A Scalable Content-Addressable Network

Sylvia Ratnasamy

November 28, 2000

Abstract

In this paper, we define the concept of a Content-Addressable Network; a system that essentially offers the same functionality as a hash table, i.e. it maps "keys" to "values". The novelty of a hash table in a Content-Addressable Network is that it may span millions of hosts across diverse administrative entities in the Internet. We describe our design of a Content Addressable Network that is scalable, highly fault-tolerant and completely self-organizing. We simulate the scalability and robustness properties of our design. Finally, we discuss some of the potential applications for a CAN.

1 Introduction

A hash table is a data structure that efficiently maps "keys" onto "values". Hash tables are widely used in the implementation of software systems and are an invaluable programming tool.

On the Internet, we conjecture that many large-scale, distributed applications and systems could utilise a *network wide* hash table (whether application specific or not) as a core system building block. For example, a distributed hash table could store (key,value) pairs of the form (domain name, IP address of name server), (URL, IP address of web server or cache), (filename, IP addresses of user PCs on the internet storing the file), (Active Badge Id, GPS coordinates of the person wearing the Active Badge) etc.

Specific examples of current Internet systems where distributed hash tables can play an important role include:

- peer-to-peer file sharing systems in the spirit of Napster, Gnutella: the index mapping file names to IP addresses can be stored in a distributed hash table.
- large scale storage management systems like OceanStore, Publius
- Ubiquitous computing environments such as those described in [11] frequently require services such as object tracking, service location etc. Similarly, networks of sensors have large numbers of sensors to be monitored and controlled. The large number of tracked objects in these environments makes it difficult to store and retrieve information related to these objects in a scalable manner. In such environments, distributed hash tables can be used as a form of scalable, robust network storage, storing for example, entries of the form (active badge ID, current GPS coordinates of badge)
- distributed, location independent name resolution services (an enhanced distributed DNS): hash tables could store entries of the form (domain name, IP address of name server), (URN, IP address of web server or cache)

More importantly, the abstraction provided by a hash table, gives internet system developers a powerful new design tool that could enable new applications and communication models.

Conceptually, the notion of a network-wide hash table is quite simple, and yet, how does one design a scalable data structure that millions of nodes can insert or retrieve entries from ?

For such a hash table to scale to Internet dimensions, it must be distributed, highly scalable and tolerant to network and machine failures.

The difficulty of the problem begins to sink in when we consider some of the large scale indexing systems in existence. Centralised solutions like Napster [14] have scalability problems under high load [15], suffer from a single point of failure and are expensive. Peer-to-peer systems such as Gnutella [7] locate content by flooding search requests over a self-organised overlay network. While truly distributed in design, flooding search requests is not scalable [9] and, because the flooding has to be curtailed at some point, may fail to find content that is actually in the system. Systems like the Web and

the DNS impose strict restrictions on how content may be named which brings its own set of problems [1, 3] while systems like the DNS are heavyweight in terms of configuration, maintainance and updates.

In this paper, we define the concept of a Content-Addressable Network; an internet-scale, distributed hash table. Viewed from the outside, the basic operations performed on a CAN are the insertion, lookup and deletion of (key,value) pairs. On the inside, individual CAN nodes store a chunk of the entire hash table. In addition, a node holds information about a small number of other chunks in the table much like IP routers hold state for a tiny fraction of the routers in the Internet. Just as IP routing algorithms enable communication between any two nodes in the network, similarly, we propose a routing algorithm that uses this hash-table state to allow any node in the system to retrieve any portion of the hash table . In designing a CAN, we thus address two basic problems: the first is a storage problem, i.e. how are the contents of the hash table divided amongst individual nodes ? The second problem is more of a routing problem: how does a node, retrieve an arbitrary hash table entry ?

We present the design of a CAN that is completely distributed requiring no form of centralised control, coordination or configuration, scalable because nodes maintain only a small amount of control state that is independent of the number of nodes in the system, and highly fault-tolerant because of built in redundancy and the ability to route around trouble. Unlike systems such as the DNS or IP routing, our design does not impose any form of rigid hierarchical structure to achieve scalability. Finally, our design can be implemented entirely at the application level.

On the downside, as with all self-organising overlay networks, our design faces the problem of factoring in underlying network topology information into application level routing decisions. While we incorporate some simple heuristics in an attempt to capture underlying network characteristics, we cannot guarantee that the performance of the path between two nodes across the CAN network is comparable to that on the underlying IP Internet. Note however that a CAN is primarily used to look up a table entry and not for the actual transfer of large data files; thus this performance hit (which we quantify in section 3) might very well be acceptable to applications.

In what follows, we describe our design for a CAN in section 2 ,present analysis and simulation results in section 3. We discuss related work in section 4, and finally conclude.

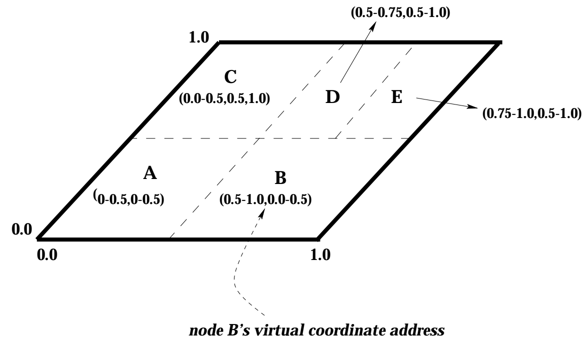


Figure 1: *Example: 2-d coordinate overlay with 5 nodes*

2 Design

In this section, we describe the design of our Content Addressable Network. We start with a description of our algorithm in its most basic, stripped down form and later describe additional design features and components that greatly improve overall system performance.

