pDomus: a prototype for Cluster-oriented Distributed Hash Tables

José Rufino*, Albano Alves, José Exposto
Polytechnic Institute of Bragança, 5300-854 Bragança, Portugal
{rufino, albano, exp}@ipb.pt

António Pina
University of Minho, 4710-057 Braga, Portugal
pina@di.uminho.pt

Abstract

The Domus architecture for Distributed Hash Tables (DHTs) is specially designed to support the concurrent deployment of multiple and heterogeneous DHTs, in a dynamic shared-all cluster environment. The execution model is compatible with the simultaneous access of several distributed/parallel client applications to the same or different running DHTs. Support to distributed routing and storage is dynamically configurable per node, as a function of applications requirements, node base resources and the overall cluster communication, memory and storage usage.

pDomus is a prototype of Domus that creates an environment where to evaluate the model embedded concepts and planned features. In this paper, we present a series of experiments conduced to obtain figures of merit i) for the performance of basic dictionary operations, and ii) for the storage overhead resulting from several storage technologies. We also formulate a ranking formula that takes into account access patterns of clients to DHTs, to objectively select the most adequate storage technology, as a valuable metric for a wide range of application scenarios. Finally, we also evaluate client applications and services scalability, for a select dictionary operation. Results of the overall evaluation are promising and a motivation for further work.

1. Introduction

In the field of Cluster Computing, data-intensive applications may build on *distributed dictionaries*. A *dictionary* is a data repository that holds <*key*, *data*> records, which may be uniquely accessed using the *key* field. Dictionaries require support for a typical set of basic operations like insertions, retrievals, removals, membership queries, etc.

*Supported by the portuguese grants PRODEP III/5.3/N/199.006/00 and FCT SAPIENS/41739/CHS/2001

Distributed Hash Tables (DHTs) are often used to implement distributed dictionaries. Research in DHTs started with models tailored to small/medium-scale networks of workstations (NOWs) [10, 8, 9], and continued with more recent contributions focused in large-scale/internet-wide peer-to-peer (P2P) scenarios [12, 18, 14]; the later demand support for i) efficient routing, ii) wide-area intermittent network connections, iii) continuous arrival/departure of nodes, iv) security/anonymity, etc; this requisites may be relaxed in a typical cluster, a tightly integrated hardware/software environment, of a much lower scale, running on private (and often very high-bandwidth) local networks.

Deploying a P2P-oriented DHT platform [13, 18] in a cluster environment may thus not be advisable. When implementations are targeted to specific scenarios, they usually include functionalities that may be useless in others (and even result in performance penalties). For instance, a distributed lookup strategy may be unappropriate when the number of nodes of a DHT is relatively small, in which case complete lookup information may be replicated at each node, or a deterministic lookup algorithm may also be used.

This paper elaborates on the evaluation of pDomus, a prototype of the Domus architecture. pDomus allows to deploy, operate and manage multiple DHTs and its supporting services. Management facilities include the deactivation/(re)activation of DHTs and the shutdown/restart of services or even entire Domus deployments. Domus DHTs also benefit from a rich set of user-level attributes, providing support for a range of i) hash functions, ii) storage platforms, iii) distribution constraints, iv) lookup strategies, etc.

The evaluation of pDomus focused 1) on the performance of the set of storage technologies currently supported, and 2) on the scalability of a class of client/server deployments. In addition, we introduce a ranking formula function to select the most appropriate storage technology.

The remaining of the paper is organized as follows: section 2 revises the Domus architecture and its foundations,

15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)
0-7695-2784-1/07 $20.00 © 2007 IEEE

section 3 introduces pDomus and the evaluation framework, section 4 evaluates storage technologies, section 5 introduces a ranking function for the later, section 6 measures the scalability of the prototype and section 7 concludes.

2. Previous Work

The Domus architecture [17] derived from our previous work on models for the balanced distribution of the range R_h of an hash function h, over a set of heterogeneous nodes [16, 15]. These models define basic units for coarse-grain and fine-grain balancement: a) the *vnode*, and b) the *partition*. A *partition* is a contiguous subset of the range R_h. At any moment, R_h is fully divided in a set of P disjoint partitions, all of the same size and such that $\#P$ is always a power of 2. A *vnode* may be regarded as a dynamic set of partitions. Depending on their base resources, and its (dynamic) availability, a cluster node n claims a certain number $\#V.n$ of vnodes of a DHT, in which case the node expects to be allocated an ideal quota $Q^i.n = \#V.n/\#V$ of R_h, where $\#V$ is the overall number of vnodes. Nodes, however, are ultimately responsible for partitions and so each node n manages $\#P.n$ partitions, for a real quota $Q^r.n = \#P.n/\#P$ of R_h. In order to keep $Q^i.n$ and $Q^r.n$ as close as possible, for a dynamic number of vnodes and nodes, vnodes will often give/grab partitions to/from others, and the overall number of partitions, $\#P$, may also change.

