Enhanced Routing in the BiCH cluster oriented DHT

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

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

Abstract

We present several enhancements to the basic routing mechanism of BiCH, a cluster oriented Distributed Hash Table, where partitions are organized into a binary *partition tree* resulting from a Binary Consistent Hashing approach to the distribution and location of partitions. We first introduce routing using all the *routing tables* of a BiCH node, with an $O(log^2 N)$ bound (with respect to the total number of partitions, N) for the routing state inspected, per node. Secondly, we show how to reduce that bound to $O(log(logN))$, by summarizing the routing tables in a *joint resume*. Thirdly, we present a *routing cache* with the same bounds of a resume, but allowing to explore eventual lookup patterns. The combination of these new mechanisms results in a new routing algorithm, where the routing state inspected per node is bounded by $O(log(logN))$ and the average number of routing hops to solve a lookup request, although still bounded by $O(logN)$, is now expected to be smaller. Finally, we show that BiCH supports a dual mode of operation, where transition from using *total* to *partial* routing information is automatic and adapts to the cluster size.

Keywords: Distributed Hash Tables, Binary Consistent Hashing, Hash Routing

1 Introduction

In this paper we present a set of techniques to enhance the basic routing algorithm of the BiCH cluster oriented Distributed Hash Table (DHT) [1], designed to support the distribution and location of the huge volumes of data managed in the context of SIRe [2], an ongoing research project for WWW information retrieval and cataloging.

BiCH is based on the Binary Consistent Hashing (BCH) approach to the distribution and location of partitions, primarily rooted in the Consistent Hashing classic model [3, 4]. However, BiCH differentiated from its origins on a fundamental issue: each partition of the distributed hash table is obtained by a *binary split* operation, that halves an ancestor partition, thus resulting in partitions that only may exist in power of two sizes. The continuous process of halving partitions creates a genealogical hierarchy of partitions – the *partition tree* – which extents to a maximum of $O(logN)$ levels, on a total of N partitions. This hierarchy may be exploited to navigate from any partition to any other in no more than $O(logN)$ hops, with each partition having to know only about $O(logN)$ other partitions in its genealogical vicinity.

The basic operation in a DHT is the lookup of hash indexes, which performs the mapping of an hash index into a node that hosts the partition that contains the index. BiCH lookups follow a pathway through the partition tree, from the lookup requester, to the final target, with a routing decision being taken at each partition visited.

When using the basic routing algorithm, a single routing table is inspected, at each visited partition, in order to choose the best next–hop. This basic algorithm has a fundamental problem: it may generate routing hot-spots resulting from the overloading of the nodes that bridge lookup requests from one large subtree of the partition tree to the other side, as is always the case of the nodes that host the partitions closer to the root of the partition tree.

To alleviate the well known hot-spot problem and accelerate routing we investigated new techniques and algorithms for balanced and fast routing. We start from the assumption that whenever a computational node hosts several routing tables, each one assigned to specific local partition, by taking all the routing tables together it is possible to cover a larger *topological scope* of the partition tree. In this situation, an adapted algorithm that uses all the local routing tables in conjunction, will deliver a next-hop that will be, on average, closer to the target partition, thus minimizing the need to go up in the partition tree. The local routing state to be searched, per routing decision, increases though, from $O(logN)$ to $O(log^2N)$.

The adapted algorithm may be severely optimized by extracting all non redundant information from the node set of routing tables, while exploiting the balanced nature of routing decisions made possible by the use of multiple routing tables. A routing algorithm based on a *joint resume* of all local routing tables, searches the resume $O(log(logN))$ times in order to find the next-hop.

A routing algorithm based on joint resume is significantly faster than the previous ones. However it always results in *static* routing decisions that doesn't take into account eventual lookup patterns that could be used to accelerate routing. We thus added to BiCH a *dynamic* or *adaptable* routing decision policy based on a *routing cache*, per node. The cache of a node is feed with the dynamically generated information about the target partition of a lookup chain, initiated in any partition of the node. The routing throughput when using the cache depends on its size; in order to to achieve a $O(log(logN))$ bound for the routing state inspected per local routing decision, the cache should use the same number or records as a joint resume. Even in the absence of any locality of references, routing in BiCH will benefit from the cache because of the randomness of its references, which are expected to be generated by hash functions.

All these new routing mechanisms may be integrated by a *combined routing algorithm* where, most of the times, the cache will be able to provide the best next-hop, in competition with the resume; routing tables will still be used for certain specific situations, like recovery procedures. The local routing state inspected by the new combined routing algorithm, in order to take a routing decision, is still bounded by $O(log(logN))$. The maximum number of routing hops needed to solve a lookup request is still of order $O(logN)$, but in average it is expected to be reduced, as each routing decision is now supported by additional and more detailed routing information.

The results of the investigation also lead to the definition of a dual mode of operation for BiCH based DHTs. The first mode, allowing for maximum throughput, is suited to small clusters whose computational nodes have the potential to have in memory *total* routing information. The second mode favors the scalability of the routing mechanism, being appropriate to large clusters whose nodes only maintain in memory *partial* routing information. The transition from a mode to the other one is automatic in the presence of a dynamic set of nodes and partitions, thus resulting in a overall approach to routing in the DHT, that is both adaptive and scalable.

The remaining of the paper is organized as follows: section 2 revises the fundamentals of the Binary Consistent Hashing approach to the distribution and location of partitions, section 3 presents the enhancements to the base routing algorithm, section 4 discusses how the enhancements may result in adaptive and scalable routing decisions, section 5 references some related work and section 6 concludes, also pointing directions for future work.

2 Binary Consistent Hashing

In this section we present the main concepts of the Binary Consistent Hashing (BCH) approach to the distribution and location of partitions of a large Distributed Hashed Table [1].

In what follows we use the term *node* to refer any real or virtual computational node, assigned to a specific partition set of an hash function range.

2.1 Distribution of Partitions

Every node X is mapped into a point $H(X) \in r_H$, where H is an hash function of b_H bits with range $r_H(b_H) = \{h \in \mathbb{N}_0 : 0 \leq h < 2^{b_H}\}$. $H(X)$ defines a partition of r_H to be hosted at node X.

When a DHT is created, a first node is responsible for the entire hash function range. Whenever a new node X joins the DHT and requests a share of r_H, that node receives half – *binary split*– of the partition previously containing $H(X)$. For instance, in case $H(X)$ is already included on a partition owned by node Y, that partition is halved as follows: if $H(X) \leq H(Y)$ then X takes the left half and Y maintains the right half, else the reverse split takes place.

In order to ensure that r_H is sliced uniformly, each computational node may host several partitions, scattered along r_H. A computational node X should host at least $n_X = (c_X/c_{min}).b_H$ partitions, where c_X measures the computational capacities of node X and c_{min} is the computational capacity of the weakest node of the cluster thats going to support the DHT [1]. To map a computational node X onto n_X points of r_H, the hash function is applied to n_X different identifiers – $X_1, \ldots, X_{n_X}$ –, simulating the existence of n_X *virtual nodes* per each computational node X.