Our design centers around the notion of a virtual coordinate space. At any point in time, the *entire* virtual coordinate space is dynamically partitioned among all the nodes in the system such that every node is responsible for its own distinct sub-space within the overall space. For example, Figure 1 shows a 2-dimensional $[0, 1] * [0, 1]$ coordinate space partitioned between 5 CAN nodes. Thus, every node in the system "owns" a sub-space within the overall coordinate space.

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 sub-space within which the point P lies. To retrieve an entry corresponding to key K_1 any node merely applies the same hash function to discover the point P and then retrieves the corresponding value from the node within whose sub-space P lies. Note that the ability to route between arbitrary points in the coordinate space implies that any node in the system can retrieve any (key,value) pair i.e. can retrieve any entry in the distributed hash table.

The nodes in our CAN system 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 sub-spaces adjoining its own sub-space. This set of immediate neighbors serves as a coordinate routing table that enables routing between arbitrary points in the coordinate space (we use the exression "routing to a point in the coordinate space" to mean routing to the node the owns the sub-space within which that point lies). Thus, our CAN system works by creating and maintaining a virtual coordinate overlay network wherein individual nodes are assigned distinct sub-spaces of the coordinate space.

In what follows, we first describe the three most basic pieces of our design: CAN routing, construction of the CAN coordinate overlay, and maintainance of the CAN overlay. We then discuss certain 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 from source to destination coordinates.

A CAN node maintains a coordinate routing table, that holds the IP addresses and virtual coordinate sub-space of its neighbors in the coordinate space. This purely local neighbor state is sufficient to route between two arbitrary points in the space. A packet includes the destination coordinates. Using its coordinate neighbor set, a node routes a packet towards it's destination by simple greedy forwarding to a neighbor with coordinates closest to the destination coordinates. Thus, the routing metric, as described above, is the progress (in terms of cartesian distance) made towards the destination. In practice, in order to factor the underlying IP topology into the routing process, a node measures the network level round trip time (rtt) between itself and each of its neighbors and uses the ratio of the progress made to the rtt as routing metric. i.e. for a given destination, a packet is forwarded to the neighbor with the maximum ratio of progress to rtt. This allows the routing process to favor lower latency paths.

Note that many different paths exist between a source and destination coordinate. Hence, 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 uses an expanding ring search to locate a node that is closer to the destination than itself. The packet is then forwarded to this closer node, from which greedy forwarding is resumed.

2.2 CAN construction

As described earlier, at any point in time, the entire CAN coordinate space is divided amongst the nodes currently in the system. A new node that joins the system, must be allocated its own portion of the coordinate space. This is done by having an existent node split its allocated sub-space in half, retaining half its old sub-space while the remaining half is assigned to the new node. This is achieved as follows: A new CAN node must first learn the IP address of any node currently in the system. We achieve this initial bootstrap in a manner similar to that described in [5] (It is worth pointing out however, that all our design requires is that a new node be able to learn of at least one node currently in the CAN and is largely independent of the exact bootstrap mechanism, i.e. if someone comes up with an improved distributed bootstrap method, we could use it)

As in [5] we assume that a CAN has an associated domain name . A CAN also runs a bootstrap host such that the CAN domain name resolves (through DNS) to the IP address of the bootstrap host (for robustness, one could run multiple bootstrap hosts and DNS round-robin between them). A bootstrap node maintains a list of nodes it believes are currently in the system. Simple techniques to keep this list reasonably current are described in [5].

To join a CAN system a new node A starts by performing a DNS lookup on the CAN domain name to retrieve the bootstrap IP address. A then contacts the bootstrap node and retrieves the IP addresses of one or more random nodes currently in the system. We use the notation R_a to denote this set of A's initial contact nodes.

A then picks (at random) a point P with coordinates C_{new}, within the space and requests one or more nodes from R_a to route a JOIN request to the point P. Using the routing algorithm described in section 2.1, the JOIN request is routed to the node in the system that currently owns the sub-space within which point P lies. This current occupant node could then split its space in

half and assign one half to the new node A. To achieve a uniform partitioning of the overall space, we put in an additional volume balancing check at this time: Instead of directly splitting its own space, the current occupant node first compares the volume of its own sub-space with the volume of the sub-spaces occupied by its immediate neighbors in the coordinate space (this is information it already holds locally). The node with the largest volume, say B is then selected and its space is split in two halves of which A is assigned one while B retains the other. In section 3 we evaluate the usefulness of this added volume balancing check.

This volume balancing check thus tries to achieve a uniform partitioning of the total volume over all the nodes in the system. Since (key,value) pairs are spread across the coordinate space using a uniform hash function, the volume of the sub-space a node owns 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 because certain (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. Caching and replication techniques can be used to alleviate this hot spot problem in CANs)

Having obtained its sub-space, A must now learn the IP addresses of its coordinate neighbor set N_a. In a d-dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along $d-1$ dimensions and abut along 1 dimension. (For example, in figure 2, node 1 is a neighbor of B because its coordinate sub-space overlaps with B's along the Y axis and abuts along the X-axis, node 5 on the other hand is not a neighbor of B because their coordinate sub-spaces abut along both the X and Y axes) Thus the set N_a is a subset of the set N_b, node B's coordinate neighbor set. Thus A simply obtains from B its coordinate neighbor set N_b and selects from it those nodes that are now its neighbors. Similarly, B updates its neighbor set to eliminate those nodes that are no longer its neighbors. Finally, A and B's neighbors must be informed of this reallocation of space. Every node in the system periodically sends update messages with its currently assigned coordinates to all its neighbors. These soft-state style updates ensure that all of A and B's neighbors will eventually learn about the entry of new node A and will update their own neighbor sets accordingly. Figures 2 and 3 show an example of a new node joining a 2-d CAN. As can be inferred from the above description, the effects of adding a new node into the system are restricted to a small number of nodes in a very small locality of the coordinate space.

