# A Scalable Content-Addressable Network

Sylvia Ratnasamy[1,2] Paul Francis[2] Mark Handley[2] Richard Karp[1,2]
Scott Shenker[2]

[1] Dept. of Electrical Eng. & Comp. Sci.
University of California, Berkeley
Berkeley, CA, USA

[2] ACIRI
AT&T Center for Internet Research at ICSI
Berkeley, CA, USA

## Abstract

Hash tables – which map "keys" onto "values" – are an essential building block in modern software systems. We believe a similar functionality would be equally valuable to large distributed systems. In this paper, we introduce the concept of a Content-Addressable Network (CAN) as a distributed infrastructure that provides hash table-like functionality on Internet-like scales. The CAN design is scalable, fault-tolerant and completely self-organizing, and we demonstrate its scalability, robustness and low-latency properties through simulation.

## 1 Introduction

A hash table is a data structure that efficiently maps "keys" onto "values" and serves as a core building block in the implementation of software systems. We conjecture that many large-scale distributed systems could likewise benefit from hash table functionality. We use the term *Content-Addressable Network* (CAN) to describe such a distributed, Internet-scale, hash table.

Perhaps the best example of current Internet systems that could potentially be improved by a CAN are the recently introduced peer-to-peer file sharing systems such as Napster [20] and Gnutella [11]. In these systems, files are stored at the end user machines (peers) rather than at a central server and, as opposed to the traditional client-server model, files are transferred directly between peers. These peer-to-peer systems have become quite popular. Napster was introduced in mid-1999 and, as of December 2000, the software has been downloaded by 50 million users, making it the fastest growing application on the Web. New file sharing systems such as Scour, FreeNet, Ohaha, Jungle Monkey, and MojoNation have all been introduced within the last year.

While there remains some (quite justified) skepticism about the business potential of these file sharing systems, we believe their rapid and wide-spread deployment suggests that there are important advantages to peer-to-peer systems. Peer-to-peer designs harness huge amounts of resources - the content advertised through Napster has been observed to exceed 7 TB of storage on a single day[1], without requiring centralized planning or huge investments in hardware, bandwidth, or rack space. As such, peer-to-peer file sharing may lead to new content distribution models for applications such as software distribution, file sharing, and static web content delivery.

Unfortunately, most of the current peer-to-peer designs are not scalable. For example, in Napster a central server stores the index of all the files available within the Napster user community. To retrieve a file, a user queries this central server using the desired file's well known name and obtains the IP address of a user machine storing the requested file. The file is then down-loaded directly from this user machine. Thus, although Napster uses a peer-to-peer communication model for the actual file transfer, the process of locating a file is still very much centralized. This makes it both expensive (to scale the central directory) and vulnerable (since there is a single point of failure). Gnutella goes a step further and de-centralizes the file location process as well. Users in a Gnutella network self-organize into an application-level mesh on which requests for a file are flooded with a certain scope. Flooding on every request is clearly not scalable [12] and, because the flooding has to be curtailed at some point, may fail to find content that is actually in the system.

We started our investigation with the question: could one make a scalable peer-to-peer file distribution system? We soon recognized that central to any peer-to-peer system is the indexing scheme used to map file names (whether well known or discovered through some external mechanism) to their location in the system. That is, the peer-to-peer file transfer process is inherently scalable, but the hard part is finding the peer from whom to retrieve the file. Thus, a scalable peer-to-peer system requires, at the very least, a scalable indexing mechanism. We call such indexing systems Content-Addressable Networks and, in this paper, propose a particular CAN design.

However, the applicability of CANs is not limited to peer-to-peer systems. CANs could also be used in large scale storage management systems such as OceanStore [15], Farsite [3], and Publius [18]. These systems all require efficient insertion and retrieval of content in a large distributed storage infrastructure, and a scalable indexing mechanism is an essential component. In fact, as we discuss in Section 5, the OceanStore system already includes a CAN in its core design (although the OceanStore CAN, based on Plaxton's algorithm[21], is somewhat different from what we propose here).

Another potential application for CANs is in the construction of wide-area name resolution services that (unlike the DNS) decouple the naming scheme from the name resolution process thereby enabling arbitrary, location-independent naming schemes.

Our interest in CANs is based on the belief that a hash table-like abstraction would give Internet system developers a powerful design tool that could enable new applications and communication models. However, in this paper our focus is not on the use of CANs but on their design. In [1], we describe, in some detail, one possible application, which we call a "grass-roots" content distribution system, that leverages our CAN work.

As we have said, CANs resemble a hash table; the basic operations performed on a CAN are the insertion, lookup and deletion of (key,value) pairs. In our design, the CAN is composed of many individual nodes. Each CAN node stores a chunk (called a *zone*) of the entire hash table. In addition, a node holds information about a small number of "adjacent" zones in the table. Requests (insert,

[1] Private communication with Yin Zhang and Vern Paxson

lookup, or delete) for a particular key are routed by intermediate CAN nodes towards the CAN node whose zone contains that key. Our CAN design is completely distributed (it requires no form of centralized control, coordination or configuration), scalable (nodes maintain only a small amount of control state that is independent of the number of nodes in the system), and fault-tolerant (nodes can route around failures). Unlike systems such as the DNS or IP routing, our design does not impose any form of rigid hierarchical naming structure to achieve scalability. Finally, our design can be implemented entirely at the application level.

In what follows, we describe our basic design for a CAN in Section 2, describe and evaluate this design in more detail in Section 3 and discuss our results in Section 4. We discuss related work in Section 5, directions for future work in Section 6.

## 2 Design

First we will describe our Content Addressable Network in its most basic form; in Section 3 we will add additional design features that improve system performance.

Our design centers around a virtual $d$-dimensional Cartesian coordinate space on a $d$-torus[2]. This coordinate space is completely logical and bears no relation to any physical coordinate system. At any point in time, the *entire* coordinate space is dynamically partitioned among all the nodes in the system such that every node "owns" its individual, distinct zone within the overall space. For example, Figure 1 shows a 2-dimensional $[0,1] \times [0,1]$ coordinate space partitioned between 5 CAN nodes.

This virtual coordinate space is used to store (key,value) pairs as follows: to store a pair ($K_1$,$V_1$), key $K_1$ is deterministically mapped onto a point $P$ in the coordinate space using a uniform hash function. The corresponding key-value pair is then stored at the node that owns the zone within which the point $P$ lies. To retrieve an entry corresponding to key $K_1$, any node can apply the same deterministic hash function to map $K_1$ onto point $P$ and then retrieve the corresponding value from the point $P$. If the point $P$ is not owned by the requesting node or its immediate neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone $P$ lies. Efficient routing is therefore a critical aspect of our CAN.

Nodes in the CAN self-organize into an overlay network that represents this virtual coordinate space. A node learns and maintains as its set of neighbors the IP addresses of those nodes that hold coordinate zones adjoining its own zone. This set of immediate neighbors serves as a coordinate routing table that enables routing between arbitrary points in the coordinate space.

We will first describe the three most basic pieces of our design: CAN routing, construction of the CAN coordinate overlay, and maintenance of the CAN overlay. We will then, in Section 3, discuss additional design pieces that greatly enhance system performance and robustness.

### 2.1 Routing in a CAN

Intuitively, routing in a Content Addressable Network works by following the straight line path through the Cartesian space from source to destination coordinates.

A CAN node maintains a coordinate routing table that holds the IP address and virtual coordinate zone of each of its neighbors in the coordinate space. In a $d$-dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along $d-1$ dimensions and abut along one dimension. For example, in Figure 2, node 5 is a neighbor of node 1 because its coordinate zone overlaps with 1's along the Y axis and abuts along the X-axis. On the other hand, node 6 is not a neighbor of 1 because their coordinate zones abut along both the X and Y axes. This purely local neighbor state is sufficient to route between two arbitrary points in the space: A CAN message includes the destination coordinates. Using its neighbor coordinate set, a node routes a message towards its destination by simple greedy forwarding to the neighbor with coordinates closest to the destination coordinates. Figure 2 shows a sample routing path.

For a $d$ dimensional space partitioned into $n$ equal zones, the average routing path length is thus $(d/4)(n^{1/d})$ and individual nodes maintain $2d$ neighbors. [3] These scaling results mean that for a $d$ dimensional space, we can grow the number of nodes (and hence zones) without increasing per node state while the path length grows as $O(n^{1/d})$.

Note that many different paths exist between two points in the space and so, even if one or more of a node's neighbors were to crash, a node would automatically route along the next best available path.

