Canon in G Major: Designing DHTs with Hierarchical Structure

Prasanna Ganesan
Stanford University
prasannag@cs.stanford.edu

Krishna Gummadi
University of Washington
gummadi@cs.washington.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

Abstract

Distributed Hash Tables have been proposed as flat, non-hierarchical structures, in contrast to most scalable distributed systems of the past. We show how to construct hierarchical DHTs while retaining the homogeneity of load and functionality offered by flat designs. Our generic construction, Canon, offers the same routing state vs. routing hops trade-off provided by standard DHT designs. The advantages of Canon include (but are not limited to) (a) fault isolation, (b) efficient caching and effective bandwidth usage for multicast, (c) adaptation to the underlying physical network, (d) hierarchical storage of content, and (e) hierarchical access control. Canon can be applied to many different proposed DHTs to construct their Canonical versions. We show how four different DHTs—Chord, Symphony, CAN and Kademlia—can be converted into their Canonical versions that we call Crescendo, Cacophony, Can-Can and Kandy respectively.

1. Introduction

A Distributed Hash Table (DHT) is simply a hash table that is partitioned among a dynamic set of participating *nodes*. There is no central directory describing which node manages which partition. Instead, nodes are arranged in an *overlay network*, so that queries for any key can efficiently be *routed* to the appropriate node.

DHTs have been proposed as a substrate for large-scale distributed applications. The traditional approach to building scalable distributed applications has almost always revolved around exploiting a hierarchical structure. Applications ranging from overlay multicast and distributed file systems to the current internet architecture and the DNS system, all achieve scalability via hierarchical design. In stark contrast, all DHT solutions we know of have been flat and non-hierarchical, which has both advantages and disadvantages. In this paper, we argue that one can obtain the best of both worlds, without inheriting the disadvantages of either, by designing hierarchically structured DHTs using a paradigm we call **Canon**.

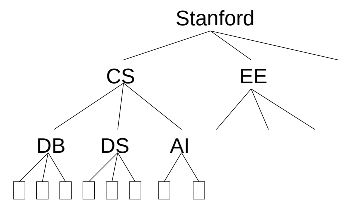

Figure 1: A portion of a hierarchy of nodes

Why flat design? The primary advantage of flat DHT design is that there is a uniform distribution of functionality and load among the participating nodes which also ensures that there is no single point of failure.

Why hierarchical design? Herbert Simon, in *The Architecture of Complexity* [1], argues that hierarchy emerges inevitably in any complex system. Butler Lampson, when describing the design of a global name system [2] observes: "Hierarchy is a fundamental method for accommodating growth and isolating faults". In our DHT context, hierarchical design offers the following advantages: fault isolation and security, effective caching and bandwidth utilization, adaptation to the underlying physical network, hierarchical storage, and hierarchical access control.

Our proposed design, Canon, inherits the homogeneity of load and functionality offered by flat design, while providing all the above advantages of hierarchical design. The key idea behind Canon lies in its recursive routing structure. Figure 1 depicts an example fragment of the hierarchy of machines at Stanford University. The rectangular boxes stand for participant nodes in the DHT. We refer to the internal nodes in the hierarchy as *domains*, to distinguish them from the actual system nodes. When we refer to the "nodes in domain D", we refer to all the system nodes in the subtree rooted at D. The design of Canon ensures that the nodes in any domain form a DHT routing structure by themselves. Thus, for example, the nodes in the "DB" domain would form a DHT structure by themselves, as will the set of all nodes in the CS domain, and the entire set of nodes at Stanford.

The DHT corresponding to any domain is synthesized by *merging* its children DHTs by the addition of some links. Thus, the DHT for CS is constructed by starting with the individual DHTs for domains DB, DS and AI, and adding links carefully from each node in one domain to some set of nodes in the other domains. The challenge we face is to perform this merging in such a fashion that the *total* number of links per node remains the same as in a flat DHT design, and that global routing between any two nodes can still be achieved as efficiently as in flat designs.

The Canon principle can be applied to transform many different DHT designs into their Canonical versions. Much of this paper will focus on Crescendo, the Canonical version of the popular Chord [3] DHT. However, we will also describe how to adapt other DHTs, including nondeterministic Chord [4, 5], Symphony [6], CAN [7] and Kademlia [8], a variant of Pastry [9].

The rest of this paper is organized as follows. In Section 2, we discuss the design of the basic routing framework for Canon, explaining how it is used to construct Crescendo, and show how it provides fault isolation. In Section 3, we explain how to construct Canonical versions of other DHTs, and offer enhancements to provide support for physical-network proximity in all our constructions. In Section 4, we discuss the usage of the hierarchy in content storage and retrieval, access control, and caching policies. In Section 5, we validate our design and quantify its advantages by means of experiments. Section 6 discusses related work.

2. Crescendo: A Canonical version of Chord

In this section, we discuss a hierarchical version of Chord that we call **Crescendo**[1]. We first describe the "static" structure of Crescendo, discuss how routing occurs in this structure, and then explain how this structure is maintained dynamically.

2.1. The Routing Structure of Crescendo

Chord: Chord [3] is a distributed hash table storing key-value pairs. Keys are hashed into a circular N-bit identifier space $[0, 2^N)$. (Identifiers on the circle are imagined as being arranged in increasing order clockwise.) Each node is assigned a unique *ID* drawn uniformly at random from this space, and the *distance* from a node m to a node m' is the clockwise distance on the circle from m's ID to m''s. Each node m maintains a link to the closest node m' that is at least distance 2^i away, for each $0 \leq i < N$. We will refer to the set of nodes forming a Chord network as a *Chord ring*.