The scheme above may produce an overall number of partitions that prevents the maintenance of full tables <partition, node>. We have thus investigated distributed lookup mechanisms based on *Chord* and *de Bruijn* graphs [18, 6], and algorithms for *aggregated routing*, suited to our distribution models. These algorithms combine all the routing information of the partitions held by a node to ensure that lookup requests hop between nodes. This allows for shorter routing chains, on average, in comparison to hopping between partitions (conventional distributed lookup).

2.1. The Domus Architecture

The Domus architecture for Distributed Hash Tables (DHTs) was specially designed to support the concurrent deployment of multiple and heterogeneous DHTs, in a dynamic shared-all cluster environment. The execution model is compatible with simultaneous access of several distributed/parallel client applications to the same/different DHTs. Moreover, to improve the utilization of cluster resources, Domus allows the assignment of the *routing* and *storage* functions of a DHT partition, to different cluster nodes.

Domus comprises five major architectural entities – see figure 1: i) client applications (denoted by a_i), ii) DHTs (denoted by d_j), iii) services (denoted by s_k), iv) base services (message passing, resource monitoring [7] and remote execution) and v) cluster nodes (n_k). Figure 1 also illustrates basic architectural properties: a) client applications may access multiple DHTs; b) DHTs may be shared by several client applications; c) DHTs build on a set of Domus services; d) applications and services, Domus-akin or external (not shown), may co-exist on the same node; e) a node runs a single Domus service, per each Domus deployment.

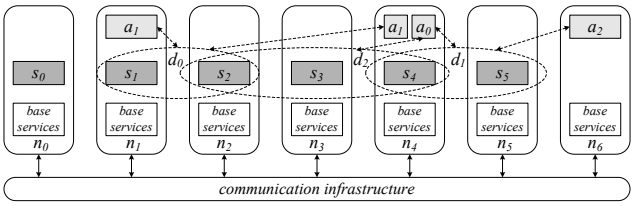

Figure 1. A possible Domus deployment.

Domus DHTs benefit from user-level attributes tailorable to match specific requirements of users/applications; such attributes include i) hash functions, ii) storage platforms, iii) distribution constraints, iv) lookup strategies, etc. Apart from the usual dictionary operations (insertions, retrievals, removals, etc.), Domus DHTs also support deactivation and reactivation operations; 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. Shutdown and restart are similar operations, applicable to Domus services; these operations may imply the deactivation, reactivation or migration of the DHTs supported by the involved services. The deactivation of a DHT and the shutdown of a service depend both on secondary storage to preserve offline state.

Tasks requiring global coordination are managed by a *supervisor* service: a) creation/destruction, shutdown/restart of the overall Domus deployment; b) adding/removal, shutdown/restart of specific services; c) creation/destruction, shutdown/restart and dynamic (re)distribution of DHTs.

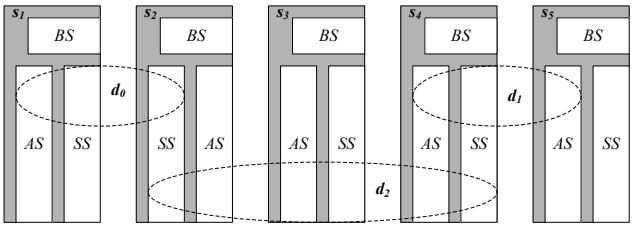

Figure 2. Services, subsystems and DHTs.

Domus services are structured into *balancement* (BS), *addressing* (AS) and *storage* (SS) subsystems – see figure 2. The AS/SS subsystem performs routing/storage functions for subsets of partitions, of one or more DHTs. The AS subsystem keeps a routing table, per partition, to perform distributed lookups along the routing overlay (graph)

that links all the partitions of a DHT; it keeps also a storage reference, per partition, to the service that stores the data records that map onto the partition (thus effectively decoupling routing and storage functions). The SS subsystem maintains local repositories capable of holding dictionaries. Repositories may build on different storage platforms and storage media combinations (e.g., BerkeleyDB [11] over RAM or Disk, etc.), selectable on a per DHT basis. The possible enrollment of a service in several DHTs, at several domains, is illustrated by figure 2; it shows how the AS and SS subsystems of the services $s_1, ..., s_5$ support the DHTs $d_0, ..., d_2$, in the context of the specific scenario of figure 1.

Dynamic load balancing involves all the subsystems. With the help of base services, the BS subsystem monitors the utilization of several 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. The reaching of certain thresholds triggers a) the re-allocation of local partitions, of one or more DHTs, to other services, and/or b) the changing of the overall number of partitions of the DHTs.

