Survey on Structured Peer-to-Peer Networks

Guo Shuqiao, Rakesh Kumar Gupta, Yao Zhen

School of Computing
National University of Singapore
Lower Kent Ridge Road
Singapore, 119260

{guoshuqi, rakeshku, yaozhen}@comp.nus.edu.sg

Abstract

Peer-to-peer (P2P) networks are becoming more and more popular because their promising applications in file-sharing systems. As peers can join, leave and fail at any time, the P2P network is highly dynamic. Thus, how to design efficient topology for P2P network has drawn a lot of research efforts. Based on the network topology, P2P networks can be classified into unstructured and structured. In this survey report, we would like to explore the field of structured network. In a structured P2P network, There are currently two broad approaches, namely, DHT-based (Distributed Hash Table) and non-DHT-based. Recent works in both approaches are covered in our survey. More specifically, we present the proposed network topologies, the updating and querying algorithms. Some important issues involved, such as load balancing and consistency-preserving are discussed. We also compare some of the DHT's by examining their underlying geometry and comparing them in terms of flexibility of neighbor and route selection.

Keywords: *pear-to-peer networks, DHT, routing, searching, hash table, Skip List.*

1 Introduction and Background

Peer-to-peer (P2P) networks are becoming more and more popular because their promising applications in file-sharing systems. In a P2P network, nodes (*peers*) play equal roles and communicate with each other in a decentralized fashion. As peers can join, leave and fail at any time, the P2P network is highly dynamic. Thus, how to design efficient topology for P2P network has drawn a lot of research efforts. Based on the network topology, P2P networks can be classified into *unstructured* and *structured*. In this survey report, we would like to explore the field of structured network.

In a *structured P2P network*, the overlay network architecture and the data placement are defined, though not necessarily precisely determined. There are currently two broad approaches, namely, DHT-based (Distributed Hash Table) and non-DHT-based. A *DHT* is a hash table whose table entries are distributed among different peers located in arbitrary locations. Each data item is assigned a unique key value by hashing functions, as well as each node. Each node is responsible for a particular set of keys.

Recent years, many DHT-based P2P lookup approaches are proposed. Chord [SML+2001] uses a ring structure for the ID space and each node maintains a finger table to support key query as binary search. Pastry [RD2001a] uses a tree-based data structure and the routing table kept in each node is based on shared prefix. P-Grid [Aberer2001] is based on a virtual distributed search tree. CAN [RFH+2001] implements DHT using a *d*-dimensional space. These systems are all scalable access structures for P2P and share the DHT abstraction. However, there are many issues in the DHT approach needing worth more investigation. Among them, load balancing and neighbor-table consistency preserving are two important and challenging problems. In recent works, [GLSKS2004] employs the concept of virtual server and directories to address the problem of load balancing, whereas [LL2004b] maintains the consistency while optimizing the neighbor-tables by proposing a join protocol and several optimization algorithms under an optimization rule.

Some non-DHT P2P structures are developed by several research groups such as SkipNet [SNL2003], SkipGraph [AS2003] and TerraDir [SBK2002]. These approaches try to overcome the problems of DHT-based P2Ps by avoiding hashing. Searching in such systems follows the specified neighboring relationships between nodes.

The rest of the report is organized as follows. In Section 2, some DHT-based P2P systems including Chord, Pastry, P-Grid and CAN are reviewed. Two important issues relating to DHT Design including load balancing and consistency preserving are discussed in Section 3 and a comparison of several DHT's are covered in Section 4. SkipNet, a non-DHT P2P system is presented in Section 5. Finally, Section 6 summarizes the report.

2 DHT-Based Structured P2P Systems

Among the structured P2P system, most implement is based on a distributed hash table (DHT). Since a structured system can determine the locations for any node or data, a DHT can perform as a routing table to determine the route of queries or messages. Chord [SML+2001] and Pastry [RD2001a] are typical systems in this class and they scale well to large number of nodes. P-Grid [Aberer2001] organizes the nodes as a virtual binary search tree, and CAN [RFH+2001] implements DHT using a *d*-dimensional space.

2.1 Chord

Chord [SML+2001] is a scalable P2P lookup service for internet application. It is one of the typical non-hierarchical DHT-based structured P2P systems.

2.1.1 Structure of Chord

In Chord, all the nodes and keys are assigned IDs in the same one dimensional ID space. IDs of the nodes that participate in the system make up of a ring, which is the basic data structure for Chord. A data is assigned to a node whose ID is nearest to and larger than the key of the data. Each node maintains a finger table which is a hash table with *logN* entries, where *N* is the size of the ID space in the system. A node's finger table contains the IP address of nodes that are$1/2^i$ (i=1, 2, …, log$N$) of the way around the ID space from it. Figure 2.1 shows an example of Chord in ID space of size 16 with 6 participant nodes. Node 4 responses for data with key of value 2, 3 and 4. There are 4 (log16) entries in node 4's finger table, pointing to nodes halfway around the ring from it, one-fourth away, one-eighth and one-sixteenth away on the ring from it respectively.

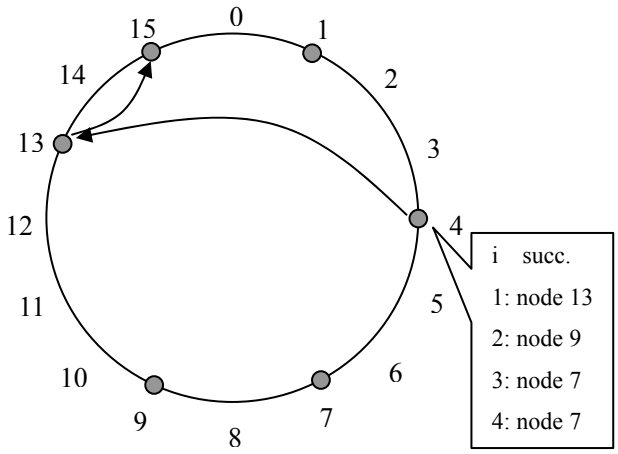


Figure 2.1 An Example of Chord

2.1.2 Searching in Chord

The searching algorithm of Chord is quite simple. In each step, a node forwards a query with key *k*

to a node in its finger table whose ID is highest and is not larger than *k*. In the example of Figure 2.1, a query with key 2 is sent to node 4. Node 4 looks up its finger table and forwards the query to node 13. Again, node 13 forwards the query, according to its finger table, to node 15, who can process this query locally.

2.1.3 Properties of Chord

Chord is completely decentralized. A query can be sent to an arbitrary node in the system. It is also self-organized. A new arrival node *X* can find its place in the ring by asking any living node in the system to lookup *X*'s ID. Then, *X* updates the fingers and predecessors of existing nodes to reflect the addition of it and takes over its data from the successor.

Since the finger table has a power-of-two structure, a key lookup in Chord ring is like a binary search, which ensures that the searching cost, in terms of exchanging messages, is bounded by $O(logN)$. Chord assumes that the keys and node IDs are spread roughly uniformly in the ID space. Therefore, if the IDs or keys are skewed, Chord can not perform well due to the load unbalance.

2.2 Pastry

Pastry [RD2001a] is a P2P object location and routing scheme based on dynamic hash table. It is a self-organizing overlay network which supports global data storage, data sharing and other P2P applications. Two P2P services are developed use Pastry: PAST [DR2001] [RD2001b] and SCRIBE [RKDC2001].

2.2.1 Structure of Pastry

In Pastry each node is assigned a unique numeric identifier (ID), which is 126 bits. Since the node ID is randomly chosen when a node joins the system, the nodes with adjacent IDs are not necessary nearby in geography, ownership and other properties. The node IDs are presented as a sequence of digits based on 2^b, where *b* is a configuration parameter with typical value *4*. It assumes that node IDs are uniform in the circular ID space. In Pastry system, the data with a numeric key *k*, which has the same length with node ID and is also presented as a sequence of digits with base 2^b, is assigned to a node whose node ID is numerically closest to *k* in the network.

To support the prefix-based routing schema, which will be covered in next subsection, each node *A* maintains three routing states including a leaf set, a neighborhood set and a routing table. The leaf set *L* contains |*L*| nodes with closest node IDs to *A*' ID, where half IDs in *L* are larger than *A*' ID and half are smaller. Leaf set is useful in message routing. The neighborhood set *M* is a set of |*M*| nodes that are closest, according to the proximity metric, to *A*. Neighborhood set is useful in maintaining locality properties.

A routing table *R* has *l* rows and 2^b columns. *l* is the length of the node ID, $[log_2{}^bN]$, where *N* is the number of nodes in the system. In node *A*'s routing table, the node IDs at row *i* share a

common prefix of length *i* with *A*'s ID. And the node IDs in column *j* has value *j* at next digit after the common prefix with *A*'s ID. Figure 2.2 shows an example of node *A* with ID 10233102 based on 4 in a 16-bit node ID system. In this example, *b* is 2, *l* is 8, |*L*| and |*M*| are both set as $2\times2^b=8$. The routing table has 8 rows with $2^b-1=3$ entries each. In each entry, among all the nodes whose IDs satisfy the common prefix and next digit, only one is chosen such that it is closest to *A* according to the proximity metric.