Crescendo, our hierarchical DHT, requires all the nodes in the system to form a *conceptual hierarchy* reflecting their

1 A sequence of ever-rising Chords

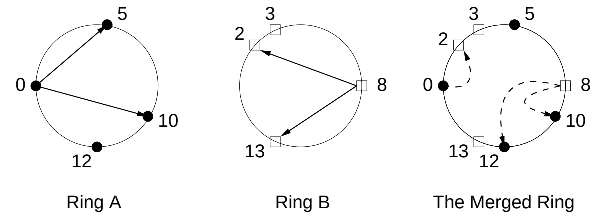

Figure 2: Merging two Chord rings

real-world organization, such as the one in Figure 1. We note that no global information about the structure of the hierarchy is necessary; it suffices for each node to know its own position in the hierarchy, and for two nodes to be able to compute their lowest common ancestor. (One possible practical implementation is to assign each node a hierarchical name as in the DNS system.) The hierarchy may also evolve dynamically with the introduction of new domains.

Each node in Crescendo is assigned a unique ID from the circular N-bit space, just as in Chord. However, the link structure in Crescendo is recursive in nature. Each set of nodes in a leaf domain[2](e.g., DB in Figure 1) forms a Chord ring just as in Chord. At each internal domain, the Crescendo ring, containing all nodes in that domain, is obtained by *merging* all the "children" Crescendo rings into a single, larger Crescendo ring. Applying this construction recursively at higher levels of the hierarchy leads to merging larger and larger rings to eventually produce the global DHT containing all the nodes in the system.

We first use an example to show how *two* separate Chord rings are merged into one Crescendo ring. Say there are two Chord rings A and B, each with four nodes as shown in Figure 2. All nodes choose a globally unique random integer ID in $[0, 16)$. We will focus on the edges created by two nodes: node 0 in ring A and node 8 in ring B. Recall that node 0 establishes its links in ring A by finding, for each $0 \leq k < 4$, the closest node that is at least distance 2^k away. Consequently, it creates links to nodes 5 (corresponding to distances $1, 2$ and 4) and 10 (distance 8). Similarly, in ring B, node 8 forms links to nodes 13 and 2.

When the two rings are merged, nodes retain all their original links. In addition, each node m in one ring creates a link to a node m' in the other ring if and only if:

(a) m' is the closest node that is at least distance 2^k away for some $0 \leq k < N$, and

(b) m' is closer to m than any node in m's ring.

Note that condition (a) is just the standard Chord rule for creating links, applied on the union of the nodes in the two rings. Condition (b), however, says that node m should cre-

2 Since our hierarchy is a "conceptual hierarchy", nodes are assumed to be hanging off the leafs rather than being leafs themselves.

ate only a subset of these links, specifically, only the links to nodes that are closer to it than any other node in its own ring.

Returning to our example, let us consider the links to be created by node 0. Condition (a) suggests that node 0 link to node 2 (for distances 1 and 2), and to node 8 (for distance 8). However, condition (b) rules out node 8, since it is further away than the closest node in Ring A (node 5). Thus, node 0 establishes an additional link only to node 2. Note that there is no link from node 0 to node 3. As another example, consider node 8 in Ring B. Condition (a) suggests nodes 10 (distances 1 and 2) , 12 (distance 4) and 0 (distance 8) as candidates. We again use condition (b) to rule out node 0.

Note that some nodes may not form any additional links at all. For example, node 2 has node 3 in its own ring as the closest node, due to which condition (b) is violated for all other nodes. One may wonder whether our construction leads to a skewed degree distribution among the nodes. However, such is not the case. Our evaluation in Section 5 will show the actual skew in degree distribution compared to standard Chord.

The above approach for merging two rings naturally generalizes to merging any number of rings rather than just two. Each node once again forms links to nodes *other than those in its own ring* if they satisfy conditions (a) and (b). This algorithm for link creation is applied bottom-up on the hierarchy, merging sibling rings to construct larger and larger rings until all the nodes belong to the same ring. We state the following theorems on node degrees in Chord and Crescendo. (Note that the degree of a node refers to its out-degree, and does not count incoming edges.) We present formal proofs in a technical report [10].

Theorem 1 *In a Chord ring of n nodes, with nodes choosing their ID uniformly at random, the expected degree of a node is at most* $\log(n-1)+1$, *for all* $n > 1$.

The above theorem bounds the expected degree of a node in Chord and our work appears to be the first to claim it. The following theorem provides a somewhat weaker bound on the expected degree for Crescendo. However, our experiments in Section 5 show that, in practice, *the average degree of a node in Crescendo is slightly less than in Chord*, and that it decreases as the number of levels in the hierarchy increases.

Theorem 2 *In a Crescendo ring of n nodes, with nodes choosing their ID uniformly at random, the expected degree of a node is at most* $\log(n-1)+\min(l, \log n)$ *if* $n > 1$, *where l is the maximum number of levels in the hierarchy.*

The following theorem shows that a node in Crescendo has a logarithmic degree with high probability.

Theorem 3 *The degree of any node in Crescendo is* $O(\log n)$ *with high probability (w.h.p.) irrespective of the structure of the hierarchy.*

2.2. Routing in Crescendo

Routing in Crescendo is identical to routing in standard Chord, namely, greedy clockwise routing. If a node wishes to route a message to a destination d, it simply forwards the message to its neighbor that is closest to d while not overshooting the destination.

