, valor_stock);
    }
}
```

Por último, resta referir que a compilação do agente, e subsequente obtenção do programa

executável, implica a utilização de todos os módulos produzidos pelo *stubber* do ILU[11], ou seja, quer os módulos destinados ao servidor quer aqueles destinados ao cliente terão que ser incluídos no processo de compilação e ligação do agente. Isto deve-se ao facto de o agente ser simultaneamente cliente e servidor da mesma interface. Na verdade, a função do agente é servir de intermediário entre o cliente e o servidor.

6.3.3 Instalação de Agentes

Por cada cliente, ou conjunto de clientes, que necessite de garantias de operação isolada, procede-se à instalação de um agente. Na generalidade dos casos, este será instalado na máquina onde se o encontra o cliente. No entanto, se existirem clientes dispersos por várias máquinas entre as quais há garantias de comunicação permanente, poder-se-á instalar um único agente para todos esses clientes. Esta poderá ser a situação de uma das lojas do caso em estudo, onde poderá existir uma rede local interligando várias máquinas com software de facturação (clientes). Um aspecto importante na instalação de agentes é que não deve ser criado um novo ponto de falha entre estes e os clientes a que se destinam.

Aquando da instalação do agente, os vários objectos por ele disponibilizados (os mesmos que o servidor exporta) serão registados pelo ILU[12]. Dado que estes objectos apenas interessam a um determinado cliente (ou num caso mais alargado a um subconjunto de todos os clientes possíveis de um servidor), não haverá necessidade de os publicar fora do contexto do agente a que dizem respeito. Assim, poder-se-á utilizar um directório local, em vez do directório partilhado por NFS.

Por forma a tornar mais flexível a utilização de agentes, a sua concretização permite que no momento da instalação sejam indicados como parâmetros os identificadores dos vários objectos a registar, mais concretamente, o SID do agente e o IR de cada objecto. Analogamente, terão que ser indicados os identificadores dos objectos com os quais o agente interactua, ou seja, dos objectos disponibilizados pelo servidor.

Depois de instalado o agente será necessário reiniciar o cliente (ou clientes), especificando os novos objectos a utilizar; o cliente deixará de interactuar com os objectos do servidor, passando a invocar métodos dos objectos disponibilizados pelo agente.

[11] Recorde-se que, para além do processamento da interface anotada através do `c++builder`, também será processada a interface original (sem anotações) pelo *stubber* do ILU.

[12] Para todos os efeitos, o agente é um servidor ILU que regista objectos no serviço de directório, a fim de serem utilizados por clientes.

6.4 Análise de Desempenho

O objectivo primordial deste trabalho é a criação de mecanismos que aumentem a disponibilidade e a fiabilidade das aplicações distribuídas. Estes mecanismos têm obviamente um preço ao nível computacional, resultando numa penalização do desempenho das aplicações. Com o intuito de avaliar o impacto do modelo CASs (sistema de replicação de objectos em conjunção com o sistema de suporte à desconexão) na operação das aplicações, foram efectuados alguns testes.

Em primeiro lugar, procedeu-se à avaliação do desempenho do sistema GTS. Para o efeito foram usadas duas máquinas de uma rede local Ethernet de 10Mbits: um Dual Pentium Pro com Red Hat Linux 4.1 (kernel 2.0.27) e um Sparc Server 10 com SunOS 2.5.1. Na primeira foram instalados o sequenciador GTS e um cliente destinado ao envio de mensagens e eventual recepção de respostas. Na segunda foram instalados vários servidores para recepção de mensagens e eventual envio de respostas. A carga média destas máquinas (obtida com o comando `w`) durante o período de realização dos testes era de 3 e 0.2, respectivamente.

