Proceedings of the 25th IASTED International Multi-Conference
Parallel and Distributed Computing and Networks
February 13–15, 2007, Innsbruck, Austria
ISBN Hardcopy: 978-0-88986-637-9 / CD: 978-0-88986-639-3

FULL-SPEED SCALABILITY OF THE PDOMUS PLATFORM FOR DHTS

José Rufino[1]* António Pina[2] Albano Alves[1] José Exposto[1] Rui Lopes[1]

{rufino,albano,exp,rlopes}@ipb.pt, [1]Polytechnic Institute of Bragança, 5300-854 Bragança, Portugal
pina@di.uminho.pt, [2]University of Minho, 4710-057 Braga, Portugal

ABSTRACT
Domus is an architecture for Distributed Hash Tables (DHTs) tailored to a shared-all cluster environment. Domus DHTs build on a (dynamic) set of cluster nodes; each node may perform routing and/or storage tasks, for one or more DHTs, as a function of the node base (static) resources and of its (dynamic) state. Domus DHTs also benefit from a rich set of user-level attributes and operations. pDomus is a prototype of Domus that creates an environment where to evaluate the architecture concepts and features. In this paper, we present a set of experiments conduced to obtain figures of merit on the scalability of a specific DHT operation, with several lookup methods and storage technologies. The evaluation also involves a comparison with a database and a P2P-oriented DHT platform. The results are promising, and a motivation for further work.

KEY WORDS
Cluster Computing, Distributed Hash Tables, Evaluation.

1 Introduction

Cluster Computing applications often deal with huge amounts of data. This may require mechanisms for distributed data storage and access, once conventional (centralized) data structures may not cope with the storage and performance requisites of such demanding applications.

Under such scenario, *dictionary*-like distributed data structures may be required. In essence, a *dictionary* is a repository of <*key*, *data*> records, with support for a basic set of *key*-based access operations like insertions, retrievals and removals (more complex operations may also be allowed, including support for iterations, locking, etc.).

Distributed Hash Tables (DHTs) have been effectively used to implement distributed dictionaries. Research in DHTs range from 1st generation models, adequate to small/medium-scale clusters [1, 2, 3], to more recent approaches [4], tailored to internet-wide peer-to-peer (P2P) scenarios. The later demand support for i) efficient routing, ii) wide-area intermittent network connections, iii) continuous arrival/departure of nodes, iv) security/anonymity, etc. These requisites may be relaxed in a cluster, a tightly integrated hardware/software/management environment, of a much lower scale, running on high-bandwidth local networks. As such, deploying a P2P-oriented DHT platform in a cluster environment may result inappropriate: when implementations target specific scenarios, they usually include functionalities that may be of little use (and even counterproductive) in others. On the other hand, some research contributions from P2P-oriented DHTs, like efficient distributed lookup, may be attractive to cluster-oriented DHTs, once clusters node counts keep increasing.

This paper presents an evaluation of pDomus, a prototype of the Domus architecture [5]. It supports the deployment, operation and management of multiple DHTs and its supporting services, in a shared-all cluster environment. Domus DHTs build on a (dynamic) set of cluster nodes; each node performs routing and/or storage tasks, for one or more DHTs, as a function of the node base (static) resources and of its (dynamic) state. Domus DHTs also benefit from a rich set of user-level attributes and operations.

The results presented in this paper focus on the scalability of several client/server deployments, measured by the throughput of a specific dictionary operation, using several storage technologies and lookup strategies. They also include a comparison with a database (MySQL) and a P2P-oriented platform for DHTs (Bamboo [6]). Overall, the results are promising, and a motivation for further work.

The remaining of the paper is organized as follows: section 2 revises basic concepts of the Domus architecture and relevant features of the pDomus prototype, section 3 establishes the evaluation framework, sections 4 and 5 present the evaluation results, and section 6 concludes.

2 Research Framework

The pDomus prototype builds on a set of Python modules; in addition, an external C-based module provides an efficient implementation of some routing and storage functionalities; these use Red-Black trees as the core data structure.

The Domus architecture [5] for DHTs derived from our models for the balanced distribution of the range of an hash function over a set of heterogeneous nodes [7]. Domus was designed to support multiple and heterogeneous DHTs, in a dynamic shared-all cluster environment. To improve the utilization of cluster resources, it enables the assignment of the *routing* and *storage* functions of a DHT to specific sets of clusters nodes, defined separately (those sets may eventually overlap or be strictly disjoint). The decoupling of the *routing* and *storage* functions al-

*Supported by grants PRODEP III/5.3/N/199.006/00, FCT SAPI-ENS/41739/CHS/2001.

lows the best nodes for a certain task to be (dynamically) chosen; for instance, *routing* or RAM-based *storage* are CPU+RAM+Network bound, whereas Disk-based *storage* is mostly Disk+Network bound; as such, those functions will be performed by the nodes that present the best combination of i) base resources and ii) spare utilization levels.

Domus services are structured into *balancement* (BS), *addressing* (AS) and *storage* (SS) subsystems. The AS/SS subsystems perform routing/storage functions (respectively), for subsets of buckets, of one or more DHTs. The AS subsystem keeps a routing table, per bucket, to perform distributed lookups along a routing graph that links all buckets of a DHT; it keeps also a storage reference, per bucket, to the service that stores the data records that map onto the bucket (thus decoupling routing and storage functions). The SS subsystem hosts local *repositories* of dictionaries. Repositories may build on different *storage platforms* and *storage media* combinations (e.g., BerkeleyDB [8] over RAM or Disk, etc.), selectable on a per DHT basis.

