Shortcut Routing for Chord Graphs in the Domus Hash Space

José Rufino[1] António Pina[2] Albano Alves[1] José Exposto[1]

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

ABSTRACT

We present and evaluate shortcut routing algorithms for Chord graphs in the hash space, specifically developed to accelerate distributed lookups in the Distributed Hash Tables (DHTs) of the Domus framework. The algorithms explore our findings about the relation of *exponential* and *euclidean* distances in Chord graphs, in conjunction with the availability, in each DHT node, of multiple routing tables of the underlying Chord graph. The outcome are routing decisions capable of achieving average distances as low as 40% of those offered by Chord's conventional routing method. Moreover, the supplemental computational effort to take the shortcut routing decisions is sufficiently low to make the algorithms useful in a broad set of application scenarios.

KEY WORDS
Distributed Hash Tables, Distributed Lookup, Evaluation.

1. Introduction

Distributed Hash Tables (DHTs) have been widely used as an effective approach to the distributed storage of data dictionaries, the distributed lookup of objects or even a combination of both. A set of 1st generation models [8] essentially targeted Cluster environments for distributed storage purposes; in this models, the need to cope with dynamic storage needs and to make efficient use of the storage resources, dictated the expansion or contraction of the DHTs, with base on distributed versions of Dynamic Hashing [5]. Later approaches [18, 2] focused on Peer-to-Peer (P2P) environments [17], where the DHT paradigm is mainly used as a scalable solution (in space and time) to the distributed lookup of objects. Given that Cluster and P2P environments have almost dual properties (in what concerns scale, composition, reliability, bandwidth, etc.), using distributed lookup schemes in the Cluster may seem misplaced. However, our investigation framework – the Domus architecture and platform for Cluster-based DHTs [15, 16] –, provides a good opportunity for the effective use of such schemes: if many DHTs are deployed in the Cluster, and dynamic balancing mechanisms are applied, the spacial and temporal complexity of centralized lookup schemes increases substantially, thus making attractive a distributed approach.

This paper presents the results of the investigation that allowed Chord graphs [18], as a distributed lookup tool, to fit Domus specific needs. Our starting point is a different kind of models used for the partitioning of the hash space by the DHT nodes. Contrarily to the Consistent Hashing approach [9], that underlies Chord, our partitioning models allow DHT nodes to be given subsets of the hash space that are composed of sparse (non-contiguous) values, thus preventing Chord routing in the node space (the usual case). The need to operate Chord graphs in the hash space lead us to develop shortcut routing algorithms that counterbalance the higher routing costs that would result from using Chord's conventional routing algorithm in that space. Moreover, the shortcut algorithms even improve on the conventional routing costs on equivalent Chord graphs in the node space, and their applicability is not limited to Domus.

The remaining of the paper is organized as follows: section 2 makes a brief reference to our partitioning models; section 3 traces our path towards Chord graphs in the hash space; section 4 presents the theoretical foundations of the shortcut algorithms described in section 5; section 6 provides several evaluation results and section 7 concludes.

2. Previous Work

In a DHT, the *partitioning* of the hash space bounds a certain subset of hashes to each DHT node. More formally: let N be the dynamic set of nodes supporting a DHT where the hash space is $H = \{0, 1, ..., 2^{\mathcal{L}-1}\}$ for a certain number of $\mathcal{L}$ bits[1]; then, $H(n)$ is the subset of H specific to the node $n \in N$, such that $\bigcap_{n\in N} H(n) = \oslash$ and $\bigcup_{n\in N} H(n) = H$; that is, the hash space H is fully divided in mutually exclusive subsets, one per node; the set of the subsets that result from the partitioning process is referred to as a *partition*.

[1] Akin to the *splitlevel* concept of Dynamic Hashing schemes [5].

Partitions Hashing	Homogeneous	Heterogeneous
Static	M1	M3
Dynamic	M2	M4

Table 1. Domus Partitioning Models.

Domus *partitioning models* only define the *number* of hashes of each subset, not its *identity* (*i.e.,* the hashes themselves). The models ensure a *perfect distribution*[2] of the number of hashes per DHT node, for different kinds of hashing and partitions (see Table 1.) A detailed description of all the models, refining and expanding previous work [14], may be found at [13]. In the scope of this paper, only M2 is relevant, as it was the base model for the simulations.

3. Distributed Lookup for the Domus DHTs

As already stated, our partitioning models only define the *number* of hashes per node, allowing many possible ways to define the *identity* (e.g., random, alternate, contiguous, etc.). Whichever method is used to define the identities, a *lookup* method is also needed to recover them (that is, a method to discover which node is responsible for a certain hash). We thus needed to investigate lookup algorithms suitable to our partitioning models and target environment (the Cluster).

Using distributed lookup schemes in a Cluster environment may seem misplaced: they are typical of Peer-to-Peer (P2P) environments which, contrarily to Clusters, have a much wider scale, intermittent node composition and lower bandwidth connections, all factors that claim for the distribution of the storage and access loads induced by the lookup process. Domus, however, supports multiple and dynamically balanced DHTs; compared to a single and static DHT, such translates in much more lookup information, frequently updated, making the case for a distributed approach.

Among the possibilities that were investigated, Chord [18] was found suitable and adaptable to Domus needs.

