A Scalable Content-Addressable Network

Sylvia Ratnasamy, Paul Francis, Mark Handley,
Richard Karp, Scott Shenker

Written by Vladimir Eske
Saarland University, Department of Computer Science

Contents

1 Motivation

Internet population explosion promotes different peer to peer systems. Several millions of users are connected to the P2P system at the same time is an everyday occurrence. There are two key pieces in a P2P system: the lookup mechanism used to locate a desired file and the actual file downloading. The decentralized storage in P2P systems makes the file transfer process inherently scalable; the hard part is finding the peer(s) from which to retrieve the file. Lookup solutions in deployed systems to date fall into two categories centralized like Napster and decentralized like Gnutella. Centralized solutions are typically cited as being vulnerable due to a single point of failure and being hard to scale for many millions of users.

2 Basic CAN architecture

2.1 Introduction

CAN is a distributed system that provides hash table functionality - mapping "keys" onto "value" on Internet-like scale. The basic operation in CAN is a lookup(key) which returns the corresponding "value" for the given "key". In spite of work on CAN was motivated by the P2P file sharing systems, the utility of CAN is not limited to them. There are many other directions in which CAN can be used: application layer multicast [5], event notification services [4], chat services [3] and others. Also there are a lot of similar to CAN projects - DHT (distributed Hash Table) systems like Chord [6]. The main difference between them is the routing algorithm which is used to provide a lookup operation.

The CAN's routing algorithm is designed to provide the following features:

- Scalability - every part of the system maintains only a small amount of control state and is independent of the number of parts
- Distributivity - the system does not require any centralized control
- Efficiency and fault-tolerance - a routing should be optimal and be provided even when some parts of DHT are crashed
- Balanced load

2.2 Data Model

CAN's design centers around the virtual d-dimension Cartesian coordinate space on d-torus. It is completely logical coordinate space which is cyclical in every dimension. However we can use the following example to illustrate it. We consider 1-dimension [0,1] coordinate space. The space can be represented as a circle, see figure 1. The perimeter of the circle equals to the size of space (length of interval [0,1]). The distance between two points in such a space equals to the length of the shortest arc between them.

CAN resembles a hash table and provides three basic operations: insertion, deletion and lookup. The system is composed of many individual nodes which store a chunk of the entire hash table. A node is not a peer, it is more like a dedicated server which basically should provide only indexing of information. A chunk of the hash table is called "Zone" in CAN. On the figure 2 there is an example of 2d coordinate space split on several zones. Zones in CAN can have different size like three gray zones on the picture, however they must have a squared shape.

Every node owns only one distinct zone and provides direct access to the stored zone for all users connected to this node. However, it is more likely that most of requested data is out of the current node; to provide user's queries on the entire DHT a node has to forward user's queries to one of its partner nodes. In CAN node can forward a query only to its "neighbor" and two nodes are neighbors if their coordinate spans overlap along d-1 dimensions and abut along one dimension. The figure 3 shows an example of neighbors in CAN. The node which owns the green zone has five nodes, they own light green zones. The green zone and red zone owners are not neighbors because they don't abut.

This neighbor relationship creates a virtual grid and allows to forward a query rationally using some simple metrics, for example cartesian distance. A node in CAN stores also a list of its neighbors which contains neighbors' IP addresses and their zones' coordinate.

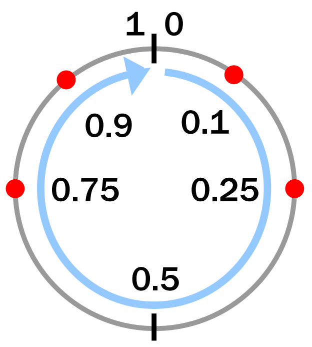


Figure 1: 1d Cartesian coordinate space on 1-torus

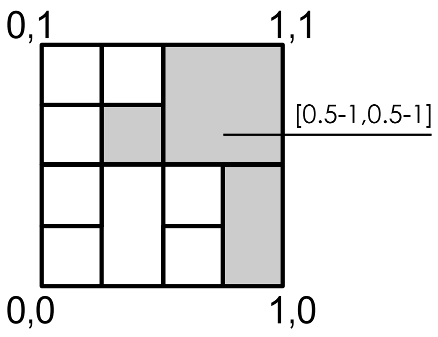


Figure 2: Zones example

2.3 Access Model

The first step of interaction between user and P2p system is getting an access to the system. CAN provides the following bootstrap mechanism which