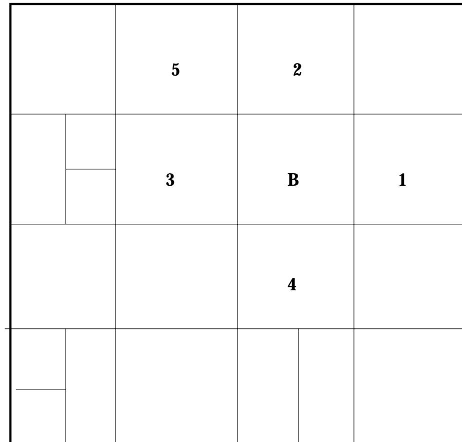


Figure 2: *Example: 2-d coordinate overlay before node A joins*

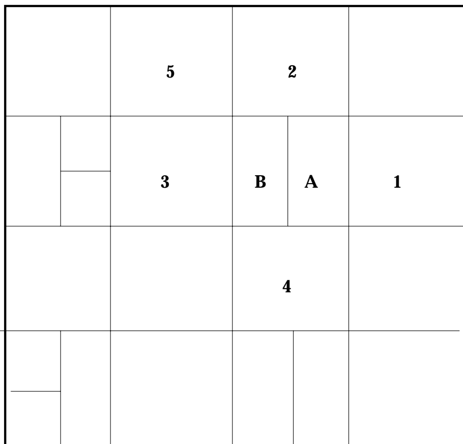
2.3 CAN recovery and maintainance algorithms

The algorithms described in this section are used to handle node failures. Our strategy for dealing with node failures consists of two pieces: the recovery piece is an immediate "quick-fix" takeover algorithm that enables quick recovery from a node failure. The second piece is a background maintainance algorithm that ensures that no single node is assigned a disproportionately large fraction of the overall coordinate space.

2.3.1 Recovery: Immediate takeover algorithm

When a node fails, one of its neighboring nodes "takes over" the failed node's now vacant sub-space.

Our algorithms draw heavily from the soft-state [17] style recovery algorithms described in [4]. Nodes send periodic update messages to their neighbors. These periodic message updates include the node's virtual coordinates, and its list of neighbors (both their IP addresses and coordinate sub-spaces) The absence of up-



B's coordinate neighbor set = {2,3,4}
A's coordinate neighbor set = {1,2,4}

Figure 3: *Example: 2-d coordinate overlay after node A joins*

dates from a node over a predefined period of time (typically some multiple of the update period) is regarded as an indication of that node's failure and triggers the repair mechanisms.

When a node I fails, it stops sending out periodic update messages to its neighbors. Each neighboring node thus independently detects I's failure and sets a takeover timer with timer interval in direct proportion to the volume of its current sub-space. When a node's takeover timer expires it sends out a *TAKEOVER* message to each of node I's neighbors (recall that this information was obtained from I's periodic updates). A *TAKEOVER* message includes the volume of the sub-space of the node initiating the takeover. A takeover bid from a node I is considered *better* than one from node J if I's sub-space volume is lower than J's or if their volume's are equal and $I < J$. Our recovery process thus favors nodes with smaller sub-spaces to avoid having a single node assigned to large fractions of the overall space. Additional metrics such as the load on a node, the quality of its connection to the Internet etc could also be factored into the takeover process.

A node that receives a better takeover bid than its own cancels its takeover timer. Because timer values are selected in proportion to a node's sub-space volume this form of timer cancellation should ensure that many nodes will hear better takeover bids before their own timer expires and can refrain from sending out takeover bids.

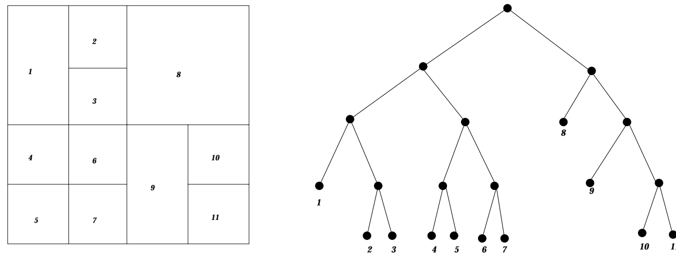


Figure 4: *Effect of dimensions on path length*

For a predefined recovery interval (typically some multiple of the maximum takeover timer interval) a node tracks the best takeover bid it has heard. At the end of the recovery interval the node that made the best takeover bid takes over the sub-space of the failed node.

Under certain failure scenarios such as the simultaneous failure of multiple, adjacent nodes, it is possible that a node A detects the failure of a neighboring node B but does not know node B's neighbors. In such cases, prior to triggering repair mechanisms, node A performs an expanding ring search for the nodes neighboring B's coordinate sub-space.

Note, that the above recovery algorithm need not be invoked every time a node leaves the system. Rather, if a participating node gracefully exits an application, it could, as part of the exit procedure, hand over its state to a neighboring node which would then takeover the departing node's sub-space. The above recovery algorithms are required for true node and/or network failures.

2.3.2 Maintainance: Background sub-space reassignment

The immediate takeover algorithm described above may result in a single node being assigned multiple sub-spaces. Ideally, we would like to retain a one-to-one (or many-to-one in the case of sub-space overloading) assignment of nodes to sub-spaces, both because this reduces the amount of control state a node must maintain and because it prevents the coordinate space from becoming highly fragmented. If for example, the membership of a CAN drops in half, we would like the number of sub-spaces to drop accordingly.