3.1. Chord Graphs in the Node Space

In Chord, Consistent Hashing [9] is used for partitioning of the circular (modulo $\#H$) hash space $H = \{0, 1, ..., 2^{\mathcal{L}} - 1)$ of $\#H = 2^{\mathcal{L}}$ hashes, produced by an hash function f of $\mathcal{L}$ bits, in $\#N$ contiguous subsets, one for each node n of the DHT[3]; the left/right limits of the subsets are simply defined by the hashes $f(n)$ that result from feeding the hash function f with each node identifier n; thus, a node n is responsible for the subset $\{f(p) + 1\ mod\ \#H, f(p) + 2\ mod\ \#H, ..., f(n)\}$, where p is the node for which $f(p)$ immediately precedes (modulo $\#H$) the hash $f(n)$ of node n. Figure 1 complements this generic description; it depicts a possible partition of the hash space $H = \{0, 1, ..., 7)$ among nodes $N = \{n_0, n_1, n_2\}$, including the resulting *partition table*.

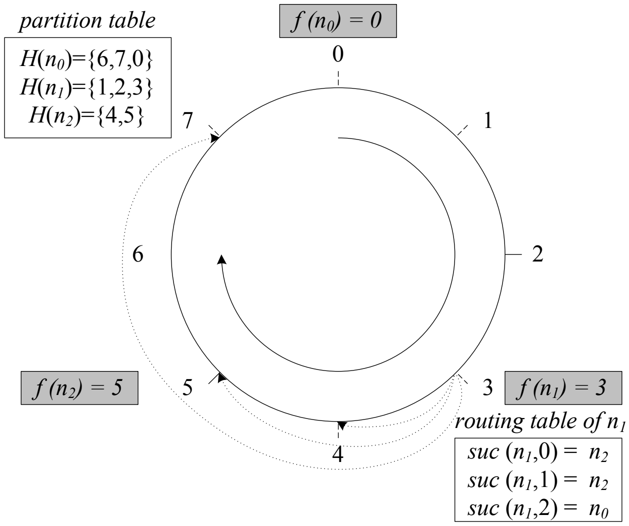

Figure 1. Chord in the Node Space.

In order to avoid the need to replicate the full partition table, or to query it in a centralized location, Chord extends the original Consistent Hashing approach, described above, by building a graph G^N, on the node space N of the DHT. Each DHT node will function as a vertex on G^N and, as such, it will store a *routing table* of $O(log\#H)$ size, carefully crafted to allow the discover of the hosting node of any hash in $O(log\#N)$ steps. A routing table will have $\mathcal{L} = log_2\#H$ entries, with indexes $l = 0, 1, ..., \mathcal{L} - 1$; for a node n, the l'th entry of its routing table will hold the identifier of the node responsible for the hash $(f(n) + 2^l)\ mod\ \#H$; this node is denoted by $suc(n, l)$, that is, the l'th successor of n in G^N. Figure 1 also shows the routing table of n_1.

To discover the node responsible for an arbitrary hash t, starting from node n, requires to: 1) find the largest l such that $[(f(n) + 2^l)\ mod\ \#H] \leq t$ (modulo $\#H$); 2) forward the request to node $suc(n, l)$; 3) in each visited node, repeat steps 1) and 2). This algorithm ensures a maximum distance of $d_{max} \approx log_2\#N$, with an average of $d_{avg} \approx \frac{d_{max}}{2}$ [10].

3.2. Chord Graphs in the Hash Space

Contrarily to the Consistent Hashing approach used by Chord, our partitioning models don't ensure contiguous subsets of H, for each DHT node[4]. For that reason, distributed lookup in Domus, based on Chord, cannot use the

[2] The *perfect distribution* concept is borrowed from the *balls-into-bins* models [12], in which for m objects (*balls*) and n servers (*bins*), the load of each server won't surpass $(m/n) + 1$ nor will be less than (m/n) [3].

[3] Chord further maps hashes to reals in $[0, 1)$, an extra step not used by Domus and thus not considered here (without any loss of generality).

[4] A possible initial contiguity would be quickly destroyed by Domus dynamic balancing, that exchanges arbitrary hashes between DHT nodes.

graph G^N, built in the node space. Instead, it uses a graph G^H, in the hash space – a graph where the vertexes are all the possible hashes of H. In such context, each DHT node n with a subset of hashes $H(n)$ will host $\#H(n)$ routing tables, one per each hash $h \in H(n)$; the tables will also have $\mathcal{L} = log_2\#H$ entries, with indexes $l = 0, 1, ..., \mathcal{L} - 1$; but, for each hash $h \in H(n)$, the l'th entry of its routing table will now identify the hosting node of the hash $suc(h, l) = (h + 2^l)\ mod\ \#H$ (l'th successor of h in G^H).

Moreover, the routing algorithm presented above for G^N is trivially adaptable to G^H. To discover the hosting node of an hash t, starting from the hosting node of an hash c, requires: 1) find the largest l such that $[(c+2^l)\ mod\ \#H] \leq t$ (modulo $\#H$); 2) forward the request to the hosting node of $suc(c, l)$; 3) in each visited node, repeat steps 1) and 2).