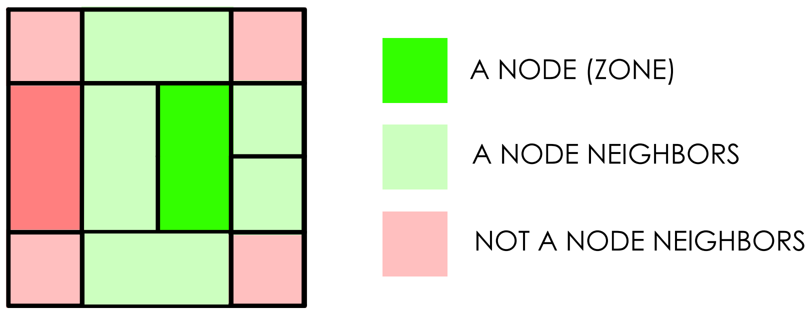


Figure 3: Zones example

provides this functionality. Assume CAN has an associated DNS domain name which is resolved to the IP address of one of the CAN bootstrap nodes. A bootstrap node maintains a partial list of CAN nodes which are currently in the system. A user in this model sends a request, using CAN's domain name. His client gets an answer from one of the bootstrap nodes and automatically establishes the connection to any available CAN node. In details the bootstrapping mechanism is described in [7].

2.4 CAN routing

The routing algorithm is very important for DHT systems because it defines the response time for the query and it means data availability. A wrong routing algorithm in the distributed system will grade all advantages of such a system, like it happened with Gnutella. CAN routing algorithm uses the greedy search on the node-graph, nodes are connected by the neighborhood relation. In the best case, when every node has exactly 2 neighbors in each dimension, this graph is a d-dimensional grid structure.

The cartesian distance is used as a routing metrics. CAN routing algorithm basically consists of 3 main steps:
First, the destination point should be defined: a user sends a request (add, delete, or lookup) to the CAN node which he is connected to. The CAN node maps the requested key using a hash function into a point in the Cartesian coordinate space. Than the request is forwarded to the node which owns the zone containing the destination point. The current node in the trace checks whether it owns the destination point:

1. If it owns, this node performs the requested action and establishe a direct connection with the user to send a result.

2. If not, the request is forwarded to the neighbor node of the current one which is the closest one to the destination point due routing metrics - cartesian distance.

Fault tolerance routing

If node loses all its neighbors in an optimal direction defined by routing metrics and the repair mechanism described later have not build the void zone, then greedy forwarding may fail. To prevent this situation the basic routing algorithm should be extended by the following rule: before forwarding the request the current node checks for its neighbors availability. The request is forwarded to the closest available node. In this case the path may be nonoptimal, but the data is still available. The figure 4 shows an example of fault tolerance routing.

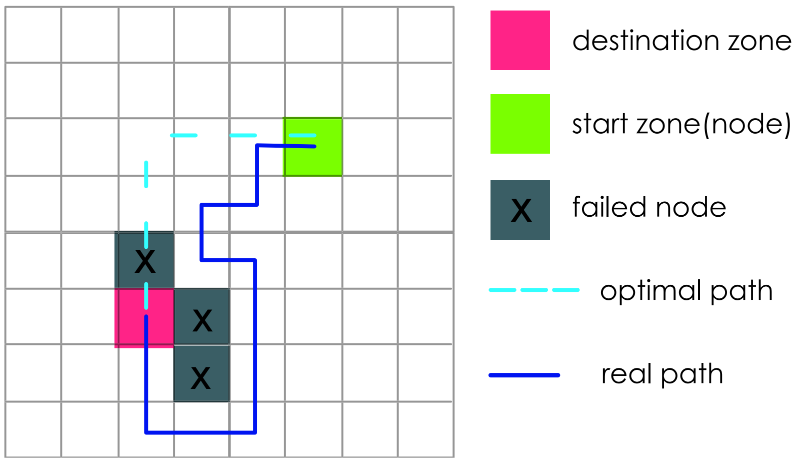


Figure 4: Fault tolerance routing

2.4.1 Average path length

It is important to estimate an average path length to be able to manage the CAN. Assume that we have a perfect situation, when every node has 2 neighbors in each dimension and, as said before, node graph becomes a d-dimension grid. The number of zones per dimension is the same for all dimensions and it equals $\sqrt[d]{n}$, where n is the total number of zones and d is the number of dimensions. The maximal path length in each dimension is $\frac{\sqrt[d]{n}}{2}$ because CAN uses a cartesian space on d-torus which is cyclical. The maximal path length for the CAN is the sum of the maximal path lengths in each dimension and it equals $d * \frac{\sqrt[d]{n}}{2}$ The average path length [1] is not greater than the maximal one.

$$\bigcirc(d * \sqrt[d]{n}) \tag{1}$$

There is a 2-dimension example on the figure 5. The number inside square (zone) means a path length from the square marked up with 0 to this one.