Observe that greedy clockwise routing in Crescendo is naturally hierarchical. In order to get to a destination d, a node m initially attempts to take the largest possible steps towards the destination, which implies that the node implicitly routes to the closest predecessor of d *in the lowest-level Crescendo ring it belongs to*. In Figure 2, if node 2 in ring B wished to route to node 12, it would route along ring B to node 8. Node 8 then switches to routing on the *merged ring*, i.e., using the ring at the next level of the hierarchy. It uses greedy, clockwise routing to forward to node 10, which in turn forwards to node 12, completing the route.

In general, when there are multiple levels of the hierarchy, greedy clockwise routing routes to the closest predecessor p of the destination at each level, and p would then be responsible for switching to the next higher Crescendo ring and continue routing on that ring. We can now see two crucial properties of this routing protocol.

Locality of intra-domain paths: The route from one node to another never leaves the domain in the hierarchy, say D, that contains both nodes. This is clearly true, since routing uses progressively larger Crescendo rings, and would be complete when the ring contains all nodes in D.

Convergence of inter-domain paths: When different nodes within a domain D (at any level of the hierarchy) route to the same node x outside D, all the different routes exit D through a common node y. This node is, in fact, the closest predecessor of x within domain D.

The locality of intra-domain paths provides fault isolation and security, since interactions between two nodes in a domain cannot be interfered with by, or affected by the failure of, nodes outside the domain. We discuss its implications for hierarchical storage and access control in Section 4. The convergence of inter-domain paths enables efficient caching and multicast solutions layered on Crescendo. We discuss caching in more detail in Section 4. We now characterize the number of hops required for routing in Chord and Crescendo. We refer the reader to our technical report [10] for proofs.

Theorem 4 *In a Chord ring of n nodes, with nodes choosing their integer ID uniformly at random from* $[0, 2^N)$, *the* *expected number of routing hops between two nodes is at most* $\frac{1}{2}\log(n-1)+\frac{1}{2}$*, if* $n>1$.

The above theorem pertains to the routing cost in Chord and, to the best of our knowledge, has not been proved prior to this work. The following theorems offer a weak upper bound on the expected number of routing hops in Crescendo, and show that routing between any two nodes takes only $O(\log n)$ hops with high probability. In Section 5, we experimentally show that routing in Crescendo is almost identical in efficiency to routing in Chord, irrespective of the structure of the hierarchy.

Theorem 5 *In a Crescendo ring of* n *nodes, with* $n>1$ *and nodes choosing their ID uniformly at random, the expected number of routing hops between two nodes is at most* $\log(n-1)+1$*, irrespective of the hierarchy structure.*

Theorem 6 *In a Crescendo ring of* n *nodes, with nodes choosing their IDs uniformly at random, the number of routing hops to route between any two nodes is* $O(\log n)$ *w.h.p.*

2.3. Dynamic Maintenance in Crescendo

So far, we have discussed the Crescendo structure without describing how it is constructed and maintained in the face of node arrivals and departures. Dynamic maintenance in Crescendo is a natural extension of dynamic maintenance in Chord. We describe only the protocol for nodes joining the system. The protocol for nodes leaving is similar.

When a new node m joins the system, it is expected to know at least one other existing node in its lowest-level domain. (If m is the first node in this lowest-level domain, then m is expected to know an existing node in the lowest domain of m in which some other node exists in the system.) This knowledge can be provided by many different mechanisms. For example, a central server could maintain a cache of live nodes in different portions of the hierarchy, and new nodes could contact the server for this information. Alternatively, each domain could have its own server maintaining a list of nodes in the system. (For example, the local DNS server could be modified to provide this information.) As a third alternative, this information can be stored in the DHT itself, and a new node can simply query the DHT for the requisite information if it knows any live node in the system.

Let us say the new node m knows an existing node m' in its lowest-level domain. Then, the new node "inserts" itself using the standard Chord technique for insertion, applied at each level of the hierarchy. Specifically, node m routes a query through m' for its own ID, and the query reaches the predecessor of m's ID at each level of the hierarchy. (This is due to the convergence of inter-domain paths.) At each such level, going successively from the lowest-level domain to the top, m inserts itself after this predecessor and sets up appropriate links to other nodes in that domain. (As an optimization, it can use its predecessor's links in each domain as a hint for finding the list of nodes m needs to link to in that domain.)

Once m has established its links in all the domains, m informs its successor in each domain of its joining. The successor at each level, say s_l, ensures that all nodes at that level which now "erroneously" link to s_l instead of to m, are notified. This notification can either be done eagerly, or can be done lazily when such an erroneous link is used to reach s_l for the first time. The total number of messages necessary to ensure all links in the system are set up correctly after a node insertion is $O(\log n)$ which is the same as in normal Chord.

Leaf Sets: In Chord, each node needs to "remember" a list of its successors on the ring, called the leaf set, to deal with node deletions. In Crescendo, each node maintains a list of successors at every level of the hierarchy. Note that leaf sets are cheap to maintain since they can be updated by passing a single message along the ring, and do not cause state overhead since they do not correspond to actual TCP links.

3. General Canon and Physical-Network Proximity

Having seen how to construct a hierarchical version of the Chord DHT, we now generalize our approach to create other Canonical constructions. We then discuss how to adapt all our constructions to optimize for the proximity of different nodes in the physical network.

3.1. Canonical Symphony : Cacophony

Symphony [6] is a randomized version of Chord, where each node m creates O($\log n$) links (where n is the number of nodes in the system) to other nodes, each chosen independently at random, such that the probability of choosing a node m' as a neighbor is inversely proportional to the distance from m to m'. In addition, each node maintains a link to its immediate successor on the ring.