Figure 1 shows a sequence of binary splits in the range $r_H(10) = \{0, 1, \ldots, 1023\}$, after which node A and B both host 3 partitions and node C hosts 2 partitions. Because there is a direct relationship between a partition and a (virtual) node, we assume as a convention that partitions are named after them. The *split level* of a partition in each stage is shown in superscript, as an indication of its relative size with regard to r_H: a partition in split level s has $1/2^s$ the size of r_H.

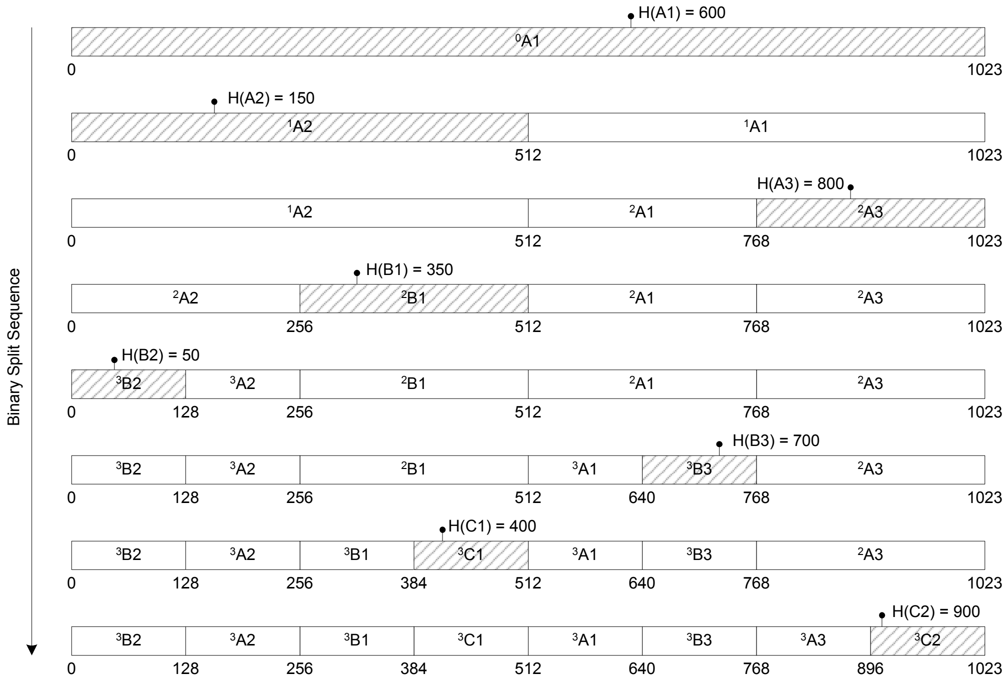

Figure 1: A sequence of binary splits in $r_H(10)$.

The *partition tree* of Figure 2 provides another view of the binary split process that shows genealogical relationships between partitions. However, it does not necessarily reflects the full order of the binary splitting sequence; for instance, by merely observing Figure 2, it is not possible to deduce if partition $^3C1$ was created before or after $^3C2$.

In the partition tree, only the (shadowed) leaves represent partitions which currently exist. The leaves have a maximum split level of b_H, once no more than b_H binary splits may be applied to any partition; in fact, only the first/root partition – r_H – supports b_H consecutive binary splits.

2.2 Location of Partitions

The location of partitions is the process by which "hash index $\hookrightarrow$ partition $\hookrightarrow$ node" mappings are performed. The location is done trough a scalable distributed lookup service running on node of

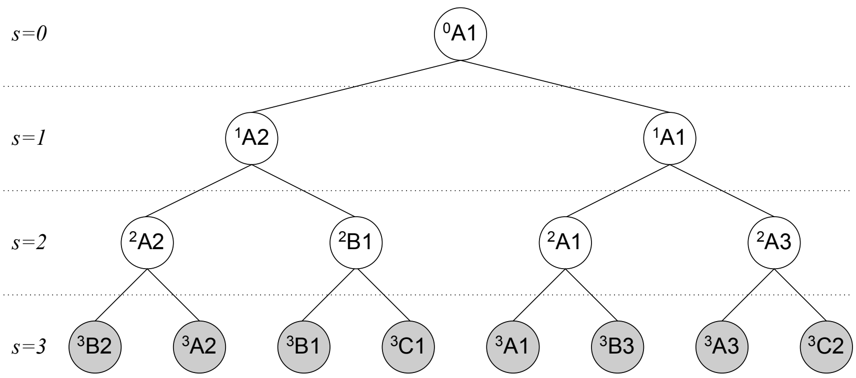

Figure 2: Partition Tree for the binary split sequence of Figure 1.

the DHT. The scalability of the location process is based on the existence of logarithmic bounds for: **a)** the amount of routing information preserved per partition and **b)** the maximum number of nodes visited during the location process.

2.2.1 Routing Tables

Each node maintains a *routing table* for each partition it hosts. The routing table $t(X_i)$ of a partition X_i currently in split level s, holds s records about the s binary splits that took place until X_i reached split level s. Once $s \leq b_H$, a routing table needs no more than b_H records. Because the maximum number of partitions is 2^{b_H}, the height of a routing table is of order $O(logN)$, with respect to the total number of partitions, N.

Each record of a routing table describes a binary split by using the following fields: $\langle pid : partid\_t, prange : range\_t, cid : partid\_t, crange : range\_t\rangle$. The pid field identifies a parent (ancestor) partition whose range has been halved into $prange$ and $crange$. The cid field identifies a child (descendent) partition with range $crange$. Both $prange$ and $crange$ are structures with scheme $\langle left : r_H, right : r_H\rangle$, where $left$ and $right$ specify the limiting points of the partition.

There are two indexes, $o(X_i)$ and $a(X_i)$, per routing table $t(X_i)$; they preserve, respectively, the *original* and *actual* split level of partition X_i. Records whose index is less or equal to $o(X_i)$ are inherited from the ancestor of X_i; the other records describe further binary splits of X_i. Figure 3 shows the routing tables (and their indexes) after the final stage of Figure 1; in each routing table, the shadowed records are relative to binary splits that occurred after the partition was created.