Dynamic load balancing involves all subsystems. With the help of specialized services [9], the BS subsystem monitors the utilization of node resources (CPU, RAM, Disk, Network). In turn, the AS and SS subsystems monitor a) the load induced by distributed lookups and access to data records, and b) the storage utilization levels. Reaching certain thresholds triggers a) the re-allocation of local buckets, of one or more DHTs, to other services, and/or b) the changing of the overall number of DHT buckets.

Tasks that require global coordination are managed by a well-known *supervisor* service and include: a) creation/destruction and shutdown/restart of a Domus deployment (set of services); b) addition/removal and shutdown/restart of specific services; c) creation/destruction, activation/deactivation[1] and (re)distribution of DHTs.

Domus DHTs have a rich set of user-level attributes, from several categories: i) hash functions, ii) distribution constraints, iii) storage technologies, iv) routing technologies, etc. In the context of this paper, the most relevant are those that define the storage and routing technologies used.

A *storage technology* refers to a combination of *storage media* (`_sm`) and *storage platform* (`_sp`) attributes. In pDomus, `_sm` may be set to `ram` or `disk`, depending on the data persistence requirements of client applications. In turn, `_sp` comprises several dictionary implementations, including those selected for the scalability evaluation of this paper: a) `python-dict` (Python `ram`-based built-in dictionaries); b) `domus-bsddb-btree` (C-crafted Red-Black trees of BerkeleyDB databases [8], the later operated with the `btree` access method). In another paper [10] we evaluate all storage technologies of pDomus, both in terms of i) performance and ii) storage utilization; in the same work we also develop metrics to assist the selection of the most appropriate technology for a certain application scenario.

Defining a *routing technology* involves the choice of a *routing graph* (`_rg`), a *routing algorithm* (`_ra`) and a *routing strategy* (`_rs`). In pDomus, `_rg` may be set to `chord` or `bruijn`, depending on the choice of *Chord* [11] or *de Bruijn* [12] graphs, respectively; for each `_rg`, there are several routing algorithms that may be used, for different balances of i) number of routing hops and ii) computational effort; a routing strategy `_rs` refers to a specific combination of several *routing methods* available – see section 4.1.2.

[1]The deactivation of a DHT brings it to an offline state and frees up cluster resources that may be reassigned to other DHTs still online.

3 Evaluation Framework

In this paper we focus on the study of the throughput scalability for insertions (*put* operations), more specifically 1st insertions (*put1* operations), using the storage technologies i) `python-dict` over `ram` and ii) `domus-bsddb-btree` over `disk`. The respective platforms were selected because, accordingly to a previous evaluation [10], a) they are the fastest platforms currently provided by pDomus, for each kind of storage media (`ram` and `disk`), that b) still offer the full set of basic dictionary operations. Moreover, for each one of these technologies, the average (unitary) time consumed by the other basic dictionary operations (reinsertion/*put2*, retrieval/*get* and removal/*del* is identical to the average insertion/*put1* time [10]; as such, the results of our scalability evaluation for the *put1* operation are extensible to all basic operations, whether isolated or intermixed.

During each test, all active clients placed *put1* requests on services as fast as possible, without any additional user-level processing between successive requests, *i.e.*, clients tried to saturate the services. These *full-speed* tests hardly match real application scenarios, where clients activity is multiplexed between many tasks, not necessarily related to DHT access. However, they provide upper bounds on the expected performance level of a Domus deployment, for a certain target cluster. Moreover, they are independent of the specificities of client applications. For instance, cluster based Web Crawling and Ray Tracing applications may exhibit very different access patterns to the same Domus deployment; thus, any performance figures collected for a scenario become meaningless to the other.

Several classes of clients–services deployments were tested. A clients–services deployment relates to a specific combination of a) number and placement of client applications, and b) number and placement of Domus services. We have considered the following classes of combinations:

1. *N_Xcli_M_1srv* – N client nodes with X client instances each, and M server nodes with one service each[2], requiring a total of $N+M$ different cluster nodes, for an overall of $x = N \times X$ client instances (threads) and $y = M \times 1 = M$ service instances;
2. *N_Xcli1srv* – N cluster nodes, each shared by X client and one service instances, for an overall number of $x = N \times X$ clients and $y = N \times 1 = N$ services.

[2]An architectural constraint in Domus is that one cluster node may have no more than one Domus service per local network interface.

These deployment classes represent two opposite situations: one where clients and services always run in disjoint nodes, and another in which they always co-exist; in between, there are many variations that naturally arise from specific application scenarios or cluster utilization constraints. Even so, the knowledge about the kind of performance achievable under opposite/extreme situations is surely useful when choosing hybrid scenarios: they may bend to one extreme or to the other, as found convenient.

For any deployment of the above classes, with x client and y service instances, we repeated the same procedure:

1. set up a Domus deployment based on y services and create a DHT with its routing and storage domains evenly spread across the y services (*i.e.*, each service was assigned a quota $1/y$ of the hash function range);
2. deploy x clients and assign to each one a specific set of 2×2^{20} random integers; these are used to valuate both the key and $data$ fields of <*key*, *data*> records during *put1* insertions; the set size was chosen to allow continuous insertions for 210 seconds, the lapse of any test; this lapse was defined indirectly by constraints of the cluster monitoring services (because of the sampling rate used, and because the metrics produced are based on exponential moving averages, it takes $\approx$ 210 seconds for any metric to reach its peak level, starting from the base value of an idle scenario).