However, using this algorithm, the average distance on G^H (approximated by $\frac{log_2\#H}{2} = \frac{\mathcal{L}}{2}$) will be higher than on G^N , once $\#H \geq \#N$. On the other hand, because each DHT node will now host multiple routing tables, there's a clear opportunity to enhance Chord's conventional routing, with new algorithms that exploit the topological information of multiple routing tables, at once, in order to find routing shortcuts. In order to lower the average distance on G^H, towards the average distance on G^N, those new algorithms should, at least, prevent the same DHT node to be visited more than once, along a *routing chain*, like happens with the conventional algorithm on G^N. Ideally, the new routing algorithms should even attain smaller average distances.

4. Euclidean and Exponential Distances in G^H

The algorithms proposed in the paper explore our findings about the *euclidean* and the *exponential* distances of vertexes in G^H (or, equivalently, of hashes in H). In what follows, we first introduce the necessary concepts and notation, and then explore the relations between those distances.

4.1. Base Concepts and Notation

The distance $d(x, y)$, between the vertexes x and y of a graph is, by definition, the number of edges of the shortest path between x and y. For a Chord graph in the hash space, G^H, $d(x, y)$ may be regarded as an *exponential distance*: it is the minimum number of exponential hops of length 2^l (with $l = 0, 1, ..., \mathcal{L} - 1$ and $\mathcal{L} = log_2\#H$) that are necessary, in order to reach y from x, by hoping in the hash space $H = \{0, 1, ..., 2^{\mathcal{L}} - 1)$, left to right, modulo $\#H = 2^{\mathcal{L}}$.

Another measure of distance between x and y is the *euclidean distance*, $d_{euc}(x, y)$, as given by Formula 1:

$$d_{\text{euc}}(x, y) = \begin{cases} y - x & \textbf{if} \quad x \leq y \\ 2^{\mathcal{L}} - (x - y) & \textbf{if} \quad y < x \end{cases} \tag{1}$$

Accordingly with Formula 1, the euclidean distance between the vertexes x and y, $d_{euc}(x, y)$, measures the number of hashes separating x from y, in the hash space $H = \{0, 1, ..., 2^{\mathcal{L}} - 1)$, left to right, modulo $\#H = 2^{\mathcal{L}}$. The exponential distance may be easily derived from the euclidean distance; simply put, the exponential distance $d(x, y)$ is measured by the number of bits with value `1` in the binary representation of the euclidean distance $d_{euc}(x, y)$. This is a well established relationship, as previously noted in [6].

Before proceeding, we also recall some useful concepts:

- x is *predecessor* of y, in G^H, if $d(x, y) = 1$;
- x is *successor* of y, in G^H, if $d(y, x) = 1$.
- x is *anterior* to y, in H, if $x < y$, modulo $\#H$;
- x is *posterior* to y, in H, if $x > y$, modulo $\#H$;

It follows that the *set of anteriors* of a certain hash h, $Ant(h)$, and each specific *anterior*, $ant(h, l)$, are given by:

$$Ant(h) = \{ant(h, l) : l = 1, 2, 3, ..., 2^{\mathcal{L}} - 1\} \tag{2}$$

$$ant(h, l) = (h - l)\ mod\ 2^{\mathcal{L}} : l = 1, 2, 3, ..., 2^{\mathcal{L}} - 1 \tag{3}$$

4.2. Impact of the Minimization of d_{euc} on d

The former concepts are necessary to understand the gist of our shortcut algorithms. Basically, they explore our findings about the impact of the minimization of the euclidean distance, on the minimization of the exponential distance. The relations involved may be found by comparing the monotony of sequences S and S_{euc}, as defined bellow:

- $S = < d(Ant(h), h) >$: the sequence of exponential distances, from the *anteriors* of any hash h, to h;
- $S_{\text{euc}} = < d_{\text{euc}}(Ant(h), h) >$: the sequence of euclidean distances, from the *anteriors* of any hash h, to h.

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

Table 2. S **and** S_{euc}**, for** $\mathcal{L} = 1, 2, 3, 4, 5$**.**

Table 2 shows sequences S and S_{euc}, for $\mathcal{L} = 1, 2, 3, 4, 5$; for each $\mathcal{L}$, the vertical alignment of the table allows to properly couple exponential and euclidean distances, measured from the same *anterior* of an arbitrary hash h; the symbol ":" separates a group of 2^l distances, from its right-side sibling, of 2^{l+1} distances (with $l = 0, 1, ..., \mathcal{L} - 1$).

By definition, S_{euc} sequences are arithmetic progressions, of ratio 1, with growing monotony. However, as Table 2 clearly shows, S sequences have no monotony (they grow and decay, alternately). This means that "the minimization of euclidean distance" does not always imply "the minimization of exponential distance". For instance, in the section $\mathcal{L} = 4$ of the Table 2, the euclidean and exponential distances represented in bold show that we have $d_{\text{euc}}(ant(h, 11), h) = 11 < 12 = d_{\text{euc}}(ant(h, 12), h)$, although $d(ant(h, 11), h) = 3 > 2 = d(ant(h, 12), h)$.