A1 *o(A1)=0* *a(A1)=3*
t(A1)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A1	512,767	^{2}A3	768,1023
3	^{3}A1	512,639	^{3}B3	640,767

A2 *o(A2)=1* *a(A2)=3*
t(A2)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A2	0,255	^{2}B1	256,511
3	^{3}A2	128,255	^{3}B2	0,127

A3 *o(A3)=2* *a(A3)=3*
t(A3)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A1	512,767	^{2}A3	768,1023
3	^{3}A3	768,895	^{3}C2	896,1023

B1 *o(B1)=2* *a(B1)=3*
t(B1)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A2	0,255	^{2}B1	256,511
3	^{3}B1	256,383	^{3}C1	384,511

B2 *o(B2)=3* *a(B2)=3*
t(B2)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A2	0,255	^{2}B1	256,511
3	^{3}A2	128,255	^{3}B2	0,127

B3 *o(B3)=3* *a(B3)=3*
t(B3)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A1	512,767	^{2}A3	768,1023
3	^{3}A1	512,639	^{3}B3	640,767

C1 *o(C1)=3* *a(C1)=3*
t(C1)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A2	0,255	^{2}B1	256,511
3	^{3}B1	256,383	^{3}C1	384,511

C2 *o(C2)=3* *a(C2)=3*
t(C2)=

1	^{1}A1	512,1023	^{1}A2	0,511
2	^{2}A1	512,767	^{2}A3	768,1023
3	^{3}A3	768,895	^{3}C2	896,1023

Figure 3: Routing tables and their indexes after the binary split sequence of Figure 1.

2.2.2 Base Lookup Algorithm

The distributed lookup service handles lookup requests to discover the partition for a specific hash index. A lookup request is initiated by scanning the routing table of a starting partition to determine if that partition contains the hash index or if it is necessary to continue the lookup taking the pathway to the routing table of another partition.

Figure 4 shows the basic lookup algorithm as a procedure *handle_lookup_request* with three parameters: $requester$ (the starting partition), h (the hash index to map into a partition) and $target$ (the current target partition of the lookup request).

The base lookup algorithm visits no more than $O(logN)$ partitions, independently of the point of depart. This logarithmic bound comes from the fact that routing tables have also logarithmic height and each record maintains information about both an ancestor and a descendant. The worst

```
handle_lookup_request (requester : reqid_t, h : r_H, target : partid_t)
{
    i : int; newtarget : partid_t;

    for i = a(target) to 1 do
        if h ∈ t(target)[i].crange then newtarget = t(target)[i].cid; break endif
        if h ∈ t(target)[i].prange then newtarget = t(target)[i].pid; break endif
    endfor

    if newtarget == target then send(LOOKUP_ACK, requester, target, h)
    else send(LOOKUP_REQ, requester, newtarget, h) endif
}
```

Figure 4: Base algorithm to process a lookup request.

case for the lookup algorithm is bounded by b_H, for the unusual situation of when DHT has at least one partition with the maximum split level.

If a next–hop is unreachable the algorithm calls a recovery procedure that involves $O(logN)$ additional steps [1]. The algorithm assumes a reliable execution environment, where a partition is unreachable only in the case it has been previously moved to a different node.

3 Balanced and Fast Routing

In BiCH, if we take the basic lookup algorithm of Figure 4, the nodes hosting the partitions positioned near the root of the partition tree may become into routing hot-spots, thus compromising the scalability of the overall lookup service. To deal with this problem, routing decisions are fed up with additional information, allowing for the discovery of *shortcuts* to the final destination and reducing the number of accesses to partitions on top of the partition tree.

3.1 Routing with Multiple Tables

A typical computational node will host several partitions, with a specific routing table per partition. In this context, a routing decision, based on a single routing table (as done by the algorithm of Figure 4), may be significantly enhanced by using routing information gathered from the routing tables of all the partitions hosted at the same node. The reasons for the expected improvement relay on the fact that partitions hosted at a node are scattered across the hash function range (meaning they are disperse at the leaf level of the partition tree). This way, the aggregated routing information of all routing tables of a node covers a broader *topological scope* than the individual routing table of each partition hosted at the node. Figure 5 and Figure 6 show, respectively, the individual topological scope of routing tables $t(C1)$ and $t(C2)$, and their joint topological scope.

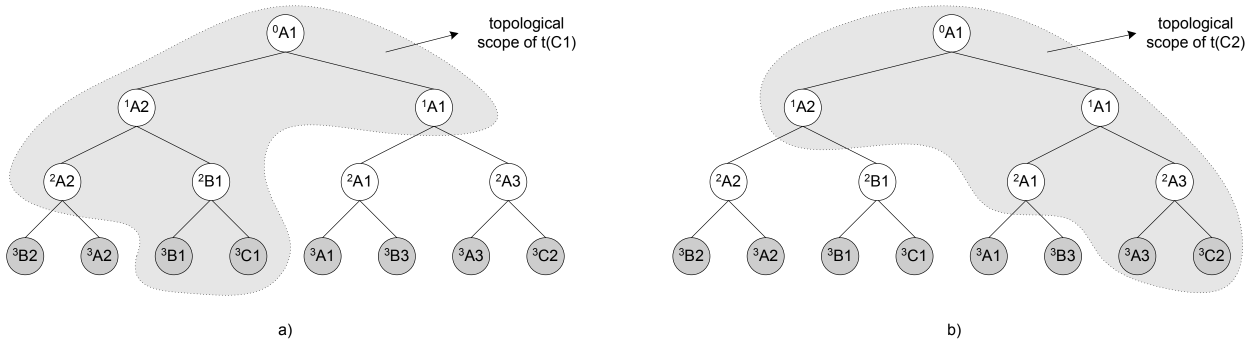

Figure 5: Topological scope of $t(C1)$ and $t(C2)$.

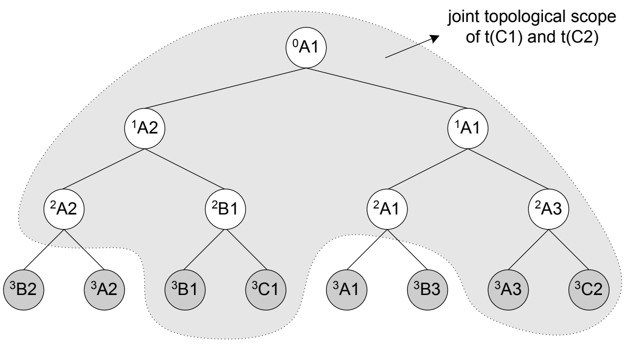

Figure 6: Joint topological scope of $t(C1)$ and $t(C2)$.

Enhanced Routing Algorithm with Multiple Tables The base routing algorithm may be modified to allow the analysis, in simultaneous, of all the individual local routing tables, in order to find the smallest partition that matches the requested mapping. As before, the recovery procedure may still be used whenever a selected partition is found to be unreachable.

Searching all local routing tables may result in an important optimization in cases where the final leaf partition containing the hash index is a local partition, thus avoiding the need to follow other pathways to reach the same point. For instance, if ${}^{3}C2$ is the final target of a lookup starting up at partition ${}^{3}C1$, in Figure 5.a), the base routing algorithm would find ${}^{1}A1$ as the next-hop; however, by using the alternative algorithm, if all routing tables in C were scanned (thus inspecting the joint topological scope shown in Figure 6), ${}^{3}C2$ would immediately be selected as the next–hop.

