TOWARDS A DISTRIBUTED GENERIC PEER-TO-PEER ROUTING ARCHITECTURE

By
Vaughan Loucks

A THESIS SUBMITTED TO
THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

SUPERVISOR: DR. NICOLA SANTORO

OTTAWA-CARLETON INSTITUTE FOR COMPUTER SCIENCE
CARLETON UNIVERSITY
OTTAWA, ONTARIO
DECEMBER 2003

© Copyright by Vaughan Loucks, 2003

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89853-9

Our file Notre référence

ISBN: 0-612-89853-9

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

To Gina and Kezia.

Table of Contents

Table of Contents	iii
List of Figures	vii
List of Symbols	viii
List of Acronyms	x
Abstract	xi
Acknowledgements	xii
1 Introduction	1
1.1 P2P Architectures	2
1.1.1 Pastry	2
1.1.2 Tapestry	3
1.1.3 Scalable CAN	3
1.1.4 Chord	4
1.1.5 CRN	4
1.1.6 D2B	5
1.2 Contribution of Thesis	6
1.2.1 Namespace and Mapping	6
1.2.2 Operations	8
1.2.3 Analysis and Proposal for Generic P2P Architecture	9
1.2.4 Potential Problems	10
1.2.5 Towards a Next Generation of P2P Architecture	12
1.3 Organization of Thesis	13

2	Overview of P2P / CAN	14
2.1	General Description of a Peer-to-Peer Architecture	14
2.2	Different Types of P2P Applications	16
2.3	History	18
2.3.1	First Generation P2P Architectures	18
2.3.2	Plaxton, et al.	20
2.3.3	Post-Plaxton	22
3	Structure of Representative Systems	24
3.1	Pastry	26
3.1.1	Name Space and Local Structures	26
3.1.2	Routing	27
3.1.3	Join	28
3.1.4	Leave	30
3.1.5	Performance	31
3.2	Tapestry	32
3.2.1	Name Space and Local Structures	32
3.2.2	Routing	33
3.3	Scalable CAN	34
3.3.1	Name Space and Local Structures	35
3.3.2	Routing	36
3.3.3	Join	36
3.3.4	Leave	37
3.3.5	Improvements	38
3.4	Chord	43
3.4.1	Name Space, Local Structures and Routing	43
3.4.2	Join	46
3.4.3	Leave	47
3.4.4	Performance	48
3.5	CRN	49
3.5.1	Name Space and Local Structures	49
3.5.2	Join	51
3.5.3	Leave	52
3.5.4	Routing	52
3.5.5	Performance	53
3.5.6	Improvements	53
3.6	D2B	54
3.6.1	Name Space and Local Structures	54

3.6.2	Routing	55
3.6.3	Join	56
3.6.4	Leave	58
3.6.5	Performance	58
4	Commonalities of Representative Systems	60
4.1	Structures	60
4.2	Namespace	63
4.2.1	Global Namespace	63
4.2.2	Area of Responsibility	65
4.3	Namespace Mapping	66
4.4	Operations	67
4.4.1	Problems for Node Joins	68
4.5	Performance	69
5	Towards a General P2P Architecture	73
5.1	Namespace	74
5.2	Mapping Function	74
5.3	Area of Responsibility	76
5.4	Area of Knowledge	77
5.5	Operations	78
5.6	Performance	80
6	Desirable Features: Towards A New Generation of P2P Architectures	82
6.1	Current Problems and Possible Solutions	82
6.1.1	High Churn Networks	83
6.1.2	Geographic Awareness - Construction	85
6.1.3	Geographic Awareness - Routing	86
6.1.4	Hotspots and Bottlenecks	87
6.2	Towards a New Generation of P2P Architectures	89
6.2.1	Overview	89
6.2.2	Multi-Layers	90
6.2.3	Namespace and Mapping	92
6.2.4	Operations	94
6.2.5	Routing in Multi-Layers	95
6.2.6	Addressing the Current Problems	96
6.2.7	Performance	98
6.3	Summary	99

6.3.1	High Churn Networks	99
6.3.2	Geographic Awareness	100
6.3.3	Hotspots	101
6.3.4	Future Peer-to-Peer Architectures	101
7	Summary and Looking to the Future	103
7.1	Overview of P2P Architecture Components	103
7.1.1	Structure	104
7.1.2	Namespace	104
7.1.3	Namespace Mapping	105
7.1.4	Operations	106
7.2	Advanced Issues	107
7.2.1	High Churn Networks	107
7.2.2	Hotspots	109
7.2.3	Geographic Awareness	110
7.3	Future	111
7.3.1	Virtual Nodes	111
7.3.2	Load Balancing	113
7.4	A New Generation of P2P Architectures	114
7.5	Conclusion	115
	Bibliography	116