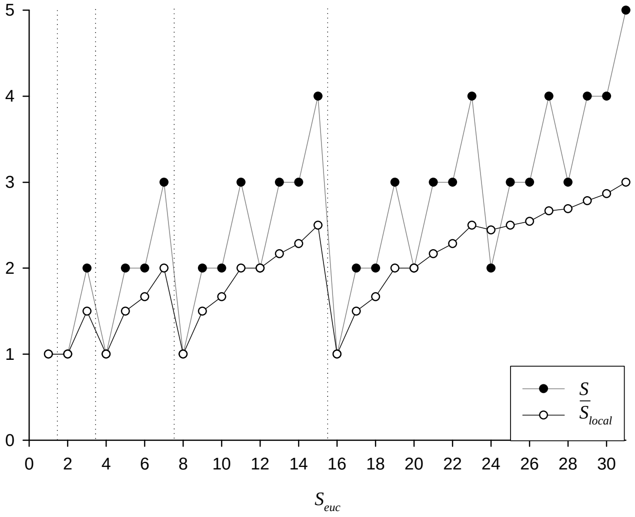

Figure 2. S **and** $\overline{S}_{local}$**, for** $\mathcal{L} = 5$ **.**

Figure 2 shows a graphical perspective of the section $\mathcal{L} = 5$ of Table 2; it plots the exponential distances of the S sequence, as a function of the euclidean distances from the S_{euc} sequence (used as the scale for the horizontal axis); notice that the vertical dotted lines divide the figure in the same zones of the Table 2 defined using the symbol ":".

It may then be observed that: i) whenever the euclidean distance is a power of 2, the exponential distance equals to a local minimum of 1, as expected upon the concepts of Section 4.1 (in other words, the *anterior* hash that is at such euclidean distance from a certain target hash h, is also a *predecessor* hash of that h); ii) between a local minimum, and the next one (from left to right, in the figure), the exponential distance exhibits a growing trend, though with periodic exceptions; nevertheless, if we plot the sequence of accumulated averages of the exponential distances between two local minimums, $\overline{S}_{local}$, it becomes more evident that, in probabilistic terms, the closer to a local minimum (from right to left), the lower will be the exponential distance to the target hash h; in essence, this is the rational behind the shortcut routing algorithms, presented in the next section: for a certain target hash h, try to find local routing tables of *predecessors* of h and, in its absence, use the routing tables of the *anterior* hashes that are closest to those *predecessors*.

5. Shortcut Routing Algorithms

This section presents the routing algorithms we have investigated, in conjunction with its auxiliary data structures.

5.1. Routing Trees

In order to perform efficient shortcut routing, the set of routing tables of each node (a total of $\#H(n)$ tables for a node n, with a table per each hash bound to n) is kept in a local balanced tree (e.g., an AVL or Red Black Tree). Basically, the tree holds registers of scheme $<hash$, $routing\ table>$, indexed and ordered by the $hash$ field.

Special requisites of the problem domain should also be supported by the tree platform. For instance, the automatic finding of the closest register in the tree when the initial target register is absent, facilitates the usage of the routing tables of *anteriors* when *predecessors* are not found. The tree structure and algorithms must also be compatible with the circularity of the hash space, $H = \{0, 1, ..., 2^{\mathcal{L}} - 1)$.

5.2. Conventional Routing (CR)

We now revisit the Conventional Routing (CR) algorithm, later used as a comparison basis. In this regard, we start by introducing notation for the core set of hashes involved in a distributed lookup: i) t (target) – the hash whose location (hosting node) is to be found; ii) c (current) – the hash that was chosen as the next hop in G^H, by the previous routing decision; iii) n (next) – the hash that will be chosen as the next hop in G^H, by the current routing decision.

Algorithm 1: Conventional Routing (CR).

```
1. compute the successors of the hash $c$ in the graph
   $G^H$, and their exponential distance to the hash $t$
2. define $n$ to be the successor of $c$ that is closest to $t$
3. find the routing table of $c$ in the local routing tree
4. find the hosting node of $n$ in the routing table of $c$
5. if $n = t$ then end (the hosting node of $n$ also hosts $t$)
   else forward the request to the hosting node of $n$ endif
```

Making use of the previous notation, Algorithm 1 is an alternative formulation of the Conventional Routing algorithm introduced in Section 3.2; also, in this new formulation, the algorithm makes explicit use of the routing tree.

5.3. Exhaustive Shortcut Routing (SR-all)

With a single access to the routing tree, the CR algorithm is the fastest, in the time to take a routing decision. On the other hand, as we shall see in Section 6, it produces the largest routing chains, in the number of network hops.

Algorithm 2: Exhaustive Shortcut Routing (SR-all).

```
1. traverse the routing tree and find the hash $b$ (best)
   with the lowest exponential distance to the hash $t$
2. do Conventional Routing (Algorithm 1), with $c = b$
```

Another extreme possibility is to analyze all routing tables of a routing tree, to make the best possible routing decision, with the available local routing information. This is the gist of the Exhaustive Shortcut Routing (SR-all) algorithm – see Algorithm 2. However, the exhaustive search of the routing tree may consume too much time. Thus, an intermediate solution (as shown in next section) is desirable.

5.4. Euclidean Shortcut Routing (SR-euc)

The Euclidean Shortcut Routing (SR-euc) algorithms achieve routing decisions almost as good as the optimal decisions of the SR-all algorithm, but with a fraction of its cost, in the number of accesses to the routing tree. As previously stated, the SR-euc algorithms exploit the relations we have found, between the minimization of the euclidean distance and the minimization of the exponential distance.