Storage and Search Cost with Multiple Tables By establishing b_H as the average number of partitions per node [1], the total amount of routing records locally maintained at each node is of order $O(log^2 N)$, with regard to the total number of partitions, N. The worst case for the search cost corresponds to the situation when all those records have to be inspected sequentially, making the local search cost of a node to be of order $O(log^2 N)$. The overall search cost for the enhanced routing lookup algorithm is still $O(logN)$; however, in average, the number of visited partitions is expected to decrease due to the additional information used when taking routing decisions.

3.2 Routing with Resumes

Searching all local tables of a node, when routing, has two drawbacks: **a)** records in different tables may have redundant information, which will happen more often the closer they are to the top of the table (that is, when those records describe binary splits that took place near the top of the partition tree); for instance, with respect to the routing tables of Figure 3, if the previous enhanced algorithm is running at node B to find the next-hop towards $h = 513$, both the 2nd and 1st records of $t(B1)$ and $t(B2)$ need to processed, despite being equal; **b)** scanning all routing tables and eventually inspecting all records, for every routing decision, may be a demanding task, with a negative impact on the routing throughput of the node. As a way to overcome these expected limitations, we propose, in what follows, a new approach based on local routing information.

3.2.1 Individual Resumes

A routing table may be compressed in a *resume*. Conceptually, the resume of a routing table represents its most detailed view of the DHT, without any redundant routing information and still covering all the hash function range.

More formally, a resume is a set of records of scheme $\langle id : partid\_t,\ range : range\_t\rangle$, with half the information of a routing table record, that is, each record of the resume keeps information only about one partition. In a resume, records must comply with the following properties: **a)** ranges for all records are mutually disjoint, meaning they never overlap; **b)** the union of the ranges of all records is r_H; **c)** records are ordered (*e.g.*, by $range.left$), to accelerate access.

Building an Individual Resume We start with an empty resume and by scanning bottom-up the source routing table, we try to find, for every partition T_i referenced there, if the resume already references a partition R_j whose range overlaps the range of T_i. If the partition R_j is not found, a new record is created for T_i on the resume. After the conclusion of the scanning process of the source routing table, the records in the resume will cover the entire original hash function range, without any range overlapping.

The resume of a routing table may also be seen as the set of references to the leaf partitions of its topological scope, as illustrated in Figure 7, where dotted arrows signal the partitions referenced in $t(C1)$ and $t(C2)$ that become referenced in the resumes $r(C1)$ and $r(C2)$.

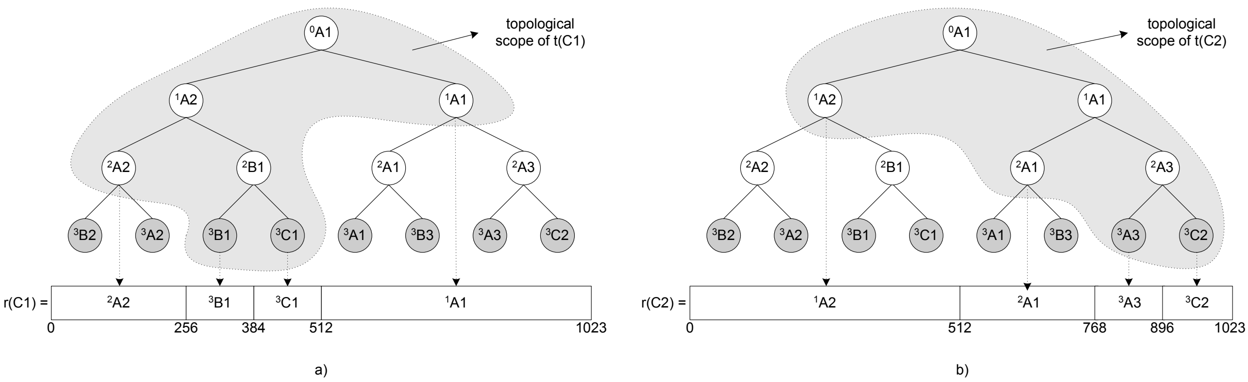

Figure 7: Individual resumes in relation with the topological scopes of $t(C1)$ and $t(C2)$.

Because the records in the resume are constructed based on information contained in its source routing table, any change in the contents of that table has to be propagated to its resume. A routing table changes as a result of structural operations on the DHT, including the creation, the merging or the moving of partitions (see [1] for details).

Enhanced Routing Algorithm with Individual Resumes The routing algorithm may now conduct a simultaneous search, in all individual resumes, for the smallest partition still containing the hash index under lookup. As a resume covers the entire hash function range and records in a resume are mutually disjoint with respect to range, there is only one possible next–hop per resume. Once again, if the selected partition is unreachable, the recovery procedure is used, this time based on the source routing table of the resume where the best next–hop was found.

Storage and Search Cost with Individual Resumes Along with redundancy, sequential scan can also be found as one of the reasons for poor routing throughput. The use of a balanced binary tree to store the resume may contribute to accelerate the lookup process. A balanced binary tree with no more than $2^{l+1} - 1$ nodes has no more than l levels. Since the resume of a routing table of height b_H has, for the worst case, $b_H + 1$ records, a balanced binary tree having a record per tree node, will have no more than l levels if $2^{l+1} - 1 \geq b_H + 1$, which gives $l \geq log_2(b_H + 2) - 1$.

Recalling that b_H is of order $O(logN)$ – with respect to the total number of partitions, N – then l is of order $O(log(logN))$. In addition, as b_H is also the average number of partitions per node [1], then, if routing is based on b_H resumes, it is necessary to inspect $O((logN) \times (log(logN)))$ routing records in the worst case. For the worst case scenario, the sequential scan of b_H routing tables would result on the need to inspect $O(log^2N)$ records, which is considerable worst when compared with the balanced tree approach.

3.2.2 Joint Resume

The advantages of using multiple individual resumes when compared with the use of multiple routing tables are not expected to be significant. In effect, fewer records to search would result in faster routing decisions but the influence of redundancy across resumes may still have a negative impact. An example of such redundancy occurs in the scenario presented in Figure 8, where we may see that the resumes $r(Z1)$ and $r(Z2)$ share a redundant record about the partition $^1X1$.