List of Figures

2.1	Routing table for node 2461 (base 8). IP addresses not shown	21
3.1	Pastry routing table for node 10233102 (base 4). IP addresses are not shown	27
3.2	Tapestry routing table for node 1642 (base 8). IP addresses not shown	33
3.3	Torus used as CAN Topology	35
3.4	CAN Network improvement examples. A. Overloading B. Multiple Realities C. Multiple hashing	40
3.5	Chord network with Finger Tables. Taken from [28]	45
3.6	Example of a CRN Butterfly Network	49
3.7	Example of a Supernode	50
3.8	Example of adding nodes to a CRN	51
3.9	Example of a D2B network	55
3.10	Building a D2B network	57
4.1	Example of a Namespace Ring	64
5.1	Mapping Nodes and Data Objects into a Namespace	75
5.2	Generic P2P Node	76
6.1	K Layers of P2P Architectures	93

List of Symbols

Symbol	Definition
d	Number of dimensions
N	Total number of nodes in the network
L	Predetermined number of neighbour nodes
p	Point in space
A, B, C	Name of example node / data items
m	Size of the namespace (eg. $2^m - 1$)
i	Example row in routing table
K	Predetermined constant in D2B architecture
k	Number of layers in multi-layer architecture
$\mathcal{S}$	Namespace
P, Q	Sample nodes
R	Area of Responsibility ranges
r_{b_i}	Beginning of range
r_{e_i}	End of range
W	Area of Knowledge set
w	Size of Area of Knowledge set
b	Boundaries of namespace
j	Number of hops a message takes

Symbol	Definition
x	Sample node name
T	Number of top data requests
t	Instant in time
n	Generic node
$h(x)$	Mapping Function
$N(t)$	Set of nodes in the network at time t

List of Acronyms

P2P	Peer-to-Peer
CRN	Censorship Resistant Network
CAN	Content Addressable Networks
DHT	Distributed Hash Table
ID	Identifier
MIT	Massachusetts Institute of Technology
VPN	Virtual Private Network
URL	Uniform Resource Locator, previously Universal Resource Locator

NOTE: Some names of systems, such as D2B, PAST, POST and DMBN do not have acronyms definitions. They may have been formed using the author's name, languages other than English or other facts that are only meaningful to the author's of the papers describing these systems.

Abstract

Peer-to-peer (P2P) networks have allowed us to use distributed computing to share, store and archive files. Newer uses of P2P networks include web-caching, improvements to multi-casting and distributed email and instant messenger servers. However, the underlying routing layer of P2P networks are still changing. Several architectures have been proposed, each offering a number of pros and cons.

This thesis will summarize the current state of technology in the field of peer-to-peer routing architectures. We will examine a number of peer-to-peer routing architectures and determine the features each provide that are portable between architectures. Finally we will investigate a new generation of peer-to-peer routing architecture which incorporates these features. Using the results of this thesis, future research will be better able to produce a more robust, reliable peer-to-peer routing architecture.

Acknowledgements

I would like to thank my wife, Gina, for her patience during this long research and writing process.

I would also like to thank my supervisor, Professor Nicola Santoro of Carleton University School of Computer Science. Without his continued guidance and patience, this thesis would not have been possible.

Finally, I would also like to thank the Ottawa-Carleton Peer-to-Peer Research Group for their ideas and support.

Ottawa, Ontario
December 2003

Vaughan Loucks

Chapter 1

Introduction

The idea of peer-to-peer networking has recently taken off in popularity with the general public. This is due in large part to file sharing systems such as Napster[1] and more recently Kazza[17]. The routing layers of peer-to-peer networks must support a large number of users, allow users to join and leave the network and to search and retrieve content at will. These peer-to-peer networks are made up of a large number of individual computers, each of which allow local storage of a small amount of the overall data available in the network. In exchange, these computers are granted access to all the data stored on the entire network.