If however, a node loses all its neighbors in a certain direction, and the repair mechanisms described in Section 2.3 have not yet rebuilt the void in the coordinate space, then greedy forwarding may temporarily fail. In this case, a node may use an expanding ring search to locate a node that is closer to the destination than itself. The message is then forwarded to this closer node, from which greedy forwarding is resumed.

### 2.2 CAN construction

As described above, the entire CAN space is divided amongst the nodes currently in the system. To allow the CAN to grow incrementally, a new node that joins the system must be allocated its own portion of the coordinate space. This is done by an existing node splitting its allocated zone in half, retaining half and handing the other half to the new node.

The process takes three steps:

1. First the new node must find a node already in the CAN.
2. Next, using the CAN routing mechanisms, it must find a node whose zone will be split.
3. Finally, the neighbors of the split zone must be notified so that routing can include the new node.

#### Bootstrap

A new CAN node first discovers the IP address of any node currently in the system. The functioning of a CAN does not depend on the details of how this is done, but we use the same bootstrap mechanism as Yallcast and YOID [8].

As in [8] we assume that a CAN has an associated DNS domain name, and that this resolves to the IP address of one or more CAN bootstrap nodes. A bootstrap node maintains a partial list of CAN

[2] For simplicity, the illustrations in this paper do not show a torus, so the reader must remember that the coordinate space wraps

[3] Several recently proposed routing algorithms for location services [21, 5] route in $O(\log n)$ hops with each node maintaining $O(\log n)$ neighbors. Notice that were we to select the number of dimensions $d$=$(\log_2 n)/2$, we could achieve the same scaling properties.We choose to hold $d$ fixed independent of $n$, since we envision applying CANs to very large systems with frequent topology changes. In such systems, it is important to keep the number of neighbors independent of the system size

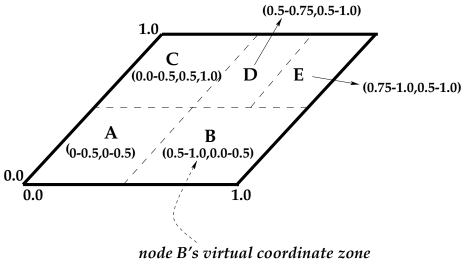
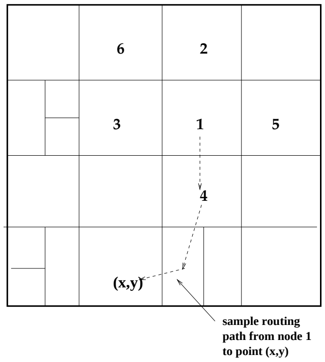
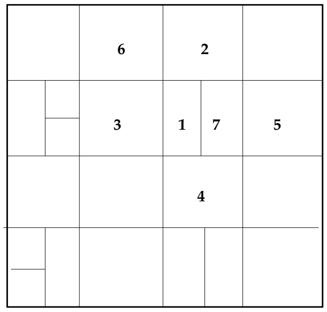


Figure 1: *Example 2-d coordinate overlay with 5 nodes*



*1's coordinate neighbor set = {2,3,4,5}*
*7's coordinate neighbor set = { }*

Figure 2: *Example 2-d space before node 7 joins*



*1's coordinate neighbor set = {2,3,4,7}*
*7's coordinate neighbor set = {1,2,4,5}*

Figure 3: *Example 2-d space after node 7 joins*

nodes it believes are currently in the system. Simple techniques to keep this list reasonably current are described in [8].

To join a CAN, a new node looks up the CAN domain name in DNS to retrieve a bootstrap node's IP address. The bootstrap node then supplies the IP addresses of several randomly chosen nodes currently in the system.

### Finding a Zone

The new node then randomly chooses a point $P$ in the space and sends a JOIN request destined for point $P$. This message is sent into the CAN via any existing CAN node. Each CAN node then uses the CAN routing mechanism to forward the message, until it reaches the node in whose zone $P$ lies.

This current occupant node then splits its zone in half and assigns one half to the new node. The split is done by assuming a certain ordering of the dimensions in deciding along which dimension a zone is to be split, so that zones can be re-merged when nodes leave. For a 2-d space a zone would first be split along the X dimension, then the Y and so on. The (key, value) pairs from the half zone to be handed over are also transfered to the new node.

### Joining the Routing

Having obtained its zone, the new node learns the IP addresses of its coordinate neighbor set from the previous occupant. This set is a subset of the the previous occupant's neighbors, plus that occupant itself. Similarly, the previous occupant updates its neighbor set to eliminate those nodes that are no longer neighbors. Finally, both the new and old nodes' neighbors must be informed of this reallocation of space. Every node in the system sends an immediate update message, followed by periodic refreshes, with its currently assigned zone to all its neighbors. These soft-state style updates ensure that all of their neighbors will quickly learn about the change and will update their own neighbor sets accordingly. Figures 2 and 3 show an example of a new node (node 7) joining a 2-dimensional CAN.

As can be inferred, the addition of a new node affects only a small number of existing nodes in a very small locality of the coordinate space. The number of neighbors a node maintains depends only on the dimensionality of the coordinate space and is independent of the total number of nodes in the system. Thus, node insertion affects only *O(number of dimensions)* existing nodes which is important for CANs with huge numbers of nodes.

## 2.3 Node Departure, Recovery and CAN Maintenance

When nodes leave a CAN, we need to ensure that the zones they occupied are taken over by the remaining nodes. The normal procedure for doing this is for a node to explicitly hand over its zone and the associated (key,value) database to one of its neighbors. If the zone of one of the neighbors can be merged with the departing node's zone to produce a valid single zone, then this is done. If not, then the zone is handed to the neighbor whose current zone is smallest, and that node will then temporarily handle both zones.

The CAN also needs to be robust to node or network failures, where one or more nodes simply become unreachable. This is handled through an immediate takeover algorithm that ensures one of the failed node's neighbors takes over the zone. However in this case the (key,value) pairs held by the departing node would be lost until the state is refreshed by the holders of the data [4].

Under normal conditions a node sends periodic update messages to each of its neighbors giving its zone coordinates and a list of its neighbors and their zone coordinates. The prolonged absence of an update message from a neighbor signals its failure.

Once a node has decided that its neighbor has died it initiates the takeover mechanism and starts a takeover timer running. Each neighbor of the failed node will do this independently, with the timer initialized in proportion to the volume of the node's own zone. When the timer expires, a node sends a TAKEOVER message conveying its own zone volume to all of the failed node's neighbors.

On receipt of a TAKEOVER message, a node cancels its own timer if the zone volume in the message is smaller that its own zone volume, or it replies with its own TAKEOVER message. In this way, a neighboring node is efficiently chosen which is still alive, and which has a small zone volume [5].

Under certain failure scenarios involving the simultaneous failure of multiple adjacent nodes, it is possible that a node detects a failure, but that less than half of the failed node's neighbors are still reachable. If it takes over under these circumstances, it is possible

[4] To prevent stale entries as well as to refresh lost entries, nodes that insert (key,value) pairs into the CAN should periodically refresh these entries

[5] Additional metrics such as load or the quality of connectivity can also be taken into account, but in the interests of simplicity we won't discuss these further here.

for the CAN state to become inconsistent. In such cases, prior to triggering the repair mechanism, the node performs an expanding ring search for any nodes residing beyond the failure region and hence it eventually rebuilds sufficient neighbor state to initiate a takeover safely.

Finally, both the normal leaving procedure and the immediate takeover algorithm can result in a node holding more than one zone. To prevent repeated further fragmentation of the space, a background zone-reassignment algorithm, which we describe in Appendix A, runs to ensure that the CAN tends back towards one zone per node.

## 3 Design Improvements

Our basic CAN algorithm as described in the previous section provides a balance between low per-node state ($O(d)$ for a $d$ dimensional space) and short path lengths with $O(dn^{1/d})$ hops for $d$ dimensions and $n$ nodes. This bound applies to the number of hops in the CAN path. These are *application level* hops, not IP-level hops, and the latency of each hop might be substantial; recall that nodes that are adjacent in the CAN might be many miles (and many IP hops) away from each other. The average total latency of a lookup is the average number of CAN hops times the average latency of each CAN hop. We would like to achieve a lookup latency that is comparable (within a small factor) to the underlying IP path latencies (between the requester and the CAN node holding the key).

In this section, we describe a number of design techniques whose primary goal is to reduce the latency of CAN routing. Not unintentionally, many of these techniques offer the additional advantage of improved CAN robustness both in terms of routing and data availability. In a nutshell, our strategy in attempting to reduce path latency is to reduce either the path length or the per-hop latency because the overall CAN path latency depends directly on the path length (*i.e.* number of hops) and the latency of each hop along the path. A final improvement we make to our basic design is to add simple load balancing mechanisms (described in Sections 3.7 and 3.8).