However, it is also possible to build and use a sole *joint resume* of the routing tables of all parti-

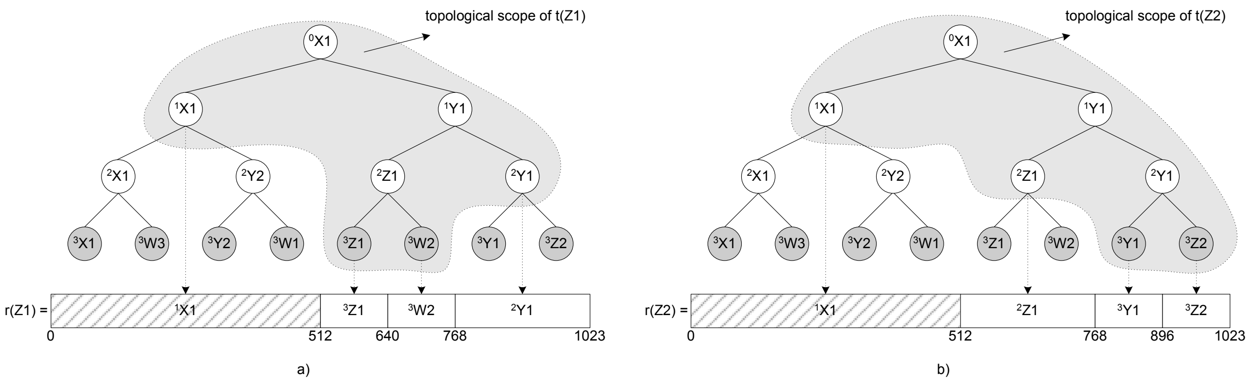

Figure 8: Individual resumes $r(Z1)$ and $r(Z2)$ with a redundant record about ${}^{1}X1$.

tions hosted at a computational node, that does not contain any redundancy. The joint resume then becomes the main routing data structure used for routing, avoiding the need for individual resumes. The source routing tables, though, must always be preserved to assist in recovery procedures.

The records of a joint resume follow the scheme $\langle id : partid\_t,\ range : range\_t,\ source : tableref\_t\rangle$, thus having one additional field, $source$, used to reference the original routing table from where the partition partition $\langle id, range\rangle$ was learned. The $source$ reference is necessary because a partition may be learned from more than one routing table and if that partition is found to be unreachable then its source table will be used by the recovery procedure.

A joint resume is also stored in a binary balanced tree and its records have the same properties of those of individual resumes.

Building a Joint Resume The construction of a joint resume is not as simple as the construction of an individual resume. To build an individual resume, only one routing table is scanned, bottom-up. In this situation, when comparing the range of each partition T_i found in the source routing table and the range of partitions R_j already in the resume, we are faced with only two scenarios: **s1)** the range of T_i is disjoint from the range of R_j (and so T_i enters the resume) or **s2)** the range of T_i is a superset of the range of at least one R_j (and so T_i does not enter the resume).

The building of a joint resume involves scanning several routing tables, possibly with redundant records; this results on having to deal with two more scenarios: **s3)** the range of T_i matches the range of one R_j (and so no update is made to the resume) or **s4)** the range of T_i is a subset of the range of a R_j (and so T_i replaces R_j in the resume).

An example involving the four scenarios is depicted in Figure 9, for the situation where the joint resume $r(Z)$ is built by first processing $t(Z1)$ – see Figure 9.a) – and then $t(Z2)$ – see Figure 9.b). The scenarios found when scanning each routing table, bottom-up, appear at the right of each partition in the partition tree. Dotted arrows signal scenarios where the resume is updated. The reference to the source routing table is at the upper right corner of each partition in the resume.

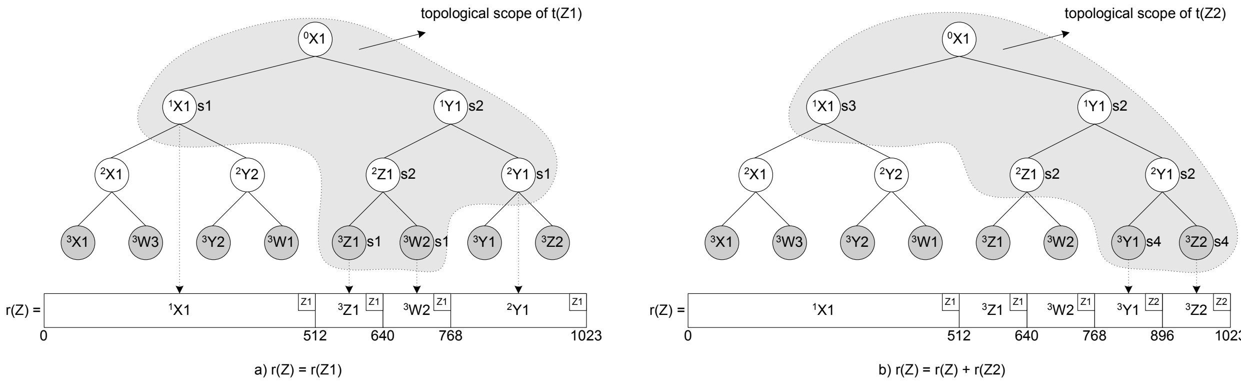

Figure 9: Building the joint resume $r(Z)$ by first processing $t(Z1)$ and then $t(Z2)$.

The order in which routing tables are scanned has no influence in the selection of the partitions that end up in the joint resume. The same is not true for the references to source routing tables. For instance, Figure 10 shows the alternate way of building the resume $r(Z)$, by first processing $t(Z2)$ and then $t(Z1)$; by comparing Figure 9 and Figure 10 it can be observed that the source routing tables for the partition ${}^1X1$ are different.

Using a source routing table instead of another means that the recovery procedure for unreachable partitions will follow a different pathway in the partition tree. We have no yet established any criteria to choose between alternate paths. However, at a first glance, it seems that no special benefits may be obtained when choosing one path in alternative to the other.

The same way an individual resume relates to an individual topological scope, a joint resume also relates to a joint topological scope: a joint resume is the set of references to leaf partitions of a joint topological scope. Figures 11.a) and 11.b) demonstrate this assertion for the joint resumes $r(C)$ and $r(Z)$, respectively; the dotted arrows point to the partitions referenced in the source

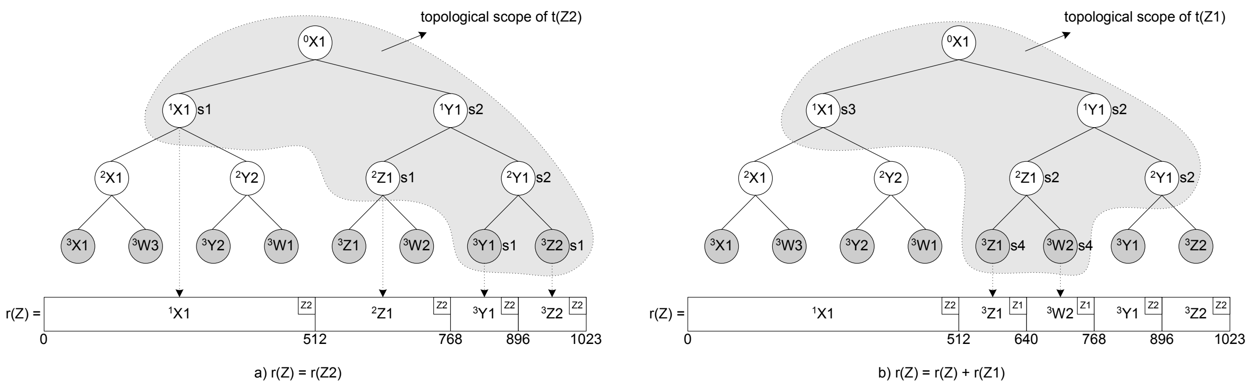

Figure 10: Building the joint resume $r(Z)$ by first processing $t(Z2)$ and then $t(Z1)$.

routing tables that become referenced in the resumes.