"Peer-to-Peer systems (P2P) are distributed systems without any centralized control or hierarchical organization." [18] There are a number of peer-to-peer routing architectures that have been proposed, each with their own pros and cons. This thesis will attempt to identify these "pros" specifically, determine which are portable to other architectures, and finally look at a future generation of architecture which

combines the strengths of the existing architectures.

From the first generation of centralized peer-to-peer networks such as Napster [1], P2P routing architectures have provided a very powerful tool for distributed computing. It supports the saying that "the overall total is greater than the sum of the parts." By spreading the work over all the members of the network in an organized manner, the peer-to-peer network can make available to all members a resource much greater than any single computer could ever accomplish. A peer-to-peer routing network can support various applications. Mostly we will study routing architectures that support distributed file systems, but the same routing network can also be used to distribute computational tasks. The optimization of this routing architecture layer will produce the most efficient use of this incredible resource.

1.1 P2P Architectures

We will study in particular six peer-to-peer architectures: Pastry, Tapestry, CAN, Chord, CRN and D2B.

1.1.1 Pastry

We will start with the Pastry architecture, a research project from Microsoft designed to be the routing layer of, among other things: PAST, a distributed file system, and Squirrel, a distributed web cache. Pastry uses a partial or incomplete hypercube

architecture and can thus achieve a message routing performance of $O(\log N)$ steps, where N is the number of nodes in the network. As nodes join the Pastry architecture, the hypercube is filled in to accommodate them.

Redundancy in Pastry is achieved by having each node keep track of a certain number of neighbouring nodes on either side of it in the namespace. The node stores a copy of the data stored at each node in this list. Adjacent nodes have overlapping lists which increases the redundancy even more.

1.1.2 Tapestry

Tapestry is very similar to Pastry. It too uses a partial hypercube as a topology and can also achieve message routing performance of $O(\log N)$. What makes Tapestry unique is its use of redundancy and load balancing by doubling the number of destinations for each element in its routing table. In addition, when a data object is searched out and found, what gets returned is not the data object itself, but rather a list of nodes where the data object is located. The requesting node may then choose the closest location from this list to retrieve the data object.

1.1.3 Scalable CAN

Scalable CAN, simply known as CAN, uses a multiple dimension Cartesian namespace that, in 2-dimensions, forms a torus topology. An area of responsibility for a node can be pictured as a rectangle within the torus and is known as a zone. A node keeps

track of the zones which neighbour its area of responsibility. When a message needs to be passed, it is passed only to one of these neighbouring zones.

The CAN architecture requires a number of improvements to improve its robustness and redundancy. Some of the suggested improvements are to allow multiple nodes to be responsible for the same zone, placing data into multiple non-neighbouring zones at the same time and allowing nodes to be part of multiple separate CAN networks at the same time.

1.1.4 Chord

Chord has been the architecture of choice when discussing peer-to-peer routing architectures in general due to its simplicity. The namespace of Chord is described as a circle, although it forms a partial hypercube when the internal links are placed. The next $\log N$ (where N is the number of real nodes in the network) nodes in the circle (clockwise) can redundantly store a node's data objects. *Finger tables* are the term used to describe a node's area of knowledge. These finger tables have the addresses of precalculated points around the circle and are used to speedily pass a message around the circle to its destination.

1.1.5 CRN

The CRN peer-to-peer architecture is unique when compared to the other architectures. It uses a static butterfly topology. Each node in the butterfly is a set of real

nodes, called *supernodes*. Nodes are placed in these supernodes so that every node is in the top layer, the bottom layer and in multiple of the middle layers of the butterfly. This allows each node to initiate a request from the top layer, route it through the middle layers and down to the bottom layer where the data objects are stored in an organized fashion.

Communication from one supernode to another is accomplished by having each real node in the sending supernode's set send the message to a corresponding real node in the receiving supernode's set. This parallel communication method provides very high probability that the message will successfully be passed to its destination.

The downside of CRN is that it is a relatively new architecture and still has a number of problems. For example, there is no easy way for a node to join the network so that the supernodes and their neighbours accommodate the new node. There is no way for the butterfly topology to grow or shrink and no way for supernodes to split or merge if their sets get too big or too small. Despite all these problems, the CRN architecture offers some very useful ideas which can be re-used in other architectures.

1.1.6 D2B

Another relative newcomer in the field of peer-to-peer routing architectures, D2B also offers some unique features. One of the most unique features of D2B is its use of the de Bruijn graph as a topology. A de Bruijn graph is very easy to grow as

nodes join the network, and offers a constant number of neighbours per node. This makes the performance of the join and leave operations to be very efficient at $O(1)$ constant time. Unfortunately, there are still some unresolved problems that need to be addressed when nodes leave the network.