Algorithm 3: 1-Euc. Shortcut Routing (SR-euc-1).

```
1. $p \leftarrow tsearch\_MinEuclideanDistance(pred(t, 0))$
2. **if** $d(p, t) < d(c, t)$ **then** $b \leftarrow p$ **else** $b \leftarrow c$ **endif**
3. do Conventional Routing (Algorithm 1), with $c = b$
```

The main difference between the several SR-euc variants lies in the number of accesses to the routing tree. For instance, Algorithm 3 (SR-euc-1) involves 1 access (in step 1.), used to find the hash $pred(t, 0)$, that is, the predecessor of the target t, in G^H, at the exponential distance of 2^0; if such predecessor is not found in the routing tree, the function $tsearch\_MinEuclideanDistance$ will return the *anterior* hash closest to that predecessor (in euclidean distance), so that its routing table is used instead; in step 2., the local hash p, found in step 1., is compared against the local hash c, chosen as the next-hop by the previous routing decision; the comparison will find which one is closest (in exponential distance) to the target t; in step 3., the local hash b, elected in step 2., will feed the Algorithm 1, to complete the routing decision, by defining the next-hop.

Once there are $\mathcal{L}$ predecessors of the target t, in the graph G^H, the SR-euc-1 algorithm may be refined to accommodate as much searches as needed, for those predecessors. Thus, an algorithm SR-euc-2 would also try to find, in the routing tree, the hash $pred(t, 1)$, at the exponential distance of 2^1 to t; similarly, an algorithm SR-euc-3 would also try to find the hash $pred(t, 2)$, at the exponential distance of 2^2 to t; and so on, for a maximum number of $\mathcal{L}$ searches in the routing tree, as illustrated by the Algorithm 4 (SR-euc-$\mathcal{L}$).

Algorithm 4: $\mathcal{L}$-Euc. Shortcut Routing (SR-euc-$\mathcal{L}$).

```
1. $b \leftarrow c$
2. **for** $l \leftarrow 0, 1, ..., \mathcal{L} - 1$ **do**
2.1. $p \leftarrow tsearch\_MinEuclideanDistance(pred(t, l))$
2.2 **if** $d(p, t) < d(b, t)$ **then** $b \leftarrow p$ **endif**
3. do Conventional Routing (Algorithm 1), with $c = b$
```

SR-euc-1 and SR-euc-$\mathcal{L}$ are thus extreme cases, in a "family" of algorithms SR-euc-l (for $l = 1, 2, 3, ..., \mathcal{L}$) and, as such, they were the only ones evaluated from that family. In this regard, the natural expectation is that SR-euc-$\mathcal{L}$ will ensure shorter routing chains, in comparison to SR-euc-1, once the later exploits much less local routing information. On the other hand, the average routing load per routing decision for the SR-euc-1 algorithm is expected to be smaller.

6. Evaluation

This section presents the results of the simulation of the routing algorithms previously presented. The simulation was performed in two phases: 1) a setup phase; 2) an evaluation phase. In both phases a platform of Red Black Trees[5] that suits the requisites referenced in Section 5.1 was used.

In the setup phase, the partitioning model M2 (referenced in Section 2) was applied, for a different number of DHT nodes, $\#N = 1, 2, 3, ..., 1024$; more specifically, for each value of $\#N$, the model M2 was used to define the overall number of hashes, $\#H$, and the specific number of hashes per each node, $\#H(n)$; then, each node was given a random subset of $\#H(n)$ hashes from H, thus completing the partitioning of H by N; this process was repeated 10 times, thus leading to 10 different partitions, for each different $\#N$; as a result, $1024 \times 10 = 10240$ partitions were generated; then, for each partition, it was necessary to define its set of routing tables, which basically define a graph G^H; in the end, 10240 different graphs were defined.

In the evaluation phase, each one of the 10240 graphs were navigated, using all 4 routing algorithms. For each graph, and each algorithm, $\#H^2$ routing chains were followed: every vertex (hash) of the graph was the starting point of a lookup for all the other vertexes, so that a total of $\approx 1.57 \times 10^{12}$ chains were followed. For each

[5] See http://libredblack.sourceforge.net

chain, a set of metrics was collected. The graphics in this section show average values for the metrics: for each $\#N = 1, 2, 3, ..., 1024$, the plotted value is an arithmetic average of the values collected for each one of 10 graphs G^H; in turn, the value collected for each graph is an average of those measured for each of the $\#H^2$ routing chains.

The typical effects of the partitioning model M2 are visible in all charts: M2 implies a saw-shaped exponential increase of $\#H$, as $\#N$ increases; this produces a logarithmic growth of the charts metrics, eventually also saw-shaped.

6.1. Average Distance per Chain ($\overline{d}_{chain}$)

Figure 3 plots the experimental averages $\overline{d}_{chain}[alg, G^H]$, for the vertex distance, when routing with the algorithm $alg \in \{$CR, SR-euc-1, SR-euc-$\mathcal{L}$, SR-all$\}$, in the hash space, under the conditions of the simulation. It plots also the theoretical average $\overline{d}_{avg}[\text{CR}, G^N] = \frac{log_2\#N}{2}$ for the Conventional Routing algorithm, in the node space. The relative (%) degree of the optimizations achieved by the SR algorithms in relation to the CR ones is also given.

A first validation of the simulation results was performed by verifying that, when using Conventional Routing, the experimental values $\overline{d}_{chain}[\text{CR}, G^H]$ almost match the theoretical values $\overline{d}_{avg}[\text{CR}, G^H] = \frac{log_2\#H}{2}$ (not plotted), thus giving us some confidence on the other simulation results.