For each test, the total number of insertions performed by each client was collected, and then used to calculate an *aggregated throughput*. The *aggregated throughput*, for a certain clients–services deployment, results from the sum of the specific throughputs achieved by all active clients of the deployment. This assumes all clients start the test at the same time (ensured by the test setup), although they may finish at different times, because of specific local load conditions at their host nodes. Additional collected information includes the utilization levels for certain resources (CPU, Network Bandwidth, etc.) of the cluster nodes where clients and services were hosted during the test. This information proved useful in order to understand the scalability behavior and limitations of the pDomus prototype.

All Domus services were configured to run single-threaded and the UDP protocol was used for the exchange of messages; pDomus also supports TCP-based communication and multi-threaded services; however, UDP-based communication and asynchronous/event-driven single-threaded servers provide the fastest setup, possibly because of the well-known weak performance of Python threads.

The physical test-bed was a ROCKS cluster, with 1 front-end node[3] and 15 homogeneous worker nodes, linked via 1Gbps Ethernet; worker nodes are based on commodity hardware: i865 chipset board, 3GHz/32bit Pentium 4 CPU, 1GB RAM, 80GB SATA HD and 1Gbps NIC onboard. More recently, small modifications on pDomus allowed its port to a high-performance ROCKS cluster of 46 3.2GHz/64bit dual-Xeon nodes with 2GB of RAM, linked via 10Gb Myrinet (1st installation in Europe) – see `http://www.di.uminho.pt/search`; in such environment, pDomus clients and services are normal user jobs, running under the supervision of Torque and Maui.

[3] Always used to host the Domus supervisor services during our tests.

4 Evaluation of RAM-based DHTs

In this section we evaluate the scalability of DHTs built with `ram`-based `python-dict` repositories. Thus, at each service that supports such a DHT, the subset of the DHT records that are assigned to that service are preserved by simply using the Python language built-in dictionaries.

4.1 N_Xcli_M_1srv Deployments

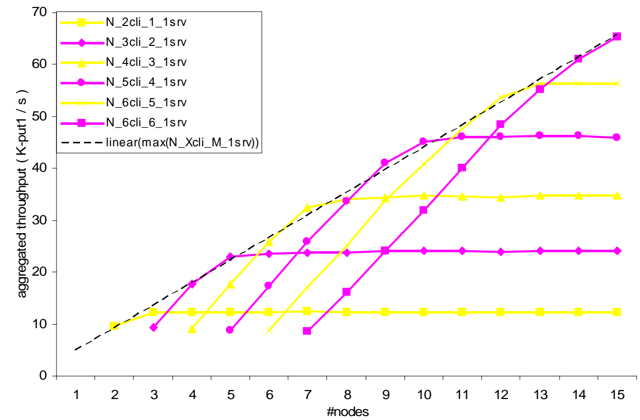

Figure 1. Aggregated throughput for *N_Xcli_M_1srv* deployments with `ram`-based `python-dict` repositories.

Figure 1 plots the aggregated throughput of *put1* operations (in thousands of operations per second), as a function of the overall number of cluster nodes involved (#nodes) in a selected subset of *N_Xcli_M_1srv* deployments (where N nodes with X clients each, plus M nodes with 1 service each, consume an overall of #nodes = $N + M$ nodes).

Each curve represents the best results attained for a subclass of *N_Xcli_M_1srv* where M is fixed and both N and X vary. Using the *N_Xcli_1_1srv* subclass ($M = 1$, *i.e.*, there is only 1 server node) as an example, the methodology used to collect the points for each curve may be described as follows: i) start with $X = 1$ (*i.e.*, with 1 client instance, per client node) and allow N to vary from 1 to 14 (once $N + M \leq 15$ and $M = 1$); the outcome is the base curve *N_1cli_1_1srv*, not shown; ii) repeat the same procedure for $X = 2$ (*i.e.*, increase the number of client instances, per client node, from 1 to 2) and, if necessary, repeat it again for $X = 3$, $X = 4$, etc., provided that each new curve is still able to increase (at least by 1%), the peak throughput of the previous curve; using this criteria, the best curve for the subclass *N_Xcli_1_1srv* is *N_2cli_1_1srv*, shown in figure 1.

For $M = 2$ and $N \in \{1, ..., 13\}$, $M = 3$ and $N \in \{1, ..., 12\}$, etc., we could use the same basic procedure, starting with the base curve given by $X = 1$. However, we can minimize the number of tests, because the best curve from a subclass implicitly defines the starting curve for the next subclass: if X client instances, per client

node, maximize the throughput with M server nodes, then we expect at least the same or a better throughput with $M + 1$ servers. For instance, having concluded that the best curve of the subclass *N_Xcli_1_1srv* ($M = 1$) is attained for $X = 2$, then the search for the best curve of the subclass *N_Xcli_2_1srv* ($M = 2$) should start with the curve *N_2cli_2_1srv* ($X = 2$) and proceed with $X = 3, 4, ...$; of course, increasing X does not necessarily provide throughput gains, as hinted by the curve *N_6cli_6_1srv* in figure 1.