The construction of Canonical Symphony, or Cacophony, is similar to that of Crescendo. Each node creates links in its lowest-level domain just as in Symphony, but choosing only $\lfloor \log n_l \rfloor$ random links, where n_l is the number of nodes in that domain. At the next higher level, it chooses $\lfloor \log n_{l-1} \rfloor$ links by the same random process, where n_{l-1} is the number of nodes in the domain at that level, but retains only those links that are closer than its successor at the lower level. In addition, it creates a link to its successor at the new level.

This iterative process continues up to the top level of the hierarchy. It is again possible to show that Cacophony achieves logarithmic routing when each node has degree $O(\log n)$. Note that both Symphony and Cacophony require the ability to estimate the number of nodes in a domain, and it is possible to perform this estimation cheaply and accurately [6]. It is actually possible to route in Symphony using only $O(\log n/\log\log n)$ hops using a modification to greedy routing. Cacophony also appears to achieve the same performance improvements as Symphony using the modified routing protocol.

3.2. Canonical Nondeterministic Chord : Nondeterministic Crescendo

Yet another variant of Chord is nondeterministic Chord [4, 5], where a node chooses to connect to any node with distance in $[2^{k-1}, 2^k)$ for each $0 \leq k < N$, instead of connecting to the closest node that is at least distance 2^{k-1} away. Nondeterministic Chord has routing properties almost identical to Symphony. The construction of nondeterministic Crescendo is very similar to Crescendo, with the nondeterministic Chord rule for link selection instead of the deterministic rule. However, when rings are merged, a node m can exercise its nondeterministic choice only among those nodes that are closer to it than any other node in its own ring.

For example, consider a node m in some ring A and say the node closest to m on A is m' at distance 12. Let us say there are two nodes p and q belonging to the next higher domain, which are distances 10 and 14 away from m. Since nondeterministic Chord only requires a link to any node between distances 8 and 15, node m may decide to consider node q to link to and not node p. However, since q is further away than m', node m would consequently decide not to link to either p or q which is erroneous. Instead, node m is allowed to exercise its nondeterministic choice only to choose among nodes which are between distances 8 and 12 away.

3.3. Canonical Pastry/Kademlia : Kandy

Pastry [9] and Kademlia [8] are hypercube versions of nondeterministic Chord. We will describe Kademlia and its Canonical version. Pastry is similar to Kademlia but has a two-level structure that makes its adaptation more complex. Kademlia defines the distance between two nodes using the XOR metric rather than the clockwise distance on a ring. In other words, the distance between two nodes m and m' is defined to be the integer value of the XOR of the two IDs. Just like in nondeterministic Chord, each node m is required to maintain a link to any node with distance in $[2^{k-1}, 2^k)$, for each $0 \leq k < N$. (For resilience, Kademlia actually maintains multiple links for each of these distances but we ignore them in this discussion.) Routing is still greedy, but works by diminishing this XOR distance rather than the clockwise distance.

Our Canon construction for nondeterministic Crescendo carries over directly to Kademlia. Each node creates its links in the lowest-level domain just as dictated by Kademlia. At the next higher level, it again uses the Kademlia policy and applies it over all the nodes at that level to obtain a set of candidate links (with the same caveat as in nondeterministic Crescendo). It then throws away any candidate whose distance is larger than the shortest distance link it possesses at the lower level. The construction is repeated at successively higher levels of the hierarchy, just as normal.

3.4. Canonical CAN : Can-Can

CAN [7] was originally proposed as a network with constant expected degree, but can be generalized to a logarithmic degree network. The set of node identifiers in CAN form a binary prefix tree, i.e., a binary tree with left branches labeled 0 and right branches labeled 1. The path from the root to a leaf determines the ID of a node corresponding to that leaf.

Since leaf nodes may exist at multiple levels of the tree, not all IDs are of the same length. We therefore make IDs equal-length by treating a node with a shorter ID as multiple virtual nodes, one corresponding to each padding of this ID by different sequences of bits. For example, if there are three nodes with IDs 0, 10 and 11, the first node is treated as two virtual nodes with IDs 00 and 01. Edges correspond exactly to hypercube edges: there is an edge between two (virtual) nodes if and only if they differ in exactly one bit. Routing is achieved by simple left-to-right bit fixing, or equivalently, by greedy routing using the XOR metric.

Canonical CAN, Can-Can, is constructed in a by-now-familiar fashion. Again, traditional CAN edges are constructed at the lowest level of the hierarchy, and a node creates a link at a higher level only if it is a valid CAN edge and is shorter than the shortest link at the lower level. Again, the properties of Can-Can are almost identical to that of logarithmic-dimensional CAN constructed in the fashion we have described here.

3.5. Further Generalizations

The use of hierarchical routing offers us even more flexibility in choosing routing structure. We observe that there is no explicit requirement that the routing structure created, and the routing algorithm used, be the same at different levels of the hierarchy. For example, say the nodes belonging to the same lowest level of the hierarchy are all on the same LAN. In such a case, it may make sense to use a routing structure other than Chord to link them up. For example, there may be efficient broadcast primitives available on the

LAN which may allow setting up a complete graph among the nodes. It may also be useful to implement messaging via UDP rather than TCP to reduce communication overhead. As another example, it may be possible to leverage detailed information about the location, availability, and capacity of nodes within an organization to build much more intelligent and efficient structures than a homogeneous Chord ring.