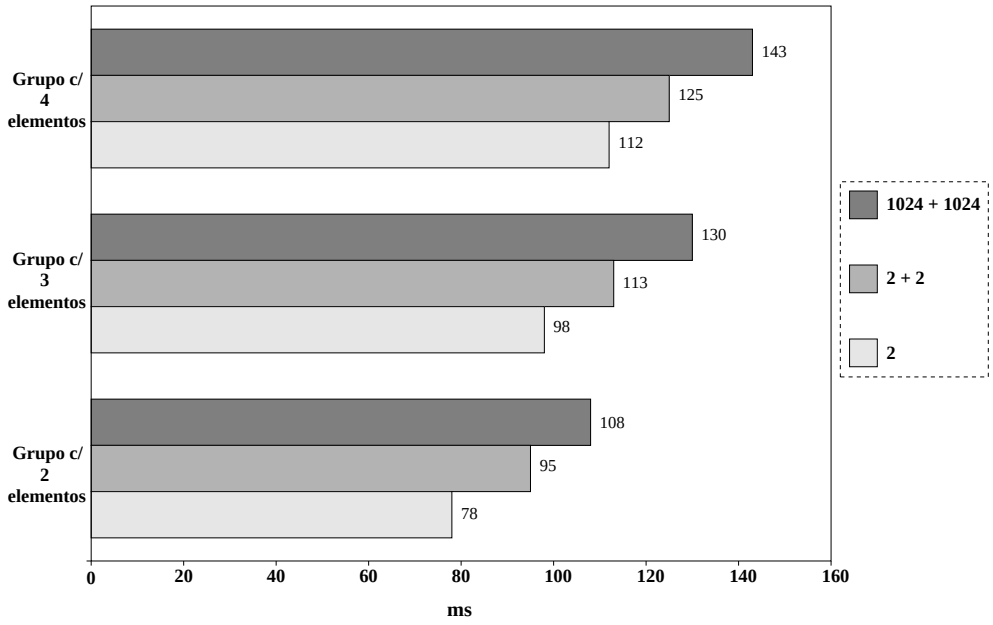

Figura 6.2: Troca de mensagens via GTS.

A figura 6.2 mostra os resultados obtidos com grupos – conjuntos de servidores – de dois, três e quatro elementos e mensagens de 2 e 1024 bytes. Para cada uma das mensagens, os servidores respondiam com uma mensagem equivalente. Os tempos (contabilizados no cli-

ente) dizem respeito às operações de preparação da mensagem[13] e sequente envio, seguidas da recepção da primeira resposta (devolvida pelo servidor mias rápido) e sua interpretação (desempacotamento). A operação de envio compreende a entrega da mensagem ao sequenciador, que por sua vez a faz chegar ao seu destino com garantia de ordenação. Note-se que a recepção de uma resposta num cliente implica, por parte do servidor, a interpretação da mensagem, a preparação da resposta correspondente e o seu sequente envio. No caso das mensagens com 2 bytes, foi ainda medido o tempo necessário para apenas submeter a mensagem ao sequenciador (preparação seguida de envio). Os resultados apresentados correspondem às médias dos valores obtidos em 100 iterações consecutivas e espaçadas de 1 segundo[14], onde cada iteração compreende a troca de 6 mensagens[15], também espaçadas de 1 segundo.