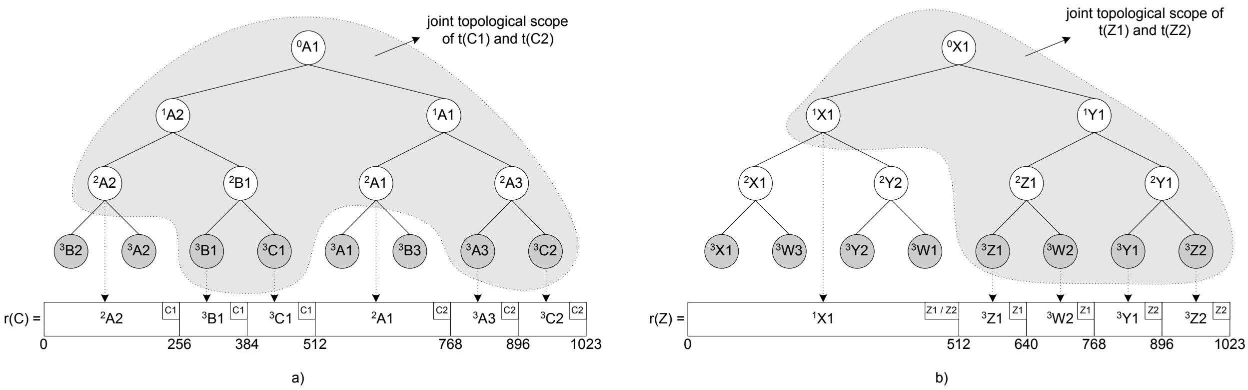

Figure 11: Joint resumes in relation with the joint topological scopes of routing tables.

Finally, a change in any source routing table needs to be propagated to the joint resume, thus triggering its reconstruction.

Enhanced Routing Algorithm with a Joint Resume In each node, the routing algorithm now scans the joint resume, for the partition that contains the hash index under lookup. Since the joint resume covers the entire hash function range and the partitions in the resume are mutually disjoint, the outcome of the search is a single next–hop. If the next–hop is unreachable, the usual recovery procedure will be used, this time based on the source routing table of the next–hop.

Storage and Search Cost with a Joint Resume A routing table participates with no more than $b_H + 1$ records to built the resume. If we choose to set b_H as the maximum number of routing tables per computational node, the joint resume will store a maximum of $b_H.(b_H+1)$ records. This number is a majorant, because some records, across routing tables, will either reference partitions already in the resume or partitions for which there are, already, sub-partitions in the resume. Thus, a balanced binary tree storing those records will have no more than l levels if $2^{l+1}-1 \geq b_H.(b_H+1)$, meaning that $l \geq log_2(b_H.(b_H+1)+1)-1 \approx log_2(b_H^2+1)-1 \approx 2.log_2 b_H - 1$, for typical values of b_H (*e.g.*, 32, 64, 128, 160 bits).

By noticing that b_H is of order $O(logN)$ for a total of N partitions, then l is $O(log(logN))$. Therefore, in the worst case, routing based on a joint resumes needs to inspect $O(log(logN))$ records, definitely a substantial improvement over the worst case scenario of the sequential scan of all local routing tables, which needs to inspects $O(log^2 N)$ records.

3.3 Routing with Caches

Routing decisions based on routing tables or resumes are *static*: for the same hash index under lookup, the next–hop will always be the same. There is one exception, though: a structural operation, like the creation, migration or merging of partitions, triggers the update of routing tables and, as a side effect, of its resumes.

To deal with situations where the same pattern of lookups may be frequently generated, we need to complement routing tables and resumes with other mechanism that benefits from the principles of spacial and time locality, as those used by contemporaneous caching mechanisms, whether software or hardware based. A *routing cache* mechanism may thus be devised in such a way that whenever a lookup requester references, for a certain amount of time, the same set of hash indexes – *locality of references* –, the cache will immediately respond with the matching partition, thus offering a *topological shortcut*.

A DHT is a distributed dictionary where two keys are involved to retrieve a piece of data: an *external key*, supplied by the client of the DHT, and an *internal key*, which is the hash index generated from the external key. The hash index is then used internally by the DHT to locate the partition where data related to the external key is maintained. By definition, a good hash function destroys any locality because it maps external keys to hash indexes randomly, which could have as a consequence the limited usefulness of the routing cache. However, the routing cache always provides a view of the partition set that is finer (although incomplete) than that of a resume. This property is fundamental to allow the routing cache to provide next-hops that are, most of the times, closer to the final target than the ones obtained by exclusively routing with resumes.

The cache stores information about partitions found at the end of a lookup chain. Cache records follow the basic scheme $\langle id : partid\_t,\ range : range\_t \rangle$, with information about the final target of a successful lookup request. Like a resume, the cache is also implemented as a balanced binary tree of records, ordered by $range.left$ and without overlapping ranges.

The entities that will support the distributed lookup service will share a single cache per node. The cache stores information about partitions already found at the end of a lookup chain. Its records have the basic scheme $\langle id : partid\_t,\ range : range\_t \rangle$, to register information about the final target of a successful lookup request. The cache is also implemented as a balanced binary tree of records ordered by $range.left$ without overlapping ranges.

Building the Cache The cache starts as a copy of the local joint resume, which has the most detailed structural information locally known. Whether a local partition is used as an entry point for an *external lookup request* (where the requester is external to the DHT), or as a starting point for an *internal lookup request*, the cache will be updated with information about the final target (partition) of the lookup chain.

The procedure to update the cache is similar to the one used to build a joint resume, but must handle specific issues: **a)** the cache stores only a limited number of records, which imposes the selection of an eviction-policy (LRU or LFU, for instance); **b)** the cache may have outdated references to partitions that should be lazily updated.

A cache may contain obsolete references to partitions resulting from single or combined reasons: **b1)** a referenced partition may not exist anymore because it has merged with other(s); the cache will eventually learn that a new partition exists whose range is a superset of the range of one or more partitions already referenced in the cache; the records of these partitions will then be replaced by a record for the new partition; **b2)** the identification of a partition may have changed because it has moved to another node; the cache will eventually learn that a new partition exists having the same range but a different identification than a partition already referenced in the cache; because the range of both partitions is the same, the old record is simply replaced by the new one.

A cache progressively derives from its source resume because it will collect records about partitions outside the joint topological scope; this, in addition to the fact that the cache has a limited capacity, means that records in cache may no longer cover the entire hash function range. Thus, in contrast to its source resume, the cache view of the hash function range may be populated with *topological holes*. Figure 12.a) illustrates this situation, showing how the cache $c(Z)$ of node Z may diverge from the resume $r(Z)$: if ${}^3Z1$ is the starting point of a lookup request (REQ) chain that ends in ${}^3Y2$, the node Z will receive an acknowledge (ACK) from node Y, containing information about ${}^3Y2$; because ${}^3Y2$ is a sub-partition of ${}^1X1$, the cache is updated from $c_0(Z)$ to $c_1(Z)$, thus creating 2 *topological holes*.