First, we describe and evaluate each design feature individually and then, in Section 4, discuss how together they affect the overall performance. These added features yield significant improvements but come at the cost of increased per-node state (although per-node state still remains independent of the number of nodes in the system) and somewhat increased complexity. The extent to which the following techniques are applied (if at all) involves a trade-off between improved routing performance and system robustness on the one hand and increased per-node state and system complexity on the other. Until we have greater deployment experience, and know the application requirements better, we are not prepared to decide on these tradeoffs.

We simulated our CAN design on Transit-Stub (TS) topologies using the GT-ITM topology generator [26]. TS topologies model networks using a 2-level hierarchy of routing domains with transit domains that interconnect lower level stub domains.

### 3.1 Multi-dimensioned Coordinate Spaces

The first observation is that our design does not restrict the dimensionality of the coordinate space. Increasing the dimensions of the CAN coordinate space reduces the routing path length, and hence the path latency, for a small increase in the size of the coordinate routing table.

Figure 4 measures this effect of increasing dimensions on routing path length. We plot the path length for increasing numbers of CAN nodes for coordinate spaces with different dimensions. For a system with $n$ nodes and $d$ dimensions, we see that the path length scales as $O(d(n^{1/d}))$ in keeping with the analytical results for perfectly partitioned coordinate spaces.

Because increasing the number of dimensions implies that a node has more neighbors, the routing fault tolerance also improves as a node now has more potential next hop nodes along which messages can be routed in the event that one or more neighboring nodes crash.

### 3.2 Realities: multiple coordinate spaces

The second observation is that we can maintain multiple, independent coordinate spaces with each node in the system being assigned a different zone in each coordinate space. We call each such coordinate space a "reality". Hence, for a CAN with $r$ realities, a single node is assigned $r$ coordinate zones, one on every reality and holds $r$ independent neighbor sets.

The contents of the hash table are replicated on every reality. This replication improves data availability. For example, say a pointer to a particular file is to be stored at the coordinate location (x,y,z). With three independent realities, this pointer would be stored at 3 different nodes corresponding to the coordinates (x,y,z) on each reality and hence it is unavailable only when all three nodes are unavailable.

Further, because the contents of the hash table are replicated on every reality, routing to location (x,y,z) translates to reaching (x,y,z) on *any* reality. A given node owns one zone per reality each of which is at a distinct, and possibly distant, location in the coordinate space. Thus, an individual node has the ability to, in a single hop, reach distant portions of the coordinate space thereby greatly reducing the average path length. To forward a message, a node now checks all its neighbors on each reality and forwards the message to that neighbor with coordinates closest to the destination. Figure 5 plots the path length for increasing numbers of nodes for different numbers of realities. From the graph, we see that realities greatly reduce path length. Thus, using multiple realities reduces the path length and hence the overall CAN path latency. Finally, multiple realities improve routing fault tolerance, because in the case of a routing breakdown on one reality, messages can continue to be routed using the remaining realities.

#### Multiple dimensions versus multiple realities

Increasing either the number of dimensions or realities results in shorter path lengths and higher per-node state. Here we compare the relative improvements caused by each of these features.

Figure 6 plots the path length versus the average number of neighbors maintained per node for increasing numbers of dimensions and realities. We see that for the same number of neighbors, increasing the dimensions of the space yields shorter path lengths than increasing the number of realities. One should not, however, conclude from these tests that multiple dimensions are more valuable than multiple realities because multiple realities offer other benefits such as improved data availability and fault-tolerance. Rather, the point to take away is that if one were willing to incur an increase in the average per-node neighbor state for the primary purpose of improving routing efficiency, then the right way to do so would be to increase the dimensionality $d$ of the coordinate space, rather than the number of realities $r$.

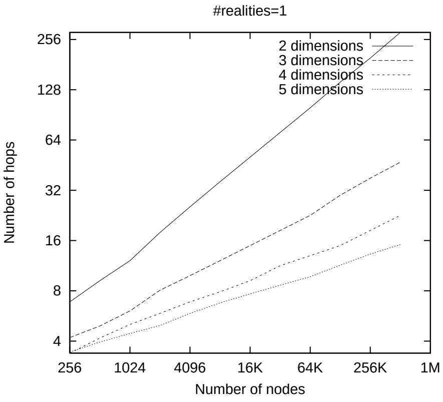


Figure 4: *Effect of dimensions on path length*

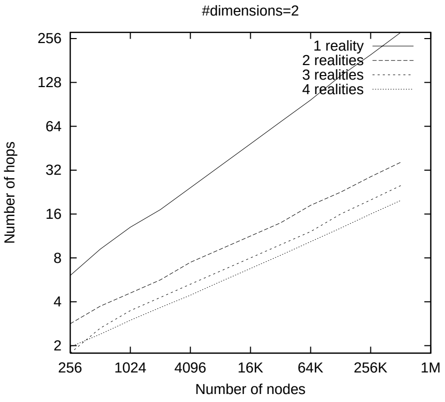


Figure 5: *Effect of multiple realities on path length*

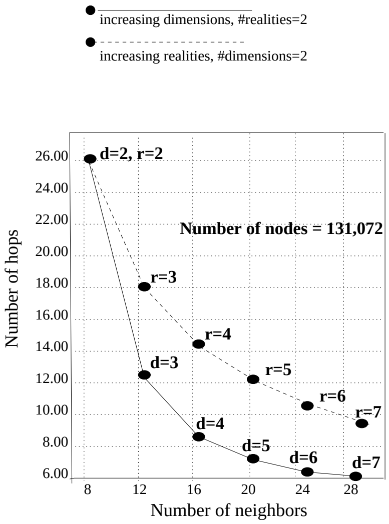


Figure 6: *Path length with increasing neighbor state*

## 3.3 Better CAN Routing Metrics

The routing metric, as described in Section 2.1, is the progress in terms of Cartesian distance made towards the destination. One can improve this metric to better reflect the underlying IP topology by having each node measure the network-level round-trip-time (*RTT*) to each of its neighbors. For a given destination, a message is forwarded to the neighbor with the maximum ratio of progress to *RTT*. This favors lower latency paths, and helps the application level CAN routing avoid unnecessarily long hops.

Unlike increasing the number of dimensions or realities, *RTT* weighted routing aims at reducing the latency of individual hops along the path and not at reducing the path length. Thus, our metric for evaluating the efficacy of *RTT* weighted routing is the per-hop latency, obtained by dividing the overall path latency by the path length.

To quantify the effect of this routing metric, we used Transit-Stub topologies with link latencies of 100ms for intra-transit domain links, 10ms for stub-transit links and 1ms for intra-stub domain links. With our simulated topology, the average end-to-end latency of the underlying IP network path between randomly selected source-destination nodes is approximately 115ms. Table 1 compares the average per-hop latency with and without *RTT* weighting. These latencies were averaged over test runs with $n$, the number of nodes in the CAN, ranging from $2^8$ to $2^{18}$.

As can be seen, while the per-hop latency without *RTT* weighted routing matches the underlying average IP network latency, *RTT* weighted routing lowers the per-hop latency by between 24 - 40% depending on the number of dimensions. Higher dimensions give more next-hop forwarding choices and hence even greater improvements.

## 3.4 Overloading coordinate zones

So far, our design assumes that a zone is, at any point in time, assigned to a single node in the system. We now modify this to allow multiple nodes to share the same zone. Nodes that share the same zone are termed peers. We define a system parameter MAXPEERS, which is the maximum number of allowable peers per zone (we imagine that this value would typically be rather low, 3 or 4 for example).

With zone overloading, a node maintains a list of its peers in addition to its neighbor list. While a node must know all the peers in its own zone, it need not track all the peers in its neighboring zones. Rather, a node selects one node from each of its neighboring zones. Thus, zone overloading does not increase the amount of *neighbor* information an individual node must hold, but does require it to hold additional state for up to MAXPEERS peer nodes.

Overloading a zone is achieved as follows: When a new node $A$ joins the system, it discovers, as before, an existent node $B$ whose zone it is meant to occupy. Rather than directly splitting its zone as described earlier, node $B$ first checks whether it has fewer than MAXPEERS peer nodes. If so, the new node $A$ merely joins $B$'s zone without any space splitting. Node $A$ obtains both its peer list and its list of coordinate neighbors from $B$. Periodic soft-state updates from $A$ serve to inform $A$'s peers and neighbors about its entry into the system.