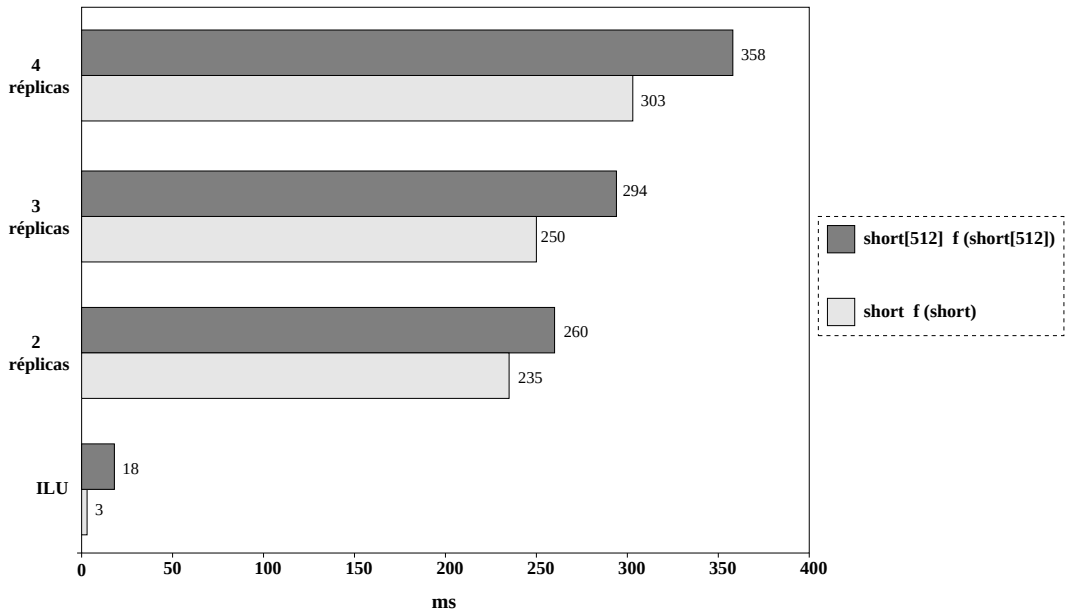

Figura 6.3: Invocações ILU com e sem replicação.

Em segundo lugar, avaliou-se o impacto da introdução de um interceptor e vários representantes entre um cliente ILU e várias réplicas de um servidor. Nos testes efectuados utilizaram-se invocações (por parte de um cliente) de dois métodos remotos: um que recebe como argumento um número inteiro e devolve como resultado esse mesmo número e outro

[13] O GTS oferece primitivas para troca de sequências de caracteres (`char *` em C++), obrigando o programador a empacotar e desempacotar todos os dados antes do envio e depois da recepção, respectivamente.

[14] Os valores correspondentes à primeira iteração foram excluídos da amostra por possuírem uma ordem de grandeza superior à dos demais.

[15] O cliente envia 2 mensagens distintas (2 e 1024 bytes) para três grupos distintos (dois, três e quatro elementos).

que recebe um *array* de 512 inteiros, procedendo também à sua devolução. Na figura 6.3 podem ser observados os resultados para o caso em que é usado um único servidor (modelo Cliente-Servidor oferecido pelo ILU) e para os casos em que se usam dois, três ou quatro servidores replicados. Na presença de replicação intervêm um cliente, um interceptor e um sequenciador GTS instalados na máquina Linux (a mesma dos testes efectuados para o GTS), e intervêm ainda os vários servidores, com os respectivos representantes, todos eles instalados na máquina SunOS (máquina esta também usada nos testes do GTS). Tal como no caso anterior, os resultados apresentados correspondem às médias dos valores obtidos em 100 iterações. Refira-se ainda que os tempos em questão não incluem processamentos adicionais respeitantes à consulta de serviços de directório. Tais processamentos apenas interferem nos resultados obtidos na primeira iteração, os quais foram desprezados.

Dado que o tamanho de um número inteiro (`SHORT INTEGER` em ILU) é de 2 bytes, a invocação destes dois métodos corresponderá aproximadamente, em termos de volume de informação trocada, ao envio e recepção das mensagens consideradas nos testes respeitantes ao GTS. Na verdade, o ILU acrescenta alguma informação de controlo aos argumentos e resultados de uma invocação. No entanto, dado que essa informação é relativamente compacta (96 bytes no pedido e 28 bytes na resposta, para os métodos em causa), poder-se-ão adoptar os tempos obtidos nos testes do GTS como valores aproximados para a duração da troca de informação entre o interceptor e os representantes, relativamente a uma invocação ILU. Assim, pode-se concluir que a invocação de métodos de objectos ILU replicados é bastante mais lenta (aproximadamente o dobro) que a correspondente utilização do GTS.