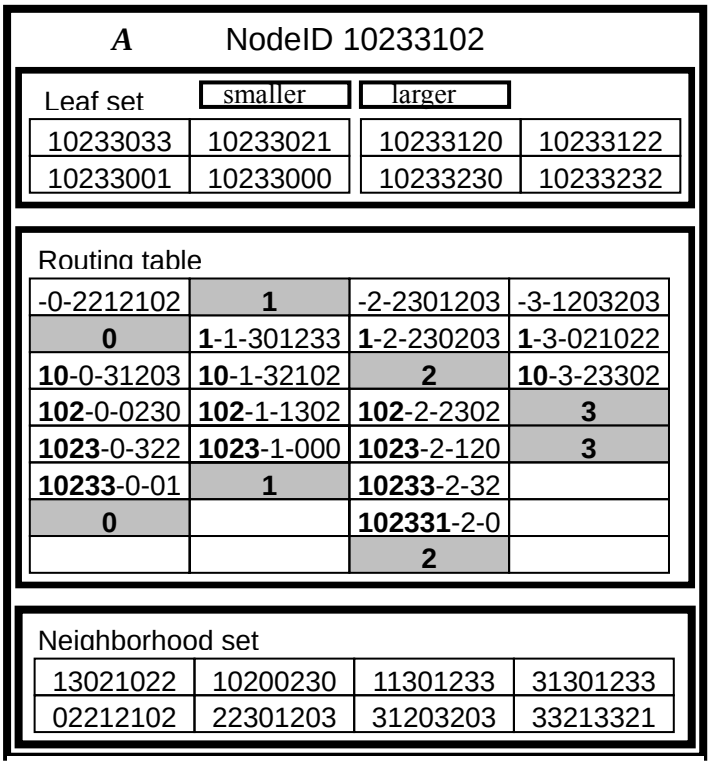


Figure 2.2 An Example of state in a Pastry node A

2.2.2 Searching in Pastry

The main idea of searching in Pastry is that, given a query of key *k*, a node *A* forwards the query to a node whose ID is numerically closest to *k* among all nodes known to *A*. At a Pastry node *A*, for a given query *Q* with key *k*, the routing algorithm is as following. First, if *k* falls within the range of node IDs covered by *A*'s leaf set, query *Q* is forwarded to a node in the leaf set whose node ID is numerically closest and closer than *A* to *k*. If *k* is not covered by leaf set range, routing table is looked up to find a node whose ID shares a longer common prefix with *k* than *A*' ID dose. Usually, a longer common prefix means a closer ID, hence the query is forwarded to such a node. If the appropriate entry in the routing table is empty, or the node in that entry failed, query *Q* is forwarded to a node in the leaf set whose ID has the same shared prefix as *A* does but is numerically closer to *k* than *A*. Finally, if such node can not be found, node *A*, therefore, is the destination node for this query.

This routing procedure of Pastry always converges, since each step chooses a node that either shares a longer prefix or shares the same long prefix, but is numerically closer to the key than the local node. If the leaf set and routing table is accurate and nodes in the table are alive, the expected number of routing steps is $log_2{}^bN$.

2.2.3 Properties of Pastry

As a DHT-based structured P2P system, Pastry has many good properties.

Pastry is completely decentralized. The construction as well as the search and update operations can be performed without any central control and global knowledge in an unreliable environment. Hence, all nodes serve as entry points for search.

Pastry is self-organized and automatically adapts to the node join. When a new node *X* joins the system, it assumes that *X* knows a nearest neighbor node, to say node *A*. *X* sends a join message with key *k*, the ID of node *X*, to *A* and *A* routes it as a general query according to the routing schema. Like any query, this join message is finally forwarded to a node *Z* whose node ID is numerically closest to *X*' ID. All the nodes along the route path from *A* to *Z* aware the arrival of *X* and update there states if necessary. Also, they send their states to *Z* to help *Z* initialize its state table. The join cost is $O(log_2{}^b N)$.

The expected routing number claimed last subsection is based on the assumption of accurate routing tables and no recent node failures. However Pastry is fault-resilient. Failed nodes in the leaf set can be corrected by connecting other nodes in the leaf set and asking for its leaf set. Hence a Pastry node can successfully forward a query unless $|L|/2$ nodes with adjacent node IDs have failed simultaneously. To repair an entry with a failed node *F* in the routing table, the local nodes connect another node in the same row with this entry and ask for its routing table to find an appropriate node.

Pastry is also scalable and support efficient search which is demonstrated by the experiment results.

2.2.4 Performance and Evaluation

The experiment result shows that the routing performance of Pastry is well bounded by $O(logN)$ with maintaining $O(logN)$ entries in the state table. The cost of join, in terms of the number of join messages exchanged is $O(logN)$, as discussed last subsection. The experiment also show that compared with complete routing table, Pastry is only cost 30% more in terms of the relative distance a query travel according to the proximity metric.

In Pastry there are several configuration parameters. *b* is the most important parameter that determines the power of the system. It can trade-off between the routing efficient ($log_2{}^b N$) and routing table size ($log_2{}^b N \times 2^b$). Each node in the system can choose its own value for $|L|$ and $|M|$ based on the node situation. These parameters make the system more flexible to meet different demand.

The author of Pastry declared that the routing schema always converges, since the routing is based on the prefix. This is not true. Since a longer shared prefix can no guarantee the closer distance in node IDs. For example, node *A* with ID 10233232 (base is 4) shares a longer common prefix (of

length 6) with key k=10233200 than node B with ID 10233133 dose (of length 5). However, node B is numerically closer to k than node A. Therefore, Pastry routing schema is not always local optimal.

2.3 P-Grid

P-Grid [Aberer2001] [ACD+2003] is a P2P lookup system based on a virtual distributed search tree. Since the structure of P-Grid is similarly to standard distributed hash tables, it is classified into DHT-based system. P-Grid can provide an advanced P2P infrastructure targeted at application domains beyond mere file-sharing.

2.3.1 Structure of P-Grid

In P-Grid, all the nodes are divided into m groups, where $m \leq N$ (N is the number of nodes in the network). The m groups are organized as a virtual binary search tree which is distributed among groups of nodes. Each group as well as nodes inside it is assigned a binary bit string, which is the path of the group from the root of the search tree and each group only takes over part of the overall search tree. This means that the data space is divided into m intervals and each is stored on one group. In other words, each group stores data (or key of data) with a prefix that is same as the path of this group. Since m can be smaller than N, this means that there can be more than one nodes in one group and these multiply nodes responsible for the same path and data. For example, as show in Figure 2.3 both node 1 and node 6 is assigned path 00 and responsible for data with prefix 00. Such replication is for fault-tolerance purpose since it assumes that nodes are not online all the time but with a very low probability.

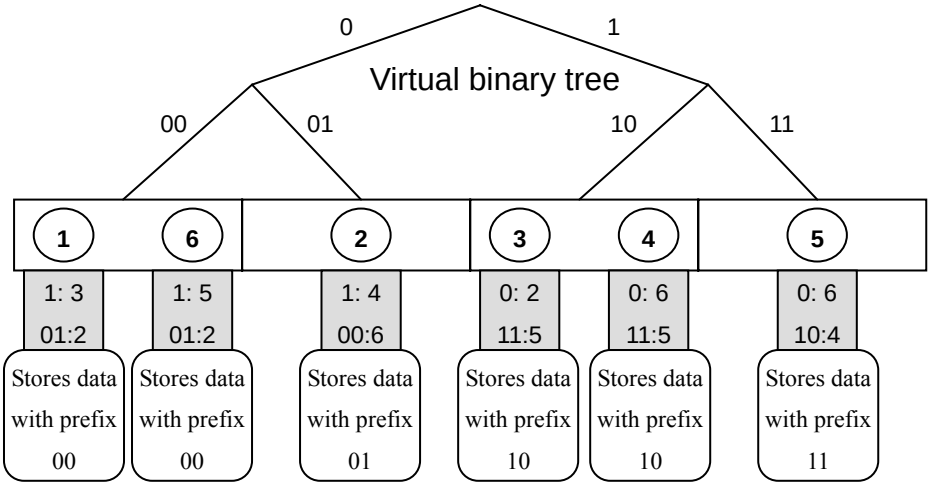


Figure 2.3 Example P-Grid

In each node, routing information is stored as a hash table, as shown in the gray part in Figure 2.3. The hash table keeps a set of pairs of prefix and node ID, which indicates the node that is closer to a certain key.

The construction of a P-Grid is based on a distributed, randomized algorithm. The construction algorithm guarantees that there always exists at least one routing path between any two nodes. This means that when a query is sent to any node in P-Grid, it can be forwarded to the destination node using the routing table maintained in the nodes. However, since the construction algorithm is based on a purely randomized algorithm, nodes in the same group may have different routing table. For example, in Figure 2.3 node 1 and node 6 has the same path and store the same data, but the hash table is not the same.

2.3.2 Searching in P-Grid

Searching in P-Grid is simple and efficient. The searching procedure is as following. A query with a key (a binary bit string) is sent to an arbitrary node in P-Grid. The node first checks the path of itself and the key of the query. If the path is a prefix of the key, it self is the destination node for this query. Else, the routing table is looked up, and an appropriate node is chosen to forward the query. The query routes in the network until arrive a destination node that store data with such key. As mentioned in last subsection, the construction algorithm guarantees that any query can be routed to a destination node in finite steps.

Figure 2.4 shows a simple example of searching in P-Grid. A query with key=100 is sent to node 6 whose path (00) is not a prefix of the query key, hence, the query is forwarded to node 5, which is in the routing table of node 6 and assigned with prefix 1. Node 5 still can not satisfy the query and according to the routing table, it forwards the query to node 4, which is destination node for this query.