Also, the kind of algorithmic effort needed to lower $\overline{d}_{chain}[\text{CR}, G^H]$ towards $\overline{d}_{avg}[\text{CR}, G^N]$ (a primary goal of the shortcut algorithms – see Section 3.2) may be judged by direct comparison of the respective plots in Figure 3.

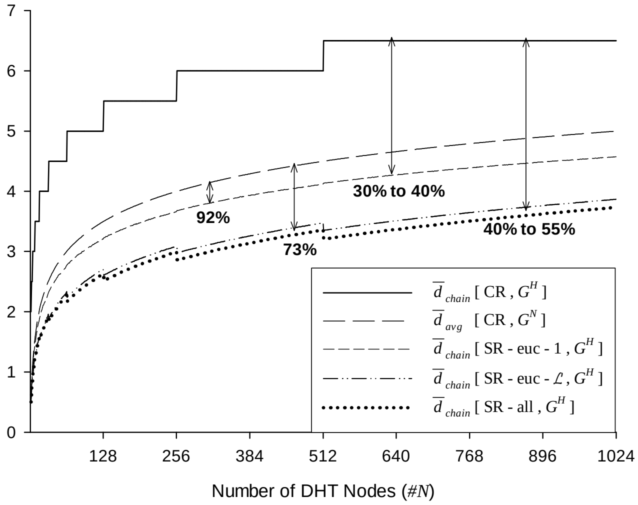

Figure 3. Average Vertex Distances.

The main results of the simulation are, however, conveyed by the plots related to the average distances $\overline{d}_{chain}[\text{SR-euc-1}, G^H]$ and $\overline{d}_{chain}[\text{SR-euc-}\mathcal{L}, G^H]$, attained using the Euclidean Shortcut Routing algorithms. As may be observed, not only such algorithms allowed lower values than $\overline{d}_{avg}[\text{CR}, G^N]$ but also, in the specific case of the algorithm SR-euc-$\mathcal{L}$, the average distances measured are very close to $\overline{d}_{chain}[\text{SR-all}, G^H]$, the optimal lower bound ensured by the Exhaustive Shortcut Routing algorithm; this means that with as little as $\mathcal{L} = log_2\#H$ searches in the local routing tree of a DHT node, the routing decision is almost as effective as that taken by traversing the entire tree.

However, an important feature of the SR-all algorithm in G^H is that it is able to ensure that no DHT node is visited twice (or more) along a routing chain, a desirable feature originally exhibited by the CR algorithm in G^N (and stated as a complementary goal of our algorithms – see Section 3.2). Thus, the distance $\overline{d}_{chain}[\text{SR-all}, G^H]$ is purely an *external distance*, that accounts only for *external hops*, between different routing tables in different nodes. In turn, SR-euc-1 and SR-euc-$\mathcal{L}$ cannot avoid some *internal hops*, made between different routing tables in the same node; however these internal hops are always consecutive (with no network access in-between), and its number tends to be rather small.

6.2. CPU Time per Routing Hop ($\overline{CPU}_{hop}$)

Another important and complementary perspective of the routing algorithms is given by their average CPU time per routing decision or hop, $\overline{CPU}_{hop}$, shown in Figure 4.

The values plotted in Figure 4 are in micro-seconds (μs) and were measured in a Pentium 4 CPU, running at 3GHz.

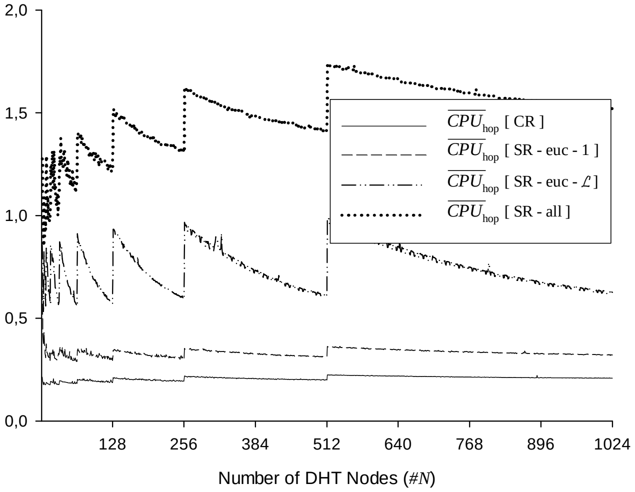

Figure 4. Average CPU Times per Hop (μs).

As expected, the ranking of the algorithms by the $\overline{CPU}_{hop}$ time is inverse to the ranking by the $\overline{d}_{chain}$ distance: in other words, the algorithms that ensure shorter routing chains are also those that strive more to achieve so. However, the merits of the algorithms differ in the two rankings: by consuming $\approx 50\%$ of the time of SR-all, SR-euc-$\mathcal{L}$ still manages to achieve similar distances; in turn, with a penalty of only $\approx$ 20% more in the distances, SR-euc-1 is able to perform in $\approx 45\%$ of the time of SR-euc-$\mathcal{L}$; finally, an increase of $\approx 60\%$ in the execution time, allows SR-euc-1 to have distances of 30% to 40% of those ensure by the CR algorithm. These observations show a clear need for a synthetic metric, that combines the network and computational efforts specific to each algorithm, and allows to select the best algorithm for a certain application scenario.