A análise detalhada da operação do sistema de replicação de objectos permite determinar as várias tarefas desencadeadas e os tempos associados a cada uma delas, desde o momento em que um cliente desencadeia a invocação até à chegada do resultado. Na verdade, poder-se-á até resumir todo o funcionamento à soma de tempos

$$t_{C-I} + t_I + t_{GTS} + t_R + t_{R-S}$$

com

- t_{C-I} – interacção entre o cliente e o interceptor,
- t_I – processamento do interceptor,
- t_{GTS} – troca de pedidos e respostas ILU via GTS,
- t_R – processamento do representante,
- t_{R-S} – interacção entre o representante e o servidor.

A interacção entre o cliente e o interceptor envolve duas fases: em primeiro lugar, haverá a construção do pedido ILU pelo cliente e a sequente recepção, por parte do interceptor, da sequência de bytes que compõem esse pedido, enquanto que na segunda fase passar-se-á ao envio (por parte do interceptor) da sequência de bytes que compõem a resposta ao pedido ILU, seguido da sua recepção e interpretação pelo cliente. Esta interacção

terá associado um tempo nunca superior a uma invocação ILU normal, dado que uma das partes – o interceptor – desempenha menos tarefas que um servidor ILU normal, pois trata os pedidos ILU como sequências de bytes, sem efectuar qualquer interpretação destes. De igual modo, a interacção entre o representante e o servidor terá um tempo também inferior a uma invocação ILU.

Os tempos de processamento do interceptor e do representante, juntamente com o tempo correspondente à troca de pedidos e respostas ILU via GTS, representarão a grande fatia do tempo gasto em invocações ILU com replicação. De facto, das tarefas mencionadas no parágrafo anterior resultaria um tempo nunca superior a duas invocações ILU, que seria muito inferior aos tempos obtidos para o sistema de replicação.

A troca de pedidos e respostas ILU via GTS, conforme já referido, equivale ao envio e recepção de uma mensagem de tamanho aproximadamente igual através do sistema GTS. Assim, se aos tempos obtidos para a troca de mensagens via GTS se adicionarem os tempos respeitantes às interacções do cliente com o interceptor e do representante com o servidor, para a utilização de duas réplicas, por exemplo, ter-se-á:

$$t_{C-I} + t_{GTS} + t_{R-S} \approx 18 + 108 + 18.$$

Este tempo está ainda bastante longe dos 260ms obtidos com o sistema de replicação. A diferença existente corresponde aos tempos de processamento do interceptor e do representante. Estes processamentos envolvem a numeração dos vários pedidos ILU recebidos e a filtragem das respostas devolvidas pelas réplicas para um dado pedido. Mesmo esta última tarefa não será muito relevante em termos de tempo gasto, visto que o interceptor devolve ao cliente a primeira resposta recebida. A maior parte do tempo é gasta numa outra tarefa desencadeada quer pelo interceptor quer pelo representante, que é a sincronização entre as operações de envio e recepção de mensagens GTS, devido ao facto de o GTS não suportar a utilização de fios de execução.

A utilização de agentes (modelo CASs) introduz duas possibilidades de operação:

- o agente resolve a invocação localmente – neste caso ter-se-á o tempo de uma invocação ILU normal;
- o agente invoca o método do servidor – haverá que adicionar o tempo correspondente a uma invocação ILU aos tempos apresentados anteriormente para a invocação de objectos replicados.

Se for usado o mecanismo de difusão de pedidos ILU, então deixará de fazer sentido efectuar uma avaliação do desempenho. De facto, o armazenamento temporário de pedidos ILU em ficheiros, com a posterior transferência (à partida em altura indeterminada) para o sistema de ficheiros de uma segunda máquina, e assim sucessivamente, não se adequa à

tarefa de avaliação de tempos de execução.