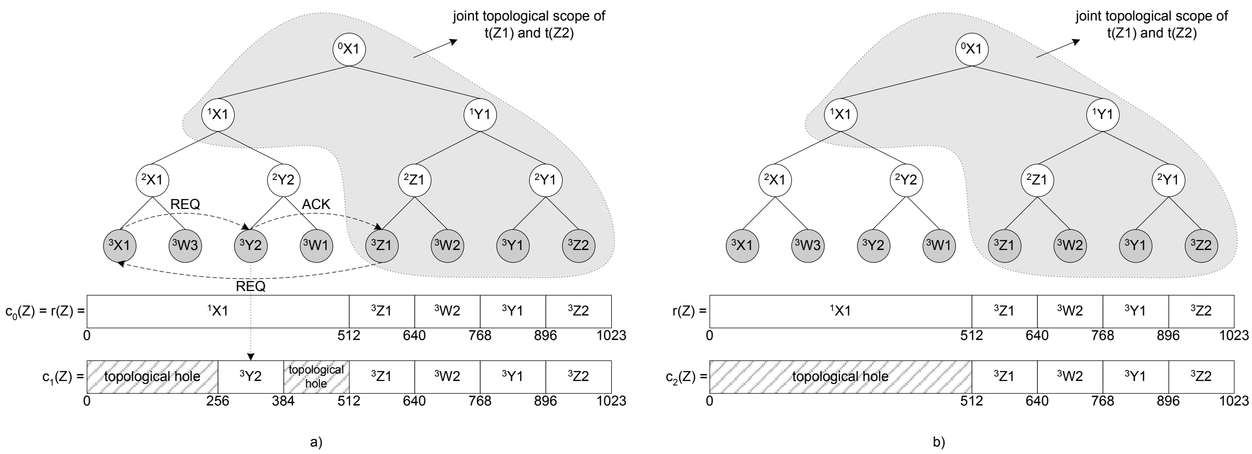

Figure 12: Topological holes in the cache $c_1(Z)$ and $c_2(Z)$.

Enhanced Routing Algorithm with the Cache A routing algorithm exclusively based on the cache will try to find out if there is a record for a partition whose range contains the hash index under lookup, in which case that partition is the next-hop. Because of the possible existence of topological holes in the cache view, the search may not find such record. In such case, routing towards the hash index is still possible via a partition referenced in cache. This partitions is the closest partition to the hash index, in topological terms, that is, in the number of hops in the partition tree. For using this heuristic, we need a way to compare the *topological closeness* between a set of (disjoint) partitions and an hash index outside their ranges.

Recalling that every partition, except for the root, results from a binary split, by stepping back in the binary splitting sequence that originated the partition, the range of the ascendant partition increases, as the split level decreases. Thus, given a set of disjoint partitions, the closest to an hash index h will be the one(s) for which we need to step back fewer times in order for the range of the ascendant to contain h. The range $\{l', l'+1, \ldots, r'\}$ of the immediate ascendant of a partition with range $\{l, l+1, \ldots, r\}$ is easily determinable: **if** $(l \; mod \; 2(r-l+1) == 0)$ **then** $\{l' = l; r' = r + (r-l+1)\}$ **else** $\{l' = l - (r-l+1); r' = r\}$ **endif**.

In reality, the comparison of the topological closeness between all the partitions referenced in the cache and the target hash index is proved to be unnecessary, because the best choice will come out of just two candidates: the partition that precedes and the partition that succeeds (in a circular way) the topological hole where the hash index falls. For instance, in Figure 12.a), only partitions $^3Y2$ and $^3Z1$ are eligible when searching the cache $c_1(Z)$ for the best next–hop towards $h = 400$; using the method above we conclude that $^3Y2$ is the best next–hop. In the same scenario, only partitions $^3Y2$ or $^3Z2$ are plausible candidates for the best next–hop towards $h = 100$; using the same logic as above, $^3Y2$ should be chosen.

If the next-hop is unreachable, the recovery procedure must deal with the fact that the cache records do not reference their source routing tables. A recovery procedure for the case when cache based routing is used in conjunction with resume based routing is presented in section 3.4.

Storage and Search Cost of the Cache The storage and search cost of the cache depends on its maximum number of records, n. Once the cache will be supported by a balanced binary tree that starts as a copy of a joint resume, n should be at least $b_H(b_H + 1)$. If $n \approx b_H(b_H + 1)$, the search cost will be, in the worst case, of order $O(log(logN))$, the same search cost of the joint resume.

3.4 Combined Routing

In the previous sections we have discussed several mechanisms to accelerate routing in the BiCH DHT. The order in which they were presented reflects a constructive process, after which a new mechanism is layered on top of another already established: routing based on a sole routing table was expanded to routing with multiple tables, a joint resume synthesizes multiple routing tables, a cache is initially based on a joint resume. We have also noticed that each new mechanism, once established, may be used almost independently of the previous ones, aside from the situation were the next-hop is unreachable and the recovery procedure falls back to the routing tables.

We now present a way in which all routing mechanisms available for routing may be explored in mutual cooperation. In particular, this solves the pending issue of dealing with an unreachable next-hop given by the cache. The procedure for *combined routing* is as follows:

1. let R_i be the best next-hop based on the local joint resume;
2. let C_j be the best next-hop based on the local cache;
3. if the range of C_j matches or is a subset of the range of R_i, then let C_j be the next-hop; if C_j is unreachable, remove its record from the cache and repeat step 2;
4. else if the range of C_j is disjoint from the range of R_i, then let R_i be the next-hop; if R_i is unreachable, use the recovery procedure with the source routing table of R_i.

The procedure puts the joint resume and the cache in competition to provide for the best next-hop (steps 1 and 2). Because the cache has a more fine-grained view of the partition set, it will supply the best next-hop most of the times (step 3). If none sub-partition C_j of R_i is reachable, the cache will only be able to provide a next-hop disjoint (never a superset of) from R_i; in such circumstances, R_i becomes the best choice (step 4).

In order to define the search cost of the combined routing procedure we make two observations: **1)** the joint resume is inspected only once and the cache may be inspected several times (although seldom); **2)** both the joint resume and the cache require no more than $O(log(logN))$ searches, per inspection. Thus, the bound for the overall number of searches still is of order $O(log(logN))$.

Figures 12 provides a representative scenario to apply the former procedure. Assuming that $c_1(Z)$ is the actual cache of node Z, it clearly provides ${}^3Y2$ as the best overall next–hop towards the hash index $h = 400$ – see Figure 12.a); in the event that ${}^3Y2$ is unreachable, its record is removed from the cache and the cache is updated to $c_2(Z)$ – see Figure 12.b); ${}^3Z1$ (or equally ${}^3Z2$) then becomes the best next-hop supplied by the cache, but because its disjoint from the best next-hop of the resume, ${}^1X1$, this one is elected.