If the zone is full (already has MAXPEERS nodes), then the zone is split into half as before. Node $B$ informs each of the nodes on it's peer-list that the space is to be split. Using a deterministic rule (for example the ordering of IP addresses), the nodes on the peer list together with the new node $A$ divide themselves equally between the two halves of the now split zone. As before, $A$ obtains its initial list of peers and neighbors from $B$.

Periodically, a node sends its coordinate neighbor a request for its list of peers, then measures the *RTT* to all the nodes in that neighboring zone and retains the node with the lowest *RTT* as its neighbor in that zone. Thus a node will, over time, measure the round-trip-time to all the nodes in each neighboring zone and retain the closest (*i.e.* lowest latency) nodes in its coordinate neighbor set. After its initial bootstrap into the system, a node can perform this *RTT* measurement operation at very infrequent intervals so as to not unnecessarily generate large amounts of control traffic.

The contents of the hash table itself may be either divided or replicated across the nodes in a zone. Replication provides higher

| Number of dimensions | Non-*RTT* weighted routing (ms) | *RTT* weighted routing (ms) |
|---|---|---|
| 2 | 116.8 | 88.3 |
| 3 | 116.7 | 76.1 |
| 4 | 115.8 | 71.2 |
| 5 | 115.4 | 70.9 |

Table 1: *Per-hop latency using RTT weighted routing*

| Number of nodes per zone | per-hop latency (ms) |
|---|---|
| 1 | 116.4 |
| 2 | 92.8 |
| 3 | 72.9 |
| 4 | 64.4 |

Table 2: *Per-hop latencies using multiple nodes per zone*

availability but increases the size of the data store at every node by a factor of MAXPEERS (because the overall space is now partitioned into fewer, and hence larger, zones) and data consistency must be maintained across peer nodes. On the other hand, partitioning data among a set of peer nodes does not require consistency mechanisms or increased data storage but does not improve availability either.

Overloading zones offers many advantages:

- reduced path length (number of hops), and hence reduced path latency, because placing multiple nodes per zone has the same effect as reducing the number of nodes in the system.
- reduced per-hop latency because a node now has multiple choices in its selection of neighboring nodes and can select neighbors that are closer (in terms of latency). Table 2 lists the average per-hop latency for increasing MAXPEERS for system sizes ranging from $2^8$ to $2^{18}$ nodes with the same Transit-Stub simulation topologies as in Section 3.3. We see that placing 4 nodes per zone can reduce the per-hop latency by about 45%.
- improved fault tolerance because a zone is vacant only when *all* the nodes in a zone crash simultaneously (in which case the repair process of Section 2.3 is still required).

On the negative side, overloading zones adds somewhat to system complexity because nodes must additionally track a set of peers.

### 3.5 Multiple hash functions

For improved data availability, one could use $k$ different hash functions to map a single key onto $k$ points in the coordinate space and accordingly replicate a single (key,value) pair at $k$ distinct nodes in the system. A (key,value) pair is then unavailable only when all $k$ replicas are simultaneously unavailable. In addition, queries for a particular hash table entry could be sent to all $k$ nodes in parallel thereby reducing the average query latency. Figure 7 plots this query latency (*i.e.* the time to fetch a (key,value) pair) for increasing number of nodes for different numbers of hash functions.

Of course, these advantages come at the cost of increasing the size of the (key,value) database and query traffic (in the case of parallel queries) by a factor of $k$.

Instead of querying all $k$ nodes, a node might instead choose to retrieve an entry from that node which is closest to it in the coordinate space.

### 3.6 Topologically-sensitive construction of the CAN overlay network

The CAN construction mechanisms described in Section 2.2 allocates nodes to zones at random, and so a node's neighbors on the CAN need not be topologically nearby on the underlying IP network. This in turn can lead to seemingly strange routing scenarios where, for example, a CAN node in Berkeley has its neighbor nodes in Europe and hence its path to a node in Stanford may traverse distant nodes in Europe. While the design mechanisms described in the previous sections try to improve the selection of paths on an existing overlay network they do not try to improve the overlay network structure itself. In this section, we present some initial results on our current work on trying to construct CAN topologies that are congruent with the underlying IP topology.

Our initial scheme assumes the existence of a well known set of machines (for example, the DNS root name servers) that act as landmarks on the Internet. We achieve a form of "distributed binning" of CAN nodes based on their relative distances from this set of landmarks. Every CAN node measures its round-trip-time to each of these landmarks and orders the landmarks in order of increasing RTT. Thus, based on its delay measurements to the different landmarks, every CAN node has an associated ordering. With $m$ landmarks, $m!$ such orderings are possible. Accordingly we partition the coordinate space into $m!$ equal sized portions, each corresponding to a single ordering. Our current scheme to partition the space into $m!$ portions works as follows: assuming a fixed cyclical ordering of the dimensions (*e.g.* xyzxyzx...), we first divide the space, along the first dimension, into $m$ portions, each portion is then sub-divided along the second dimension into $m - 1$ portions each of which is further divided into $m - 2$ portions and so on.

Previously, a new node joined the CAN at a random point in the entire coordinate space. Now, a new node joins the CAN at a random point in that portion of the coordinate space associated with its landmark ordering.

The rationale behind this scheme is that topologically close nodes are likely to have the same ordering and consequently, will reside in the same portion of the coordinate space and hence neighbors in the coordinate space are likely to be topologically close on the Internet.

The metric we use to evaluate the above binning scheme is the ratio of the latency on the CAN network to the average latency on the IP network. We call this the latency *stretch*. Figure 8 compares the stretch on CANs constructed with and without the above landmark ordering scheme. We use the same Transit-Stub topologies as before (Section 3.3) and 4 landmarks placed at random with the only restriction that they must be at least 5 hops away from each other. As can be seen, landmark ordering greatly improves the path latency.

A consequence of the above binning strategy is that the coordinate space is no longer uniformly populated. Because some orderings (bins) are more likely to occur than others their corresponding portions of the coordinate space are also more densely occupied than others leading to a slightly uneven distribution of load amongst the nodes. The use of background load balancing techniques (as described in Appendix A) where an overloaded node hands off a portion of his space to a more lightly loaded one could be used to alleviate this problem.

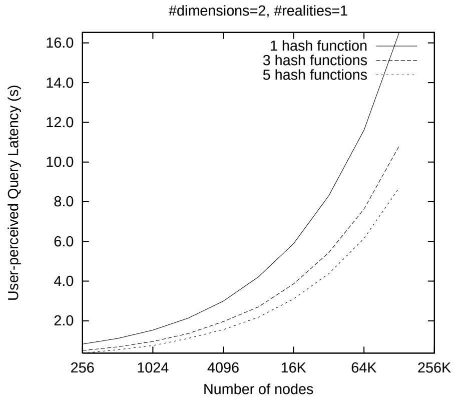


Figure 7: *Reduction in user-perceived query latency with the use of multiple hash functions*

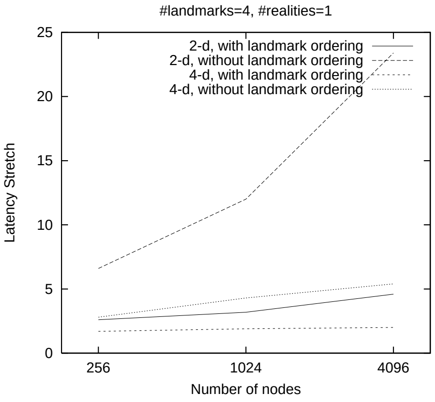


Figure 8: *Latency savings due to Landmark ordering used in CAN construction*

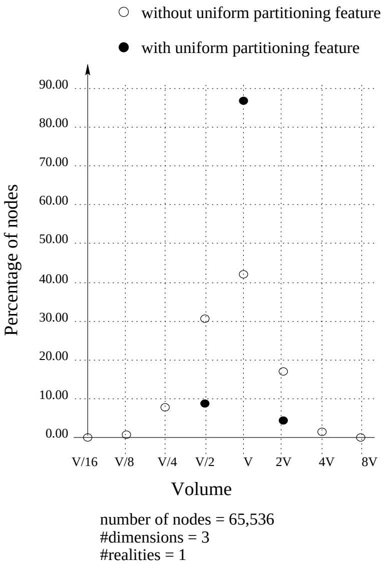


Figure 9: *Effect of Uniform Partitioning feature*

These results seem encouraging and we are continuing to study the effect of topology, link delay distribution, number of landmarks and other factors on the above scheme. Landmark ordering is work in progress which we do not discuss further (nor make use of) in this paper.