Before introducing the final metric, we give an explanation for the saw-shaped pattern of the $\overline{CPU}_{hop}$ curves: right after the number of DHT nodes, $\#N$, increases past a power of 2 boundary, the partitioning model M2 dictates the doubling of the hash space and, as a consequence, of the average number of hashes (and thus of routing tables) per DHT node; the depth of the routing tree of each node thus increases, and so does the average search effort; however, as the number of DHT nodes increases towards the next power of 2, the average number of hashes (and thus routing tables) per node will decrease, leading to faster tree searches.

6.3. Total Time per Chain ($\overline{TOTAL}_{chain}$)

A final and synthetic metric is the total average time per routing chain (or, equivalently, per hash lookup), given by

$$\overline{TOTAL}_{chain} \approx \overline{CPU}_{chain} + \overline{NET}_{chain} \qquad (4)$$

, where $\overline{CPU}_{chain}$ is the average CPU time per chain, and $\overline{NET}_{chain}$ is the average Network time per chain.

$\overline{CPU}_{chain}$ depends on the two metrics already studied:

$$\overline{CPU}_{chain} \approx \overline{d}_{chain} \times \overline{CPU}_{hop} \qquad (5)$$

In turn, $\overline{NET}_{chain}$, is given by the following formula:

$$\overline{NET}_{chain} \approx \overline{d}_{ext} \times \overline{NET}_{hop} \qquad (6)$$

Thus, $\overline{NET}_{chain}$ depends on the *external* component of the distance $\overline{d}_{chain}$, denoted by $\overline{d}_{ext}$, and also on the average Network time per *external* hop, denoted by $\overline{NET}_{hop}$. As previously established in Section 6.1, the *external distance* accounts the average number of *external hops* (*i.e.*, between different DHT nodes), along a routing chain. $\overline{NET}_{hop}$ is a parameter (and not a measure), used to capture the effect of different kinds of network technologies. In this regard, Figure 5.a) and Figure 5.b) plot $\overline{TOTAL}_{chain}$, for $\overline{NET}_{hop} = 1\mu$s and $\overline{NET}_{hop} = 10\mu$s, respectively.

Figure 5.a) is representative of a scenario where very high speed interconnects would be used, like 10Gbps Myrinet. The minimization of the network lookup hops is thus less important, once it may be counterbalanced by quickly dispatching lookup requests; this is why the CR algorithm outperforms the SR-all algorithm; still, algorithms SR-euc-$\mathcal{L}$ and SR-euc-1 manage to be more competitive than CR, despite their additional computational effort; this is because such effort is relatively small (specially for SR-euc-1), as shown in Figure 4, and the routing chains are clearly shorter (mainly for SR-euc-$\mathcal{L}$), as Figure 3 showed.

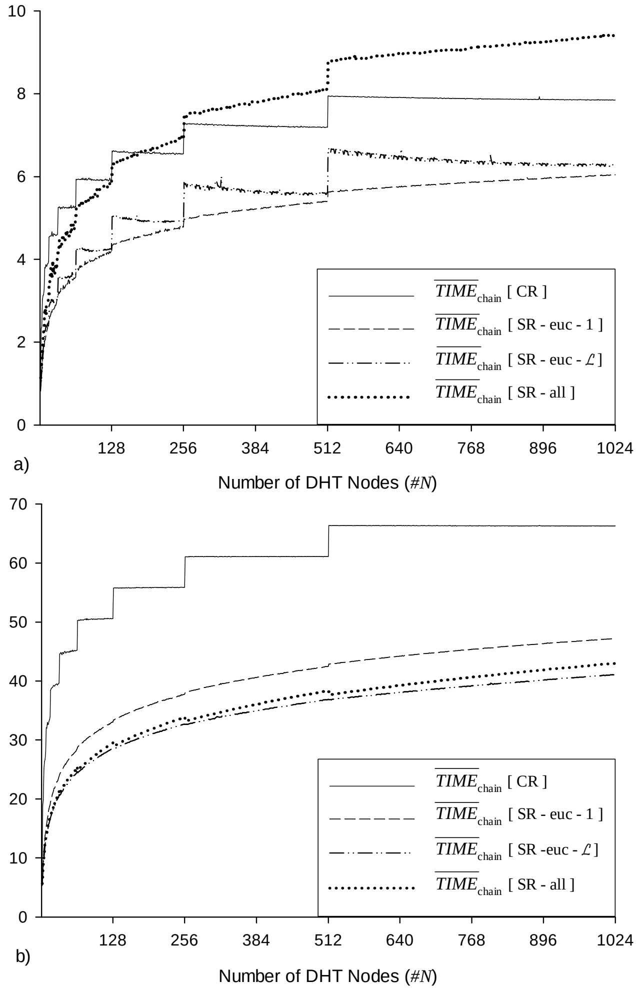

Figure 5. Total Time per Chain (in μs) for a) $\overline{NET}_{hop} = 1\mu$s, and for b) $\overline{NET}_{hop} = 10\mu$s.

With $\overline{NET}_{hop} = 10\mu$s, Figure 5.b) stands for a scenario of moderately high speed connections, like 1Gbps Ethernet. Now, the dominant time is the Network time ($\overline{NET}_{chain}$); thus, all SR algorithms are more attractive than CR, and the relative separation between the two groups is wider; also, the increase in $\overline{NET}_{hop}$ was enough to make SR-all really competitive, though still outperformed by SR-euc-$\mathcal{L}$.