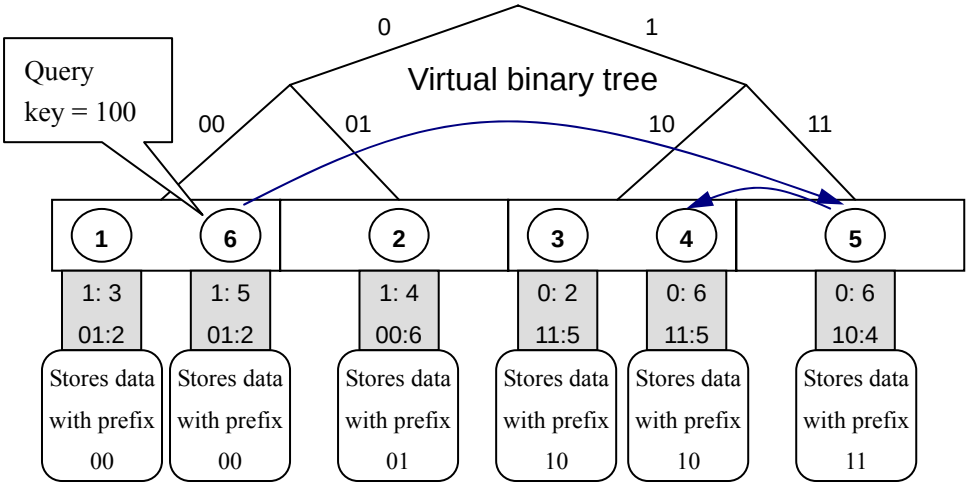


Figure 2.4 Example of query in P-Grid

2.3.3 Properties of P-Grid

The basic idea of the P-Grid system is to build a virtual binary search structure with replication that is distributed over the nodes and supports efficient search. It is comparable to some other P2P system such as [IVB1997] and [YKM1999] to construct scalable, tree-based, distributed indexing structures.

P-Grid exhibits many good properties as following.

It is completely decentralized, which is achieved by applying randomized, distributed algorithms. All algorithms including construction, update and search are realized through completely decentralized cooperation among the nodes. Consequently all nodes serve as entry points for search and interactions are strictly local.

P-Grid is scalable gracefully with the total number of nodes and data items. And, probabilistic estimates for the success of search requests can be given.

Furthermore, P-Grid is fault-resilient. Search is robust against failures of nodes due to the replication of data among the group.

Another property of P-Grid is that the path assigned to each node is not relative to the ID of the node. This is in contrast to other DHT-based P2P systems such as Chord and Pastry we have discussed previously.

2.3.4 Performance and Evaluation

Searching in P-Grid is efficient and the search time and number of generated messages grow $O(logN)$ with the number of nodes N in the network. With this expected cost it is fast even for unbalanced trees, since for a randomized selection of pointers to other nodes in the routing tables, probabilistically the search cost remains logarithmic, independently of the length of the paths occurring in the virtual tree [Aberer2002]. The experiment results show that it is efficient in search and update operation.

However, the efficient of P-Grid is based on the assumption that data distribution and behavior on nodes are uniform. If data/query distribution is skewed, P-Grid is not able to balance the load automatically.

2.4 CAN

A d-dimensional space is used to implement the DHT in CAN [RFH+2001]. The space is divided into hyper-rectangles, called zones. Each node in the system is associated with a zone and the boundaries of its zone are used as the identifier of the node. A node A maintains $2d$ pointer to its neighbors, nodes in the system whose zone share a d-1 dimensional hyperplane with A. The neighbor pointers in a node make up of the routing table for it.

In CAN, a key is mapped onto a point in the d-dimensional space. The searching algorithm is that in each step, a node forwards the query with key k to one of its neighbors, whose zone is approximately nearer to the coordinate of k than the local node. With each node maintaining $O(d)$ pointers, the searching cost is $O(dN^{1/d})$.

3 Two Important Issues in DHT-Based P2P System

There are many issues in the DHT approach needing worth more investigation. Among them, load balancing and neighbor-table consistency preserving are two important and challenging problems. The following subsections will discuss methods addressing them.

3.1 Load Balancing

3.1.1 Problem

In a P2P system using DHT approach, there is a $\Theta(\log N)$ imbalance factor in the number of items stored at a node, if node and item IDs are randomly chosen, where N is the number of nodes in the system [SMKKB2001]. The case will be even worse if the application associates semantics with the item IDs where IDs are no longer uniformly distributed. The difference in the capabilities of nodes can exacerbate the problem of load balance.

However, to balance the load in a P2P system is not easy. The challenges lie at the facts that, data items are continuously inserted and deleted; nodes join and depart the system continuously, and the distribution of data item IDs and item sizes can be skewed.

There is also a trade-off between the load balance and the amount of load moved. Intuitively, the better load balance, the more and more frequent movement of data. Thus, how to achieve load balance with tolerate load movement in a dynamic P2P system is a challenging problem.

3.1.2 Early works

[SMKKB2001][RLSKS2003] and [BCM2003] proposed some methods to address the load balancing problem. However, all of them assume a static system and most assume a uniformly distributed node- and item-ID- space.

3.1.3 One Solution

The paper [GLSKS2004] proposed an idea to achieve load balance by using the concept of directory and virtual node. The load information of each physical node is stored in a number of directories. It is these directories that periodically schedule reassignments of virtual servers among nodes to achieve better balance. By doing so, the distributed load balancing problem is reduced to a centralized problem at each directory.

In the system, each data item is associated with load, which is the amount of resources consumed on the physical node that it resides in. Each item also has a movement cost, charged each time we move the item between physical nodes. The *load* l_i of a node i at a particular time is the sum of the loads of the items stored on that node at that time. The node i also has a fixed limit of resources, *capacity* c_i. The utilization u_i of node i is l_i/c_i. If it is lager than 1, the node is said to be *overloaded*;

otherwise, *underloaded*. For all nodes in a P2P system, the *maximum node utilization* refers to the utilization of the node who has the highest utilization.

The concept of virtual server was previously proposed in [DKKMS2001]. A *virtual server* represents a peer in the DHT, which means the storage of data items and routing happen at the virtual server lever rather than at the *physical node* level. A physical node can host more than one virtual server, as shown in Figure 3.1.

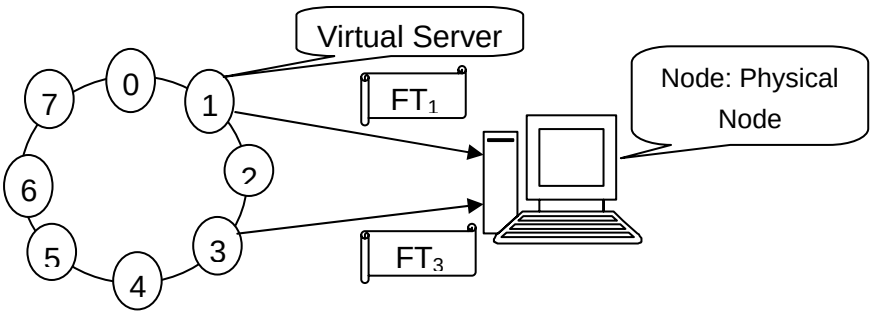


Figure 3.1. An example of virtual servers and physical nodes.

The behavior of the directories and nodes can be summarized in the flowcharts in Figure 3.2.

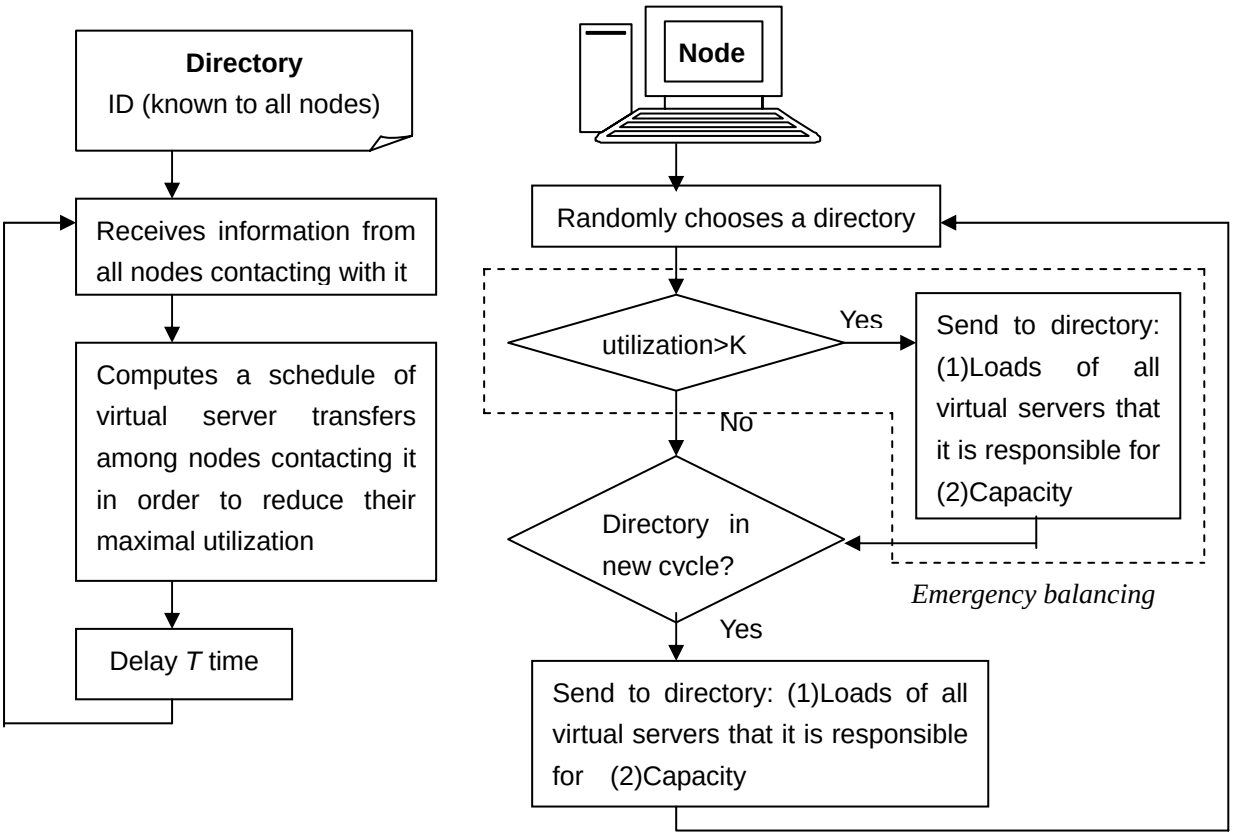


Figure 3.2 Flowcharts of load balancing algorithm for directory (left) and node (right). The part bounded by dashed-lines is called *emergency balancing*.