To achieve this one-to-one node to sub-space assignment, we use a simple algorithm that aims at maintain-

ing, 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 sub-space as a leaf of a binary “partition tree.” The internal vertices in the tree represent sub-spaces that no longer exist, but were split at some previous time. The children of a tree vertex are the two sub-spaces 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 sub-space corresponding to that leaf vertex, and for the node responsible for that sub-space. 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 want 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. This corresponds to combining sub-spaces x and y into a single sub-space which is assigned to the node y. If y, the sibling of x, is not a leaf perform a depth-first search in the subtree of the partition tree rooted at y until two leaves that are siblings are found. Call these leaves z and w. Then combine z and w, making their former parent a leaf. This corresponds to combining sub-spaces z and w into a single sub-space. Assign the node z to this combined sub-space and assign node w to sub-space x. Figure 4 shows a simple example of this reassignment process. In fig 4 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 would discover nodes 10 and 11. One of these, say 11 takes over the combined sub-spaces 10 and 11, and 10 takes over what was 9’s sub-space.

Of course we don’t really maintain the partition tree; it is just a conceptual aid. All an individual node actually has is its coordinate routing table which captures the adjacency structure among the current sub-spaces (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 sub-space 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 sub-space that forms the other half to I’s sub-space formed as a result of the last splitting along dimension d_k (gross !!).
- if the volume of J’s sub-space equals I’s volume, then I and J constitute a pair of sibling leaf nodes whose sub-spaces can be combined.
- If J’s sub-space is smaller than I’s then I forwards a depth-first search request to node J, which then repeats the same steps.
- The above process repeats until a pair of sibling nodes is found.

Section 3 measures the number of steps a depth-first search request has to travel before sibling leaf nodes can be found.

2.4 Additional Design components

The previous section, described our CAN design in its most basic, stripped down form. In this section, we point out certain features of the above design as well as certain additional mechanisms that greatly improve the performance and robustness of the above design. Each of the following additions cause quite dramatic improvements in system performance and robustness 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). 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 on the other and should be made to satisfy application specific requirements.

2.4.1 Moving to higher dimensions

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 for a small increase in the size of the coordinate routing table. For a CAN with d dimensions and n nodes, the path length grows as $O(n^{1/d})$ while the amount of neighbor state grows as $O(d)$. Because increasing the number of dimensions implies that a node has more neighbors, the fault tolerance in routing improves as a node now has more potential next

hop nodes along which packets can be routed. Similarly, the higher number of neighbors improves the efficacy of our volume balancing check thereby resulting in a more uniform partitioning of the coordinate space among the nodes in the system.

2.4.2 Adding multiple realities

The second design observation is that we can maintain multiple independent coordinate spaces with every node in the system being assigned to different, independent sub-spaces on every coordinate space. We call each such coordinate space a "reality". Every added reality improves system robustness and performance at the cost of increased control and data state per node.

The contents of the hash table are replicated on every reality. i.e. any data associated with a location, say (x,y,z), is now stored at the nodes associated with (x,y,z) on each reality. This replication improves data availabilty. For example, say a pointer to a file "ABC" 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. This redundancy means that the pointer to "ABC" is unavailable only when all three nodes are down.

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. Consequently, multiple realities improves routing efficiency (i.e. routing path length). To see this, consider the state held by a single node. For a CAN with r realities, a single node is assigned r coordinate sub-spaces, one on every reality. A node thus has r coordinate addresses, and r independent neighbor sets. A node's coordinate addresses are selected such that they lie at sufficiently different locations of the coordinate space. What this implies, is that 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 packet, a node now checks all its neighbors on each reality and forwards the packet to that neighbor with coordinates closest to the destination.

Finally, multiple realities improve routing fault tolerance, because in the case of a routing breakdown on one reality, traffic can continue to be routed using the remaining realities.

A CAN system could thus make use of multiple, multi-dimensional coordinate spaces.

2.4.3 Overloading coordinate sub-spaces

So far, our design assumes that any sub-space 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 sub-space. Nodes that share the same sub-space are termed peers. We define a system parameter $MAXPEERS$, which is the maximum number of allowable peers per sub-space (we imagine that this value would typically be rather low, 3 or 4 for example).

At all times, a node must maintain a peer-list, i.e. a list of the nodes sharing its sub-space. Nodes continually monitor the liveness of their peers. While a node must know all the peers in its own sub-space, it need not track all the peers in its neighboring sub-spaces. Rather, a node selects one node from each of its neighboring sub-spaces. As described below, a node will over time, measure the round-trip-time to all the nodes in each neighboring sub-space and retain the closest (i.e. lowest latency) node in its coordinate neighbor set. Thus, while overloading sub-spaces requires a node to track its peer nodes, it does not increase the amount of coordinate neighbor state a node must maintain.

Overloading a sub-space is achieved as follows: When a new node, say A, joins the system, it discovers an existent node, say B, whose sub-space it is meant to occupy. Rather than directly splitting its sub-space as described before, node B first checks whether it has fewer than $MAXPEERS$ number of peers. If yes, the new node A merely joins B's sub-space 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 sub-space is full (already has $MAXPEERS$ nodes), then the sub-space 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 sub-space. 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 sub-space and retains the node with the lowest rtt as its neighbor in that sub-space. Every node in the system does this for each of its neighboring sub-spaces. After its initial bootstrap into the system, a node can perform this rtt measurement

operation at very infrequent intervals so as to not unneccessarily generate large amounts of control traffic.

The benefits of overloading sub-spaces are many:

- overloading a sub-space significantly improves system fault tolerance because a sub-space becomes empty only when *all* the nodes in a sub-space crash at the same time. (in which case the repair process described above still needs to be applied)
- because a node now has multiple choices in its selection of neighboring nodes, it can select neighbors that are closer (in terms of latency) thereby reducing path latencies on the CAN overlay. Our simulation results in section 3 quantify these latency gains.
- sub-space overloading has the effect of reducing the average path length (in terms of number of hops on the CAN overlay) because placing multiple nodes per sub-space has the same effect as reducing the number of nodes in the system. Again, our simulation results quantify this effect.