Finally, we present a situation where the combined procedure will not elect the best overall next-hop. Again referring to Figure 12.a), when node Z routes towards an hash index $h \in \{0, 1, \ldots, 255\}$, it will choose ${}^3Y2$ as the best overall next-hop; we see, however, that ${}^1X1$ would have been the right choice; this anomaly emerges because $r(Z)$ has no information about the real split level of $X1$; nevertheless, the anomaly is transient: depending on the value of h, the cache will gain a record about ${}^3X1$ or ${}^3W3$, which will accelerate further accesses.

4 Adaptive and Scalable Routing

We finish the presentation of the enhanced routing mechanisms in the BiCH cluster oriented DHT by explaining how the routing cache allows for a dual mode of operation with regard to the amount of routing information maintained by each computational node.

In section 3.3 we have established the maximum number of records of a routing cache to be $b_H(b_H + 1)$. Taking into account that b_H is the standard number of partitions per node, then the cache supports complete information about the partition set of a total of $b_H + 1$ nodes. For instance, for a typical value of $b_H = 32$, the number of nodes must surpass $b_H + 1 = 33$ in order for the cache to become exhausted.

In small clusters, the routing cache in each node will end up, rapidly, to contain information about all partitions of the DHT. In such scenario, any lookup chain has only one hop, independently of the node where the location begins. However, even in this situation, the cache will always be lazily updated, following possible changes of the partition set. This means that, although the cache maintains *total* routing information, such information is not *static*.

The convergence of caches about the partition set of the DHT may further be accelerated if nodes exchange routing information between them. For instance, whenever its joint resume changes, a node could send it to all nodes that can be learned from that resume. These nodes make the *topological neighborhood* of the node (*e.g.*, $\{A, B\}$ is the topological neighborhood of node C with respect to the resume $r(C)$ in Figure 11.a)). The routing information a node receives this way, may be made part of its cache, by using the normal cache update procedure.

When more nodes join the DHT, the number of partitions increases, meaning that caches may no longer be capable of holding total routing information. With *partial* routing information, the complementary routing mechanisms based on resumes and routing tables are also used, following the *combined routing* algorithm presented in the previous section.

The cache thus makes possible a dual mode of operation: one with *total* routing information, adequate for a small number of nodes and allowing for the best performance; the other with *partial* routing information, appropriate when the DHT is supported by many nodes and scalable routing is essential to keep performance acceptable. Changing between both modes is automatic, making the routing mechanism *adaptive*.

5 Related Work

There has been numerous research addressing the field of DHTs. In this section, we make only reference to a few representative examples and restrict the focus to routing issues.

A first generation of DHTs for NOWs and cluster environments can be traced back to work based on dynamic hashing schemes, such as LH* [5] and DDH [6]. Later, Gribble et al. [7] have designed a DHT to support Internet services on clusters of workstations. From the perspective of the hash routing approach, all of them maintain total routing information.

In its inception, the classical Consistent Hashing model [3, 4] was used for cooperative Web Caching, using an approach where nodes of a distributed cache have partial views of the node set but routing inside a view is total, that is, each node knows every other node in its view.

Lately, research on DHTs has mainly addressed P2P environments [8], where lookup schemes used to find a data item usually have a DHT interface, such as Chord [9], CAN [10] and Kademlia [11]. In general, these schemes guarantee logarithmic bounds (with respect to the number of peers) on the routing state at each peer and on the number of hops needed to find a data item.

Chord is based on the classical Consistent Hashing model, with random sized partitions, but partial routing information per node; when nodes join or leave the DHT, it scales worse than BiCH. CAN uses a toroidal d-dimensional coordinate space, partitioned into zones; lookups scale worse than BiCH, but routing tables are smaller; as in BiCH, a node that joins the DHT chooses a random point and asks half of the zone (partition) where the point belongs. Kademlia implements "tree/trie-like routing", by using a prefix tree for routing lookup requests towards leaves (nodes); considering that by expanding a prefix we could obtain a large set of contiguous full hash indexes, then a prefix defines a partition; as Kademlia k-buckets allow access to any subtrie, leaves of a resume, in BiCH, may be viewed as roots of subtrees that, together, make up for the entire partition tree.

6 Conclusions

All the contributions of our investigation in order to eliminate the hot-spot problem and produce shorter routing paths come with a price: additional routing information has to be maintained at each node to be used for routing decisions.

Nevertheless, we lowered the upper bound of the search cost for the best next-hop, from $O(logN)$ when scanning a single routing table using the base algorithm, to $O(log(logN))$ when scanning the joint resume and the cache using the new combined routing algorithm.

In addition, BiCH may now be regard as an adaptive system that supports routing with automatic transition between partial or total information modes, while preserving the scalability of the routing subsystem.

The BiCH platform is planned to evolve in several areas, with emphasis in:

- *node disconnections* – the basic design already supports synchronous/administrative disconnections but is necessary to minimize the impact of asynchronous disconnections; in both cases, the idea is ensure that every partition that is on-line (not disconnected) is reachable; due to the huge volume of data sets that the BiCH DHT is intended to manage, we do not plan to support data replication;

- *skewed hash distributions* – for specific key sets, the hash function may produce biased hash indexes; in such situation, there will be an imbalance between the real data load of a node and its share of the hash function range; it should be possible to detect and correct such situations, possibly by reassigning overloaded partitions (or subpartitions) to other nodes;

- *architecture of the BiCH platform* – although the basic role of a node in the BiCH DHT is already established, it is necessary to refine its tasks; the possible coexistence of several partition sets, with origin in different DHTs, in the same cluster node, also raises other issues, such as "does a partition set has to have its own separate software context or may it benefit from the existence of a single service layer to handle all BiCH related activities?"; it is also necessary to formalize how external clients may access a BiCH DHT, namely what external mechanisms may be published to clients, to ensure a balanced and fast access to the service.

References

[1] J. Rufino, A. Pina, A. Alves, and J. Exposto. BiCH: Binary Consistent Hashing for Cluster oriented Distributed Hash Tables. 2003. (submitted to HiPC 2003; available on request).

[2] Scalable Information Retrieval environment (SIRe). SAPIENS/POSI project 41739/CHS/2001. http://sire.estig.ipb.pt.

[3] D. Kargeer, 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*, 1997.

[4] D. Kargeer, A. Sherman, A. Berkheimer, R. Bogstad, B. andDhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing. In *Proceedings of the 8th International WWW Conference*, 1999.

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

[6] R. Devine. Design and implementation of DDH: a distributed dynamic hashing algorithm. In *Procs. of the 4th Int. Conf. on Foundations of Data Organization and Algorithms*, 1993.

[7] S.D. Gribble, E.A. Brewer, J.M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[8] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Labs, 2002.

[9] 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.

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

[11] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proceedings of the 1st Int. Workshop on P2P Systems (IPTPS02)*, 2002.