Por fim, resta concluir com alguns apontamentos relativamente aos resultados obtidos:

- a relativa ineficiência do sistema de replicação deve-se à utilização do GTS;
- a opção por sistemas de comunicação em grupo que restringissem o âmbito de aplicação a redes locais, como por exemplo o xAMp, levaria a resultados bastante melhores;
- quando comparado com outros sistemas, por exemplo RPCs sobre Isis [46], cujos tempos de invocação podem ascender aos 900ms, o sistema de replicação desenvolvido revela-se bastante bom;
- a degradação do desempenho associada à solução apresentada nesta dissertação é um pequeno preço a pagar pela possibilidade de operação isolada e pela replicação de servidores, principalmente porque se trata de uma solução genérica.

Capítulo 7

Conclusões

O objectivo desta dissertação foi a criação de ferramentas/metodologias para a simplicação do processo de desenvolvimento de aplicações distribuídas tolerantes a faltas. Dada a impossibilidade, no âmbito de um trabalho deste género, de construir totalmente de raiz uma plataforma para o desenvolvimento de aplicações, optou-se por acrescentar funcionalidade a uma plataforma já existente. Foi então escolhido o sistema ILU como tecnologia de base, e criaram-se mecanismos para introduzir suporte à desconexão e replicação em aplicações desenvolvidas com base nessa tecnologia – modelo Cliente-Servidor, oferecido pelo ILU.

Apesar da sua antiguidade, o modelo Cliente-Servidor revela-se ainda uma boa opção para a concepção de aplicações distribuídas, principalmente quando integrado com a programação orientada ao objecto. Prova disso é a escolha efectuada pelo OMG na especificação CORBA. Deste modo, será de esperar que o sistema desenvolvido tenha uma boa aplicabilidade no desenvolvimento de aplicações distribuídas e até na conversão de aplicações já existentes, acrescentando-lhes suporte à desconexão e replicação.

A forma como é oferecido o suporte à desconexão – modelo CAS – permite obter soluções modulares, sem a necessidade de modificar clientes ou servidores. No entanto, a possibilidade de retirar de funcionamento o agente em qualquer instante, passando ao tradicional modelo Cliente-Servidor, permite que as aplicações dispensem o suporte à desconexão sempre que desejado.

O geração do código do agente com base nas anotações de desconexão é uma ideia completamente nova que permite automatizar significativamente a codificação de algoritmos relativamente complexos. As anotações propostas, apesar de em certos casos exigirem a escrita de algum código (geração apenas parcial do código do agente), possibilitam a obtenção de soluções para operação desconectada de forma bem estruturada e segundo uma metodologia clara e bem documentada. Se se compararem os agentes obtidos segundo esta abordagem, juntamente com a interface anotada, com as soluções específicas codificadas

no passado, poder-se-á verificar a modularidade (separação de tarefas) e clareza do código relativo ao suporte à desconexão adjacentes à nova metodologia.

O conjunto de primitivas que constitui a extensão à linguagem de especificação de interfaces do ILU foi escolhido por forma a oferecer um leque mínimo e genérico de funcionalidades a implementar num agente. No entanto, com a inclusão de primitivas que traduzissem o comportamento dos objectos de um servidor e as estruturas de dados por eles manipuladas, conseguir-se-ia gerar automaticamente agentes mais completos e mais funcionais. Este poderá ser um ponto de evolução futura do trabalho actual.

Convém também referir que o modelo CAS e a metodologia de anotação de interfaces – as anotações de desconexão – poderiam ser aplicadas/generalizadas a outras tecnologias de desenvolvimento de aplicações Cliente-Servidor, como por exemplo o RPC da Sun. O tradutor de anotações poderia até ser alterado, ou poderiam ser criadas diversas variantes, por forma a reconhecer várias linguagens de especificação de interfaces. As operações desencadeadas pelo agente, na implementação actual, são praticamente independentes da tecnologia do ILU.

O mecanismo de replicação de objectos ILU permite a instalação pacífica de múltiplas instâncias de um servidor, sem a necessidade de proceder a quaisquer alterações ao nível do código dos clientes ou do servidor. Obviamente, este sistema não é uma solução óptima para a replicação, pois não são contemplados problemas específicos da replicação que implicam a interação de réplicas. No entanto, para um grande número de aplicações, o mecanismo apresentado é suficiente, possuindo ainda a vantagem de ser bastante fácil de utilizar.