### 3.7 More Uniform Partitioning

When a new node joins, a JOIN message is sent to the owner of a random point in the space. This existing node knows not only its own zone coordinates, but also those of its neighbors. Therefore, instead of directly splitting its own zone, the existing occupant node first compares the volume of its zone with those of its immediate neighbors in the coordinate space. The zone that is split to accommodate the new node is then the one with the largest volume.

This volume balancing check thus tries to achieve a more uniform partitioning of the space over all the nodes and can be used with or without the landmark ordering scheme from Section 3.6. Since (key,value) pairs are spread across the coordinate space using a uniform hash function, the volume of a node's zone is indicative of the size of the (key,value) database the node will have to store, and hence indicative of the load placed on the node. A uniform partitioning of the space is thus desirable to achieve load balancing.

Note that this is not sufficient for true load balancing because some (key,value) pairs will be more popular than others thus putting higher load on the nodes hosting those pairs. This is similar to the "hot spot" problem on the Web. In Section 3.8 we discuss caching and replication techniques that can be used to ease this hot spot problem in CANs.

If the total volume of the entire coordinate space were $V_T$ and $n$ the the total number of nodes in the system then a perfect partitioning of the space among the $n$ nodes would assign a zone of volume $V_T/n$ to each node. We use $V$ to denote $V_T/n$. We ran simulations with $2^{16}$ nodes both with and without this uniform partitioning feature. At the end of each run, we compute the volume of the zone assigned to each node. Fig 9 plots different possible volumes in terms of $V$ on the X axis and shows the percentage of the total number of nodes (Y axis) that were assigned zones of a particular volume. From the plot, we can see that without the uniform partitioning feature a little over 40% of the nodes are assigned to zones with volume $V$ as compared to almost 90% with this feature and the largest zone volume drops from $8V$ to $2V$. Not surprisingly, the partitioning of the space further improves with increasing dimensions.

### 3.8 Caching and Replication techniques for "hot spot" management

As with files in the Web, certain (key,value) pairs in a CAN are likely to be far more frequently accessed than others thus overloading nodes that hold these popular data keys. To make very *popular* data keys widely available, we borrow some of the caching and replication techniques commonly applied to the Web.

- Caching: In addition to its primary data store (*i.e.* those data keys that hash into its coordinate zone), a CAN node maintains a cache of the data keys it recently accessed. Before forwarding a request for a data key towards its destination, a node first checks whether the requested data key is in its own cache and if so, can itself satisfy the request without forwarding it any further. Thus, the number of caches from which a data key can be served grows in direct proportion to its popularity and the very act of requesting a data key makes it more widely available.

- Replication: A node that finds it is being overloaded by requests for a particular data key can replicate the data key at each of its neighboring nodes. Replication is thus an active pushing out of popular data keys as opposed to caching which is a natural consequence of requesting a data key. A popular data key is thus eventually replicated within a region surrounding the original storage node. A node holding a replica of a requested data key can with a certain probability choose to either satisfy the request or forward it on its way thereby causing the load to be spread over the entire region rather than just along the periphery.

As with all such schemes, cached and replicated data keys should have an associated time-to-live field and be eventually expired from the cache.

## 4 Design Review

Sections 2 and 3 described and evaluated individual CAN design components. The evaluation of our CAN recovery algorithms (using both large scale and smaller scale *ns* simulations), are presented in Appendix B due to space restrictions. Here we briefly recap our design parameters and metrics, summarize the effect of each parameter on the different metrics and quantify the performance gains achieved by the cumulative effect of all the features.

We used the following metrics to evaluate system performance:

- **Path length**: the number of (application-level) hops required to route between two points in the coordinate space.
- **Neighbor-state**: the number of CAN nodes for which an individual node must retain state.
- **Latency**: we consider both the end-to-end latency of the total routing path between two points in the coordinate space and the per-hop latency i.e. latency of individual application level hops obtained by dividing the end-to-end latency by the path length.
- **Volume**: the volume of the zone to which a node is assigned, that is indicative of the request and storage load a node must handle.
- **Routing fault tolerance**: the availabilty of multiple paths between two points in the CAN.
- **Hash table availability**: adequate replication of a (key,value) entry to withstand the loss of one (or more) replicas.

The key design parameters affecting system performance are:

- dimensionality of the virtual coordinate space: $d$
- number of realities: $r$
- number of peer nodes per zone: $p$
- number of hash functions (*i.e.* number of points per reality at which a (key,value) pair is stored): $k$
- use of the *RTT* weighted routing metric
- use of the Uniform Partitioning feature described in section 2.2

In some cases, the effect of a design parameter on certain metrics can be directly inferred from the algorithm; in all other cases we resorted to simulation. Table 3 summarizes the relationship between the different parameters and metrics. A table entry marked "-" indicates that the given parameter has no real effect on that metric, while a ↑ and ↓ indicates an increase and decrease respectively in that measure caused by the corresponding parameter. The figure numbers included in certain table entries refer to the corresponding simulation results.

To measure the cumulative effect of all the above features, we selected a system size of $n$=$2^{17}$ nodes and compared two algorithms (using the same Transit-Stub topology as before):

1. a bare bones CAN that does not utilize most of our additional design features
2. a "knobs-on-full" CAN making full use of our added features (without the Landmark Ordering feature from Section 3.6)

Table 3: *Effect of design parameters on performance metrics*

| Design Parameters | path length | neighbor state | total path latency | per-hop latency | size of data store | routing fault tolerance | data store availability |
|---|---|---|---|---|---|---|---|
| dimensions:$d$ | $O(dn^{1/d})$ (fig: 4) | $O(d)$ | ↓ (due to reduced path length) | - | - | ↑ | - |
| realities:$r$ | ↓ (fig: 5) | $O(r)$ | ↓ (due to reduced path length) | - | $O(r)$ | ↑ | $O(r)$ |
| number of peer nodes per zone:$p$ | $O(1/p)$ | $O(p)$ | ↓ (due to reduced path length and reduced per-hop latency) | ↓ (table: 2) | replicated data store replicated: $O(p)$, partitioned data store: - | ↑ (due to additional backup neighbors) | replicated data store: $O(p)$, partitioned data store: - |
| number of hash functions:$k$ | - | - | ↓ (fig: 7) | - | $O(k)$ | - | $O(k)$ |
| use of *RTT* weighted routing metric | - | - | ↓ (due to reduced per-hop latency) | ↓ (table: 1) | - | - | - |
| use of uniform partitioning feature | reduced variance | reduced variance | - | - | reduced variance (fig: 9) | - | - |

| Parameter | “bare bones” CAN | “knobs on full” CAN |
|---|---|---|
| *d* | 2 | 10 |
| *r* | 1 | 1 |
| *p* | 0 | 4 |
| *k* | 1 | 1 |
| *RTT* weighted routing metric | OFF | ON |
| Uniform partitioning | OFF | ON |
| Landmark ordering | OFF | OFF |

Table 4: *CAN parameters*

| Metric | “bare bones” CAN | “knobs on full CAN” |
|---|---|---|
| path length | 142.0 | 4.899 |
| # neighbors | 4.2 | 24.4 |
| # peers | 0 | 2.95 |
| IP latency | 115ms | 82ms [a] |
| CAN path latency | 19,671ms | 134.77ms |

Table 5: *CAN metrics*

[a] The reason the IP latency is 82ms instead of 115ms is not because the average latency of the physical network is lower but because our CAN algorithm (because of the use of zone overloading and *RTT* weighted routing) automatically retrieves an entry from the closest replica. 82ms represents the average IP network level latency from the retrieving node to this closest replica

Tables 4 and 5 list the values of the parameters and metrics for each test.

We find these results encouraging as they demonstrate that for a system with over 130,000 nodes we can route with a latency that is well within a factor of two of the underlying network latency. The number of neighbors that a node must maintain to achieve this is approximately 28 (24.4 + 2.95) which is definitely on the high side but not necessarily unreasonable. We repeated the above “knobs-on-full” simulation and varied the system size *n* from $2^{14}$ to $2^{18}$ and found, somewhat to our surprise, that as *n* grows, the total path latency grows even slower than $n^{1/d}$ (with $d = 10$ in this case) because although the path grows slightly (from 4.56 hops with $2^{14}$ nodes to 5.0 with $2^{18}$ hops) the latency of the additional hops is lower than the average latency.

Extrapolating this scaling trend and making the pessimistic assumption that the total latency grows with the increase in path length (*i.e.* as $n^{1/10}$) we could potentially scale the size of the system by another $2^{10}$ before seeing the path latency increase to within a factor of four of the underlying network latency.