However, to compute an optimal virtual server reassignment in order to minimize the maximum node utilization is NP-complete. Thus, the authors proposed a greedy algorithm which runs in O($m$log$m$) time. In the algorithm, the least loaded virtual server of each heavily loaded node is moved into a *pool*. Then, for each virtual server in the *pool*, from the heaviest to the lightest, assign to a node which minimizes the resulting load.

3.1.4 Performance

The algorithm is evaluated by looking at the load movement together with the maximum node utilization. As mentioned before, there is a tradeoff between these two factors. Experiments show that, if we decrease the period (the time T a directory delays before collecting information from nodes for the next round), the maximum node utilization decreases while the movement increases. The algorithm seems effective because, even for system utilization as high as 0.9, it can keep 99.9 percent of the nodes underloaded while incurring a loa movement factor of less than 0.08. The load moved is considerable smaller than the load moved by the underlying DHT especially for small system utilization. The *emergency balancing* strategy is necessary to achieve good performance. It is also found that, the number of directories in the system has small effect, and an increased number of virtual servers helps load balance at fairly high system utilization, particularly in load movement. However, good balancing could be achieved with many fewer virtual servers per node when node capacities are heterogeneous than when they are homogeneous.

3.2 Neighbor-table Consistency

3.2.1 Problem

Recall that in hypercube routing scheme, such as Pastry, neighbor-tables are used to keep neighbor pointers for efficient routing. It's a challenge to maintain neighbor-tables consistent in a dynamic P2P system where nodes may joins, leave and fail concurrently and frequently.

Formally, a *consistent network* is one where, for every entry in neighbor tables, if there exists at least one qualified node in the network, then the entry stores at least one qualified node. A node x is a *qualified node* for an entry of a node's neighbor table, if x's ID has suffix same as the required suffix of that entry. If nodes may fail frequently in a network, a natural approach to improve robustness is to store in each table entry multiple qualified nodes. Thus comes the concept of *K-consistent network*: for every entry in neighbor tables, if there exist H qualified nodes in the network, then the entry stores at least min(K,H) qualified nodes. The goal of neighbor-table consistency problem is to maintain the P2P network as an K-consistent network while optimizing neighbor-tables.

3.2.2 General Strategy for Consistency-Preserving Optimization

It is observed that, for the hypercube routing scheme, within a subnet that is already consistent, replacing any neighbor with any other neighbor does not break consistency conditions if both neighbors belong to the consistent subnet. Thus, a general strategy for consistency-preserving optimization would be:

(1) Identify a consistent subnet as large as possible.
(2) Only allow a neighbor to be replaced by a closer one if both belong to the subnet.
(3) Expand the consistent subnet after new nodes join; and maintain consistency of the subnet when nodes fails.

Thus, what needs to be done is to:

(1) Design a join protocol so that at any time, a set of well-defined nodes form a consistent subnet.
(2) Design a failure recovery protocol that recovers *K*-consistency of the subnet by repairing holes left by failed neighbors with qualified nodes in the consistent subnet.

3.2.3 Join Protocol

[LL2004b] proposed a Join Protocol which is able to construct and maintain *K*-consistent neighbor tables for an arbitrary number of concurrent joins. In the protocol, each node maintains a state variable *status*, which can take value of `copying`, `waiting`, `notifying`, `cset_waiting`, and `in_system`. A node is called an *S-node* iff its status is `in_system`; otherwise, *T-node*. All the *S-nodes* form a *K*-consistent subnet. Figure 3.3 describes the process how a newly joining node, say *x*, changes its status and finally becomes a member of the subnet.

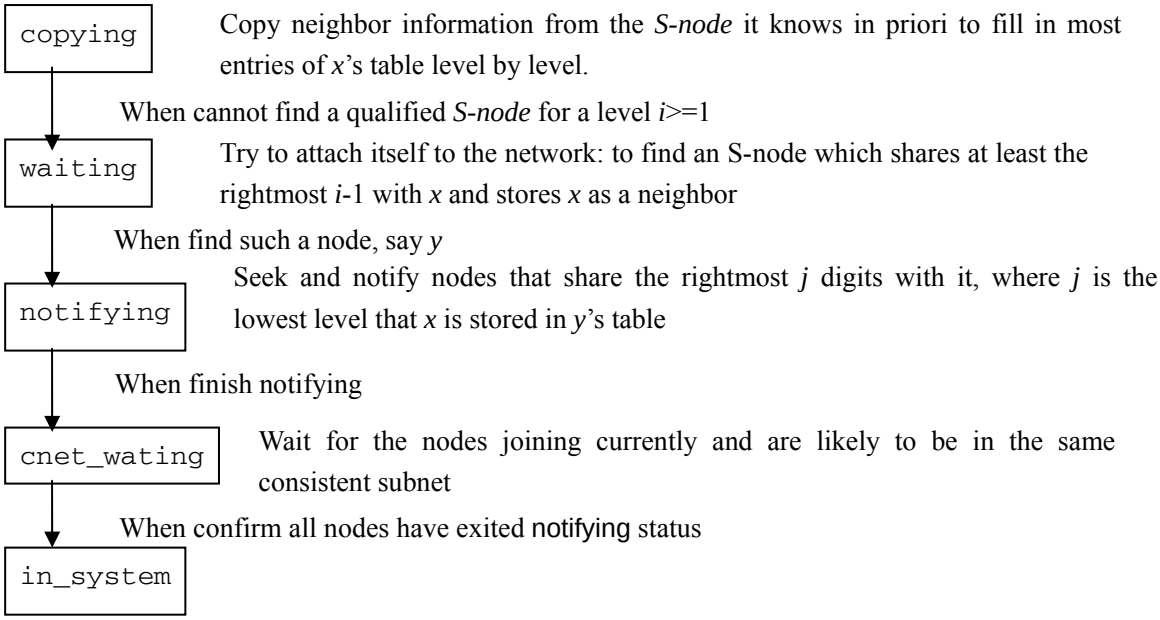


Figure 3.3 Status changes of a newly joining node *x* according to the join protocol.

The `cnet_waiting` status ensures that when two nodes have both become *S-nodes*, paths between them (in both directions) have already been established.

3.2.4 Neighbor-table Optimization

It is desired to have a set of nearest neighbors in a nodes' neighbor-table. Three heuristic algorithms are proposed by [LL2004b] to optimize neighbor-tables. Whichever algorithm, the optimization rule must be obeyed: when a node, *x*, intends to replace a neighbor, *y*, with a closer one, *z*, the replacement is only allowed when both *y* and *z* are *S-nodes*.

(1) Copy neighbor information from nearby nodes. This is done at the `copying` stage. Instead of directly copy neighbors from the S-node known in priori, say *y*, the joining node chooses the closest node in *y*'s neighbors to copy.

(2) Utilize protocol messages that include copies of neighbor-tables. Some messages exchanging in the join protocol include neighbor-table of the sender, such as during status `waiting` and `notifying`. The receiver thus can search for qualified nodes that are closer than the current ones in its neighbor-table.

(3) Optimize neighbor-tables when a node's join process terminates. When a joining node becomes an *S-node*, it sends message to its neighbors and reverse-neighbors. The receivers of these messages can then optimize their neighbor-table entries. In addition, the joining node also exchanges neighbor-table with its neighbors. Thus, the joining node can optimize its neighbor-table too.

3.2.5 Performance

Metric used is *p-ratio*, which is defined as below: for a table entry of a node, say *x*, suppose the primary-neighbor of the entry is *y*, and the closest node among all qualified nodes of the entry is *z*, then, *p-ratio of the entry* is the ratio of the communication delay from *x* to *y* to the delay from *x* to *z*. Neighbor-tables are optimal if for every table entry in a network, p-ratio is 1.

Experiments results show that, by primarily using information carried in join protocol messages, table entries can be greatly optimized. *K-consistency* was always maintained after all joins had terminated. It is observed that, when *K* is increased, the average p-ratio decreases. The reason is that, when *K* becomes larger, more neighbors are stored in each table entry, thus more information is carried in protocol messages. There is a tradeoff between the benefits and maintenance costs of *K-consistency*.

In the 2nd part of the experiments, the join protocol and optimization strategies of [LL2004b] are combined with the failure recovery protocol proposed by [LL2004a]. When there are massive joins and failures, in all experiments with $K \geq 2$, *K*-consistency was maintained when all join and failure recovery processes had terminated, and the table entries were optimized greatly: average p-ratio were less than 3. In another set of experiments, the join rate and failure rate followed Poisson distribution. It's shown that, given the rate and network size, their protocols can sustain a large stable "core" over the long term even when joins, failures, and neighbor-table optimization happen currently.