A difusão selectiva de pedidos ILU através do GTS (base do mecanismo de replicação) revelou-se algo ineficiente, em face do desempenho conseguido nas invocações tradicionais do sistema ILU. Esta degradação de desempenho deve-se ao processo de sequenciação de mensagens utilizado no GTS, que é bastante ineficiente. No entanto, a escolha deste sistema de comunicação em grupo possibilitou obter um mecanismo de replicação de objectos bastante flexível, devido à grande portabilidade e configurabilidade do GTS, e de concretização relativamente simples, devido ao armazenamento persistente de mensagens para entrega posterior a processos momentaneamente icontactáveis.

De qualquer modo, as arquitecturas do interceptor e do representante permitem migrar para um outro sistema de comunicação em grupo, dado que não utilizam particularidades do GTS. Além disso, a troca de informação entre clientes e servidores é efectuada sem nenhum mecanismo de segurança, pelo que poderia haver algum interesse em substituir o GTS por outro sistema de comunicação. Este poderá ser outro ponto de evolução deste trabalho, principalmente no contexto de duas outras teses em fase de conclusão no seio do Grupo de Sistemas Distribuídos, no âmbito da segurança e da comunicação multi-ponto.

Refira-se ainda que o mecanismo de replicação desenvolvido, apesar de bastante genérico e de aplicação extremamente simples a qualquer solução baseada no modelo Cliente-Servidor do ILU, apresenta um desempenho razoável quando comparado com outros mecanismos do mesmo género. Por outro lado, o modelo de interceptores subjacente ao mecanismo de replicação apresentado pode também ser aplicado a outras tecnologias que não o ILU, desde que se tratem de tecnologias baseadas no modelo Cliente-Servidor. Na verdade, em relação à concretização levada a cabo neste trabalho, apenas será necessário proceder a alguns ajustes no método de controlo do registo de servidores, por forma a adaptar todo o sistema a uma nova tecnologia de construção de aplicações distribuídas.

Os mecanismos oferecidos neste trabalho obviamente não solucionam na sua totalidade os problemas resultantes da necessidade de melhorar a disponibilidade e a fiabilidade das aplicações. No entanto, o conjunto de facilidades disponibilizado permite a migração imediata de aplicações desenvolvidas segundo o bastante popular modelo Cliente-Servidor, abrindo boas prespectivas para aquelas organizações que possuem software inadequado às necessidades actuais. Conforme demonstrado no capítulo 6, empresas que disponham de programas cliente ligados a um servidor poderão facilmente migrar para o modelo Cliente--Agente-Servidores, tirando partido de todas as vantagens a ele adjacentes. Um exemplo típico de aplicação é o caso da gestão de *stocks* e facturação de uma empresa, com diversos locais de movimentação (locais onde são efectuadas vendas, compras ou outras alterações relativamente à informação associada a produtos, clientes ou fornecedores). A solução aqui apresentada permitirá obter resultados similares ao de soluções comerciais e com um custo praticamente nulo.

As ferramentas desenvolvidas para a anotação de interfaces e para a gestão de réplicas de objectos permitem replicar um servidor já existente e oferecer suporte mínimo à operação desconectada de clientes sem escrever uma única linha de código, o que prova a simplicidade e conveniência da utilização do sistema desenvolvido.

Bibliografia

[1] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall International, 1992.

[2] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall International, 1982.

[3] Michael D. Schroeder. *Distributed Systems*, chapter A State-of-the-Art Distributed System: Computing with BOB. Addison-Wesley Publishing, 2nd edition, 1993.

[4] S. J. Lemer, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[5] Sun Microsystems. RPC: Remote Procedure Call, Protocol Specification. Request for Comments 1057, 1988.

[6] Graham D. Parrington. A Stub Generation System For C++. Technical Report NE1 7RU UK, Department of Computing Science, The University of Newcastle upon Tyne, 1993.