1.2 Contribution of Thesis

The goal of this thesis is to create a generic description of a peer-to-peer routing architecture by identifying the major components. To accomplish this goal, we will investigate the major components, analyze their use, discuss some current problems and look towards future generations of peer-to-peer routing architectures.

1.2.1 Namespace and Mapping

All the peer-to-peer networks we will investigate rely on *hashing functions*. These hashing functions map a name to a number. Whether that name is the name of a song, a file or other data, or if it is the name or address of a computer does not matter. This number produced by the hashing function falls into a range of numbers, typically from 0 to $2^m - 1$ called a *namespace*. m is a constant predetermined by the architecture to be sufficiently large (128 or 256) so that all possible internet addresses could easily fall within its range. All data objects and computers in the network (from now on called *nodes*) are arranged in this namespace according to the rules of

the architecture we are studying. To find a data object, we use a hashing function on its name to get a number in the namespace, and then we find the node in the namespace that is responsible for that number. Both nodes and data objects share the same namespace, typically from 0 to $2^m - 1$. Therefore each node is responsible for a certain range of the namespace and the data objects that are hashed into their range. Since we find the data by using its name, this type of network is also known as a *Content Addressable Network* or *CAN*. This is not to be confused with one of the architectures we will study called Scalable CAN, but also commonly referred to as simply CAN.

For a message to be passed between nodes, a node uses a routing table known as its *area of knowledge*. This routing table contains the addresses of some of the other nodes in the network. The number of addresses and the overall topology they form when connected together are determined by the peer-to-peer architecture being used.

To route a message, the node uses its routing table to determine the best addresses. This calculation is done so that the message is sent at least one step closer to its final destination. If the message requires that a result be returned, such as a data search, then the return result is sent back directly to the requesting node.

1.2.2 Operations

We will also find that all peer-to-peer architectures must support three basic operations, node join, node leave and search/update. When a node joins the network, it must be assigned a name, an area of responsibility and an area of knowledge. To accomplish this, the join operation begins at a node that already exists in the network that the new node contacts. The existing node gives the new node a name in the namespace and then informs the new node's neighbours of its existence. The new node and its neighbours will negotiate the new area of responsibility and transfer any necessary data objects to the new node. The new node's area of knowledge can be calculated using its neighbour's areas of knowledge plus its own inquiries as to its new surroundings.

When a node wishes to leave the peer-to-peer network, it may or may not inform its neighbours. If it is nice then it tells its neighbours it is about to leave and negotiates the takeover of its area of responsibility. If the node simply leaves without telling any neighbours, then the "hole" is discovered the next time a message is passed to the missing node. The missing node's neighbours must then decide how to takeover the missing node's area of responsibility and recover the data from copies stored at other nodes.

The final operation that must be available to any peer-to-peer architecture is the data search/update operation. When a node wishes to retrieve a data object it first

finds the name of the object within the namespace of the peer-to-peer architecture. This is usually done using some sort of hashing or mapping function. Next, a message is sent from the requesting node to the node that is determined to be the closest to the final destination according to the node's area of knowledge. How this calculation is performed is determined by the individual peer-to-peer architecture. The message is passed from one node to the next, always getting at least one step closer to its final destination. When the message finally reaches its destination, the action is performed and any results are returned directly to the requesting node.

1.2.3 Analysis and Proposal for Generic P2P Architecture

As we analyze these architectures we will find that the majority of them use topologies from the cube family of graphs. Some of the more popular topologies include the hypercube, the butterfly and the deBruijn graphs. The hypercubes used are never fully complete. Rather, partial hypercubes are used leaving room for new nodes to join the network and fill in more of the hypercube's connections. The butterfly network used in the CRN architecture is a static graph and the nodes are actually sets of nodes. This must be done since it is very difficult to grow a butterfly graph by only one node at a time. The deBruijn graph is used by the D2B architecture since it has constant degree at each node which simplifies the routing table each node must maintain.

Having all the topologies as members of the cube family means that the best routing performance that can be expected is $O(\log N)$. This is due to the fact that the diameter of a graph in this family is almost always at most $\log N$ in size where N is the number of nodes in the network. The performance of adding or removing nodes from an architecture differs according to the degree of the nodes in the topology. The higher the degree, the more nodes and routing tables that need to be updated when a topology change occurs. Performance ranges from $O(1)$ constant time to $O(\log^2 N)$.