3. The Domus Prototype (pDomus)

pDomus mostly builds on a set of Python modules; in addition, a C-crafted module provides for an efficient implementation of several routing and storage functionalities, accessible via Swig [5] midleware; these C-based functionalities use Red-Black trees [3] as the core data-structure.

Domus DHTs supports several storage attributes. These include, among others: a) the attribute `_sm` (storage media) and b) the attribute `_sp` (storage platform). In pDomus, the attribute `_sm` may be set to `ram` or `disk`, depending on the data persistence requirements of the client applications of the DHTs; the storage platforms encompass a variety of different dictionary implementations: a) `python-dict` (Python `ram`-based built-in dictionaries); b) `python-cdb` (Python module for access to `disk`-based "constant databases" [1]); c) `python-bsddb-hash` and `python-bsddb-btree` (use the Python `bsddb` module to access BerkeleyDB `ram/disk`-based databases via `hash/btree` access methods); d) `domus-bsddb-hash` and `domus-bsddb-btree` (C-based external Red-Black trees where each node references a BerkeleyDB database).

Support for new platforms may be added to pDomus with minimal effort, through appropriate midleware. However, not all <_sm,_sp> tuples are necessarily valid. For instance, `python-dict` native medium is `ram` and while a `python-dict` repository may be dumped to `disk`, it cannot be operated from there at run-time; at the opposite side, we have the `python-cdb` platform, which is only `disk`-based. Moreover, it may happen that some dictionary operations are absent: for instance, a `python-cdb` repository is of the kind *Write-once-Read-many* and its utilization comprises two strictly separated phases; in the 1st phase, only inserts are possible; in the 2nd phase, only retrievals are allowed (in pDomus, transition from the 1st to the 2nd phase is triggered by the insertion of the record <None,None>).

Presently, pDomus runs on a dedicated ROCKS cluster [4], with 1 front-end node and 15 worker (homogeneous) nodes, interconnected via 1Gbps full-duplex ethernet. Worker nodes are based on commodity hardware: i865 chipset board, 3GHz/32bit Pentium 4 CPU, 1GB RAM, 80GB SATA HD and 1Gbps NIC on-board. More recently, small modifications on pDomus allowed its execution on another (high-performance) ROCKS cluster[1] of 46 3.2GHz/64bit dual-Xeon nodes, with 2GB RAM, 80GB SATA HD, 1Gb Ethernet and 10Gb Myrinet; in such environment, pDomus clients and services are normal user jobs, running under the supervision of Torque and Maui.

In what follows we evaluate pDomus first installation. We start by evaluating the storage technologies currently supported, under common dictionary operations, and introduce a ranking function for the selection of the most appropriate technology, for an expected DHT utilization pattern. Then, we investigate pDomus throughput scalability for different combinations of client applications and services, considering their number and placement in the cluster nodes.

4. Evaluation of Storage Technologies

The performance and scalability of the various storage technologies of pDomus was measured by repeating, for each <_sm,_sp> valid pair, the following procedure: 1st) set up a new Domus deployment, with only one regular service (always in the same worker node); 2nd) create an empty DHT, based on that sole service, and with the test-specific values for the attributes `_sm` and `_sp`; 3rd) a client application, hosted by another worker node (always the same), performs several dictionary operations on the DHT. This configuration, where most message exchanges involve just two nodes, is adequate to compare the relative merit of each storage technology. Moreover, only UDP-based communication was used between the client and the service, and the service was running a single thread (pDomus also supports TCP and multi-threaded services; however, in other tests, we found UDP-based communication and event-driven single-threaded servers to be the fastest setup, due to the well-known weak performance of Python threads).

Dictionary operations followed a specific order: 1st) the DHT was populated (put1-test); 2nd) all records were retrieved (get-test); 3rd) all records were overwritten (put2-test); 4th) all records were deleted (del-test). Immediately after a get-test, the DHT was shutdown and the storage consumed by the DHT at the file system was accounted for;

[1] See `http://www.di.uminho.pt/search`.

the DHT was then restarted, before the put2-test. Each test was performed 8388608 (8×2^{20}) times. All integers from $\{0, 1, ..., 8388607\}$ were used, sequentially, as keys for <*key*,*data*> records (for the put1-test and put2-test, *data* = *key*). Using such small records (≈8 bytes for 32bit nodes) stressed out the overhead from control data structures.