# 5 Related Work

We categorize related work as related *algorithms* in the literature relevant to data location and related *systems* that involve a data location component.

## 5.1 Related Algorithms

The Distance Vector (DV) and Link State (LS) algorithms used in IP routing require every router to have some level of knowledge (the exact link structure in the case of LS and the distance in hops for DV) of the topology of entire network. Unlike our CAN routing algorithm, DV and LS thus require the widespread dissemination of local topology information. While well suited to IP networks wherein topology changes are infrequent, for networks with frequent topology changes, DV and LS would result in the frequent propagation of routing updates. Because we wanted our CAN design to scale to large numbers of potentially flaky nodes we chose not to use routing schemes such as DV and LS.

Another goal in designing CANs was to have a truly distributed routing algorithm, both because this does not stress a small set of nodes and it avoids a single point of failure. We hence avoided more traditional hierarchical routing algorithms [2, 24, 16, 6].

Perhaps closest in spirit to the CAN routing scheme is the Plaxton algorithm [21]. In Plaxton’s algorithm, every node is assigned a unique $n$ bit label. This $n$ bit label is divided into *l* levels, with each level having $w = n/l$ bits. A node with label, say $xyz$, where x,y and z are $w$ bit digits, will have a routing table with:

- $2^w$ entries of the form: $\boxed{* \; X \; X}$
- $2^w$ entries of the form: $\boxed{x \; * \; X}$
- $2^w$ entries of the form: $\boxed{x \; y \; *}$

where we use the notation $*$ to denote *every* digit in $0, ..., 2^w - 1$, and $X$ to denote *any* digit in $0, ..., 2^w - 1$.

Using the above routing state, a packet is forwarded towards a destination label node by incrementally “resolving” the destination label from left to right. *i.e.* each node forwards a packet to a neighbor whose label matches (from left to right) the destination label in one more digit than its own label does.

For a system with $n$ nodes, Plaxton’s algorithm thus routes in $O(\log n)$ hops and requires a routing table size that is $O(\log n)$. CAN routing by comparison routes in $O(dn^{1/d})$ hops (where $d$ is dimensions) with routing table size $O(dr)$ which is independent of $n$, although as mentioned earlier, setting $d = (\log_2 n)/2$ allows our CAN algorithm to match Plaxton’s scaling properties.

Plaxton’s algorithm addresses many of the same issues we do. As such it was a natural candidate for CANs and, early into our work, we seriously considered using it. However, we decided that it was not well-suited for our application. Because the reasons involve the details of the algorithm, we discuss them at some length in Appendix C.

Algorithms built around the concept of geographic routing [14, 17] are similar to our CAN routing algorithm in that they build around the notion of forwarding messages through a coordinate space. The key difference is that the “space” in their work refers to true physical space because of which there is no neighbor discovery problem (*i.e.* a node’s neighbors are those that lie in its radio range). These algorithms are very well suited to their targeted applications of routing and location services in ad-hoc networks. Applying such algorithms to our CAN problem would require us to construct and maintain neighbor relationships that would correctly mimic geographic space which appears non trivial (for example, GPSR performs certain planarity checks which would be hard to achieve without a physical radio medium). Additionally, such geographic routing algorithms are not obviously extendible to multi-dimensional spaces.

## 5.2 Related Systems

### 5.2.1 Domain Name System

The DNS system in some sense provides the same functionality as a hash table; it stores key value pairs of the form (domain name, IP address). While a CAN could potentially provide a distributed

DNS-like service, the two systems are quite different. In terms of functionality, CANs are more general than the DNS. The current design of the DNS closely ties the naming structure to the manner in which a name is resolved to an IP address, CAN name resolution is truly independent of the naming scheme. In terms of design, the two systems are very different as should be evident from this paper.

#### 5.2.2 OceanStore

The OceanStore project at U.C.Berkeley [15] is building a utility infrastructure designed to span the globe and provide continuous access to persistent information. Servers self-organize into a very large scale storage system. Data in OceanStore can reside at any server within the OceanStore system and hence a data location algorithm is needed to route requests for a data object to an appropriate server. OceanStore uses the Plaxton algorithm as the basis for its data location scheme. The Plaxton algorithm was described above.

#### 5.2.3 Publius

Publius [18] is a Web publishing system that is highly resistant to censorship and provides publishers with a high degree of anonymity. The system consists of publishers who post Publius content to the web, servers that host random-looking content, and retrievers that browse Publius content on the web. The current Publius design assumes the existence of a static, system-wide list of available servers. The self-organizing aspects of our CAN design could potentially be incorporated into the Publius design allowing it to scale to large numbers of servers. We thus view our work as complementary to the Publius project.

#### 5.2.4 Peer-to-peer file sharing systems

Section 1 described the basic operation of the two most widely deployed peer-to-peer file sharing systems; Napster and Gnutella. We now describe a few more systems in this space that use novel indexing schemes. Although many of these systems address additional, related problems such as security, anonymity, keyword searching etc, we focus here on their solutions to the indexing problem.

Freenet [10, 4] is a file sharing application that additionally protects the anonymity of both authors and readers. Freenet nodes hold 3 types of information: keys (which are analogous to web URLs) addresses of other Freenet nodes which are also likely to know about similar keys, and optionally the data corresponding to those keys. A node that receives a request for a key for which it does not know the exact location forwards the request to a Freenet node that it does know about whose keys are closer to the requested key. Results for both successful and failed searches backtrack along the path the request travelled. If a node fails to locate the desired content, it returns a failure message back to its upstream node which will then try an alternate downstream node which is its next best choice. In this way, a request operates as a steepest-ascent hill-climbing search with backtracking. The authors hypothesize that the quality of the routing should improve over time, for two reasons. First, nodes should come to specialize in locating sets of similar keys because a node listed in routing tables under a particular key will tend to receive mostly requests for similar keys and because of backtracking will become better informed in its routing tables about which other nodes carry those keys. Second, nodes should become similarly specialized in storing clusters of files having similar keys. This is because forwarding a request successfully will result in the node itself gaining a copy of the requested file, and most requests will be for similar keys and hence the node will mostly acquire files with similar keys. The scalability of the above algorithm is yet to be fully studied.

Ongoing work at U.C.Berkeley (independent of the OceanStore project described above) [5] looks into developing a peer-to-peer file sharing application using a location algorithm similar to the Plaxton algorithm (although developed independently from the Plaxton work). A novel aspect of their work is the randomization of path selection for improved robustness.

A description and evaluation of these and other file sharing applications can be found at [27]. A key difference between our CAN algorithm and most of these file sharing systems is that under normal operating conditions, content that exists within the CAN can always be located by any other node because there is a clear “home” (point) in the CAN for that content and every other node knows what that home is and how to reach it. With systems such as [4, 5, 11] however it is quite possible that even with every node in the system behaving correctly, content may not be found either because content is beyond the horizon of a particular node [11] or because different nodes have different, inconsistent views of the network [4, 5]. Whether this is an important distinguishing factor depends of course on the nature of an application’s goals.

## 6 Discussion

Our work, so far, addresses two key problems in the design of Content-Addressable Networks; scalable routing and indexing. Our simulation results validate the scalability of our overall design - for a CAN with over 65,000 nodes, we can route with a latency that is less than twice the IP path latency.

Certain additional problems remain to be addressed in realizing a comprehensive CAN system. An important open problem is that of designing a secure CAN that is resistant to denial of service attacks. This is a particularly hard problem because (unlike the Web) a malicious node can act, not only as a malicious client, but also as a malicious server or router. A number of ongoing projects both in research and industry are looking into the problem of building large-scale distributed systems that are both secure and resistant to denial-of-service attacks [18, 15, 4, 5].

Additional related problems that are topics for future work include the extension of our CAN algorithms to handle mutable content, and the design of search techniques [13, 9, 25] such as keyword searching built around our CAN indexing mechanism etc.

Our interest in exploring the scalability of our design, and the difficulty of conducting truly large scale experiments (hundreds of thousands of nodes), led us to initially evaluate our CAN design through simulation. Now that simulation has given us some understanding of the scaling properties of our design, we are now, in collaboration with others, embarking on an implementation project to build a file sharing application that uses a CAN for distributed indexing.

## References

[1] Anonymized for blind reviewing. In *Under submission to HotOS-VIII* (Jan. 2001).

[2] ARPANET WORKING GROUP REQUESTS FOR COMMENT, DDN NETWORK INFORMATION CENTER. *Internet Protocol Specification*. SRI International, Menlo Park, CA, Sept. 1981. RFC-791.