4 Comparisons of DHT-based Techniques

4.1 Motivation

Each Distributed Hash Table (DHT) Algorithm has many details which makes it difficult for comparison and analysis. [DRGR2003] The paper attempts to use a component-based analysis approach. It does so by breaking the DHT design into independent components and analyzing impact on each component source separately.

The two levels of components which will be examined are routing level components, (deals with neighbor and route selections) and system level components (deals with caching, replication, querying policy & latency). The metrics used for the comparison of the DHT include the following:

- Flexibility in the options that the algorithm has in neighbors and routes selection.
- Static resilience refers to how well the algorithm routes in the presence of node failures. This can measure how long before recovery algorithms should function.
- Proximity refers to how close the content is being stored. Content stored neared source nodes can be accessible faster. Normally measured in hop counts.
- Latency refers to the time gap between the source and the destination nodes.

4.2 Analysis of DHT's

Each DHT comprise a routing algorithm which specifies the rules for selecting neighbors and routes using the distance function (eg. CAN, Chord, PRR, Tapestry, Pastry), the geometry which specifies the underlying structure derived from the way in which neighbors and routes are chosen (Chord routes on a Ring) and the distance function which specifies the number of hops between two nodes.

The table on the right summarizes the routing algorithms and their underlying geometries that we will compare. All geometries are capable of providing $O(\log n)$ path length with $O(\log n)$ neighbors (Viceroy improve that to $O(1)$ neighbors).

Geometry	**Algorithm**
Tree	PRR
Hypercube	CAN
Butterfly	Viceroy
Ring	Chord
XOR	Kademlia
Hybrid	Pastry

4.2.1 Flexibility

Let us examine the flexibility in neighbor & route selection. Neighbor selection flexibility implies that the algorithm chooses neighbors based on proximity leading to shorter paths. Route selection flexibilities specify the number of options the algorithm has in selecting the next hops, higher options tend to have shorter and reliable paths.

Hypercube Geometry is used by the CAN which routes along a d-torus. CAN node's identifier is a binary string rep of its position in space. Each node has log n neighbors, neighbor *i* differs from

the given node only on the i^{th} bit. Routing is done greedily by correcting bits on which forwarding node differs from destination

- The distance between two nodes is the number of bits on which the identifier differs
- Greater routing flexibility as the algorithm has ($log\ n$)! nodes for next hop.
- No neighbor selection flexibility as neighbors differs from itself on exactly one bit.

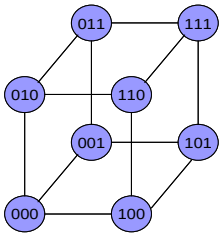


Figure 4.1 Hypercube Geometry

Tree Geometry is used by the PRR algorithm. Hierarchical organization of the tree makes it efficient in routing. The nodes in a tree-geometry constitutes of leaf nodes in a binary tree. Routing in a tree is achieved by successively "correcting" the highest order bit on which the forwarding node differs from destination, making a step closer to destination.

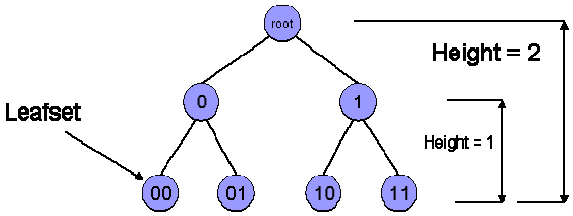


Figure 4.2 Tree Geometry

- The distance between two nodes is the height of their common sub-tree.
- Neighbor selection flexibility - has 2(i-1) options of choosing neighbor at distance i.
- No route selection flexibility as a node has only one neighbor that reduces the distance to destination.

Butterfly Geometry is used by the Viceroy but adapts the structure to be self-organising and robust in the face of nodes arrival and departure. Viceroy imposes a global ordering of all nodes in the system and requires each node to hold its immediate successor and predecessor as its neighbors in the ordering. Nodes are organized in a series of log n "stages" where all the nodes at stage i are capable of correcting the i^{th} bit in the identifier.

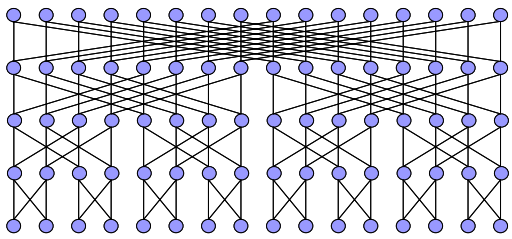


Figure 4.3 Butterfly Geometry

- Does not permit flexibility in route selection and neighbor selection.

Ring Geometry is used by the Chord. Nodes lie on a one-dimensional cyclic identifier space. Each node in Chord maintains log n neighbors apart from the immediate neighbor and can route to an arbitrary node in log n hops. The figure on the right show 2 routes selection possibilities to route a message from Node "000" to "110".

- Distance between nodes is the clockwise distance to destination on the circle.

- Flexibility in neighbor selection, node has 2^i possible options to pick its i^{th} neighbor for an approx of $n^{((\log n)/2)}$ possible routing table for each node
- The algorithm has (log n)! possible routes to route from a source to destination.

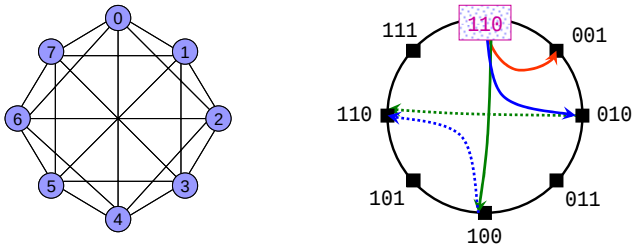


Figure 4.4. Ring Geometry

XOR Geometry is used by Kademlia. Each node picks log n neighbors where the i^{th} neighbor is any node within an XOR distance from itself. Routing is done greedily.

- Distance between nodes is XOR of their identifier.
- Node has $2^{(i-1)}$ options of choosing neighbor at i^{th} distance. Thus each node has approximately $n^{\log n / 2}$ entries per routing table.
- Route flexibility is provided by fixing lower order bits before fixing the higher bits if an optimal path is not available.

Hybrid Geometry employed by algorithms that employ dual
modes, where each mode inspires a different geometrical representation. Pastry use both a tree with a ring geometry. Nodes are regarded as both leaves of a binary tree and points to a 1-d circle.

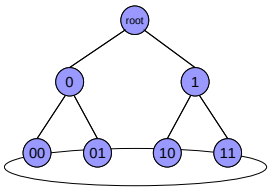


Figure 4.5. Hybrid Geometry

- Distance is either tree distance or the cyclic distance between nodes
- Node has $2^{(i-1)}$ options of choosing neighbor at i^{th} distance. Thus each node has approximately $n^{\log n / 2}$ entries per routing table.
- Route selection freedom is provided by taking hops that do not make progress on the tree, but on the ring –paths might not retain the *O(log n)* bound on routes.

In conclusion, the ring and hypercube geometries have twice the routing flexibilities of the Hybrid and XOR geometries.

Property	**Tree**	**Hypercube**	**Ring**	**Butterfly**	**Xor**	**Hybrid**
Neighbor selection	$n^{\log n / 2}$	1	$n^{\log n / 2}$	1	$n^{\log n / 2}$	$n^{\log n / 2}$
Route Selection (optimal)	1	c1(log n)	C1(log n)	1	1	1
Natural support for sequential neighbors?	no	no	yes	No	No	Deafult – no Fallback – yes

Figure 4.6 – Overview of geometries and flexibility

4.2.2 Static Resilience

Static resilience determines how well the geometry operates in a transient environment where node goes up and down frequently. A measure is to see how well each algorithm continues to routes when a percentage of nodes have gone down and recovery algorithms have not been executed. This is important as it specifies how fast and frequently recovery algorithms need to be executed to re-populate the routing tables with life nodes.

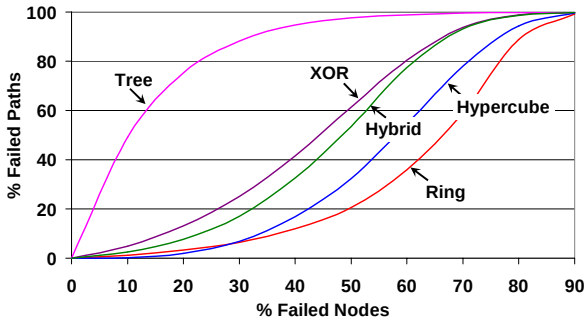


Figure 4.7 – Measure of static resilience

A test was conducted on a 65,536 node network where node failure percentages were varied. The following was observed for the different routing algorithms when 30% of the nodes in the network failed.

- Tree - 90% routes failed
- Ring, Hypercube – 7% routes failed
- Hybrid, XOR - 20% route failed

Ring & Hypercube has most routing flexibility and thus better resilience. However, when sequential neighbors were added to each algorithm, no route failures were observed in any geometry when 30% of the nodes had failed.
It is noticed that the more routing selection flexibility, the more resilience the algorithms are to node failures. This shows consistent with the degree of route selection flexibility.

4.2.3 Path Latency

The goal of path latency is to minimize the end-to-end latency of the networks. The author proposes two proximity methods when comparing the DHTs - Proximity Neighbor Selection (PNS) and Proximity Route Selection (PRS). PNS specifies that neighbors are chosen on their proximity (eg. Tree, Ring & XOR geometries implement this method) and PRS specifies that routes are selected depending on the proximity of the neighbors (eg. Hypercube, Ring & XOR implement this method).