Overloading sub-spaces adds somewhat to system complexity because nodes must now track a set of peers and ensure consistency of both data and control state across peers. And yet, this complexity appears worthwhile both because we do not envisage the $MAXPEERS$ parameter being set higher than maybe 3-4 nodes and because (as our simulation results validate) we obtain large gains in performance and robustness for what seems like a small increase in system complexity.

2.4.4 Multiple hash functions

For improved data availibility, one could use p different hash functions to map a single key onto p points in the coordinate space and accordingly store a single (key,value) pair at p distinct nodes in the system. A (key,value) pair is then unavailable only when all p nodes are simultaneously unavailable. In addition, queries for a particular hash table entry could be sent to all p nodes in parallel thereby reducing the average query latency. Of course, these advantages come at the cost of increasing the size of the hash table, and the amount of query traffic (in the case of parallel queries) by a factor of p.

3 Simulation Results

In this section, we present simulation results evaluating the performance of our CAN algorithm. We first recap our design parameters, then describe our evaluation metrics and quantify the effect of individual parameters on these metrics.

The key design parameters affecting system performance are:

- dimensionality of the virtual coordinate space: *d*
- number of realities: *r*
- routing metric: Progress-only (denoted *P*) and Progress/RTT (denoted *P/R*) as described in section 2.1
- presence or absence of the volume balancing check described in section 2.2
- number of peer nodes per sub-space: *MAXPEERS*
- number of hash functions (i.e. number of points per reality at which a (key,value) pair is stored): *p*

We use the following metrics to evalute system performance:

- path length: measures the number of hops required to route between two points in the coordinate space. Note that by ”hops” we refer to the *application level* hops required to route on the CAN overlay network and not hops on the underlying IP Internet. A single hop on the CAN network may translate to several IP network level hops.
- neighbor-state: measures the number of entries in a node’s coordinate routing table
- latency: we measure both, the overall latency of the complete routing path between two points in the coordinate space and the per-hop latency i.e. latency of individual application level hops.
- volume: measures the volume of the sub-space a node is assigned to. The fraction of the hash table stored at a node is directly proportional to the volume of the space it owns.
- Routing fault tolerance
- Hash table availibility

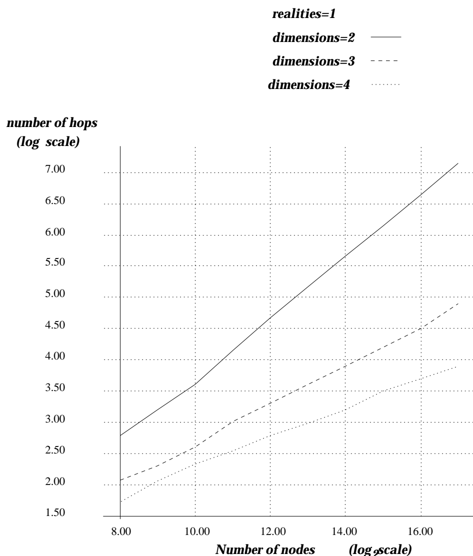


Figure 5: *Effect of dimensions on path length*

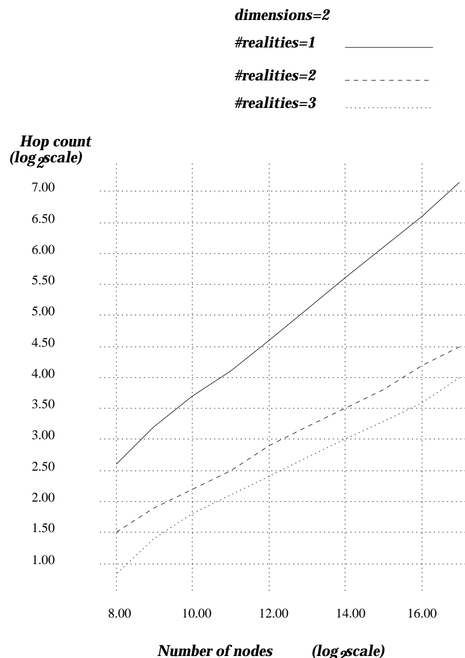


Figure 6: *Effect of multiple realities on path length*

In some cases, the effect of a design parameter on certain metrics can be directly inferred from the algorithm; in all other cases we resort to simulations. We divide our simulation work into two pieces. We first perform static simulations evaluating the CAN construction and routing process without modeling node failures (i.e. without CAN maintainance) and then add node failures into the picture and quantify their effect.

3.1 Performance under static system conditions

3.1.1 Path length

Figure 5 measures the 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. In keeping with the analytical results, we see that for a system with n nodes and d dimensions, the path length scales as $O(n^{1/d})$.

Similarly, Figure 6 plots the path length for increasing numbers of nodes for different numbers of realities.

3.1.2 Neighbor State

For a CAN system with d dimensions and r realities, each node requires at least $2dr$ neighbors. This is because a node requires at least two neighbors along each dimension, one to advance and another to retreat along that dimension, thus requiring $2d$ neighbors for a single reality. Further, a node maintains independent sets of neighbors for each reality thus requiring a total of at least $2dr$ neighbors.

Figure 7 plots the path length versus the average number of neighbors maintained per node for increasing numbers of dimensions and realities. As can be seen, the average number of neighbors per node approximately equals the lower bound of $2dr$. A second observation is 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 redundancy and fault-tolerance. Rather, the point to take away is that if one was willing to incur an increase in the average per-node neighbor state for the sole 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.