[3] BOLOSKY, W., DOUCEUR, J., ELY, D., AND THEIMER, M. Feasibility of a serverless distributed file system deployed on an existing set

of desktop pcs. In *Proceedings of SIGMETRICS 2000* (Santa Clara, CA, June 2000).

[4] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. Freenet: A Distributed Anonymous Information Storage and Retrieval System. ICSI Workshop on Design Issues in Anonymity and Unobservability, July 2000.

[5] COSTELLO, A. Private communication, Dec. 2000.

[6] CZERWINSKI, S., ZHAO, B., HODES, T., JOSEPH, A., AND KATZ, R. H. An architecture for a secure service discovery service. In *Proceedings of Fifth ACM Conf. on Mobile Computing and Networking (MOBICOM)* (Seattle, WA, 1999), ACM.

[7] FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C.-G., AND ZHANG, L. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of SIGCOMM '95* (Boston, MA, Sept. 1995), ACM, pp. 342–356.

[8] FRANCIS, P. Yoid: Extending the internet multicast architecture. Unpublished paper, available at http://www.aciri.org/yoid/docs/index.html, Apr. 2000.

[9] FRANCIS, P., KAMBAYASHI, T., YA SATO, S., AND SHIMIZU, S. Ingrid: A self-configuring information navigation infrastructure. http://www.ingrid.org/francis/www4/Overview.html.

[10] FREENET. http://freenet.sourceforge.net.

[11] GNUTELLA. http://gnutella.wego.com.

[12] GUTERMAN, J. Gnutella to the Rescue ? Not so Fast, Napster fiends. link to article at http://gnutella.wego.com, Sept. 2000.

[13] INFRASEARCH. http://www.infrasearch.com.

[14] KARP, B., AND KUNG, H. Greedy Perimeter Stateless Routing. In *Proceedings of ACM Conf. on Mobile Computing and Networking (MOBICOM)* (Boston, MA, 2000), ACM.

[15] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. Oceanstore: An Architecture for Global-scale Persistent Storage. In *Proceedings of ASPLOS 2000* (Cambridge, Massachusetts, Nov. 2000).

[16] KUMAR, S., ALAETTINOGLU, C., AND ESTRIN, D. SCOUT: Scalable Object Tracking through Unattended Techniques. In *Proceedings of the Eight IEEE International Conference on Network Protocols* (Osaka, Japan, Nov. 2000).

[17] LI, J., JANNOTTI, J., COUTO, D. D., KARGER, D., AND MORRIS, R. A Scalable Location Service for Geographic Ad-hoc Routing. In *Proceedings of ACM Conf. on Mobile Computing and Networking (MOBICOM)* (Boston, MA, 2000), ACM.

[18] MARC WALDMAN, A. D. R., AND CRANOR, L. F. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceedings of the 9th USENIX Security Symposium* (August 2000), pp. 59–72.

[19] MCCANNE, S., AND FLOYD, S. *The LBNL/UCB Network Simulator*. Lawrence Berkeley Laboratory, University of California, Berkeley.

[20] NAPSTER. http://www.napster.com.

[21] PLAXTON, C., RAJARAM, R., AND RICHA, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (June 1997).

[22] RAMAN, S., AND MCCANNE, S. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proceedings of SIGCOMM '99* (Cambridge, MA, Sept. 1999), ACM.

[23] RAMAN, S., MCCANNE, S., AND SHENKER, S. Asymptotic scaling behavior of global recovery in SRM. In *Proceedings of SIGMETRICS '98/PERFORMANCE '98 Joint International Conference on Measurement and Modeling of Computer Systems* (Madison, WI, June 1998).

[24] REKHTER, Y., AND LI, T. *A Border Gateway Protocol 4 BGP-4*. ARPANET Working Group Requests for Comment, DDN Network Information Center, Mar. 1995. RFC-1771.

[25] WELSH, M., BORISHOV, N., HILL, J., VON BEHREN, R., AND WOO, A. Querying large collections of music for similarity. Technical report, University of California, Berkeley, CA, Nov. 1999.

[26] ZEGURA, E., CALVERT, K., AND BHATTACHARJEE, S. How to Model an Internetwork. In *Proceedings IEEE Infocom '96* (San Francisco, CA, May 1996).

[27] ZEROPAID.COM. File sharing portal at http://www.zeropaid.com.

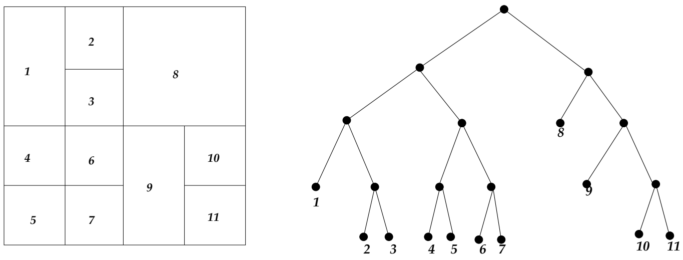


Figure 10: *Example depth-first search for a replacement node*

## A CAN Maintenance: Background zone reassignment

The immediate takeover algorithm described in Section 2.3 may result in a single node being assigned multiple zones. Ideally, we would like to retain a one-to-one assignment of nodes to zones, because this prevents the coordinate space from becoming highly fragmented. To achieve this one-to-one node to zone assignment, we use a simple algorithm that aims at maintaining, even in the face of node failures, a dissection of the coordinate space that could have been created solely by nodes joining the system.

At a general step we can think of each existing zone as a leaf of a binary “partition tree.” The internal vertices in the tree represent zones that no longer exist, but were split at some previous time. The children of a tree vertex are the two zones into which it was split. Of course we don’t maintain this partition tree as a data structure, but it is useful conceptually.

By an abuse of notation, we use the same name for a leaf vertex, for the zone corresponding to that leaf vertex, and for the node responsible for that zone. The partition tree, like any binary tree, has the property that in the subtree rooted at any internal vertex there are two leaves that are siblings.

Now suppose a node wants to hand-off a leaf $x$. If the sibling of this leaf is also a leaf (call it $y$) the hand-off is easy: simply coalesce leaves $x$ and $y$, making their former parent vertex a leaf, and assign node $y$ to that leaf. Thus zones $x$ and $y$ merge into a single zone which is assigned to node $y$.

If $x$’s sibling $y$ is not a leaf, perform a depth-first search in the subtree of the partition tree rooted at $y$ until two sibling leaves are found. Call these leaves $z$ and $w$. Combine $z$ and $w$, making their former parent a leaf. Thus zones $z$ and $w$ are merged into a single zone, which is assigned to node $z$, and node $w$ takes over zone $x$.

Figure 10 illustrates this reassignment process. Let us say node 9 fails and by the immediate takeover algorithm node 6 takes over node 9’s place. By the background reassignment process, node 6 discovers sibling nodes 10 and 11. One of these, say 11 takes over the combined zones 10 and 11, and 10 takes over what was 9’s zone.

While the partition tree data structure helps us explain the required transformations, its global nature makes it unsuitable for actual implementation. Instead we must effect the required transformations using purely local operations. All an individual node actually has is its coordinate routing table which captures the adjacency structure among the current zones (the leaves of the deletion tree). However, this adjacency structure is sufficient for emulation of all the operations on the partition tree

A node $I$ performs the equivalent of the above described depth-first search on the partition as follows:

- let $d_k$ be the last dimension along which node $I$’s zone was halved (this can be easily detected by merely searching for the highest ordered dimension with the shortest coordinate span).
- from its coordinate routing table, node $I$ selects a neighbor node $J$ that abuts $I$ along dimension $d_k$ such that $J$ belongs to the zone that forms the other half to $I$’s zone by the last split along dimension $d_k$.
- if the volume of $J$’s zone equals $I$’s volume, then $I$ and $J$ are a pair of sibling leaf nodes whose zones can be combined.
- If $J$’s zone is smaller than $I$’s then $I$ forwards a depth-first search request to node $J$, which then repeats the same steps.
- This process repeats until a pair of sibling nodes is found.

Our simulation results measure the number of steps a depth-first search request has to travel before sibling leaf nodes can be found.

## B CAN performance with node failures

This section presents simulation results quantifying CAN system performance in the face of node failures.

We first look at the question of how system performance deteriorates with the introduction of node failures in the absence of any recovery algorithms; *i.e.* if a node fails, its zone is left vacant and no takeover algorithms are invoked. Because no attempt is made to recover from node failures these tests represent the worst case scenario. Our metric for performance degradation is the increase in path length with node failures.