For the purpose of evaluating latency, the topology of the underlying network along with link latencies is required. Hence the author used the latency distribution of the Internet. From the graph plotted, it was found that algorithms that implement the PNS & PRS method do indeed experience lower path latencies, hence finding shorter paths. However, the PRS methods leads lead to variations in hop counts.

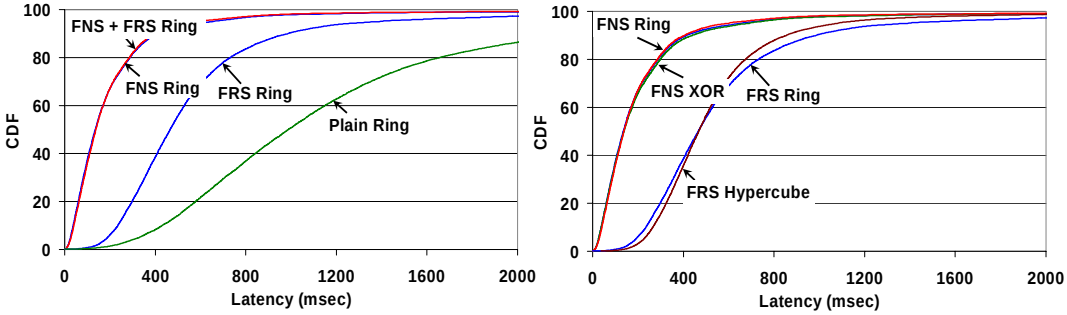


Figure 4.8 – Path Latencies among Ring, XOR, Hypercube geometry

It was noted that PNS achieves great improvement over PRS & Plain version and PRS achieve significant improvement over Plain version. PNS deterministically pick its i^{th} neighbor selected from any 2^i nodes whereas PRS chooses from any of its first i^{th} neighbors, but each neighbor is deterministically chosen. Since PNS has more options, it has improved performance. Not all geometries can implement PNS & PRS algorithm. Hence it is important to choose a routing algorithm that has a geometry that accommodates PNS. The XOR & Ring geometry can accommodate both PNS & RPS

4.2.4 Local Convergence

Local Convergence refers to the property that two messages send from two nodes nearby (in terms of latency), addressed to the same location converges at a node near the two sources. This is important as this property lead to low latencies and/or bandwidth savings in the usage of DHT's. A best case scenario would be that only one node from one domain sends a message to another domain.

The author compared the convergence of the Ring Tree & XOR Geometries. In the proximity methods identified earlier, PRS algorithm does little to limit the number of exit points in a network and PNS algorithm improves local convergence.

4.3 Limitations & Conclusion

The paper [DRGR2003] concludes that routing geometry & flexibility is important as it improves resilience & proximity. The Ring & XOR geometry are flexible in choosing neighbors and routes and are able to implement both the proximity methods : PNS & PRS. The paper favors the Ring geometry based on its flexibility and its highest performance in resilience tests. The limitation of the paper is that it has not considered all geometries and that it has also not considered other factors and performance metrics in the comparison.

5 Non-DHT Structured P2P System-SkipNet

The SkipNet adapts the Skip List Data Structure for maintaining a sorted list of data records as well as pointers that "skip" over varying number of records. Let us look at SkipList Data Structure before we proceed.

5.1 SkipList [PSL1990]

Skip list are data structures tat can be used in place of balanced trees. It makes use of probabilistic balancing techniques hence algorithms are simpler and faster. SkipList can be described as a sorted linked list in which some nodes are supplemented with pointers that skip over many list elements. A perfect skip list is one where the height of the i^{th} node is the exponent of the largest power-of-two that divides *i*. Pointers at level *h* have length 2*h*. A perfect skip list supports searches in *O(log N)*.

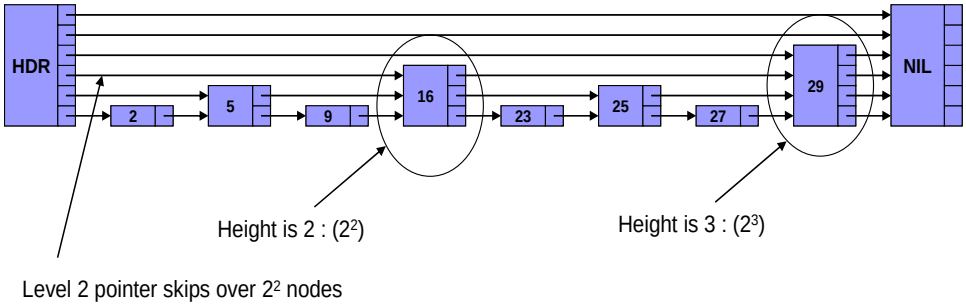


Figure 5.1 A Perfect Skiplist.

Because it is expensive to perform insertion and deletions in a perfect skip list, a probabilistic balanced skip list is proposed by consulting a random number generator. Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. The algorithm is presented below.

Initialisation: a NIL element is allocated and given the highest key.

Insertion and Deletion: Uses the search and splice methods. If an insert or deletion increases / decreases the max level of list, we update max level.

Searching: Conducted by traversing pointers that do not overshoot the node containing the element searched. If there is no progress, we move down to the next level. At level 1, we should be at the desired node or the element does not exist.

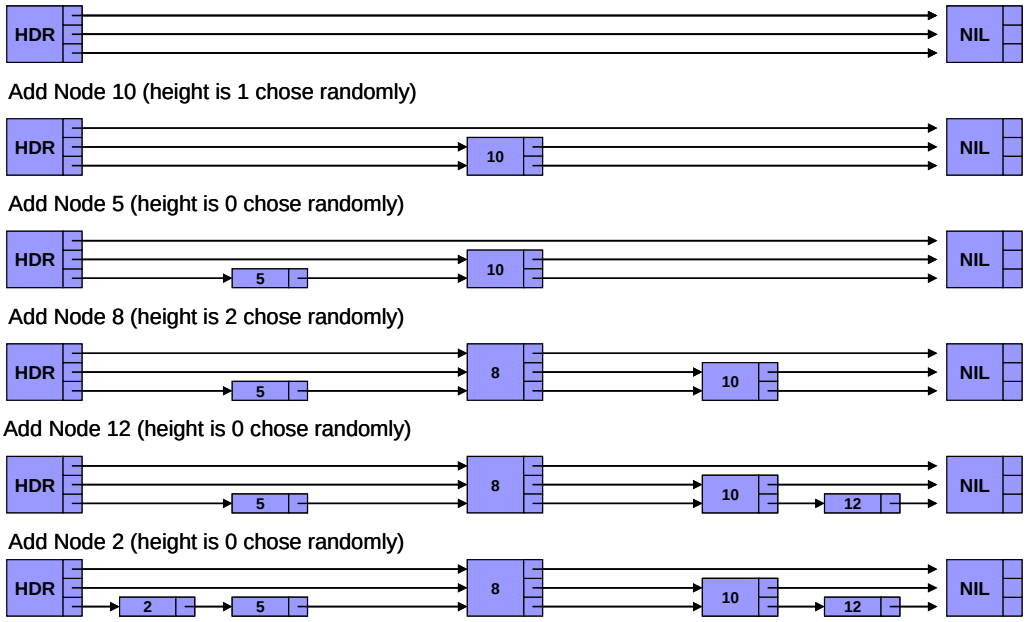


Figure 5.2 – Insertions in a probabilistic skiplist.

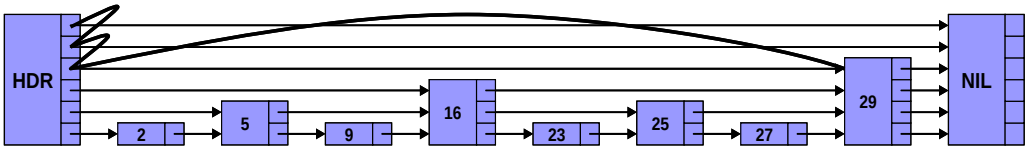


Figure 5.3 – Searching for Node 29 in a SkipList.

SkipList is space efficient as it can use approximately 1.33 pointers per element and maintains a *O(log N)* searches with high probability. When the list is significantly unbalanced, it can lead to a worse case performance. In comparison, SkipLits is slightly slower than AVL trees in searches, but insertions and deletions are faster. SkipList is faster than recursive 2-3 & self-adjusting trees when a uniform distribution is encountered, but slower for highly skewed distributions

5.2 Motivation

While DHT have load balancing properties (allows data to be uniformly diffused over all the participants in a P2P system), they do not allow us to control where the data will be stored. This gives rise to two issues – Content & Path Locality. Content locality emphasis the explicit placement of data on specific overlay nodes or distributed across nodes in a specified domain, hence it can overcome traffic analysis & denial-of-service attacks. Path Locality guarantees that routing path between two overlay nodes in a domain does not leave the domain hence security is enforced.

SkipNet [SNL2003] is a scalable overlay network that provides controlled data placement and

guarantee routing locality by organizing data by string names. SkipNet allows content to be placed on pre-defined node or distributed uniformly across nodes of a hierarchical naming subtree. SkipNet employs two separate but related address spaces: a string name id and a numeric id space. Node names and content identifier string are mapped into a name ID. The hash of the node names and content identifier string is mapped into a numeric ID. By arranging content in name id order rather than dispersing it across the system, we are able to achieve content & path locality.