The total number of zones is 64, 8 zones per dimension, the maximal path length in each dimension is 4 and the total one is 8 = 2*4. However for this kind of examples the exact average path length can be computed and it is $\frac{d}{4} * \sqrt[d]{n}$.

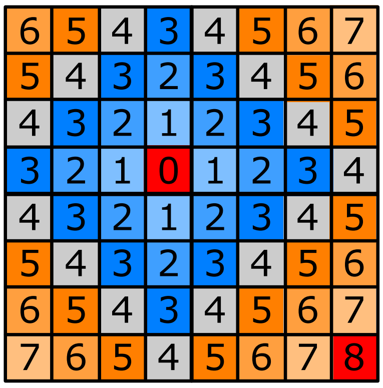


Figure 5: An average path length example

3 CAN construction

In this part we consider how CAN can be constructed. We assume that there is at least one node in the system. In this case it is important to handle 3 different situations:

1. A new node wants to join the system
2. A CAN node wants to leave the system
3. A CAN node crashes or leaves the system without any notification.

3.1 Node's arrival

A new node arrives and we need first to find a zone for it and move it onto a new place. A priori there are no free zones and a zone for the new node should be taken from the node are already in the system. The easiest way to do it is to split a zone of some CAN node in half. The next step is when a new node got a zone and should be included in "routing" network to provide all CAN's services.

Now consider all step in details:

Finding a Zone A new node first connects to any CAN node using the access mechanism described before. Then it sends a join request to the randomly chosen node, the CAN node **randomly** chooses a point in the coordinate space. The zone which contains this point should be split in half. The join request is forwarded due normal CAN routing procedure.

When the CAN node gets a join request it splits its zone in half according the following rule. Assume there is a certain ordering of the dimensions and the node always splits the zone in half along only one dimension in which the zone has largest size and the smallest order of the dimension. Figure 6 gives an example: in the first case the zone has the same size in both dimensions and should be split along the dimension #1 and in the second case the zone is larger in the dimension #2 and should be split along it. After splitting one half of the zone is replaced into the new node.

The next step is to run the new node. First the new node gets a neighbor list together with its half of zone. Then it should notify all its neighbors about the new state of the system. In results the neighbor lists of the new node, of their neighbors and also of the old node, that owns the split zone, should be updated.

3.2 Node's departure

When a node leaves the system in a normal way, it tells the system about its leaving. In this case it is necessary to hold physical integrity, to replace

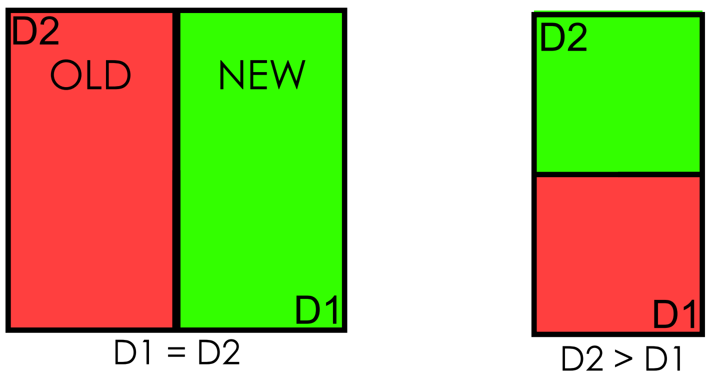


Figure 6: Splitting zone in half

leaving node's zone and logical integrity, to support a routing under this zone. CAN offer the following algorithm to do it:

- The leaving node finds such a neighbor which zone can be merged with it and forms a proper zone - squared shaped. Figures 7 a and b illustrate examples of appropriate and unappropriate merging.

  If such a neighbor does not exist the leaving node chooses any its neighbor, in this case a node covers two different zones in the same time. However CAN has an inconsistent state, one node owns 2 zones and it should be fixed by additional algorithm.