We will discuss a generic peer-to-peer architecture where we will find that every peer-to-peer routing architecture must contain certain components. A namespace is essential so that nodes and data have a common, bounded naming convention. Each node in the peer-to-peer architecture must have an area of knowledge to allow it to pass messages to its neighbours, and an area of responsibility so it can maintain a portion of the namespace and the data objects associated therein. A peer-to-peer node must also support the three basic operations, node join, node leave and data search/update.

1.2.4 Potential Problems

Some problems that all architectures face are that of *hotspots* caused by popular data. When a certain piece of data is very popular, then the node responsible for serving the requests for that data will get bogged down, slowing response times. Caching

has been proposed as a solution for hotspots although it would only be a temporary measure. More permanent solutions include *redundancy* and *load-sharing* methods which have been proposed by the various architectures.

Redundancy will be accomplished by using a hashing function that produces multiple addresses in the namespace for a single data object. Such a hashing function is proposed in [24]. If the data object is stored at each of these addresses this will produce multiple copies of the same data object randomly spread around the peer-to-peer network. A node wishing to retrieve this data object could then contact the closest copy of the data resulting in a more balanced load on the network.

Geographically aware construction and routing techniques are used to allow the peer-to-peer network to be more efficient as it sends messages between neighbouring nodes. If these logical neighbours are geographically on the opposite sides of the world, or the network, the messages passed between logical neighbours will be slow and may be lost. Faster, more reliable communications will happen between geographically close nodes. Thus some of the architectures use *proximity construction* and *proximity routing* to optimize the peer-to-peer network.

When constructing a network, a geographically aware peer-to-peer architecture will attempt to place geographically close nodes as logical neighbours. Some peer-to-peer architectures also choose to use electronic closeness instead of geographic

closeness as their measure of proximity. Although this will minimize the communications cost between neighbouring nodes, it does increase the risk of network failure due to geographic disasters like earthquakes or power failures.

Instead of using geographic or electronic proximity as a factor during network construction, other peer-to-peer architectures take proximity into account during routing. This allows the network to be constructed randomly, eliminating the concerns over geographic disasters. When a node must forward a message using its area of knowledge, it may be given a choice of several nodes which are at least one step closer to the final destination of the message. Using proximity routing, the node would choose the closest node from this list. Once again the term "closest" could refer to either geographic or electronic distances.

1.2.5 Towards a Next Generation of P2P Architecture

We will propose an idea for a new peer-to-peer architecture which makes use of some of the unique features found in the current architectures under review. This new architecture will expand on the idea proposed in CAN where the same node can be placed into multiple networks at the same time. If some of these multiple networks used different architectures then we could reap the benefits of each type of architecture while bypassing their weakness. If the namespace was the same for all the networks, and the data was hashed once (i.e. a single ID for a data object no matter what

architecture was being used), then it results in having the ability to reach the data item from any of the architectures running on a single node. Since a single node would hold multiple IDs, one for each architecture, then one ID would have to be closer to the data item than the others. This minimizes the path to find the data object. If a bottleneck or hotspot occurs, then the routing method could simply switch architectures on the fly and continue routing towards the target destination. This also provides great redundancy since each node would be responsible for a different set of data for each of the architectures it is using.

1.3 Organization of Thesis

The thesis is organized as follows. We begin by describing in Chapter 2 the various applications that can be used on a peer-to-peer routing architecture. In Chapter 3, we describe the mechanisms of each of the six architectures we have chosen to study. We investigate some of the common features in Chapter 4 and describe a generic peer-to-peer architecture in Chapter 5. We will address some current problems and areas of research including a new generation of peer-to-peer architecture in Chapter 6. In Chapter 7 we will draw conclusions.

Chapter 2

Overview of P2P / CAN

In this section, we will examine the background of peer-to-peer routing architectures. We will begin by discussing some general rules that all peer-to-peer networks must follow. Secondly we will review some of the applications that can be run on top of the peer-to-peer routing layer. Finally we will briefly look at the history of the peer-to-peer routing architecture field.