We may draw some general conclusions from figure 1: for each curve, (aggregated) throughput grows linear with the number of client nodes (N) involved, until a saturation point is reached (for each curve, M is fixed; thus, #nodes= $N + M$ only increases when N increases).

In the same figure we may also observe the curve *linear(max(N_Xcli_M_1srv))*, a linear interpolation of the curve *max(N_Xcli_M_1srv)*, not shown. The later gathers the maximum (aggregated) throughput, among all subclasses, for each value of #nodes. For instance: i) *N_2cli_1_1srv* provides the maximum throughput when #nodes $\in \{2, 3\}$; ii) *N_3cli_2_1srv* provides the maximum throughput when #nodes $\in \{4, 5\}$; etc. The interpolation *linear(max(N_Xcli_M_1srv))* shows that *max(N_Xcli_M_1srv)* also grows linear, this time with #nodes. In other words, the right choice of N, X and M ensures a linear scaling of the aggregated throughput.

The knowledge of *max(N_Xcli_M_1srv)* is useful from a cost/benefit optimization perspective: it allows to deploy the most performant clients-servers Domus setup, for any number of nodes involved. This is specially relevant in batch-oriented clusters, where users are assigned a limited number of cluster nodes, for a limited slot of time.

4.1.1 Scalability Constraints

The stabilization of the throughput, for each subclass of deployments, past a certain number of client nodes, may be understood by inspecting the utilization levels of certain resources of the client and server nodes. More specifically, for a `ram`-based DHT, the critical resources are a) CPU and b) Network bandwidth (assuming that each server node has enough RAM to accommodate its share of the DHT).

CPU Utilization Figure 2 plots the (average) peak CPU utilization of the client and server nodes that support the deployment subclasses (curves) of figure 1. For instance, *u(cpu, srv, N_2cli_1_srv)* plots the (average) peak CPU utilization, per server node, for the *N_2cli_1_srv* subclass of deployments, and *u(cpu, cli, N_2cli_1_srv)* plays the same role for the client nodes involved. Figure 2 is divided in 5 separate sections, one for each subclass; in each section, the CPU utilization values fall in the range [0,...,1].

In figure 2 we may observe the same general pattern repeated in all five sections: i) as the overall number (N) of client nodes involved increases, the CPU utilization of the (M, fixed) server nodes grows toward its peak and then stabilizes; ii) the CPU utilization of client nodes also increases, reaches a peak value close to that of servers, and then decreases. In the later case, as soon as the CPU utilization of client nodes falls bellow the CPU utilization of the servers (the cross-over is shown in figure 2, surrounded by ellipses), the aggregated throughput stabilizes in its peak value (this may be verified in figure 1, for all the values of #nodes that come after (inclusive) the elipses of figure 2).

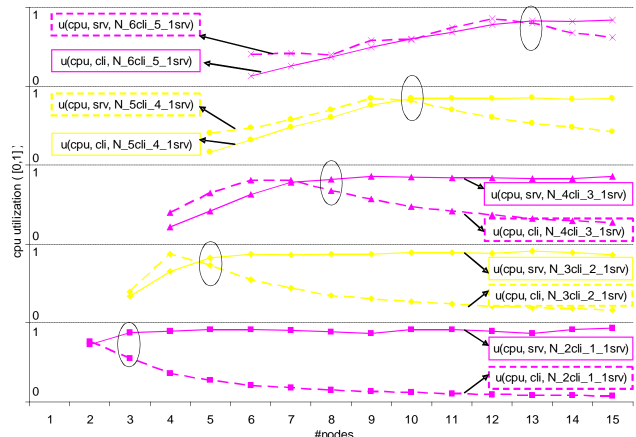

Figure 2. Peak CPU utilization, for *N_Xcli_M_1srv* deployments with `ram`-based `python-dict` repositories.

Network Utilization Figure 3 plots the (aggregated) peak Network utilization registered in the cluster, for each subclass of *N_Xcli_M_1srv* deployments. Like before, the figure was divided in 5 sections, one for each subclass of deployments; in each section, the Network utilization is plotted against the range [0.0,...,0.3]. For instance, *u(net, N_2cli_1_srv)* plots the maximum Network utilization registered during the *N_2cli_1_srv* deployments.

Because the Network is a resource shared by all cluster nodes, the value for the overall (aggregated) Network utilization results from the sum of the Network utilization induced by each cluster node. The cluster nodes exogenous to a specific deployment still generate background traffic (mainly related to cluster-wide resource monitorization), but this traffic is negligible and so was not accounted for.

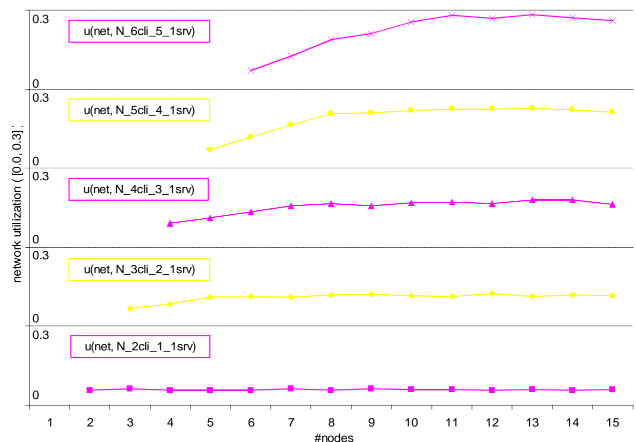

Figure 3. Peak Network utilization, for *N_Xcli_M_1srv* deployments with `ram`-based `python-dict` repositories.