With locality of content, we have improved availability, performance, manageability & security. Data stored within a domain can still be searched even if the network disjoins. Because of the SkipNet structure, it is resilience against Internet failure as nodes within a cluster gracefully survive failures that disconnect clusters from the rest of the Internet. Furthermore searches for local content are faster as data is stored nearer. Moreover, by explicit placement of data, we can administer control and deal with issues like traffic analysis and denial attacks.

5.3 Structure of SkipNet

SkipNet adapts the Skip List Structure maintaining a sorted list of data records as well as pointers that "skip" over varying number of records. Data records are replaced with computer nodes, using the string name ID of the nodes as the data record keys. The topology of the SkipList is changed from a list to a ring which is doubly-linked because the traversal can start from any node in any direction. Each node in the SkipNet stores 2 log N pointers rather than a high variable number of pointers. Each node has a routing table (R-table) and the pointer in the R-table at level h point to nodes that are roughly 2h nodes to the left and right of the given node.

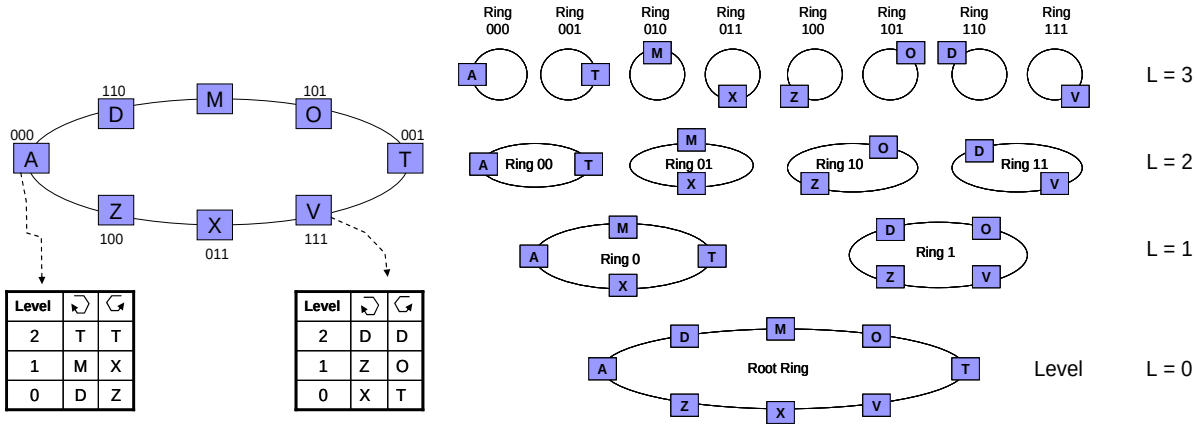


Figure 5.3 – SkipNet Structure.

The diagram on the right depicts the SkipNet from the left, arranged to show how the nodes are interconnected at each level in the routing table. All nodes are connected to the root ring formed by each node's pointers at level 0. The pointers at level 1 point to nodes that are 2 nodes away and hence the overlay nodes are divided into two distinct rings. The same goes for level 2 and level 3.

Routing in the SkipNet is done the String Name ID or the String Numeric ID.

Routing By String Name ID is similar to search in Skip List. The message is routed from the highest level pointer in either clockwise of counter-clockwise direction such that the name ID does not past the destination value. The message terminates when it arrives at a node whose name Id is closest to the destination. Because nodes are doubly linked, scheme can route in both left and right direction. Number of hops is *O(log N)*

Example:

Routing a message from Node A to Node V

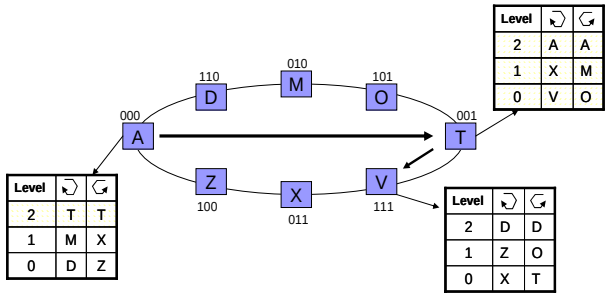
Chosen Path:

A (Level 2, clockwise) → T, "T" < "V"

T (Level 2, clockwise) → A, Failed

T (Level 1, clockwise) → X, Failed

T (Level 0, clockwise) → V




```
SendMsg(nameID, msg) {
    if( LongestPrefix(nameID,localNode.nameID)==0 )
        msg.dir = RandomDirection();
    else if( nameID<localNode.nameID )
        msg.dir = counterClockwise;
    else
```

```
// Invoked at all nodes (including the source and
// destination nodes) along the routing path.
RouteByNameID(msg) {
    // Forward along the longest pointer
    // that is between us and msg.nameID.
    h = localNode.maxHeight;
    while (h >= 0) {
        nbr = localNode.RouteTable[msg.dir][h];
        if (LiesBetween(localNode.nameID, nbr.nameID,
msg.nameID, msg.dir)) {
            SendToNode(msg, nbr);
```

Figure 5.3 – Example of routing by Name ID & algorithm.

Routing By String Numeric ID starts from level 0 until a node is found whose numeric ID matches the destination numeric ID in the first digit. Messages forwarded from ring in level h, R_h, to a ring in level $h+1$, R_{h+1}, such that nodes in R_{h+1} share $h+1$ digits with destination numeric ID. The routing terminates when the message is delivered to node with the correct numeric ID. If none of the nodes in R_h share $h+1$ digits with destination numeric ID then we pick node with numeric ID that is closest to destination's ID. Figure 5.4 depicts an example.

The benefits of the Skip Net support routing with the same data structure by name ID and numeric ID. Bottom ring in the SkipNet is sorted by name ID and top rings are sorted by numeric ID. For a given node, the SkipNet rings to which it belongs to precisely form a Skip List that is a ring & double linked.

Example:

Route from A→ 1011.

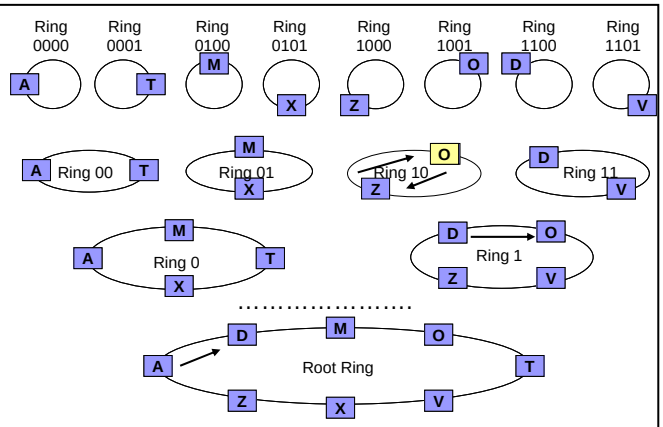
Path: A(0000) →D (1100 – move up level)

→O (1001 – move up level)

→Z (1000)

→O (1001 – closest match for 1011)

(deliver).




```
// Invoked at all nodes (including the source and destination nodes) along the routing path.
// Initially:    msg.ringLvl = -1,    msg.startNode = msg.bestNode = null & msg.finalDestination = false
RouteByNumericID(msg) {
        if (msg.numID == localNode.numID || msg.finalDestination) {
                DeliverMessage(msg.msg);
                return;
        }
        if (localNode == msg.startNode) { // Done traversing current ring.
                msg.finalDestination = true;
                SendToNode(msg.bestNode);
                return;
        }
        h = CommonPrefixLen(msg.numID, localNode.numID);
        if (h > msg.ringLvl) {    // Found a higher ring.
```

Figure 5.4 – Example of routing by Numeric ID & algorithm.

When a new node joins the SkipNet, it finds the top level ring that matches its numeric ID. Using the name ID search, it finds a neighbor in the top-most ring. Starting from one of the neighbors, it searches for its name ID at the next lower level and thus finds neighbors at the lower level. This process is repeated until it reaches the roots. The existing nodes only point to the new node only after it has joined the root ring.

Example:

Join - Insert node O (101)

- Search by numeric ID 101
  - Highest attainable level is 2
  - joins ring containing Z at level 2
  - Z forwards message to D at lower level
- Proceed by searching in lower levels
  - D, V are neighbors in level 1
  - M, T are neighbors in level 0

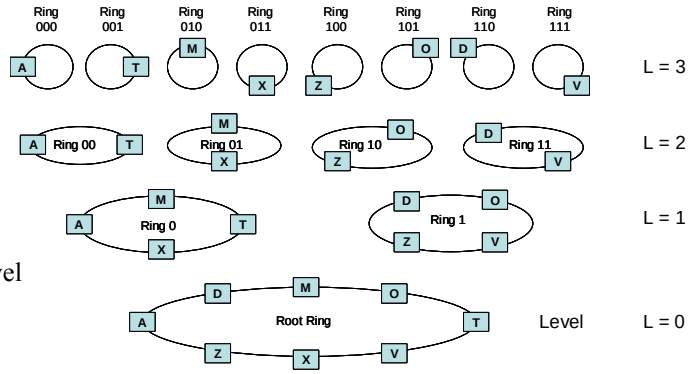


Figure 5.5 – Example of node joins.

5.4 Properties of SkipNet

SkipNet provides content and path locality. We can name nodes like a DNS entry. Hence by reversing DNS names (*john.microsoft. com* becomes *com.microsoft.john*), we can maintain path locality for groups in which nodes share a single DNS suffix. Furthermore, by incorporating name ID into content names will guarantee that content will be hosted on the specified node. Specifying "com.microsoft.john/ secretDoc.html" will store the content of secretDoc.html onto the node "john.microsoft.com".