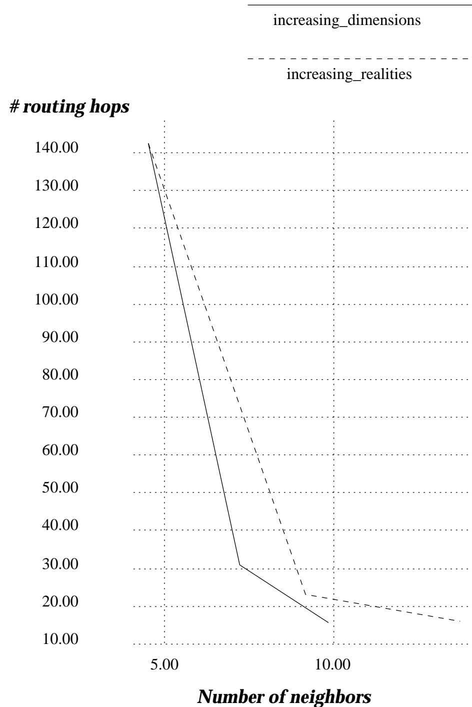


Figure 7: *Path length with increasing neighbor state*

Number of dimensions	P routing	P/R routing
3	116.7	76.08
4	115.8	71.2

Table 1: *Per-hop latencies using P and P/R routing*

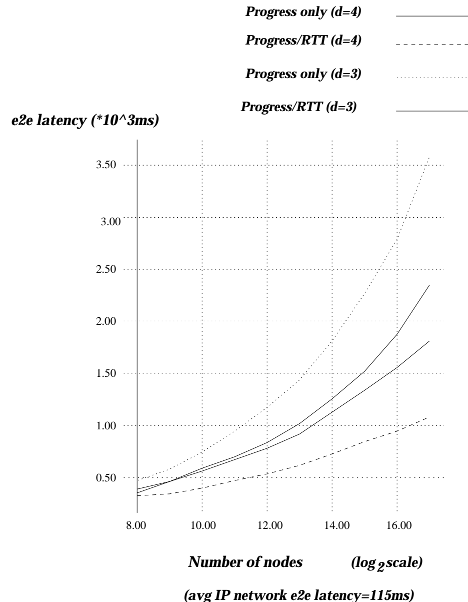


Figure 8: *Comparison of ProgressRTT vs. Progress-only Routing*

3.1.3 Latency

The overall path latency depends on the per-hop latency and the number of hops (i.e. path length). We can thus lower path latency by reducing either the path length or the per-hop latency. The path length can be reduced by increasing the number of dimensions, realities or the number of peers (nodes per sub-space) as seen above. P/R routing and sub-space overloading aim at reducing the per-hop latency. We now quantify the reduction in overall and per-hop latencies through the use of P/R routing and sub-space overloading.

The following simulations were carried out using the GT-ITM transit-stub topology generator [19]. Transit-stub topologies model networks using a 2-level hierarchy of routing domains. *Stub* domains only carry traffic that originates or terminates in their domain while *transit* domains serve to interconnect lower level stub domains. We assign link latencies of 100ms to intra-transit domain links, 10ms to stub-transit links and 1ms to intra-stub domain links The average end-to-end latency of the underlying IP network path between two nodes in our simulated topology is approximately 115ms.

Figure 8 compares the overall path latency using the P and P/R routing metrics. We plot the latency versus increasing number of nodes. Clearly, P/R routing reduces the path latency. We also see that the absolute values of the latencies are significantly lower for higher numbers of dimensions. This is due to the reduction in the number of hops with increasing dimensions. To factor out the effect of path length and quantify the effect of P/R routing on the per-hop latency, Table 1 lists the per-hop latencies, obtained by normalizing the overall latencies from Figure 8 by the average path length. As can be seen, while the per-hop latency using P routing matches the underlying network per-hop latency, P/R routing lowers the per-hop latency by around 36 %.

Figures 9 and 10 show the effect of sub-space overloading on overall path latency. Figure 9 shows the overall path latency for an increasing number of nodes with different values of $MAXPEERS$, the maximum number of peers per sub-space. With increasing $MAXPEERS$, a node has increasingly more choices

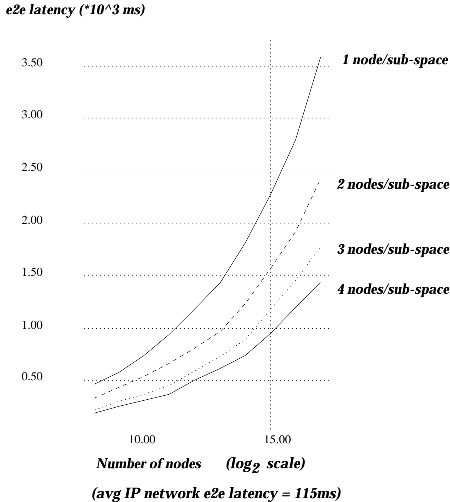


Figure 9: *Reduction in End-to-end latency due to replication*

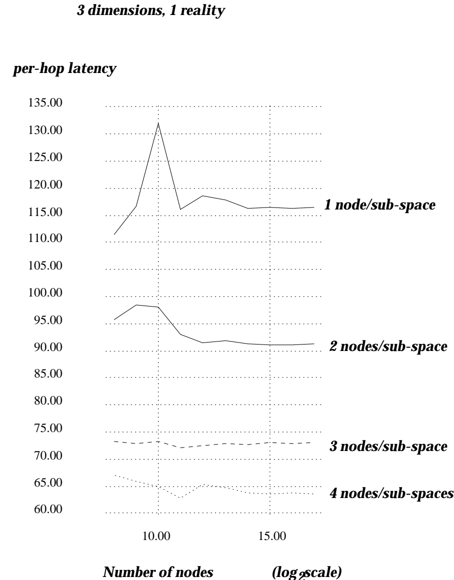


Figure 10: *Reduced per-hop latency with replication*

in selecting its neighbors and can thus select neighbors that are closer (i.e. lower latency) thus reducing the latency of individual CAN links. Once again, to isolate the latency reduction obtained because of overloading we normalize the overall latencies by the number of hops and plot the per-hop latencies in Figure 10. As can be seen, sub-space overloading with a maximum of 4 nodes per sub-space, yields a reduction of about 50% over the network level per-hop latency.