Figure 3 shows that the Network is not a limiting resource of the *put1* throughput, for any deployment. In fact,

even during the most demanding scenario (*N_6cli_5_srv*), the maximum network utilization remained under 30%.

Discussion Based on the previous observations, we conclude that the availability of CPU power in the server nodes is the critical factor for the scalablity of the *N_Xcli_M_srv* class of deployments. In other words, such class scales as long enough server nodes, with spare CPU power, are provided. Otherwise, increasing the number of client nodes (and/or the number of client instances per client node) will just decrease the fraction of work accomplished by each server on the behalf of each client; in turn, this leads to the sub-utilization of the client nodes, as shown by their decreasing CPU utilization levels, after the servers have reached their saturation point – see figure 2.

4.1.2 Impact of Lookup Methods

In this section we analyze the general impact on throughput performance that results from the choice of a specific *lookup/routing strategy* when operating a Domus DHT.

In a DHT, access to a <*key*, *data*> record involves a lookup operation to find the (putative) service/node responsible for the storage of the record. In pDomus, the lookup process may exploit several *routing methods*, weather conventional or distributed, in the scope of a *routing strategy*.

pDomus supports two kinds of *routing graphs* for distributed lookup purposes, namely *Chord* graphs [11] and *de Bruijn* graphs [12]. For each kind, there are several *routing algorithms* available. In the experiments of this paper, all distributed lookups were operated with *Chord* graphs, using the routing algorithm that minimizes the routing hops.

Lookup Methods pDomus implements three lookup methods: *direct* (*D*), *cached* (*C*) and *random* (*R*). A lookup strategy is defined by an ordered sequence of methods. A lookup starts with the 1st method. An automatic fallout takes place from the current method to the next one in the sequence if the current method translates into a lookup miss. The valid strategies (sequences) are: <*D*>, <*D*,*C*,*R*>, <*C*,*R*> and <*R*>. These strategies allow a gracious decay in performance, from more error-prone (but faster) methods, to more accurate (but slower) methods.

The rationale behind the *direct* method is as follows: during the creation of a DHT, the supervisor service performs a (weighted) distribution of DHT buckets, among a set of regular services; this initial distribution follows a simple, deterministic algorithm, such that the only information that is needed to reconstruct that distribution is the number of buckets assigned to each service; later, when a client application opens/creates/reactivates a DHT, the supervisor is asked that information which, thereafter, will be used to resolve any lookup request; if the DHT has not yet been redistributed since its creation, then a 100% hit-ratio will be achieved; if the DHT has been somehow redistributed, lookup-misses will accumulate and, past a certain configurable threshold, the *direct* method will be turned off.

The *cached* method consists on *distributed* lookup, enriched with a client-side LRU cache, per DHT; the cache is feed by distributed lookup results; initially, the cache is empty and the *random* method (see bellow) is necessary to initialize it; the cache contents will then be used to perform *aggregated routing*; this exploits several cache records, in order to define a starting service for a distributed lookup chain, as close as possible (in the routing graph) to the end of the chain (the same rationale is used along the lookup chain, when a routing service inspects many local routing tables in order to make a routing decision); lookup-misses from the *cached* method trigger an automatic fallout to the *random* method and an update of the cache with its results.

The *random* method operates on minimal information about the current distribution of the DHT. Basically, it needs to know only the number of buckets (and not which buckets) assigned to each of the services that support the DHT; the amount of information is small, when compared to the full mapping of buckets to services; this information may be asked to the supervisor or broadcast from there after any redistribution. Thus, in order to define the starting service for a lookup chain, one can simply perform a random choice, among the services that support the DHT, weighted by the number of buckets assigned to each service.

Comparison The experimental results of figure 1 were collected using the strategy <*D*>, in order to achieve the maximum possible throughput. Figure 4 presents the best results obtained by repeating those experiment using other lookup strategies. Each curve *max(N_Xcli_M_1srv,cs, [p])* relates to a specific strategy s, with an eventual parameter p. The curve *max(N_Xcli_M_1srv,<D>)* is directly related to the results of figure 1: in that figure, *linear(max(N_Xcli_M_1srv))* is a linear interpolation of *max(N_Xcli_M_1srv,<D>)*. The curves *max(N_Xcli_M_1srv,* $<C,R>$, 0.5*)* and *max(N_Xcli_M_1srv,* $<C,R>$, 0.25*)* result from the use of cached and random routing, with a cache size of 50% and 25%, respectivelly; the cache size refers to the number of cache records; the size was defined relative to the overall number of buckets of the DHT (100% cached lookup information would require one cache entry per bucket).

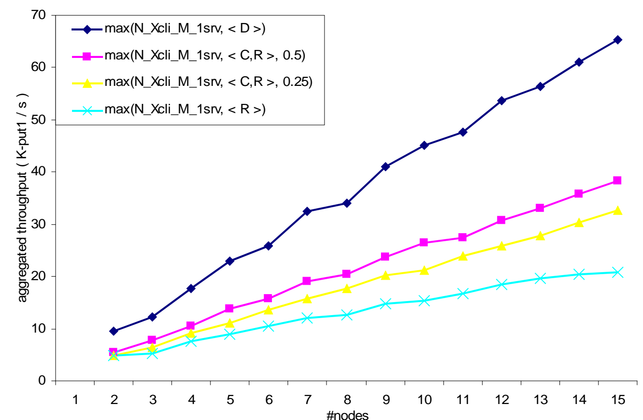

Figure 4. Effect of lookup methods on aggregated throughput, for the *N_Xcli_M_1srv* deployments of Figure 1.