When "merging" different LANs at the next higher level of the hierarchy, we could still use the same approaches as described earlier, for example, to construct Crescendo. Each node creates links to some nodes outside its LAN, but ensuring that the distance covered by the link is smaller than the distance to its closest neighbor within the LAN (in the same ID space as earlier). Routing takes place hierarchically. At the lowest level, the complete graph is exploited to reach the appropriate node in one hop. This node then forwards on the Crescendo ring at the next level of the hierarchy using greedy, clockwise routing.

3.6. Adapting to Physical-Network Proximity

All the constructions we have seen so far exploit the likelihood of nodes within a domain being physically close to each other to produce natural adaptation to the physical network. However, this natural adaptation is likely to break down at the top levels of the hierarchy. For example, the top level of the hierarchy could have hundreds of children domains spread all over the world. Some of these domains may be in North America while others are in Europe, Africa and Asia. In such a case, we would like to preferentially connect nodes in North America to other nodes in North America, (and among such nodes, nodes on the West coast to other nodes on the West coast) and so on, without having to explicitly create additional levels of hierarchy to capture such preferences.

In such a case, it is possible to introduce such preferential connections, or *adaptation to the physical network*, in a transparent fashion that is independent of the DHT structure being constructed. The intuition is to introduce *multiple choices* of neighbors for each link that a node needs to set up, alowing the node to randomly sample some small number of these choices, and select the "best" one as its neighbor.

Such a construction can be applied both to flat and hierarchical DHTs, with deterministic or nondeterministic topologies. The details of this construction are discussed in [11], and in our extended technical report. The only point we make here is that the use of such proximity adaptation in a hierarchical DHT affects the link construction only at the top level of the hierarchy, and is applicable to any hierarchical DHT. We will refer to the versions of Chord and Crescendo using such *proximity adaptation* as *Chord (Prox.)* and *Crescendo (Prox.)* respectively.

4. Storing and Retrieving Content

In this section, we first discuss the basic mechanism for storing and retrieving content in a hierarchical fashion. We then discuss how caching may be exploited by the system. We discuss how to achieve partition balance, i.e., ensure that content is distributed across the nodes in as even a fashion as possible, in the technical report [10].

4.1. Hierarchical Storage and Retrieval

DHTs are designed to store and retrieve content, consisting of key-value pairs, by hashing the key into the space $[0, 2^N)$. For convenience, we will refer to the hash value of a key as the key itself. In a flat DHT, the hash space is *partitioned* across the different nodes, and the key-value pair is stored at the unique node "responsible" for the partition containing the key. The assignment of responsibility is simple. Each node is responsible for all keys greater than or equal to its ID and less than the next larger existing node ID on the ring[3]. Thus, there is no choice available in determining where a key-value pair is stored. A query for a specific key is answered simply by routing with the key as the "destination", which automatically results in routing terminating at the node responsible for that key.

The hierarchical design of a DHT offers more alternatives for content storage. When a node n wishes to insert content, it can specify the content's *storage domain*, i.e., a domain containing n within which the content must be stored. Node n can also specify the content's *access domain*, a superset of the storage domain, to all of whose nodes the content is to be made accessible.

Say, node n requires a key-value pair $\langle k, v\rangle$ to be stored within storage domain D_s. In such a case, the key-value pair is stored at the node in D_s whose ID is closest to, but smaller than, k, i.e., it is stored at the location dictated by the DHT consisting solely of nodes in D_s. If the content is to be accessible within a larger access domain D_a, rather than only within D_s, an additional *pointer* to the content is stored at the node in D_a whose ID is closest to, but smaller than, k.

A search for a key k occurs by hierarchical, greedy routing just as described earlier, with two changes. The first change is that a node m along the path, which switches routing from one level to the next, may have "local" content that matches the query key. A key-value pair $\langle k, v\rangle$ will be returned by m as a query answer if and only if its access domain is no smaller than the domain defined by the current routing level[4].

3 Chord actually inverts this definition to make a node responsible for keys less than it and greater than the next smaller node ID. However, our definition is an improvement on Chord's both in terms of efficiency and coding complexity.

If the application allows only one value for each key, then search can terminate when the first node along the path finds a hit for the key. *Note that this implies that a query for content stored locally in a domain never leaves the domain.* If the application requires a partial list of values (say one hundred results) for a given key, the routing can stop when a sufficient number of values have been found for the key.

The second change to the routing algorithm occurs because a node m may have *pointers* to content matching the key, without having the content itself. In such a case, this indirection is resolved and the actual content is fetched by m (and possibly cached at m) before being returned to the initiator of the query.

Greedy routing thus automatically supports both hierarchical storage and access control. A query initiated by a node automatically retrieves exactly that content that a node is permitted to access, irrespective of whether parts of this content are stored locally in a domain or globally.

4.2. Caching

The hierarchical routing of queries naturally extends to take advantage of the caching of query answers. As we have already seen, the convergence of inter-domain paths imply that, in each domain D, a query Q for the same key initiated by any node in D exits D through a common node $p_{Q,D}$ which we call the *proxy node* for query Q in domain D. (This node $p_{Q,D}$ is also responsible for storing content with the same key and storage domain D.) Thus, answers to Q may be cached at $p_{Q,D}$ for any, or all, choices of the level of domain D in the hierarchy.

We propose caching the answer to query Q at the proxy node $p_{Q,D}$ at each level of the hierarchy encountered on the path to the query's answer. If nodes exhibit locality of access, it is likely that the same key queried by a node m would be queried by other nodes close to m in the hierarchy. Say another node m' initiates a query for the key queried by m. The routing algorithm ensures that the cached copy of the answer is discovered at the lowest-level domain which contains both m and m'.