Further tenfold increases of $\overline{NET}_{hop}$ (not charted) also induce a tenfold increase of $\overline{TOTAL}_{chain}$ for all algorithms, though with a minor modification in its ranking: starting from $\overline{NET}_{hop} = 100\mu$s (typical of 100Mpbs Ethernet), SR-all and SR-euc-$\mathcal{L}$ exchange positions, once Network time becomes, effectively, the unique relevant factor.

These results show that our shortcut routing algorithms are useful in a wide range of network scenarios where DHTs may be deployed, from Cluster LANs to P2P WANs.

7. Discussion

Often, partitioning and lookup are tightly integrated, like in Chord [18], but in other approaches, like Dipsea [11] and ours, they are decoupled, for increased flexibility.

In Chord [18], a Consistent Hashing [9] approach bounds each DHT node to a contiguous subset of the hash space; such subset is derived and constrained by the node identifier (or, more precisely, by the hash of the identifier). In other approaches, like P-Grid [1] and ours, there isn't a fixed bound between a DHT node and its hash subset.

Shortcut routing in Chord graphs was first used in the CFS [4] distributed file system. In CFS, each physical DHT node appears as a collection of virtual nodes, each with a routing table in the underlying Chord graph. However, details about CFS's shortcut algorithms are scarce to none.

Godfrey et al. [7] also reference CFS, asserting that routing shortcuts allowed by virtual nodes may ensure, under certain conditions, average distances of $O(log\#N)$; this is in line with our own results (recall Figure 3) that show average distances of even lower order (though we have not used virtual nodes, the comparison is possible, once our model M2 ensured several hashes / routing tables per node).

We only explored the unidirectional variant of Chord graphs, clockwise oriented. With bidirectional graphs, routing tables store twice the topological information, once anticlockwise edges are also considered. This should allow even lower average distances. For instance, Ganesan et al. [6] developed *optimal* algorithms for bidirectional Chord graphs with a power of 2 vertexes (thus in the hash space) allowing average distances as low as $\mathcal{L}/3$, instead of $\mathcal{L}/2$. With unidirectional graphs, but multiple routing tables per DHT node, we have achieved average distances around $\mathcal{L}/3$ for SR-euc-1 and around $\mathcal{L}/4$ for SR-euc-$\mathcal{L}$ and SR-all.

We conclude by restating the main contribution of the paper: a set of algorithms that 1) ensure a high degree of acceleration to distributed lookups on DHTs based on Chord graphs in the hash space, and 2) are applicable to a broad set of network scenarios, ranging from Cluster to Peer-to-Peer environments. In the later case, peers that participates in a Chord-based DHT in the node space, may still use our shortcut routing algorithms with minor modifications, provided that peers run a protocol to exchange topological information (routing tables content), with a minimum set of neighbors in the Chord graph. A platform that demonstrates the use of these and other previous contributions may be found at `http://www.ipb.pt/~rufino/domus`.

References

[1] K. Aberer, A. Datta, and M. Hauswirth. Multifaceted Simultaneous Load Balancing in DHT-based P2P systems: A new game with old balls and bins. *Self-* Properties in Complex Information Systems*, Springer LNCS 3460:373–391, 2005.

[2] H. Balakrishnan, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43–48, 2003.

[3] A. Czumaj, C. Riley, and C. Scheideler. Perfectly Balanced Allocation. In *Proceedings of the 7th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '03)*, 2003.

[4] F. Dabek, M. Kaashoek, D. Karger, and R. Morris. Wide-area Cooperative Storage with CFS. In *Procs. of the 18th ACM Symposium on OS Principles (SOSP '01)*, 2001.

[5] R. Enbody and H. Du. Dynamic Hashing Schemes. *ACM Computing Surveys*, 20(20):85–113, 1988.

[6] P. Ganesan and G. Manku. Optimal Routing in Chord. In *Proceedings of the 15th ACM Symposium on Distributed Algorithms (SODA '04)*, pages 169–178, 2004.

[7] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Procs. of the 23rd Annual Joint Conf. of the IEEE Computer and Commun. Societies (INFOCOM '04)*, 2004.

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

[9] D. Karger, E. Lehman, F. Leighton, D. Levine, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[10] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routig Distances and FaultResilience. In *Proceedings of the 2003 ACM SIGCOMM*, 2003.

[11] G. Manku. *Dipsea: A Modular Distributed Hash Table*. PhD thesis, Stanford University, 2004.

[12] M. Raab and A. Steger. Balls into Bins — A Simple and Tight Analysis. In *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '98)*, pages 159–170, 1998.

[13] J. Rufino. *Co-Operation of Distributed Hash Tables in Heterogeneous Clusters*. PhD thesis, Department of Informatics, Engineering School, University of Minho, 2008.

[14] J. Rufino, A. Pina, A. Alves, and J. Exposto. Toward a Dynamically Balanced Cluster oriented DHT. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN '04)*, 2004.

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

[16] J. Rufino, A. Pina, A. Alves, and J. Exposto. pDomus: a Prototype for Cluster-oriented Distributed Hash Tables. In *Procs. of the 15th Euromicro Inter. Conf. on Parallel, Distributed and Network-based Processing (PDP '07)*, 2007.

[17] R. Steinmetz and K. Wehrle, editors. *Peer-to-Peer Systems and Applications*. Number 3485 in LNCS. Springer, 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 2001 ACM SIGCOMM*, pages 149–160, 2001.