4.1. Overall Time

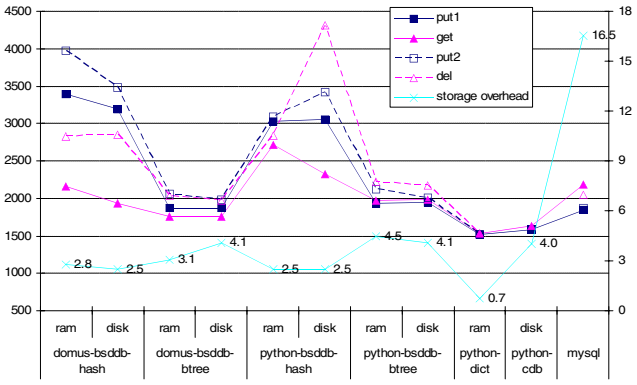

Figure 3. Overall Time and Storage Overhead.

Figure 3 plots the time (in seconds, relative to the left vertical axis) spent by each test (put1/get/put2/del-test), for all the <_sm, _sp> combinations (shown along the horizontal axis). It may be observed that, for all operations (put1/get/put2/del), `python-dict` on `ram` is the fastest technology, closely followed by `python-cdb` on `disk`; however, `python-dict` is volatile at run-time, and although `python-cdb` is persistent, its usage pattern (discussed in section 3) may be too restrictive (in fact, it is compatible with only two kinds of tests: put1-test and get-test).

As such, the best compromise, both in performance and usage flexibility, seems to be `domus-bsddb-btree`, whether on `ram` or `disk`, closely followed by `python-bsddb-btree`. It should also be noted that `hash`-based access methods for the BerkeleyDB technology, as provided by `domus-bsddb-hash` and `python-bsddb-hash`, have the worst performance, again with a small advantage of `domus`-based over `python`-based platforms. Finally, it may be observed that, for `bsddb`-based technologies, `ram` as storage media is not always better that `disk`, as one might expect; in fact, with `_sm=ram`, BerkeleyDB still creates the databases in temporary files, at `/var/tmp`, and in such case database access is underperforming `disk`-native database access.

In order to gain insight about the competitiveness of pDomus with other storage tools, we also tested the MySQL database [2] against the pDomus storage technologies. We repeated the same put1/get/put2/del-test sequence, between a small Python-based MySQL client (hosted by the same node of the Domus client application), and the MySQL server (hosted by the same node of the Domus regular service). The tests were done on a fresh MySQL installation, over a new database, having just one user-level table, with two integer columns (one for the (primary) *key*, the other for the *data* of Domus records). MySQL access operations were not aggregated: each operation was performed by one specific network transaction. This makes the comparison fair with pDomus, once DHT access requests are not aggregated by the `domus_libusr` library at the client-side. As figure 3 shows, the `mysql` technology performance rates near the `bsddb-btree`-based technologies (it would not be fair to compare `mysql` to `python-dict` and `python-cdb`, as these lack persistence or support for some basic operations). We may thus conclude that our pDomus prototype, though mainly Python-based, provides a fair performance.

Other general conclusions may also be attained from figure 3: i) typically, put1-operations are more time consuming than put2-operations and so evaluating only put1-operations would have been misleading with regard to insertion operations; ii) get-operations are the fastest in most of the cases, which is somehow expected; iii) the categorization of del-operations is more irregular and technology-dependent.

4.2. Storage Overhead

Figure 3 also shows the "*relative* storage overhead" (against the right vertical axis), for each evaluated technology. In our context, an "*absolute* storage overhead" refers to the additional amount of storage necessary to hold, at the file system level, all the 8388608 8-byte records, above the absolute minimum that would be required just for the data (8388608 × 8 = 64 Mbytes). For instance, `python-dict` over `ram` would require an overall of (1 + 0.7) × 64 Mbytes = 108.8 Mbytes of storage space, for a "*relative* storage overhead" of 0.7 (i.e., 70%). For `mysql`, indexing and transactional support(which requires intensive logging) imposes a heavy storage overhead of 16.5 (that is, 1650%). Also, `bsddb-btree`-based storage technologies, that exibit fair performance levels, show almost the double storage overhead against `bsddb-hash`-based technologies.

4.3. Throughput Evolution

By only taking into account the overall time of an operation, we may ignore issues relevant to the user perception of the way by which the system accomplishes its work. Basically, these issues are related to the "*turnaround-time* versus *response-time*" dichotomy: operations may take little/reasonable time to execute, but they may severely disturb the system. In these circumstances, users may opt for slower but smoother operations (with a "less intermittent"

/ "more regular" behavior). Moreover, a certain operation may run faster/slower during the startup phase and, conversely, it may execute slower/faster during its final stage.

To understand these issues, we registered the evolution of the throughput (operations per second) for all operations and storage technologies. The results are shown in figures 4 to 8 (to facilitate comprehension, storage technologies were split into several subsets). Curve legends are grouped by operation type (except for figure 4) and, for each group, the tag relative to the technology with better performance comes first, in top-bottom order. Also, curves were smoothed and spikes were removed, to emphasize differences.