We discuss caching, and cache-replacement policies, in more detail in the extended technical report. We simply note here that the above caching scheme offers many advantages over caching in flat DHTs. Caching solutions for flat DHT structures all require that the query answer be cached all along the path used to route the query, creating many cached copies of each query answer, and leading to higher overhead. Moreover, the absence of guaranteed local path convergence implies that these cached copies cannot be exploited to the fullest extent, and may not capture locality of access patterns.

4 This routing level can either be maintained as a field in the query message, or can be computed by finding the lowest common ancestor of m and the query source.

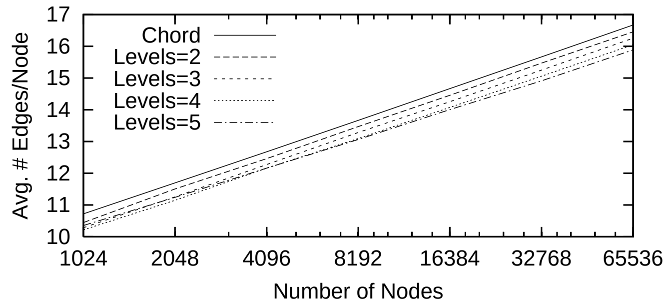

Figure 3: Average Number of Links per Node

5. Evaluation

We now present an experimental evaluation of the different routing and path convergence properties of Crescendo.

5.1. Basic Routing Properties

Our first set of experiments evaluates the number of links vs. number of routing hops tradeoff offered by Crescendo and shows that Crescendo is very similar to Chord. All our experiments for this subsection use a hierarchy with a fanout of 10 at each internal node of the hierarchy. The number of levels in the hierarchy is varied from 1 (a flat structure) to 5. The number of nodes in the system is varied from 1024 to 65536, and all nodes choose a random 32-bit ID.

We used two different distributions to assign nodes to positions in the hierarchy: (a) uniformly random assignment of each node to a leaf of the hierarchy (b) a Zipfian distribution of nodes where the number of nodes in the k^{th} largest branch (within any domain) is proportional to $1/k^{1.25}$. The results obtained with the two distributions were practically identical and here, we show graphs corresponding to the Zipfian distribution.

Figure 3 plots the average number of links per node, as a function of the size of the network, for different numbers of levels in the hierarchy. Note that Chord corresponds to a one-level hierarchy. *We notice that the number of links is extremely close to* $\log n$ *irrespective of the number of levels. We also observe that the number of links decreases slightly as the number of levels increases.* (The reason for this drop in edges lies in Jensen's inequality. To illustrate, consider the merging of two rings A and B with m nodes each. The expected number of B nodes between two consecutive A nodes is 1. However, the expected number of inter-domain links set up by an A node is less than 1 because $E(\log(X+1)) \leq \log(E(X)+1)$ for any random

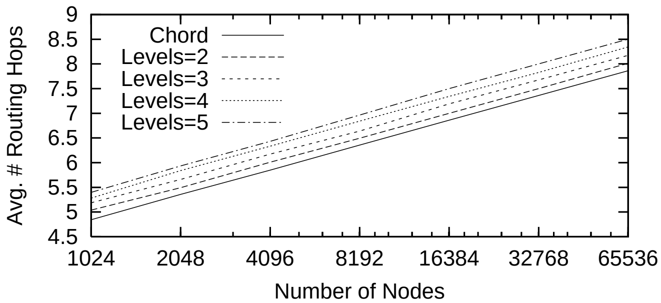

Figure 4: Average Number of Routing Hops

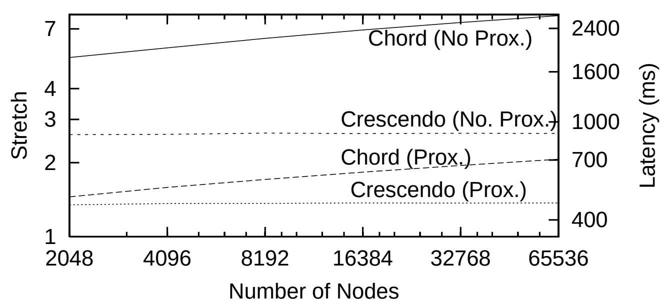

Figure 5: Routing Stretch on Crescendo and Chord

variable $X > 0$.) We discuss the *distribution* of the number of links per node in the extended technical report.

Figure 4 depicts the average number of hops required to route between two nodes, as a function of the network size. We see that the number of routing hops is $0.5 \log n + c$, where c is a small constant which depends on the number of levels in the hierarchy. The number of hops required increases slightly when the number of levels in the hierarchy increases, which is explained by the corresponding drop in the number of links created. We note, however, that this increase is at most 0.7 irrespective of the number of levels in the hierarchy (even beyond what we have depicted on this graph).

5.2. Adaptation to Physical-Network Proximity

We now evaluate routing in terms of actual physical-network latency, rather than in terms of the number of hops used. All experiments hereonin use the following setup: We used the GT-ITM [12] topology generator to produce a 2040-node graph structure modelling the interconnection of routers on the internet. In this model, routers are broken up into transit domains, with each transit domain consisting of transit nodes. A stub domain is attached to each transit node, and the stub domain is itself composed of multiple stub nodes interconnected in a graph structure. The latency of a link between two transit nodes is assigned to be 100ms, with transit-stub links being 20ms and stub-stub links being 5ms.