3.1.4 Volume

Figure 11 quantifes the efficacy of our volume balancing check applied at the time a node joins the CAN. We ran simulations with 2^{16} nodes both with and without the volume balancing check. At the end of each run, we compute the volume of the sub-space assigned to each node. 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 sub-space of volume V_T/n to each node. We use V to denote V_T/n. Fig 11 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 sub-spaces of a particular volume. From the plot, we can see that without the volume balancing check a little over 40% of the nodes are assigned to sub-spaces with volume V as compared to almost 90% with the use of the volume balancing check. With the volume balancing check, the largest sub-space is only $2V$ compared to $8V$ without. Thus, the variance in the volumes of individual node sub-spaces drops dramatically with the use of the volume balancing check. This in turn implies that the contents of the hash table are likely to be evenly partitioned over all the nodes.

3.2 Dynamic metrics

We now present 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 sub-space 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 12 plots the probability with which a node

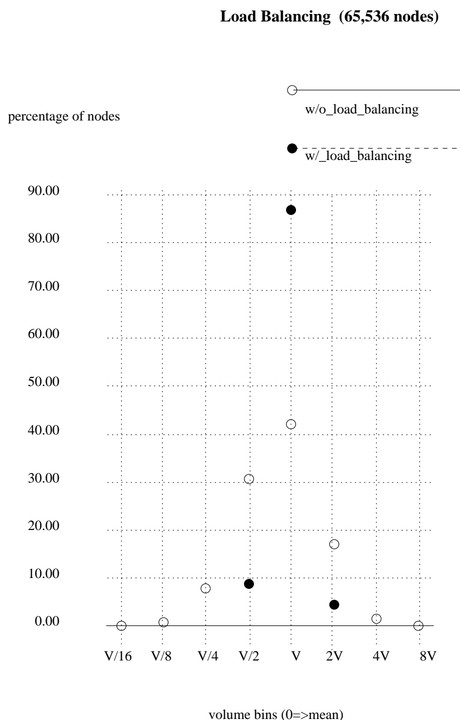


Figure 11: *Effect of Volume Balancing check*

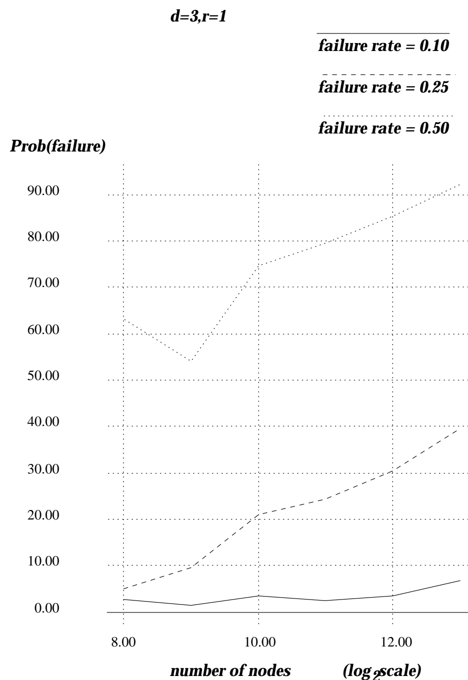


Figure 12: *Routing failure rate without repair*

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

Table 2: *Background sub-space reassignment*

must perform an expanding ring search as a function of the number of nodes for different failure rates. Figure 13 plots the search radius at which the ERS successfully terminates and figure 14 plots the total path length for increasing numbers of nodes and different node failure rates.

The background sub-space reassignment algorithm described in 2.3 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 sub-spaces.

Table 2 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 sub-space to.

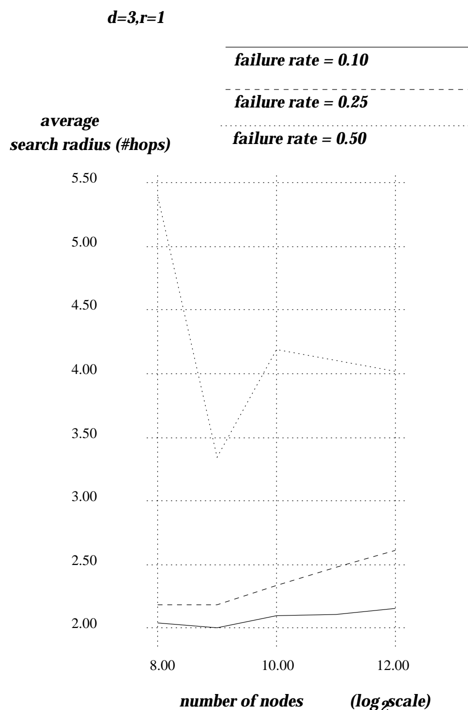


Figure 13: *Search Radius using ERS*

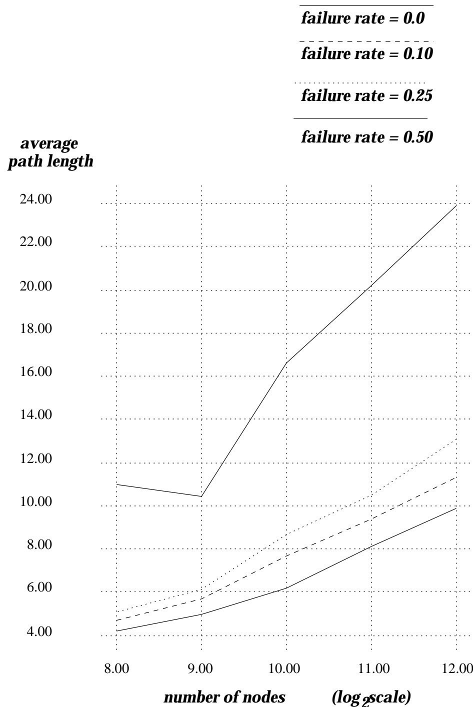


Figure 14: *Path Length using ERS*

4 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.