Figure 4 aggregates the technologies that exhibit the most sustained throughput: `python-dict` on `ram`, `python-cdb` on `disk`, and `mysql`. They were all able to preserve a constant operation rate, except for the slightly decaying rate of `python-cdb` for get operations.

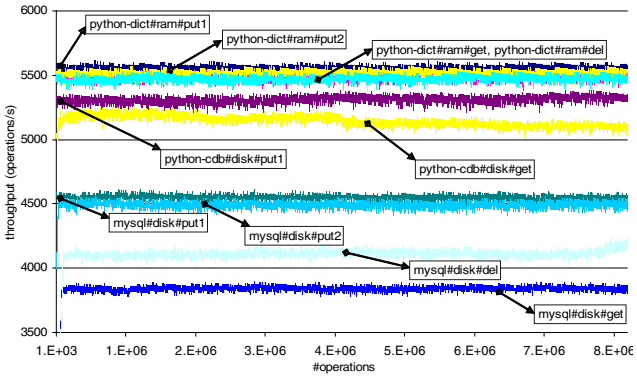

Figure 4. Throughput (`python-dict` on `ram`, `python-cdb` on `disk` and `mysql` on `disk`).

Figures 5 and 6 show the throughput evolution for operations under the `bsddb-hash`-based technologies, over `ram` and `disk`, respectively. In both figures, we may clearly see that throughput evolution for put1 operations follows a saw/jagged line; this is a direct consequence of dynamic hashing, which must periodically double the number of buckets of the hash table; as the current bucket set fills, performance decreases, until the number of buckets doubles; if records were spread uniformly among buckets, they would fill approximately at the same time, when the overall number or records reached a power of 2; in practice, however, records do not evenly spread, and so growing events are slightly delayed from the ideal point, as may also be seen.

With regard to the remaining operations, we may observe that put2 operations (re-insertions) do not suffer from the "growing pains" induced by put1 operations, but their throughput follows the same descending pattern that would be visible by interpolating the jagged lines (in fact, put1 and put2 lines superimpose at their right end). Get operations also follow a descending pattern, although above the put1 operations. Finally, del operations are the only operations that exhibit a constant (and slightly increasing) throughput. We also note that, in both figures, `domus`-based operations are generally shown to be faster than `python`-based ones, as expected in accordance to the results of figure 3.

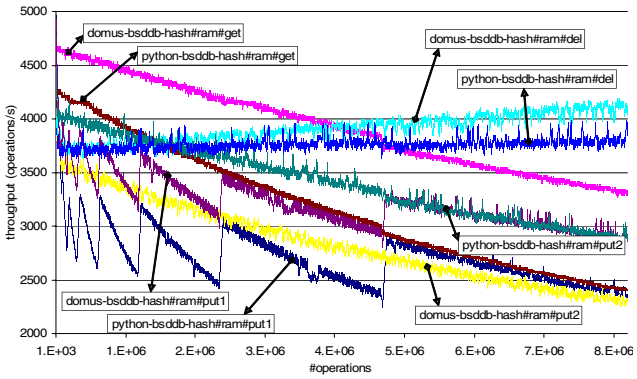

Figure 5. Throughput (`bsddb-hash` on `ram`).

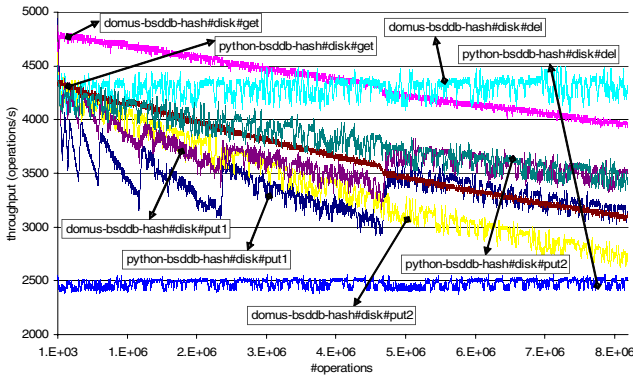

Figure 6. Throughput (`bsddb-hash` on `disk`).

Figures 7 and 8 show a more sustained throughput for `bsddb-btree`-based technologies, when compared with its `bsddb-hash`-based counterparts of figures 5 and 6. Throughput for put1 operations still decays, as the number of already inserted records increases, but the decay rate follows a different pattern. Overall, for the same operations, throughput is higher than that of the `bsddb-hash`-based technologies (as may be also hinted by the different scale used at the vertical axis), and `domus`-based technologies preserve their performance lead over `domus`-based ones.