To construct a Crescendo network with a desired number of nodes, we uniformly attach an appropriate number of Crescendo nodes to each stub node, and assume that the latency of the link from a Crescendo node to its stub node is 1ms. This GT-ITM structure induces a natural five-level hierarchy describing the location of a node (root, transit domain, transit node, stub domain, stub node).

We evaluate the performance of four different systems: Chord and Crescendo, with and without the top-level proximity adaptation discussed in Section 3.6. Figure 5 depicts the performance of these four systems using two different measures on the y-axis. The right axis shows the average routing latency, while the left axis shows the *stretch*, which is the same quantity normalized by the average shortest-path latency between any two nodes in our internet model. Thus, a stretch of 1 implies that routing on the overlay network is as fast as directly routing between nodes on our model of the internet.

We observe that the routing latency of plain Chord is again linearly related to $\log n$, which is not too surprising. Plain Crescendo, on the other hand, fares much better than Chord, and produces an almost constant stretch of 2.7 even as the number of nodes increases. The reason why the stretch is constant is that an increase in the number of nodes only results in the lowest-level domain under each stub node increasing in size. Thus, the "additional hop" induced by the number of nodes quadrupling is effectively a hop between two nodes attached to the same stub, which costs only 2ms in our model.

When proximity adaptation is used, the stretch for Chord (Prox.) improves considerably but is still about 2 on a 64K-node network. Again, we notice that the stretch is a linear function of $\log n$, albeit with a smaller multiplicative factor than earlier. Crescendo (Prox.), which uses proximity adaptation at the top level of the hierarchy, again produces a constant stretch of 1.3 for all network sizes, and thus continues to be considerably better than Chord (Prox.). These curves illustrate the important scaling advantage of Crescendo-style bottom-up construction, which results in constant stretch, compared to random sampling to find a nearby node, whose performance is a function of the number of nodes in the system.

5.3. Locality of Intra-domain paths

We now illustrate the advantages of intra-domain path locality. Again using our GT-ITM models, we compare the expected latency of a query as a function of its locality of access. Thus, a "Top Level" query would be for content present anywhere in the system, a "Level 1" query initi-

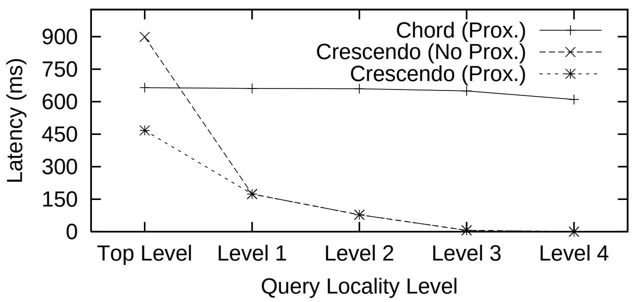

Figure 6: Latency as a function of query locality

ated by a node would be for content within its own transit domain, and so on. Figure 6 plots the latency as a function of the locality of the query for three different systems: a 32K-node Chord network with proximity adaptation, and a 32K-node Crescendo network with and without proximity adaptation. We do not show plain Chord since its performance is off by an order of magnitude. Also note that the use of proximity adaptation in Crescendo only improves the performance of top-level queries, since it applies only to the top level of the hierarchy.

The left end-points of the lines depict the query latency for top-level queries and, as observed earlier in Figure 5, Crescendo with proximity adaptation performs the best. As the locality of queries increases, the latency drops drastically in Crescendo and becomes virtually zero at Level 3, where all queries are within the same stub domain. On the other hand, Chord, even with proximity adaptation, shows very little improvement in latency as query locality increases.

This graph establishes two points: (a) search for local content is extremely efficient in Crescendo, from which we deduce that local caching of query answers will improve performance considerably, and (b) Chord does not offer good locality of intra-domain paths.

5.4. Convergence of inter-domain paths

We now illustrate the advantages of inter-domain path convergence for caching. Evaluation for multicast is described in [10].

Say a random node r initiates a query Q for a random key. Let r belong to domain D, and let the path taken by the query be P. Consider a second node chosen at random *from* D which issues the same query Q, and let P' be the path taken by the query from this node. We define the *hop overlap fraction* to be the fraction of the path P' that overlaps with path P. We define the *latency overlap fraction* to be the ratio of the latency of the overlapping portion of P' to the latency of the entire path P'. The expected values of these quantities can be viewed as simple metrics capturing the

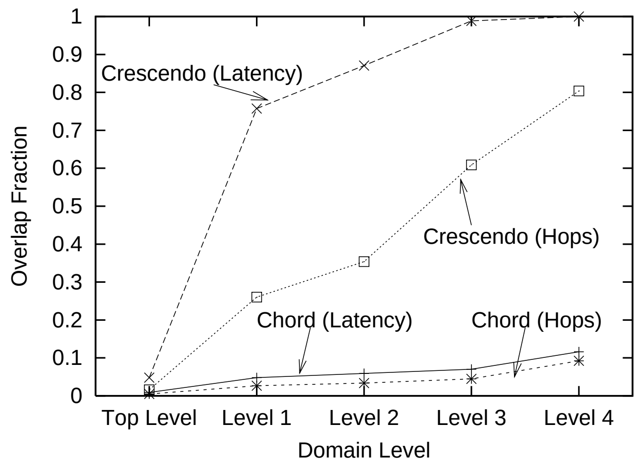

Figure 7: Overlap Fraction versus domain level

bandwidth and latency savings respectively, obtained due to the first node's query answer being cached along the query path. (We note that practical bandwidth savings would be much higher, because inter-domain bandwidth is generally much more expensive than intra-domain bandwidth.)

