Content-Addressable Network

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 and Gnutella .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, 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 peerto-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 and, because the flooding has to be curtailed at some point, may fail to find content that is actually in the system.

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, Farsite, and Publius. 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. 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.

2.Design

Our design centers around a virtual *d* -dimensional Cartesian coordinate space on a *d* -torus. 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] x [0,1] coordinate space partitioned between 5 CAN nodes.

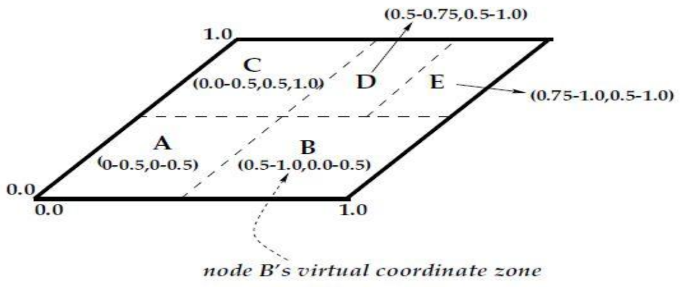


Figure 1: *Example 2-d coordinate overlay with 5 nodes*

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 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.

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.

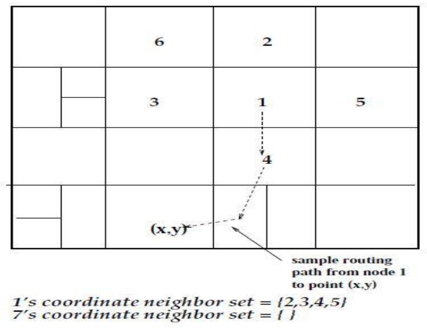

6
2
3
1
5
4
(x,y)
sample routing path from node 1 to point (x,y)
1's coordinate neighbor set = {2,3,4,5}
7's coordinate neighbor set = { }

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

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. 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.

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 nodes it believes are currently in the system.

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.

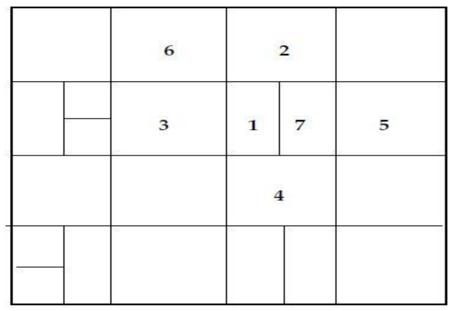
Finding a Zone

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 transferred 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 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.



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*

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 are 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.

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 than 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.

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 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 runs to ensure that the CAN tends back towards one zone per node.