[7] Sun Microsystems. XDR: External Data Representation Standard. Request for Comments 1014, 1987.

[8] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. Comp. Syst.*, 6(1):109–133, 1988.

[9] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. ORCA: A Language for Parallel Programming of Distributed Systems. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit, 1991.

[10] Henri E. Bal. A Comparative Study of Five Parallel Programming Languages. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit, 1991.

[11] Dimitri Konstantas. Object Oriented Interoperability. Technical report, Centre Universitaire d'lnformatique, University of Geneva, 1993.

[12] Douglas C. Schmidt and Steve Vinoski. Object Interconnections to Distributed Object Computing. *C++ Report*, 1995.

[13] J. Dilley. OODCE: A C++ Framework for the OSF Distributed Computing Environment. In *Winter Usenix Conference*. USENIX Association, 1995.

[14] IBM. *SOMobjects: Managing DSOM*, chapter SOM and Distributed SOM. IBM Corporation, first edition, 1995.

[15] Nuno Furtado da Silva. Memória Distribuída e Partilhada Flexível. Master's thesis, Universidade do Minho, 1996.

[16] Microsoft. *OLE 2. O : Programmer's Reference Manual*, 1994.

[17] IBM. The System Object Model (SOM) and the Component Object Model (COM): A comparison of technologies from a developer's perspective. Technical report, IBM Corporation, 1994.

[18] IBM. OpenDoc vs. OLE 2.0: Superior by Design – A Developer's View. Technical report, IBM Corporation, 1994.

[19] Kurt Piersol. A Close-Up of OpenDoc. *AIXpert*, 1994.

[20] Brian Marsh, Fred Douglis, and Ramón Cáceres. Systems Issues in Mobile Computing. Technical report, Matsushita Information Technology Laboratory, 1993.

[21] Don J. Belisle. OMG Standards for Object-Oriented Programming. *AIXpert*, 1993.

[22] B. R. Badrinath, Arup Acharya, and Tomasz Imielinski. Impact of Mobility on Distributed Computations. *Operating Systems Review*, 1993.

[23] M. Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu. Experience with Disconnected Operation in a Mobile Computing Environment. Technical Report CMU-CS-93-168, School of Computer Science, Carnegie Mellon University, 1993. Appeared in Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing.

[24] Object Management Group. The Common Object Request Broker: Architecture and Specification. 2.0 (draft), 1995.

[25] Steve Vinoski. Distributed Object Computing With CORBA. *C++ Report*, 1993.

[26] IONA. The Orbix Architecture. Technical report, IONA Technologies Ltd., 1993.

[27] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[28] Katy Ring. Is the OMG failing the market with CORBA? Power PC News.

[29] Joe Salemi. Guide to Client/Server Databases. *PC Magazine*, 1993.

[30] Ken Henderson. Client/Server Developer's Guide with Delphi. *HW Sams*, 1997.

[31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Summer Usenix Conference*, pages 119–130, 1985.

[32] Ajay Bakre and B. R. Badrinath. M-RPC: A Remote Procedure Call Service for Mobile Clients. Technical report, Department of Computer Science, The State University of New Jersey, 1995.

[33] Dietmar Kottmann and Christian Sommer. Stublets: A Notion for Mobility Aware Application Adaption. Technical report, University of Karlsruhe, Institute of Telematics, 1996.

[34] Anthony D. Joseph, Alan F. de Lespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A Toolkit for Mobile Information Access. In *SIGOPS' 95*, pages 156–171, 1995.

[35] Vitor Guedes and Francisco Moura. Replica Control in MIo-NFS. In *ECOOP' 95 Workshop on Mobility and Replication*, 1995.

[36] Raquel Menezes, Carlos Baquero, and Francisco Moura. A portable lightweight approach to NFS replication. In *Rose' 94 Conference*, 1994.

[37] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 1990.

[38] Marc E. Fiuczynski and David Grove. A Programming Methodology for Disconnected Operation. Technical report, Department of Computer Science and Engineering, University of Washington, 1994.