2.1 General Description of a Peer-to-Peer Architecture

Content Addressable Networks (CAN) or Peer-to-Peer Architectures are described as "any network which addresses its content (data) using the same namespace as the network nodes" [21]. Each node in CAN is assigned a portion of the namespace known as its area of responsibility. This area of responsibility does not overlap with any other node's area of responsibility. However, collectively all areas of responsibility

must completely cover all points in the namespace. Data objects use their name within the namespace to determine which node will be responsible for their storage and retrieval.

To allow both nodes and data objects to share the same namespace, a DHT (distributed hash table) is spread across all nodes in the network. We will refer to the portion of the DHT contained at each node as a node's *area of knowledge*. The area of knowledge for a specific node represents the other nodes in the peer-to-peer network of which that node is aware. Typically these nodes are stored in a table whose organization is determined by the overlying peer-to-peer architecture. When a node wishes to send or forward a message, it uses the distributed hashing table in its area of knowledge to determine the best node that is at least one step closer to the final destination of the message. The calculation that determines the best node is also defined by the overlying peer-to-peer architecture.

These peer-to-peer networks only need to perform three tasks: join, leave and search/update. "Join" happens when a node wishes to join the network. "Leave" happens when a node either wishes to leave the network or is discovered missing by its peers. "Search" or "update" is a targeted message forwarded to a specific node to perform a data operation.

CAN networks must guarantee three things:

- Atomicity deals with data integrity. A request for data from a node must

produce the same results when sequentially performed multiple times.

- Termination of all join, leave and search/update operations must eventually occur if no new join, leave or search/update operations begin.
- Stabilization: Data and link information should eventually stabilize once all join and leave operations complete and no new ones have started.

2.2 Different Types of P2P Applications

Of the types of peer-to-peer networks examined, we can determine that different problems are trying to be solved. The top layer file applications that run on top of these routing architectures fall into several categories. Some are *file archive networks*, some are designed for *file sharing* while yet others are more inclined towards being *file systems*.

File archives, such as Oceanstore [16] are more concerned with ensuring that data is never lost. They maintain version control of the data and reconstruct and redistribute files occasionally in the background to ensure that any version is available for retrieval, even if it hasn't been requested for a long time. They also tend to maintain data ownership so that only the user who stored the file is able to retrieve it, by encryption and/or by maintaining user IDs as part of the meta-data for the file. This extra protection adds a significant amount of overhead, which can cause these

archives to provide slower response for a data retrieval request.

File sharing systems are on the opposite end of the spectrum from archives. Systems like GNUTella [26] are meant for public (or at least to some extent open) sharing of a file. The data is only available on the network as long as a node attached to the network contains that file. No attempt is made to preserve the data within the network after the node leaves. Data that is popular spreads throughout the network as more and more users download it, providing more copies that can be downloaded. This provides redundancy and load balancing, but only lasts for as long as that data is popular.

In the middle of this spectrum is the *distributed file system*. It is fast enough to provide constant access, secure enough that a node leaving the network does not remove data from the network, and private enough that individual users can control which other users can access their data.

Another idea for the use of a peer-to-peer routing architecture is a distributed *web cache*. This would allow all the browsers on a network to share their web caches with each other, minimizing the wait time for popular pages or sites.

There is an ever growing list of ideas on how to use these distributed peer-to-peer routing architectures. On the Pastry architecture alone the following applications have been built:

- the Squirrel web cache [14]

- the PAST file archival storage mechanism [10]
- the Scribe group/event notification and communications system using multicast [5]
- the SplitStream high bandwidth content multicast system [4]
- the POST secure message passing service for email, instant messaging, shared calendars, notebooks, whiteboards etc. [20]
- the Scrivener resource sharing facility for peer-to-peer networks [22]

2.3 History

Peer-to-peer routing was developed as part of the popularity in internet file sharing mechanisms. Napster [1] and GNUTella [26] are two of the more popular of these file sharing services. From their popularity, academia began to investigate more fully to the potential behind using peer-to-peer networks for uses other than simply file sharing. As we have seen in the previous section, the number of applications available on peer-to-peer networks has grown and will continue to grow.

2.3.1 First Generation P2P Architectures

A successful peer-to-peer architecture must be able to scale to the potential number of users on the internet. Napster fell short of this requirement due to its design. It

uses a centralized server for indexing the data available on the network. A request is sent to this central server where the name of the file required is searched for in the index. Addresses of nodes containing the file are returned to the requesting node and the file transfer takes place independent of the central database. The Napster system has a weak point at its central server. This single point of failure also becomes a bottleneck as the number of users grows.