As described in section 2.1 a node forwards a packet to its neighbor that makes the maximum progress towards the destination. If, on account of node failures, no neighbor exists that makes forward progress, a node performs an expanding ring search for a node that is closer to the destination than itself. Increase in the routing path length thus depends on how often a node must perform an ERS and, (when it does perform an ERS) on the average radius of the expanding ring search.

Figure 11 plots the probability with which a node must perform an expanding ring search as a function of the number of nodes for different failure rates. Figure 12 plots the search radius at which the ERS successfully terminates and Figure 13 plots the total path length for increasing numbers of nodes and different node failure rates.

To evaluate and capture the dynamics of the recovery algorithms, we implemented our CAN system (together with the immediate takeover algorithm) in *ns* [19]. Because *ns* does not scale to very large scale topology simulations, our results are restricted to simulations with a few hundred nodes. For each of the following simulations, we select a fixed time window and for every simulation run, kill an increasing fraction of the nodes in the system during that fixed time window. We then measure the amount of traffic generated by takeover bids and flooded neighbor discovery requests.

The immediate takeover algorithm uses timer mechanisms to suppress unnecessary takeover bids. Ideally, one would like to have exactly one takeover message transmitted for each node failure. Figure 14 plots the number of takeover bids transmitted as a function of the node failure rate for different size topologies. As can be seen, the amount of takeover traffic is very close to the ideal case. This is because in most cases, the dead node’s sibling (which is the best possible takeover node) immediately transmits a takeover bid that suppresses all other bids.

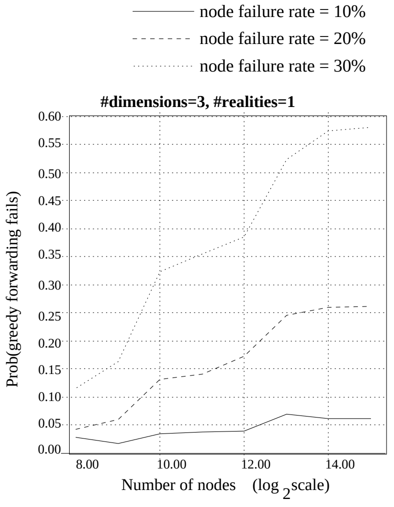


Figure 11: *Routing failure rate without repair*

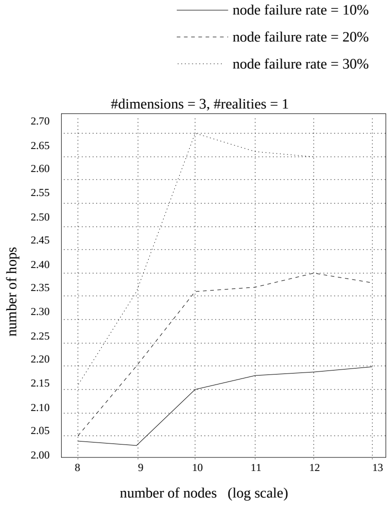


Figure 12: *Search Radius using ERS*

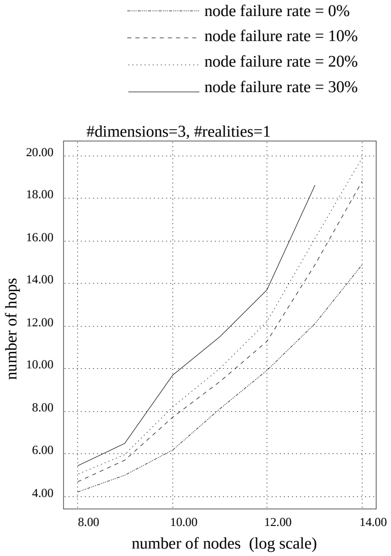


Figure 13: *Path Length using ERS*

| Number of dimensions | avg(# hops) | max(# hops) |
|---|---|---|
| 2 | 1.12 | 3 |
| 3 | 1.09 | 3 |
| 4 | 1.07 | 3 |

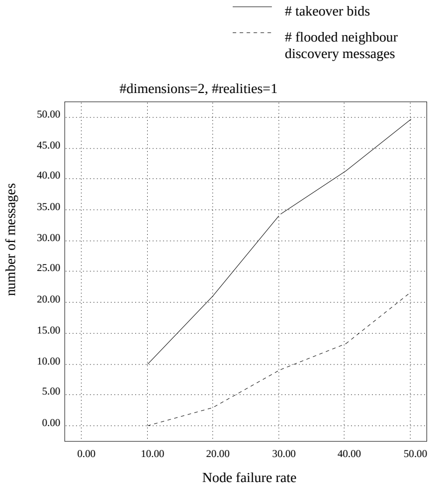
Table 6: *Background zone reassignment*

In the event of complex failure modes where multiple adjacent nodes fail simultaneously, a node might have to resort to a scoped form of flooding to discover its new neighbors. Ideally, we would like such events to be rare. Figure 14 also plots the number of flooded search requests as a function of the node failure rate.

The above *ns* simulations are really intended more as a proof of concept than as a comprehensive evaluation of the recovery algorithms which requires an in-depth understanding of soft-state and timer-based suppression algorithms [7, 23, 22] that is currently beyond the scope of this paper.

The background zone reassignment algorithm described in A requires a node to send out a "depth-first search" query to find a node to which it can hand off one of its extra zones.

Table 6 lists the number of hops away from itself that a node would have to search in order to find a node it can hand off an extra zone to. Because of the more or less uniform partitioning (due to our uniform partitioning feature from Section 3.7) of the space, a pair of sibling nodes is typically available very close to the requesting node (*i.e.* the dissection tree is well balanced).



number of CAN nodes = 100
periodic update interval = 20ms
recovery period = 6s
takeover timer interval selected in the range [1-4]seconds

Figure 14: *Recovery Traffic*

## C Plaxton's Algorithm

The Plaxton algorithm described in Section 5 addresses many of the same issues as our CAN algorithm. Here we describe some of the details of the algorithm that, we believe, make it inappropriate for a large-scale peer-to-peer environment.

The Plaxton algorithm was originally proposed for web caching environments which are typically administratively configured, have fairly stable hosts, and maximal scales on the order of thousands.

While the Plaxton algorithm is well suited to such environments, the peer-to-peer contexts we address are quite different. We require a self-configuring system which is capable of dealing with a very large set of hosts (millions), many of them potentially quite flaky. The Plaxton algorithm does not seem well-suited for such an environment. First, every node arrival and departure affects a logarithmic number of nodes which, for large systems with high arrival and departure rates, appears to be on the high side because nodes could be constantly reacting to changes in system membership. Another potential problem involves the discovery of neighbors. Plaxton's algorithm, because it was designed for administratively configured systems, does not provide a solution whereby nodes can discover their neighbors in a decentralized manner.

This neighbor discovery problem aside, we believe that, for large-scale self-organizing systems, our CAN algorithm is simpler and more robust in the face of fluctuating node membership. This is best understood through an example: Consider a Plaxton system, wherein nodes are assigned 9 bit labels with 3 levels. Continuing with the notation used above, let us say that at some point in time there is no node in the system with a label of the form 13X . All nodes with labels of the form 1XX must store a routing table entry of the form 13X. Since there is no node currently in the system with a label of the form 13X, nodes instead store a pointer to a node currently in the system, with a label that is a good "approximation" of the label 13X (what constitutes a good approximation is defined in [21]). Consider what happens when a node, say 136, enters the system. All nodes of the form 1XX must now be informed about node 136's arrival and update their routing tables. If node 136 were to subsequently crash, every node of the form 1XX must now again locate a good "approximation" node.

The Plaxton algorithm defines rules to determine a unique node currently in the system that is the best approximation to a missing node, however their definition of this approximation node assumes some form of global knowledge of all the nodes currently in the system without which different nodes might independently pick different approximation nodes for a single missing node. These inconsistencies might make data that is actually in the system unreachable by some nodes. Applying Plaxton's algorithm to self-organizing systems thus requires designing a distributed mechanism (which could potentially be quite complex) whereby nodes can independently develop consistent views of the entire network. Because the CAN algorithm always has a fully occupied address space (*i.e.* coordinate space), it does not face any such problems associated with approximations of node labels.

While it might very well be possible to adapt the Plaxton algorithm to our application environment (in fact, discussions with colleagues [6] seem to indicate that constraining the neighbor set in the Plaxton problem might make the above problem of approximating node labels more tractable), we opted to go with our CAN algorithm both because it offers the flexibility of operating with very low per-node state and because of the simplicity of our basic design under normal operation.

[6] Private communication with Ion Stoica