5. Selection of Storage Technologies

The previous results stress the need for *objective* criteria that allows the selection of the storage technology that best suits specific application scenarios. For instance, in certain

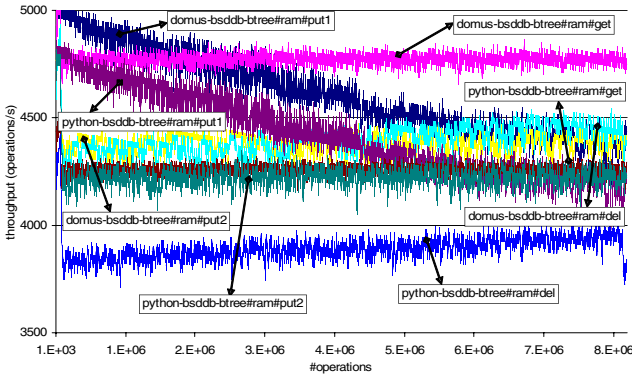

Figure 7. Throughput (`bsddb-btree` on `ram`).

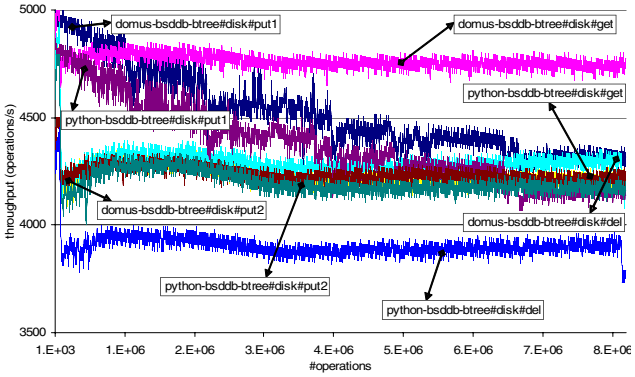

Figure 8. Throughput (`bsddb-btree` on `disk`).

situations, pure performance may be the only relevant factor; in other situations, data persistence may be required for long periods of time; in the later case, minimizing storage overhead may also be of fundamental importance. As such, we introduce the linear metric $r(x)$ – see equation 1, as a ranking weighted formula that takes into account client access patterns to DHTs, to objectively select the most adequate storage technology, when operating with pDomus:

$$r(x) = w(time) \times r(time, x) + w(storage) \times r(storage, x) \quad (1)$$

Under this metric, i) a storage technology x is ranked with a certain value $r(x)$ (with $0 \leq r(x) \leq 1$), ii) time and storage have complementary weights, denoted by $w(time)$ and $w(storage)$ (with $w(time) + w(storage) = 1$), and iii) these weights are applied to the specialized rankings $r(time, x)$ and $r(storage, x)$, given by equations 2 and 3:

$$r(time, x) = p(put) \times [\, p(put1) \times \frac{time(put1, x)}{\max(time(put1, *))} + p(put2) \times \frac{time(put2, x)}{\max(time(put2, *))} \,] + p(del) \times \frac{time(del, x)}{\max(time(del, *))} + p(get) \times \frac{time(get, x)}{\max(time(get, *))} \quad (2)$$

$$r(storage, x) = \frac{storage\_overhead(x)}{\max(storage\_overhead(*))} \quad (3)$$

In Equation 2, $p(put)$, $p(del)$ and $p(get)$ are the probability of write, read and remove operations, expected for a certain scenario, with $p(put) + p(del) + p(get) = 1$; for write operations, $p(put1)$ is the probability of writing a record once and only once, whereas $p(put2)$ is the probability of the record to be rewritten, with $p(put1) + p(put2) = 1$. In addition, i) $time(op, x)$ denotes the time consumed by operation op when using technology x, with $op \in$ {put1,get,put2,del}, and ii) $\max(time(op, *))$ is the maximum value of $time(op, x)$, for all technologies.

In Equation 3, the storage overhead of technology x is denoted by $storage\_overhead(x)$, and its maximum is given by $\max(storage\_overhead(*))$, for all technologies.

Storage technologies may thus be ranked by metric $r(x)$ and, for a certain combination of parameters, the best technology is the one that minimizes the metric. Several combination of parameters are possible; each one conforms to a specific access pattern to the DHT, which ultimately depends on the application scenario and user-level requisites.

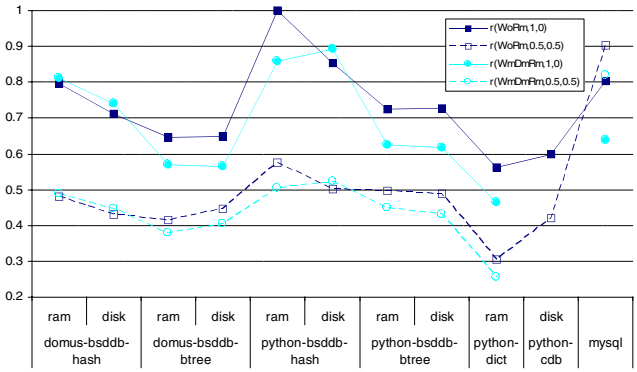

Figure 9. Ranking of Storage Platforms.