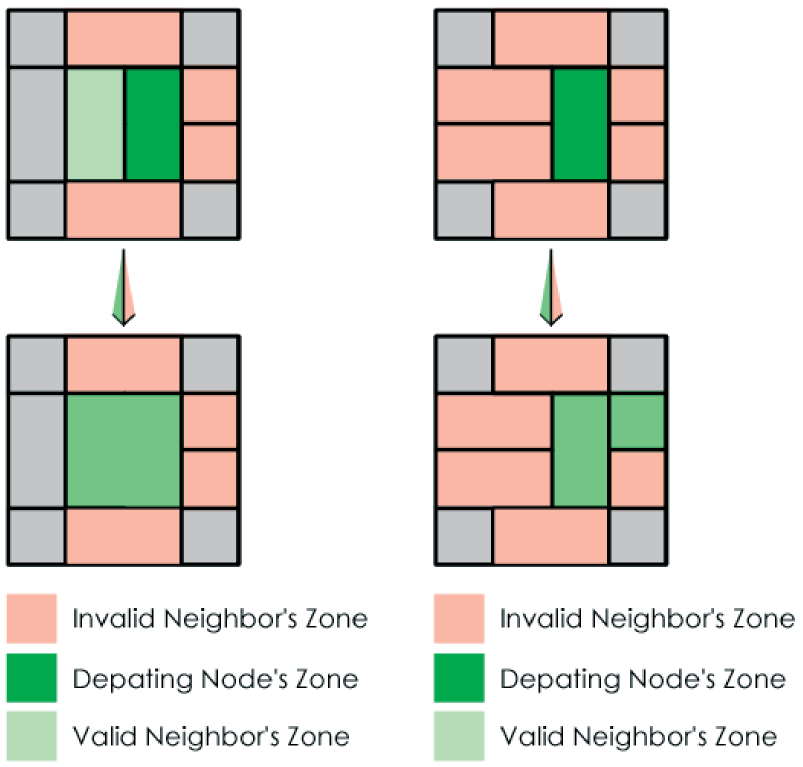
- The leaving node's zone is replaced to the chosen neighbor, physical integrity is held.

- To provide a routing under this zone the system should update its state. Leaving node's neighbors are notified that another node is now their neighbor instead of the leaving one. The node which receives the zone changes its neighbor list and notifies all its neighbors.

3.3 Node's crash

In this case the node does not notify the system about its departure. This is handled through an takeover algorithm which ensures that one of failed node's neighbors takes over the zone. However the data, (key, value) pairs, owned by the failed node would be lost until the state is refreshed by data owners, in the case of P2P file distributed system users will connect to the CAN and share their files again.

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



a. Appropriate zones merging   b. Unappropriate zones merging

Figure 7: zone merging

neighbor signals its failure. If some node has decided that its neighbor has failed it initiates a TAKEOVER mechanism. Note that several neighbors can start a TAKEOVER mechanism independently.

TAKEOVER mechanism:

1. The node initializes a timer in proportion to its zone volume
2. If a timer is expired it sends a TAKEOVER message to all failed node's neighbors which contains the volume of its sender zone
3. A neighbor which gets a TAKEOVER message compares its own zone volume to the sender zone volume and if its zone is smaller then this node sends a new TAKEOVER message as described above
4. A failed node's neighbor which did not get a TAKEOVER message with smaller zone should take the zone of the depastured node.

The two main advantages of this mechanism are that

1. it allows to assign a failed node's zone to the smallest node (balanced load

2. it works without any centralized control

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 described in [2] (Appendix A).

4 Advanced CAN architecture

The basic CAN architecture described above provides a balance between low per-node state (every node has O(d) neighbors for a d dimensional space) and short average path length with $O(d * \sqrt[d]{n})$ hops. However they are application level hops. In real life all "application" hops are different: nodes are located in different parts of networks and at least ping time is different for different hops. Moreover the basic architecture does not provide any additional mechanism to increase data availability, in the case when some nodes crashed the data is lost.

This part of the report presents, first, different techniques whose primary goal is to reduce the latency of CAN routing:

1. Per path latency: reduction of the average number of hops per path
2. Per hop latency: reduction of average time needed to make one hop.

In the end several approaches which increase data availability and balanced load will be presented.