4.1 Related Algorithms

The Distance Vector (DV) and Link State (LS) algorithms used in IP routing require every router to have some level of knowledge of the topology of entire network. The Bellman Ford algorithm used in DV does this iteratively by having every router periodically announce its distance from all network destinations to its local neighbors while LS works by simply announcing its link status to every router in the 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, DV and LS are not well suited to networks with frequent topology changes as this would generate large amounts of routing updates. Because we wanted our CAN design to scale to large numbers of nodes and cope well with high degrees of node flakiness, we chose not to use routing schemes such as DV and LS that require a router to have a full topology map.

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 [18, 11, 2].

Perhaps closest in spirit to the CAN routing scheme is the Plaxton algorithm [16]. 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 $d = n/l$ bits. A node with label, say xyz, where x,y and z are d bit digits, will have a routing table with entries of the form

- $*XX$
- $x * X$
- $xy*$

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

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 does.

For a system with n nodes, Plaxton's algorithm thus routes in $O(logn)$ steps and requires a routing table size that is $O(logn)$. CAN routing by comparison routes in $O(n^{1/d})$ hops (where d is dimensions) with routing table size $O(dr)$ which is independent of n.

Both Plaxton and CAN routing thus have good performance bounds. Plaxton's algorithm achieves slightly shorter paths at the expense of slightly larger routing tables as compared to our CAN algorithm.

We believe however that for self-organising systems, CAN routing offers greater robustness and simplicity 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 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 (details of what constitutes a good approximation are in [16]). 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 entry into the system 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. In short, maintaining accurate, up-to-date routing tables in a scalable manner, with nodes entering and leaving the system, is non-trivial with Plaxton's algorithm. 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.

It should be pointed out however that the Plaxton algorithm was originally proposed for web caching environments which are typically administrator configured as opposed to self-organising and the number of caches in the administrator's network is fairly stable.

4.2 Related Systems

4.2.1 Domain Name System

The DNS system in some sense provides the same functionallity 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.

4.2.2 OceanStore

The OceanStore project at U.C.Berkeley [10] is building a utility infrastructure designed to span the globe and provide continuous access to persistent information. Servers self-organise 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 for data location. The Plaxton algorithm was described above.

4.2.3 Publius

Publius [12] 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. Publius content is encrypted by the publisher and spread over some subset of the web servers on the list. The self-organising 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.

4.2.4 Peer-to-peer file sharing systems

Recently, a number of systems and applications [14, 7, 13, 6] make use of what is being termed a "peer-to-peer" model of communication. In peer-to-peer systems, files are stored on individual user machines (rather than a central server). The transfer of files takes place from one user machine to another directly without passing through a server. Another way of stating this is to say that every machine in the system plays the role of both client and server. Because files can potentially be located at any node in the system, peer-to-peer systems need a mechanism to discover the IP addresses of nodes in the system storing a particular file. In Napster [14] the index mapping file names to IP addresses is stored at a central server hence the search process itself does not fall under the peer-to-peer banner, only the only the actual file transfer process is peer-to-peer based.

The index in Gnutella [7] is distributed across the set of users. Search requests are essentially flooded (with some form of scoping) over the Gnutella network. The distributed search component of Gnutella thus has scalability problems [9] and, because the flooding has to be curtailed at some point, may actually fail to find content that is actually in the system.

In all the above applications, our CAN design can be used for the construction and maintainance of a distributed index that is at once scalable and fault tolerant. Hence, while our CAN design does not by any means solve all the problems (such as anonymity, security, accountability etc) tackled by systems like FreeNet, Gnutella and MojoNation, it can serve as a core application building block within all of them.

4.2.5 Distributed Data Structures

In [8], Gribble et al. implement a distributed hash table designed to run on a cluster of workstations. Their goal is to ease the development of scalable, available services running on a cluster of workstations by providing persistent data structures that encapsulate the vagaries of clustered platforms. Because their targetted environment, i.e. clusters of workstations, is very different from the wide-area Internet, we view their work as orthogonal to our own.

5 Conclusion

In this paper, we defined the concept of a Content-Addressable Network, a key component in building large scale, distributed storage systems. We presented the design of a CAN that is distributed, self-organizing and scalable. Simulation results validate the scalability and robustness of our design. This paper focused primarily on the content routing and indexing aspects of a CAN. For a comprehensive design of a CAN system certain key problems, such as security, resistance to DoS attacks and data consistency remain to be addressed.

6 Acknowledgements

This work was done in collaboration with Mark Handley, Richard Karp and Scott Shenker and greatly benefitted from discussions with Brad Karp, Steven McCanne, Jitendra Padhye and Vern Paxson.

References

[1] CNN. Rosa Parks wins domain name battle. article at http://www.cnn.com/2000/LAW/09/20/rosa.ap/index.html, Sept. 2000.

[2] 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.

[3] DOMAINNAMES FORSALE. http://www.domainnames-forsale.net.

[4] 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.

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

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

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

[8] GRIBBLE, S., BREWER, E., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed structures for internet service construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation* (San Diego, CA, Oct. 2000).

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

[10] 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).

[11] 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).

[12] 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.

[13] MOJONATION. http://www.mojonation.com.

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

[15] NAPSTER. Napster faq: Why can't i connect to napster ? http://www.napster.com/help/faq/top10.html.

[16] 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).

[17] 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.

[18] TSUCHIYA, P. The Landmark Hierarchy: a new hierarchy for routing in very large networks. In *Proceedings of SIGCOMM '88* (Palo Alto, CA, Sept. 1988), ACM.

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