As expected, we observe the strategy $< D >$ to be the most performant, and $< R >$ to represent the worst case. In between, we find the strategies $< C, R >$. Overall, the other strategies preserve the linear scaling shown by strategy $< D >$. Also, as expected, the bigger the cache, the better the throughput achieved. However, we note that a cache of 50% translates in $\approx$ 35% (on average) of the throughput achieved by the $< D >$ strategy, and not 50%; this is because i) querying the cache is inherently slower than the deterministic procedure of the *direct* method, and ii) unless the location of the final-hop is found in the cache, it will be necessary to follow a distributed lookup chain. Finally, we note that halving the cache size, from 50% to 25% does not halve the throughput; instead, the resulting throughput is of $\approx$ 60% (on average) of the throughput achieved by a cache of 50%; this seems to imply that *aggregated routing*, used both at the cache level and along a lookup chain, indeed provides sensible gains. Finally, the results of this experiment stress the importance that a simple, yet very efficient *direct* method may have for high performance scenarios, where short-lived and/or static DHTs are all that is required. It shows also the kind of penalty that is expectable when switching among different methods.

4.2 N_Xcli1srv Deployments

We now investigate the aggregated throughput of the *N_Xcli1srv* class of deployments and make a comparison with the *N_Xcli_M_1srv* class. We recall that the *N_Xcli1srv* class includes deployments where N cluster nodes are each shared by X client instances and one service instance, *i.e.*, a cluster node plays both the role of a client and a server. The co-existence of clients and services may be a frequent situation, specially in small scale clusters, or in batch-managed clusters, were users are assigned limited node subsets. Therefore, it is of relevance to investigate the impact on performance that results from such configuration.

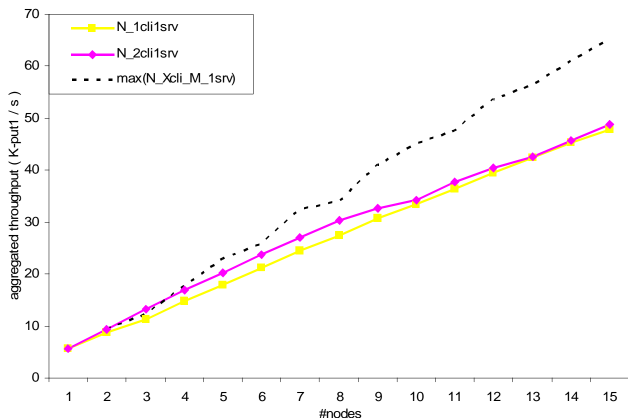

Figure 5. Aggregated throughput for *N_Xcli1srv* deployments with `ram`-based `python-dict` repositories.

Figure 5 plots the aggregated throughput of *put1* operations (in thousands of operations per second), as a function of the overall number of cluster nodes involved (#nodes) in a selected subset of *N_Xcli1srv* deployments (where #nodes = N). The methodology used to select the subclasses and specific deployments that should be investigated, was analogous to the one used for the *N_Xcli_M_1srv* class of deployments. The lookup strategy $<D>$ was again used.

As may be observed, the best subclass is given by *N_2cli1srv* (placing more than 2 client instances per node didn't improve the throughput). More important, the figure also shows that, except for a small number of nodes, *N_Xcli1srv* deployments always fall behind (and diverge) from *N_Xcli_M_1srv* deployments, as shows the comparison of the curves *N_2cli1srv* and *max(N_Xcli_M_1srv)*.

5 Evaluation of Disk-based DHTs

In this section we discuss the evaluation of `disk`-based DHTs built using the `domus-bsddb-btree` platform. As previously stated, the `domus-bsddb-btree` and `domus-bsddb-hash` platforms offer an upper-level access layer to BerkeleyDB, that basically builds on a Red-Black tree of BerkeleyDB databases; this provides a higher-level of organization for DHT records (a DHT bucket may be implemented by a separate BerkeleyDB) and enables increased parallelism for better record access performance.

5.1 N_Xcli_M_1srv Deployments

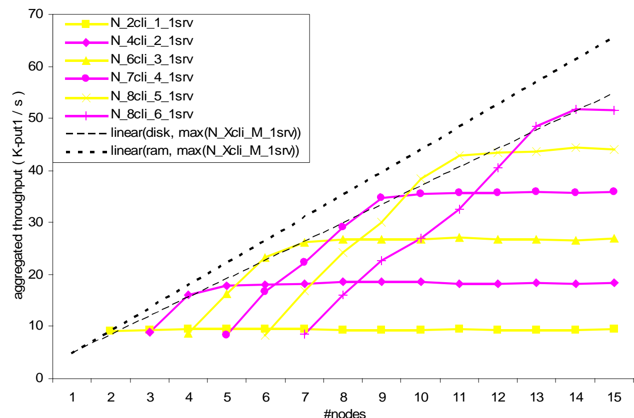

Figure 6. Aggreg. throughput for *N_Xcli_M_1srv* deployments w/ `disk`-based `domus-bsddb-btree` repositories.