Figure 9 plots the ranking of the storage technologies we have studied so far, under two basic classes of scenarios: 1) *Write-once-Read-Many* (WoRm) and 2) *Write-many-Delete-many-Read-many* (WmDmRm). In the WoRm class, only put1 and get operations are possible; furthermore, $p(get) >> p(put1)$ and so $p(put) \approx 0$, $p(get) \approx 1$ and $p(del) = 0$. In the WmDmRm class, all operations are admissible, with writes plus removes having

the same probability as reads, that is, $p(put) = 0.25$, $p(del) = 0.25$ and $p(get) = 0.5$; furthemore, we assume $p(put2) >> p(put1)$ and so $p(put1) \approx 0$ and $p(put2) \approx 1$. For each class, two sub-scenarios are further considered: a) $w(time) = 1$ and $w(storage) = 0$ (time is the only relevant factor); b) $w(time) = 0.5$ and $w(storage) = 0.5$ (time and storage are of equal importance); these weights are used to name the four sub-scenarios of figure 9: <WoRm,1,0>, <WoRm,0.5,0.5>, <WmDmRm,1,0> and <WmDmRm,0.5,0.5>.

Several conclusions may be derived from figure 9. Firstly, for WoRm scenarios, `python-dict` on `ram` is the best choice, except if persistence is required, in which case `python-cdb` on `disk` should be choosen. Secondly, for WmDmRm scenarios, `python-dict` on `ram` is again the best choice for volatile repositories, but `domus-bsddb-btree` on `disk` is now the right choice for persistent repositories (note that `python-cdb` is not eligible as it does not support put2 and get operations).

Finally, we note that if storage (namely "storage overhead") is the only relevant factor, the resulting ranking is already given by the storage overhead plot, in figure 3. As it may be observed, `python-dict` would be the most competitive platform, followed by `bsddb-hash`-based ones.

6. Scalability Evaluation

In a second set of tests we investigated the scalability of the pDomus prototype, relative to the aggregated throughput of put1 operations, under several combinations of a) number and placement of client applications, and b) number and placement of Domus services. These combinations fall into two categories: 1) *nCli_mSrv* (n client applications and m services, each hosted by a different cluster node, thus requiring an overall of $n+m$ nodes); 2) *nCli_nSrv* (n client applications and n services, coupled in the same node, thus requiring an overall of n nodes, one for each pair).

For all tested combinations of x clients and y services we i) set up a Domus deployment based on the y services and ii) created a DHT equally spread across those y services (*i.e.*, each service s was assigned an ideal quota $Q^i.s = 1/y$ of the hash function range; also, during the tests, no redistributions were allowed). Each client was assigned a slot of $\approx 2^{20}/x$ sequential integers from the range $\{0, 1, ..., 2^{20} - 1\}$, for an overall of x disjoint slots; those integers were used to valuate the *key* and *data* fields of <*key*, *data*> records for put1 operations; also, like before, the *data* field always reused the same value of the *key*.

In this preliminary evaluation of the prototype, we focused on the scalability of the insert operations. More specifically, we concentrated on the 1st insertions (put1-operations), taking into account just the fastest storage technology currently supported for the Domus DHTs which, accordingly with the previous evaluations, is `python-dict` over `ram`. Also (and again), services were configured to run single-threaded and message exchanges were UDP-based.

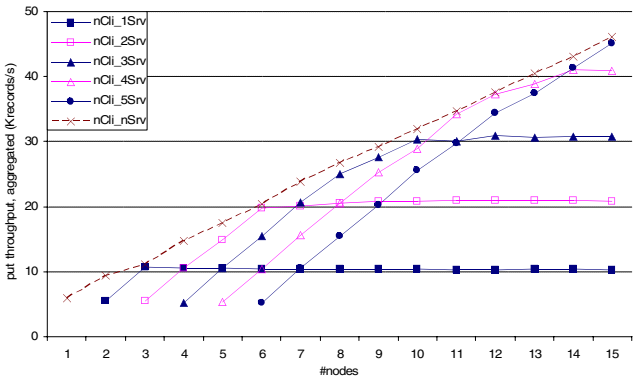

Figure 10. Put1 Scalability.

Figure 10 shows the aggregated throughput of put1-operations for several *nCli_mSrv* and *nCli_nSrv* scenarios. The aggregated throughput, for a scenario, results from the sum of the specific throughputs achieved by all active clients in that scenario; this assumes all clients start doing insertions approximately at the same time (ensured by the test setup), and end almost at the same time (verified to occur). The total nodes involved in a test (#nodes) grows along the horizontal axis; note that, for *nCli_mSrv* and *nCli_nSrv* scenarios, #nodes = $n + m$ and #nodes = n, respectively.