4.1 Path latency improvement

4.1.1 Multiple dimensions

The simplest way to improve path latency is to increase the number of dimensions. The average path length is $O(d * \sqrt[d]{n})$, formula 1. Assume n is constant, it is easy to find such value of d which minimizes the average path length. You can see an example in table 4.1.1.

d	Average Path length
2	200
3	30
6	18.973
7	**18.778**
8	18.970
10	19.952
20	28.250

Table 1: an optimal number of dimension, n = 1000

4.1.2 Multiple realities

This approach is based on the idea to maintain multiple, independent coordinate spaces. Each such coordinate space is called "reality". The content of the hash table is physically replicated on every reality. Each node owns a different zone in each coordinate space that means it stores several different chunks of hash table independently. When a new node joins the system it sends r (number of realities) join-requests, one for one reality. As was said before a zone is assign to the node randomly, by choosing a random point in the coordinate space. Using this mechanism the new node gets r zones which should be mostly different (random assign). However the important thing is that the CAN still uses one hash function to map real objects onto coordinate space for all realities. Add and delete operation are performed on all realities independently using the basic routing mechanism that holds data consistency for all realities. However every node should store r different neighbor lists: one for one reality. Such an architecture uses an advanced routing algorithm: now node checks for destination point in all realities and forwards the request to such its neighbor which is the closest one to the destination point over all realities. All realities use the same hash function that is why the coordinate of the destination point is the same for all of them but every node owns different zones and it means that the average path length should be decreased with a factor of r.

This CAN architecture requires much more space than the original one, it grows in r times. However it strongly reduces the average path length and in the same time increases the data availability with a factor of r.

4.2 Hop latency improvement

Usually there are several paths from the current zone to the destination point which have minimal distance. The basic routing chooses one, probably randomly (the original paper does not say anything about this). To improve a per hop latency for a given destination, a message is forwarded to the neighbor with the maximum ratio of progress to RTT, round-trip-time - some kind of ping time. Such a metric is called RTT metrics. Also the number of different optimal paths grows together with the number of dimensions or realities that makes using RTT metrics especially effective together with two approaches described before.

The table 4.2 shows the results of performing experiments under the CAN containing from 2^8 to 2^18 nodes. As can be seen using the RTT metrics archives 24-40% of per hop latency decrease in average.

4.3 Overloading coordinate zones

Unlike all previous approaches the "overloading coordinate zones" approach modifies the basic CAN rule: one zone - one node. Assume now that every zone can be owned by several nodes and the constant MAXPEERS defines the maximum number of allowable peers per zone. With the following CAN architecture, a node maintains a list of its peers in addition to its neighbor list, which still contains only the node's neighbors but not its peer's

number of dimensions	routing without RTT (ms) per hop	routing with RTT (ms) per hop
2	116.8	88.3
3	116.7	76.1
4	115.8	71.2
5	115.4	70.9

Table 2: Per-hop latency using RTT weighted routing

neighbors. Each zone is replicated among all nodes assigned to it.

Such architecture needs a new construction algorithm. Consider a new join mechanism. When a new node joins a system it discovers a zone which is used to be split in half, using the basic approach described before, and checks whether this zone has not more than MAXPPERS peers. If so the new node becomes a new peer, replicates a data and send an update request to all its peers, otherwise the zone is split into half as before, the peer list is also split in half. The half nodes from the peer list own one half of this zone, another half and the new node owns the second half zone. All nodes should update their neighbor and peer lists.

Periodically, a node sends each of its neighbors a request for its list of peers, then estimates the RTT (round-trip-time) to all the nodes from this peer list and replaces the current neighbor to the node which has the minimal RTT among all peers. After all updates it may happen that some nodes will not occur in other's neighbor lists, they will have no in-link in the CAN graph, however they will have an out-links and can be used as a start search point and recovery data storage.

This approach archives many advantages:

- it reduces path latency, because placing of multiple nodes per zone has the same effect as reducing the number of nodes in the system
- it reduces per-hop latency by periodical neighbor list updates which select neighbors that are closer in term of RTT latency
- it increases a fault tolerance, because each zone is replicated several times and it is vacant only when all peer nodes crashed. However when the basic routing is used, there is no direction between peers, only between neighbors and the CAN still requires the repair process described in section "Node's crash", otherwise some users don't get an answer.