Figure 6 shows the maximum aggregated throughput, for selected *N_Xcli_M_1srv* deployments, where N client nodes, with X client instances each, insert records into a `disk`-based DHT, supported by M server nodes, using the `domus-bsddb-btree` platform. The methodology used to find the best combinations of N, X and M was the same as the one used before, for `ram`-based deployments.

Some of the conclusions that were drawn previously for the `ram`-based deployments are still reproducible: 1) for each curve, throughput grows almost linear with N, before stabilization occurs at peak values; 2) the linear interpolation of the best points of all curves, for all values of #nodes, also increases linearly with #nodes – see the curve *linear(max(disk,N_Xcli_M_1srv))* in figure 6.

However, to maximize throughput, `disk`-based deployments typically need more client instances per client

node than the ram-based counterparts of the same subclass. For instance, for the subclass *N_Xcli_3_srv*, ram-based deployments maximize the throughput with $X = 4$ client instances per client node (see figure 1), whereas disk-based deployments require $X = 7$ instances (see figure 6). The reason for this is as follows: each access to a record on a disk-based domus-bsddb-btree repository is intrinsically slower than the same access performed on ram-based python-dict repository [10]; thus, each server node takes more time to dispatch each request, and so client nodes will be idle for longer periods, if the number of local client instances is too small; therefore, in order to maximize the overall throughput, client nodes need more client threads.

The fact that disk-based domus-bsddb-btree repositories have an inherently slower access also helps to explain why the absolute maximum throughput for ram-based python-dict DHTs is always better; this may be verified by comparing the plots *linear(disk,max(N_Xcli_M_1srv))* and *linear(ram,max(N_Xcli_M_1srv))*, in figure 6. Nevertheless, the ratio *linear(disk,max(N_Xcli_M_1srv))* / *linear(ram,max(N_Xcli_M_1srv))* $\approx$ 85%, on average, meaning that disk-based domus-bsddb-btree repositories are very atractive. In fact, during the tests, Disk IO activity for the server nodes was observed to be very modest (presumably because there was always plenty of free RAM for the file system cache). The cluster monitorization also revealed a similar behaviour for CPU and Network utilizations, as measured previously for the ram-based deployments; thus, the same conclusions apply.

5.2 N_Xcli1srv Deployments

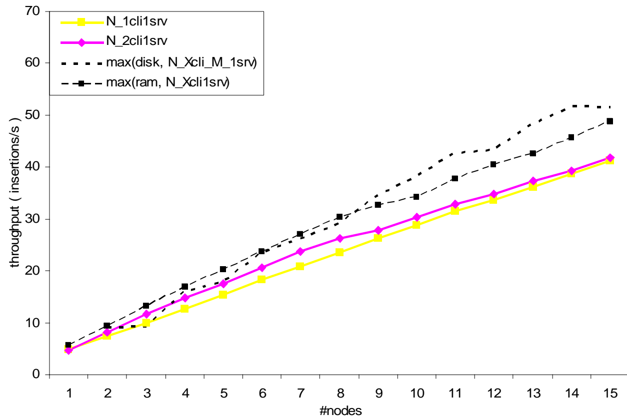

Figure 7. Aggreg. throughput for *N_Xcli1srv* deployments with disk-based domus-bsddb-btree repositories.

Figure 7 shows the aggregated throughput for *N_Xcli1srv* deployments, using disk-based DHTs with domus-bsddb-btree repositories. The experimental methodology used was the same as that employed for the *N_Xcli_M_1srv* and *N_Xcli1srv* classes already evaluated.

The general conclusions previously drawn for the ram-based deployments, of the class *N_Xcli1srv*, still hold: i) *N_2cli1srv* is again the best subclass; ii) *N_2cli1srv* competes with the best disk-based *N_Xcli_M_1srv* deployments (denoted by *max(disk, N_Xcli_M_1srv)*), but only for a small number of nodes, after which both approaches diverge. Moreover, *N_2cli1srv* always falls behind *max(ram, N_Xcli1srv)*, the ram-based *N_2cli1srv* subclass of figure 5; this situation is expected, as a natural result from the intrinsic performance differential between RAM and Disk.

5.3 Comparison with Other Disk-based Platforms

We now provide some figures on the performance of other Disk-based platforms, in order to better understand the real meaning of the pDomus performance evaluation results.

5.3.1 Comparison with MySQL

The choice of MySQL, a widely used database platform, helped us to gain perspective on the relative merits of Domus Disk-based technologies with regard to production-level storage tools. More specifically, we have evaluated MySQL with *N_1cli_1_1srv* deployments, where Python-based MySQL clients (one per client node, for an overall of N client nodes) place insert requests to a single MySQL server, running at a dedicated node. Each deployment operated on a fresh MySQL installation on the server node, with a single user-level database, having just one table with two integer columns (one for the storage of a (primary) *key*, the other for the storage of the related *data*); MySQL access requests were not aggregated at the client-side, that is, each request was performed by one specific network transaction (this makes the comparison fair with pDomus, once DHT access requests are not aggregated by DHT clients).

The results of the experiment were observed to be comparable to those of the *N_1cli_1_1srv* deployments that we have tested in the context of figure 6 (the figure plots only the results for the *N_2cli_1_1srv* deployments, but they are only slightly better than the results for the *N_1cli_1_1srv* deployments). Basically, such reinforces the competitiveness of the performance of pDomus, mainly Python-based.