6.1. *nCli_mSrv* Deployments

The tested *nCli_mSrv* scenarios were primarily chosen to find out how many clients were necessary to saturate a certain number of services. However, the opposite thinking is also valid: for a given parallel/distributed client application, for which the parallelization/distribution degree is known/set *a priori*, one should be able to foresee the number of Domus services necessary, to maximize throughput.

We started by testing scenario *nCli_1Srv* and concluded that increasing n from 1 to 2 doubled the throughput, but for $n \geq 2$ (*i.e.*, for #nodes = $n+m \geq 3$) there were no throughput gains. Then, we got the same kind of conclusions for i) *nCli_2Srv* and $n \geq 4$, ii) *nCli_3Srv* and $n \geq 6$, iii) *nCli_4Srv* and $n \geq 8$, iv) *nCli_5Srv* and $n \geq 10$. The lack of more nodes prevented the testing of scenarios *nCli_6Srv* and beyond, but a clear pattern seems to emerge: the maximum aggregated throughput is reached for $n >= 2m$, *i.e.*, "when the number of clients was twice the number of services".

Moreover, by considering only the peak throughputs of the *nCli_mSrv* scenarios, we may also conclude that put1-throughput is able to grow linear under such scenarios; it is only a matter of making the right choice of m and n.

6.2. *nCli_nSrv* Deployments

For clusters with a small/moderate number of nodes, a *2mCli_mSrv* scenario may consume a large fraction of the node set and so *2mCli_mSrv* scenarios may not be the best option for such clusters. Moreover, *nCli_mSrv* scenarios require clients and services to be given a specific node each, but Domus provides mechanisms for dynamic load balancing, compatible with the sharing of a node by many clients and services. This, and the quest for the combination of clients and services that achieves the best cost/performance ratio, motivated the testing of other classes of scenarios.

We thus have studied the *nCli_nSrv* class, for all the 15 possibilities allowed by our test-bed (we recall that in the scenarios of such class, a node is shared by a client and a service, and #nodes=n). As it may be observed, in figure 10, the aggregated put1-throughput for the *nCli_nSrv* scenarios grows linear with n, but near to the peak throughputs achieved by the *nCli_mSrv* scenarios, that is, *nCli_nSrv* and *nCli_mSrv* scenarios perform equally well, when $n = 2m$, although with a slight advantage to the *nCli_nSrv* scenarios.

Though somehow inconclusive, these observations pave the way for further investigation on *nCli_nSrv*, *nCli_mSrv* and other classes of scenarios. For instance, an issue that deserves to be studied is the load induced by each class on the cluster. Thus, for the same performance levels, the light-loaded class becomes automatically the most attractive approach, once nodes involved will have increased spare resources to execute additional applications and services.

7 Discussion

pDomus is a first implementation of Domus that creates a cluster running environment where to evaluate the model embedded concepts and planned features. It supports several representative storage technologies that may cope with different application scenarios, and is easily extensible to include other technologies. Evaluation focused on the i) range of storage technologies currently supported by pDomus, and ii) the study of client/server scalability under different classes of scenarios. In addition, we introduced a ranking formula that proved its usefulness to evaluate the merit of each technology for specific application demands.

Scalability appears to be somehow limited but it is necessary to take into account that, in the experiments, clients tried to saturate the DHT services, an extreme situation that hardly matches real applications, where clients activity is multiplexed between many tasks, not necessary related to DHT access. In the future, we plan to i) refine the study of the prototype scalability, ii) investigate the impact of using different lookup strategies and iii) perform a comparison with other DHT platforms (both cluster and P2P-oriented).

References

[1] Constant Database Library. http://cr.yp.to/cdb.html.

[2] MySQL Database. http://www.mysql.org.

[3] RB-Trees Library. http://libredblack.sourceforge.net.

[4] Rocks Clusters. http://www.rocksclusters.org.

[5] SWIG. http://www.swig.org.

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

[7] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of the IEEE Cluster 2003 Conference*, 2003.

[8] 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.

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

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

[11] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Procs. of the FREENIX Track: 1999 USENIX Annual Technical Conference*, 1999. (see also http://www.sleepycat.com).

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content–Addressable Network. In *Proceedings of the ACM SIGCOMM'01*, 2001.

[13] 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. (see also http://bamboo-dht.org).

[14] 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.

[15] J. Rufino, A. Alves, A. Pina, and J. Exposto. A cluster oriented model for dynamically balanced DHTs. In *Procs. of the IEEE Int. Parallel and Distributed Processing Symposium (IPDPS '04)*, Santa Fe, New Mexico, USA, April 2004.

[16] 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.

[17] 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.

[18] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balkrishnan. Chord: A Scalable Peer–to–Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM'01*, 2001. (see also http://pdos.csail.mit.edu/chord).