4.4 Construction improvement - Uniform partitioning

The uniform partitioning is very important for CAN, the average path length is minimal when the coordinate space is split into n equal zones. In basic

CAN architecture a new node gets a zone by splitting randomly chosen zone in half. The node which own this zone also knows about its neighbor's zones. And in order to improve the partitioning this node can easily forward the join request to the neighbor which zone is the largest one among all neighbors and larger than its own. Note that this is not sufficient for true load balancing because different key, value pairs has different popularity and even uniform hash function does not distribute a real data perfectly. However this simple technique archives a good results. Assume V_T is a total volume of the entire coordinate space and n is a number of nodes then a perfect partitioning would assign a zone of volume V_T/n to each node. The figure 8 shows that using of uniform partitioning features increases the number of normal zones (V_T/n) from 40 % to 90 %.

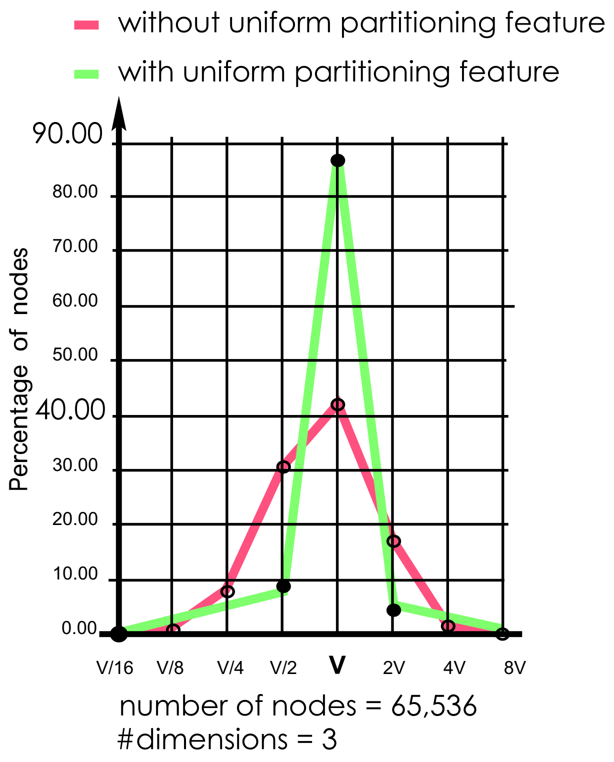


Figure 8: Effect of uniform partitioning feature

5 Summary

CAN is scalable, distributed Hash Table which can be used in many different applications. It provides:

- Dynamical Zone allocation
- Fault Tolerance Access Algorithm
- Stable Fault Tolerance Routing Algorithm

There are many improvement techniques which

- Increase Routing Latency
- Increase Data availability
- Increase Fault Tolerance

The authors performed several experiments which goal was to value the utility of several improvement techniques. There were built two different CAN system called "bare bone" which provides only basic CAN architecture and "knobs on full" which uses many improvement techniques. See table 5.

Parameter	bare bones CAN RTT	knobs on full CAN
# of dimensions	2	10
MAXPEERS	0	4
RTT weighted routing metrics	OFF	ON
Uniform partitioning	OFF	ON

Table 3: CAN parameters

Metric	bare bones	knobs on full
Avg. Path length	142.0	4.899
# of neighbors	4.2	24.4
# of peers	0	2.95
Avg. Path Latency	19671 ms	135 ms

Table 4: CAN metrics

The experiments were performed under CAN containing 2^17 nodes and using the Transit-Stub topology. The results are in table 5. You can see a great improvement in real path latency, more than 100 times. However it is archived mostly by increasing of the number of dimensions. The "knobs on full" CAN also has a certain improvement in data availability and balanced load.

References

[1] Gerhard Weikum: Introduction to P2P systems, http://www.mpi-sb.mpg.de/units/ag5/teaching/ws03_04/p2p-data/10-21-intro.pdf, 2003.

[2] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker: A Scalable Content-Addressable Network, Proceedings of ACM SIGCOMM 2001. http://citeseer.nj.nec.com/ratnasamy01scalable.html

[3] CAN based Chat. http://di.jxta.org/, 2001.

[4] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a global event notification service. In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau/Oberbayern, Germany, May 2001.

[5] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Applicationlevel multicast using content-addressable networks. In Proceedings of NGC, London, UK, November 2001.

[6] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of SIGCOMM 2001, August 2001.

[7] Paul Francis. Yoid: Extending the internet multicast architecture. Unpublished paper, available at http://www.aciri.org/yoid/docs/index.html, April 2000.