5.3.2 Comparison with Bamboo

Finally, we have also took the opportunity to gain some insight about the performance of a popular P2P-oriented DHT when instantiated in our test-bed cluster. More precisely, we have put to the test the Bamboo [6] platform. Bamboo is based on Pastry [13], but includes optimizations for high churn-rates, especially in bandwidth-limited environments. Bamboo is written in Java and uses BerkeleyDB (with transactional support) as the underlying storage platform; records are replicated and have a limited lifetime, after which they are removed. Bamboo nodes constantly try optimize their routing tables, to handle the asynchronous arrival or departure of nodes from the DHT.

We have evaluated Bamboo for the full *N_Xcli_1_1srv* class of deployments, using Java-based clients; these tried to saturate the Bamboo servers, during 210s, by inserting $<$*key*, *data*$>$ pairs of random integers. Clients used random servers as gateways, to ensure uniform spreading of routing

load. Proximity neighbor selection by servers was turned off, as the server node set would remain static during a test.

The results of the evaluation are as follows: i) the *put1* (aggregated) throughput was independent of the number ($N \times X$) of client instances; ii) as the number of server nodes (M) increased, from 1 to 4, the *put1* (aggregated) throughput decreased, from $\approx$ 200 put1/s, to $\approx$ 107 put1/s; then, for $M \geq 5$, the throughput remained constant, with a value of $\approx$ 93.5 put1/s. The later results are consistent with the four-level replication of the data records used by Bamboo[4]. Moreover, we have observed that even when $M \geq 5$, Bamboo always used the same four nodes for the storage of records; instead, we were expecting the storage load to be evenly spread among the M nodes of a deployment, something that should not be incompatible with the replication.

The previous throughput figures are at least one order of magnitude bellow than those attainable by the less performant Disk-based pDomus deployments that we have evaluated (see figure 7). Basically, such reinforces our argument in favor of the "right tool for the right job": for a cluster scenario, pDomus is more suitable and performant.

6 Discussion

pDomus is a first implementation of Domus that creates a cluster environment where to evaluate the model embedded concepts and planned features. It supports several storage technologies that may cope with different application scenarios, and is easily extensible to other technologies.

The pDomus evaluation discussed in this paper focused on the study of client/server full-speed scalability for two classes of deployments. For both classes, scalability appears to be linear, with the right combination of client nodes, client threads and server nodes. However, the class that separates clients and services into different cluster nodes showed the best performance.

The impact on performance from the various lookup methods available in pDomus was also studied; specifically, it was possible to understand the performance loss introduced by distributed lookup methods, whether *cached* or almost stateless like the *random* method; these results are specially relevant for large scale DHTs, which may become more common in cluster environments as the average number of nodes per cluster keeps growing; for smaller, short-lived DHTs, the *direct* method is more feasible and its benefits were stressed out by the experiments results.

Finally, we have also investigated the relative merit of the pDomus prototype, with regard to other storage platforms: a database and a DHT; we have observed encouraging results that motivate us to further enhance pDomus.

References

[1] W. Litwin, M.-A. Neimat, and D.A. Schneider. LH*: Linear Hashing for Distributed Files. In *Proceedings of the ACM SIGMOD - International Conference on Management of Data*, pages 327–336, 1993.

[2] R. Devine. Design and implementation of DDH: a distributed dynamic hashing algorithm. In *Proceedings of the 4th Int. Conf. on Foundations of Data Organization and Algorithms*, pages 101–114, 1993.

[3] V. Hilford, F.B. Bastani, and B. Cukic. EH* – Extendible Hashing in a Distributed Environment. In *Proceedings of the COMPSAC '97 - 21st Int. Computer Software and Applications Conference*, 1997.

[4] E.K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Survey and Tutorial*, 6(1), March 2004.

[5] J. Rufino, A. Pina, A. Alves, and J. Exposto. Domus - An Architecture for Cluster-oriented Distributed Hash Tables. In *Procs. of the 6th International Conferenece on Parallel Processing and Applied Mathematics (PPAM'05)*, Poznan, Poland, September 2005.

[6] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of ACM SIGCOMM 2005*, August 2005.

[7] J. Rufino, A. Pina, A. Alves, and J. Exposto. Toward a dynamically balanced cluster oriented DHT. In M. H. Hamza, editor, *Procs. of the International Conference on Parallel and Distributed Computing and Networks (PDCN'04)*, Innsbruck, Austria, February 2004.

[8] M.A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, 1999.

[9] Federico D. Sacerdoti, Mason J. Katz, Matthew L. Massie, and David E. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of the IEEE Cluster 2003 Conference*, 2003.

[10] J. Rufino, A. Alves, A. Pina, and J. Exposto. pDomus: a Platform for Cluster-oriented Distributed Hash Tables. In *Proceedings of the 15th Euromicro Conference on Parallel, Distributed and Network-based Processing*, February 2007. (to be presented).

[11] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balkrishnan. Chord: A Scalable Peer–to–Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM'01*, 2001.

[12] J-C. Bermond, Z. Liu, and M. Syska. Mean Eccentricities of de Bruijn Networks. Technical Report RR-2114, CNRS - Université de Nice-Sophia Antipolis, Valbonne, France, August 1993.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large Peer–to–Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[4] Replication is deeply embedded in Bamboo and cannot be turned off.