SkipNet also provides constrained load balancing (CLB). SkipNet stores documents using two parts – a CLB Domain and CLB suffix. For example a document using the name "msn.com/DataCenter!TopStories.html". This specifies that the name ID to be store on is "msn.com/DataCenter" and the numeric ID is the hash of the "TopStories.html". Searching for node in the CLB Domain is done by using the name ID search first. Then a search by numeric ID for the hash of the CLB suffix constrained by domain ID.

SkipNet is also fault tolerance. The nodes in a SkipNet only maintains correct neighbors at the root level (level 0). Each node at level 0 has 16 neighbors. A background stabilization mechanism is executed whenever a failure is encountered. Also failures across organizational boundaries only segments the network and thus the SkipNet gracefully survives and once the failure is rectified, it gracefully joins the network again.

SkipNet also provides security. Nodes cannot create global names containing suffix of registered domains. Hence, administrative domain maintenance is controlled. Furthermore, path locality ensures that local paths do not leave the domain and thus can avoid traffic analysis. However, out-bound traffic is still prone to analysis easily.

5.5 Proposed Enhancements & Design Alternatives

The author proposes the following enhancements to the SkipNet algorithm. Use a density parameter k and a non-binary random digit to the base of k for representing numeric ID of nodes. This will give rise to sparse & dense routing tables where rings of different dimensions can be created at every level. Furthermore, it is noted that a 25% improvements can be achieved over routing by storing replacement pointers over duplicate pointers in the routing table.

Moreover, the author suggest two new routing tables, called the P & C –table. The P-table incorporates network proximity for routing by name id. The goal of P-table is to maintain routing in *O(log n)* hops. This P-table ensures that each hop has low latency by keeping track of the nodes whose network distance is close by. The C-table incorporates network proximity for routing by numeric id. The C-table keeps track of nodes that are close by and within the CLB domain.

The author compares SkipNet to IP routing & DNS, Single Overlay & Multiple Overlay Networks. In IP Routing & DNS, content placement is done by IP using DNS lookup. For Single Overlay, we could use a 2-part naming segment (numeric ID and nameID) like SkipNet, however this results in a static form of constrained load balancing (static CLB). For multiple overlay networks, access to other overlay networks is mediated via gateways. Hence data is constrained and load balanced within each single overlay network and not accessible to clients outside except via gateway. This can lead to heterogeneous networks.

5.6 Performance and Evaluation

Experiments are conducted against the Basic SkipNet (using the R-table), Full SkipNet (using R-table, P-table, C-table), Pastry and Chord. The metrics Relative Delay Penalty (RDP - latency of overlay path compare to IP), number of neighbors, physical network hops and number of failed lookups were used.

Basic routing costs measures the routing costs for each of the algorithms. It was found that Full SkipNet and Pastry performed better as they are locality aware while Basic SkipNet and Chord are not. Hence they had more routing entries per node, which allowed it to choose routes that yielded shorter routing costs.

Chord	Basic SkipNet	Full SkipNet	Pastry
16.3	41.7	102.2	63.2

Routing Entries per Node

Locality of Placement measures the number of physical network hops to retrieve data. Chord and Pastry have constant physical network hops because they are oblivious to locality of data since they diffuse data throughout network. SkipNet shows performance improvements as the locality of the data references increased.

Fault Tolerance measures how well the algorithms survives when a failure occurs and the organisation is disconnected from the Internet. Chord, Pastry fails totally for lookups at data

diffused throughout the network and thus is unable to access the network. Both SkipNet continues to functions and does local lookups.

Constrained Load Balancing (within a domain) studies the Relative Delay Penalty (RDP) - latency of overlay path compare to IP. It was found that Basic CLB using R-Table cause higher delays penalties as it is oblivious to locality of nodes. Full CLB causes intermediate delays penalties and Pastry has low delay penalties.

Network proximity studies the effect of Relative Delay Penalty (RDP) over a density *k* which control the number of P-Table entries. It was notice that RDP levels off after *k*=8 because of the increase in the number of pointers in P-Table

5.7 Summary

In summary, SkipNet is the first p2p system that achieves explicit path and content locality. Content locality is provided at a desired degree and granularity. Clustering node names in SkipNet allows SkipNet to perform gracefully in face of linkages failure. Performance of SkipNet is similar to other p2p systems such as Chord and Pastry under uniform access pattern. Under access patterns where intra-organisation traffic predominates, SkipNet performs better. SkipNet is also more resilience to network partitions than other p2p.

6 Conclusion

In this report, we first looked at various DHT-based P2P lookup approaches which include Chord, Pastry, P-Grid & CAN. Chord uses a ring structure for the ID space and each node maintains a finger table to support key query as binary search. Pastry uses a tree-based data structure and the routing table kept in each node is based on shared prefix. P-Grid is based on a virtual distributed search tree. All these tree approaches are complete decentralized, scalable with the total number of nodes and data items, fault-resilient, and support efficient search with cost $O(\log N)$ according to the number of massage exchange. And join cost of these three system are $O(\log^2 N)$. CAN implements DHT using a d-dimensional space. The search cost is $O(dN^{1/d})$ and join cost is $O(dN^{1/d}+d\log(N))$. These systems have many aspects in common. They all share the DHT abstraction, and this has been shown to be beneficial in a range of distributed P2P applications. However, all of them are based on uniform assumption. If data/query distribution is skewed, they are not able to balance the load automatically

In the next part, we discussed two important issues relating to DHT Design namely load balancing and neighbor-table consistency preserving. The concept of virtual server and directories is used to address the problem of load balancing, whereas the problem of consistency is addressed by optimizing the neighbor-tables and proposing a join protocol and several optimization algorithms under an optimization rule.

We have then gone on to compare some of the DHT's by examining their underlying geometry and comparing them in terms of flexibility of neighbor and route selection. We have also looked at several other properties like static resilience and local convergence. We also examined path latencies by looking at the Proximity Neighbor Selection (PNS) and Proximity Route Selection (PRS) methods and deduced that geometries that implement these methods have shorter path latencies. We concluded in that section that the Ring is favored as it has the greatest flexibility and its highest performance in resilience tests.

Finally we looked at SkipNet which overcomes the problems of DHT in that they do not allow users to control where the data will be stored. SkipNet provides both content and path locality in one single network. Clustering node in SkipNet allows SkipNet to perform gracefully in face of linkages failure when a network disconnects from the Internet. SkipNet provides enhanced performance where intra-organisation traffic predominates.

References

[Aberer2002] K. Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. In *Workshop on Distributed Data and Structures (WDAS-2002)*, 2002. http://www.p-grid.org/Papers/WDAS2002.pdf.

[Aberer2001] K. Aberer, P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *proc. Of 6th International Conf. on Cooperative Information Systems (CoopIS 2001),* Trento, Italy, Lecture Notes in Computer Science 2172, Springer Verlag, Heidelberg, 2001.

[ACD+2003] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva and R. Schmidt. P-Grid: A Self-organizing Structured P2P System., In *SIGMOD Record*, September 2003.

[AS2003] J. Aspnes, and G. Shah, Skip Graphs. *technical report*, Yale University, 2003.

[BCM2003] J. Byers, J. Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables", in *Proc. IPTPS*, Feb. 2003.

[DKKMS2001] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area Cooperative Storage with CFS", in *Proc. ACM SOSP*, Banff, Canada, 2001.

[DR2001] P. Druschel and A. Rowstron. PAST: A large0scle, persistent peer-to-peer storage utility. In *Proc. HotOS VIII*, Germany, May 2001.

[DRGR2003] K. Gummadi, R. Gummadiy, S. Gribble, S. Ratnasamy, S. Shenker, I. Stoicak, The Impact of DHT Routing Geometry on Resilience and Proximity, *SIGCOMM'03,* August 25–29, 2003.

[LL2004a] S. S. Lam and h. Liu. Failure recovery for structured P2P networks: Protocol design and performance evaluation. In *Proc. Of ACM SIGMETRICS,* June 2004.

[LL2004b] Consistency-preserving Neighbor Table Optimization for P2P Networks, *Technical Report TR-04-01,* Dept. of CS, Univ. of Texas at Austin, January 2004.

[IVB1997] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on Air: Organization and Access. *IEEE Transactions on Knowledge and Data Engineering*, May/June 1997.

[PSL1990] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, June 1990 supported by an AT&T Bell Labs Fellowship and by NSF grant CCR–8908900.

[RD2001a] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and

routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conf. on Distributed Systems Platforms,* November 2001.

[RD2001b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer systems. In *Proc. 18th ACM Symposium on Operatin Systems Principles*, October, 2001.

[RKDC2001] A. Rowstron, A. –M. Kermarrec, P. Druschel, and M. Castro. Scribe: The design of a large-scale event notification infrastructure. *In Proc. of 3rd International COST264 Workshop on Networked Group Communication*, 2001

[RLSKS2003] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P Systems", in *Proc. IPTPS*, Feb. 2003.

[RFH+2001] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, CA, August 2001

[SBK2002] B. Silaghi, B. Bhattacharjee, and P. Keleher, Query routing in the TerraDir Distributed Directory. In *Proc. of SPIE ITCom'02*, 2002.

[SMKKB2001] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", in *Proc. ACM SIGCOMM*, San Diego, 2001, pp. 149—160.

[SML+2001] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for compute*, 2001

[SNL2003] Nicholas J.A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems* (USITS '03), Seattle, WA. March 2003

[YKM1999] H. Yokota, Y. Kanemasa, and J. Miyazaki. Fat-Btree: An Update-Conscious Parallel Directory Structure. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.