[39] Thomas Kirste. Aspects of an object model for Mobile Information Systems. Technical report, Computer Graphics Center, Wilhelminenstr, 1995.

[40] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in Partitioned Networks. *AGM*, 17(3), 1985.

[41] Mark Hettler. Lotus Notes vs. Microsoft Exchange. *BYTE*, 21(12):112–116, 1996.

[42] Silvano Maffeis, Walter Bischofberger, and Kai-Uwe Matzel. A Generic Multicast Transport Service to Support Disconnected Operation. Technical report, Department of Computer Science, Cornell University, and UBILAB, Union Bank of Switzerland, 1996.

[43] Luís Rodrigues and Paulo Veríssimo. xAMp: A Protocol Suite for Group Communication. In *11th Symposium On Reliable Distributed Systems*, 1992.

[44] Bill Janssen and Mike Spreitzer. *ILU 2.0alpha, Reference Manual*. Xerox Corporation, 1996.

[45] Xerox Corporation. Courier: the Remote Procedure Call protocol. Technical report, Systems Integration Standards, Stanford, 1981.

[46] Artur Manuel Pereira Romão. Integração de comunicação multi-ponto em sistemas de invocação de procedimentos remotos. Master's thesis, Universidade Nova de Lisboa, 1996.

[47] Sarr Blumson, Mark Carter, and Daniel Hyde. Automatic Insertion of Performance Instrumentation for Distributed Applications. Technical Report 95-4, Center for Information Technology Integration, University of Michigan, 1995.

[48] Robert A. Whiteside and Emest J. Friedman-Hill. idldoc: the IDL Documentation Generator. Sandia National Laboratories, Livermore, 1997.

[49] Luís Soares Barbosa and José João Almeida. CAMILA: A Reference Manual. Technical Report DI-CAM-95:11:2, Departamento de Informática, Escola de Engenharia, Universidade do Minho, 1995.

[50] Luís Soares Barbosa and José João Almeida. From Prototypes to Implementations and Back. Technical Report DI/INESC-94-10-2, Instituto de Engenharia de Sistemas e Computadores, Universidade do Minho, 1994.

[51] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, second edition edition, 1992.

[52] Terence John Parr. *Language Translation Using PCCTS and C++ (A Reference Guide)*, 1995.

[53] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1993. draft version.

[54] Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *USENIX Conference on Object-Oriented Technologies*, 1995.

[55] R. van Renesse and K. P. Birman. Fault-Tolerant Programming using Process Groups. IEEE Computer Society Press, 1994.

[56] K. P. Birman and R. van Renesse. Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, 1994.

[57] Luís Rodrigues, Ellen Siegel, and Paulo Veríssimo. A Replication-Transparent Remote Invocation Protocol. Technical report, INESC, 1994.

[58] Karim R. Mazouni, Benoit Garbinato, and Rachid Guerraoui. Building Reliable Client-Server Software Using Actively Replicated Objects. Technical report, Laboratoire de Systemes d'Exploitation, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1995.

[59] Karim R. Mazouni, Benoit Garbinato, and Rachid Guerraoui. Filtering Duplicated Invocations Using Symmetric Proxies. Technical report, Laboratoire de Systemes d'Exploitation, Département d'Informatique, École Polytechnique Fédérale de Lausanne, 1996.

[60] Sun MicroSystems. *SunOS 5.2, Guide to Multi-Thread Programming*, 1993.

[61] SunSoft. pthreads and Solaris threads: A comparison of two user level threads APIs. Technical report, Sun Microsystems, 1994.

[62] Brian Kantor and Phil Lapsley. Network news transfer protocol: A proposed standard for the stream-based transmission of news. Request for Comments 977, 1986.

[63] Carlos Baquero. Indirect Calls: Remote Invocations on loosely coupled Systems. Technical report, Universidade do Minho, Grupo de Sistemas Distribuídos, Departamento de Informática, 1996.