Figure 7 depicts the expected value of the hop overlap fraction and the latency overlap fraction for both Crescendo and Chord (with proximity adaptation), as a function of the level of the domain within which the nodes are drawn. We see that the overlap fraction, for both hops and latency, is extremely low for Chord, even for low-level domains. On the other hand, the overlap fraction increases considerably as the level of the domain increases in Crescendo. As is only to be expected, the overlap fraction is higher for latency, since the non-overlapping hops have very low latency.

6. Related Work

There have been many different Distributed Hash Table designs that have been proposed recently [7, 3, 9, 13, 6, 14, 15, 16, 17] all of which use routing structures that are variants of the hypercube. All of them can be viewed as providing routing in $O(\log n)$ hops on an n-node network when each node has degree $\Theta(\log n)$. (Some of these constructions are for constant-degree networks but they may be generalized to use base $\log n$, and thus have logarithmic number of links [15, 14].)

Some of these networks also use locality heuristics [9, 18, 13, 5, 11] to ensure that nodes nearby on the physical network are preferentially connected to each other. In consequence, they achieve some convergence on inter-domain paths due to this "clustering" effect. However, such convergence is heuristic in nature and is dependent on the number of nodes in the system, the characteristics of the underlying physical network and the relative stability of the different nodes.

Another recent system which provides some DHT functionality is SkipNet [19]. SkipNet behaves just like a normal DHT when routing for content outside the local domain (and thus provides no, or heuristic, convergence for inter-domain paths). However, SkipNet provides explicit path locality when searching for content within the domain. This locality is achieved by using a separate routing protocol which requires maintaining more state at each node. Furthermore, SkipNet does not ensure convergence of inter-domain paths, a property essential for efficient caching and multicast implementations.

SkipNet also offers storage of content within an arbitrary storage domain, but at the expense of modifying the key of the content. This may be acceptable, or desirable, for some applications such as DNS. Our aim, on the other hand, is to allow arbitrary storage domains without modifying the key, which is necessary for true DHT functionality. Finally, SkipNet offers the additional ability to query in a namespace, a feature not present in other DHTs. It is possible to inherit this feature by building a Canonical version of SkipNet, the details of which we postpone to a future work.

7. Conclusion

We have described Canon, a general technique for constructing hierarchically structured DHTs. We have shown how this technique can be applied to construct different DHTs, Crescendo, Cacophony, Can-Can and Kandy. We demonstrated the advantages of hierarchical DHT construction and routing in terms of fault isolation, and quantified the advantages of our design in terms of caching, bandwidth utilization, adaptation to the physical network, hierarchical storage and hierarchical access control, by means of experiments.

Acknowledgments

We thank Steve Gribble for valuable discussions on the use of hierarchies in DHTs which helped crytallize the ideas presented in this paper.

References

[1] H. Simon, *The Sciences of the Artificial*, MIT Press, 1996.

[2] B. Lampson, "Designing a global name service," in *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, 1986, pp. 1–10.

[3] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. ACM SIGCOMM 2001*, 2001.

[4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Wide-area cooperative storage with CFS," in *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, 2001, pp. 202–215.

[5] K. Gummadi, R. Gummadi, S. Gribble, S.Ratnasamy, S.Shenker, and I.Stoica, "The impact of DHT routing geometry on resilience and proximity," in *Proc. ACM SIGCOMM*, 2003.

[6] G. S. Manku, M. Bawa, and P. Raghavan, "Symphony: Distributed hashing in a small world," *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[7] S. Ratnasamy, P. Francis, M. Handley, and R. M. Karp, "A scalable content-addressable network," in *Proc. ACM SIGCOMM 2001*, 2001, pp. 161–172.

[8] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Proc. 1st Intl. Workshop on Peer-to-Peer Systems (IPTPS 2002)*, 2002.

[9] A. I. T. Rowstron and Peter Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001, pp. 329–350.

[10] P. Ganesan, K. Gummadi, and H. Garcia-Molina, "Canon in G major: Designing DHTs with hierarchical structure," Tech. Rep., Stanford University, 2003, http://dbpubs.stanford.edu/pub/2003-74.

[11] G. S. Manku and P. Ganesan, "DHT design: A modular approach," Technical Report.

[12] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee, "How to model an internetwork," in *IEEE Infocom*, San Francisco, CA, March 1996, IEEE, vol. 2, pp. 594–602.

[13] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao, "Distributed object location in a dynamic network," in *Proc. 14th ACM SPAA*, 2002, pp. 41–52.

[14] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," in *Proc 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002, pp. 183–192.

[15] F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal hash table," in *Proc. 2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS 2003)*, 2003.

[16] G. S. Manku, "Routing networks for distributed hash tables," in *Proc. 22nd ACM Symp. on Principles of Distriuted Systems (PODC 2003)*, Jul 2003.

[17] I. Abraham, B. Awerbuch, Y. Azar, Y. Bartal, D. Malkhi, and E. Pavlov, "A generic scheme for building overlay networks in adversarial scenarios," in *Proc. Intl. Parallel and Distributed Processing Symp.*, Apr 2003.

[18] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Topology-aware routing in structured peer-to-peer overlay networks," in *Proc. Intl. Workshop on Future Directions in Distrib. Computing (FuDiCo 2002)*, 2002.

[19] N. J. A. Harvey, M. Jones, M. Theimer, and A. Wolman, "Skipnet: A scalable overlay network with practical locality properties," *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.