GNUTella's goal was to remove the requirement of a central server. It does not have a central index for file search, but rather uses a controlled flood for searching by limiting the number of network hops. Nodes self organize into hierarchial groups of subgraphs. A node wishing to search for a file floods its local subgraph with the search request. Flooding obviously does not scale well to internet sized groups of users. In addition, the flooding is stopped at a certain point before it reaches every node in the network, therefore it is possible that data that does exist in the network remains unfound.

Another peer-to-peer system that has been proposed is the Freenet [7] architecture. Freenet is also self organizing and decentralized, but it is also anonymous. A file stored in Freenet is not assigned to a specific server, rather it is cached within the architecture. This produces the desired anonymity since a file is not stored on the machine that originally made it available to the network. Unfortunately this forces Freenet to not put lower bounds on the cost of retrieving a file or even be able to

guarantee the retrieval of existing files.

The Ohaha [2] file sharing system is very similar to Freenet, but it does map files to servers using a hash-like function. Another system called Archival Intermemory [6] maps file names to servers using an off-line computed tree. If new data objects must be added to the network, the tree must be recalculated and distributed.

Globe [3] is a peer-to-peer architecture that breaks the internet down into a logical hierarchy by geographic, topological or administrative groups. This forms a static search tree that is used to find data objects. The individual groups cache their indexes of available data at the roots of the hierarchy allowing for faster searches.

2.3.2 Plaxton, et al.

A system proposed by Plaxton, Rajamaran and Richa [23] is the basis for the second generation of peer-to-peer routing architectures. The Tapestry architecture that we study in a future chapter, is based upon the Plaxton, et al. architecture. The assumption is made that this is a static network without nodes joining or leaving.

A namespace of 160 bits is used to hold all node and data objects. Messages are routed towards their destination by matching digits of the data object from right to left. For example to find data object 2461 would follow (in the worst case) $** * 1 \Rightarrow ** 61 \Rightarrow * 61 \Rightarrow 2461$, where the "*" represents a wildcard.

Each node contains a routing table that contains multiple levels, one for each

xxx0	xxx1	xxx2	xxx3	xxx4	xxx5	xxx6	xxx7	→
←								
xx01	xx11	xx21	xx31	xx41	xx51	xx61	xx71	→
←								
x061	x161	x261	x361	x461	x561	x661	x761	→
←								
0461	1461	2461	3461	4461	5461	6461	7461	→

Figure 2.1: Routing table for node 2461 (base 8). IP addresses not shown

digit of suffix matching the node's identifier. For example, a routing table may look like figure 2.1. For each step a message takes, it is at least one digit closer to its destination.

When looking at the overall topology, a partial hypercube gets formed by using this type of routing table. We will see many more partial hypercubes later as we investigate other peer-to-peer architectures, so based directly on the Plaxton proposal.

Once the destination is reached, the data object is not found. Instead, what is found is the root of a tree of nodes embedded into the architecture. Each leaf node in the tree contains the data object being sought.

To assign these root nodes to the object, Plaxton used an algorithm that required global knowledge. This is clearly a major downfall in a peer-to-peer architecture. In addition the idea of the root node produced a single point of failure for locating an object. Finally, the assumption made that there would not be any nodes joining or leaving the network is not valid.

2.3.3 Post-Plaxton

After the release of the paper by Plaxton et al. a new wave of peer-to-peer routing architectures were proposed. These include many of the architectures we will study including Tapestry [30], Pastry [27], Scalable CAN [24] and Chord [28]. At this time, a greater interest in the field of peer-to-peer routing started to take off. In addition, it was found that peer-to-peer systems could be used for more than simply file sharing, so the routing and application layers were separated.

The Viceroy [19] and DMBN [8] architectures were also proposed around this time, both based on the butterfly topology. They tended to be dismissed due to their overall complexity making their implementation very difficult.

After this wave of peer-to-peer architectures based on the Plaxton et al. architecture, a few architectures were proposed as improvements to Plaxton including Censorship Resistant Networks (CRN) [11] and D2B [12] which we will also study in a future section. Most recently the Koorde [15] and Kelips [13] architectures have been proposed.

Kelips, from Cornell University, is a fresh look at the peer-to-peer routing problem since it concentrates on finding an index to the data rather than finding the data itself. It uses multiple subgroups to contain all the nodes in the architecture. Each subgroup contains the entire index mapping of data objects to the nodes where they are stored, whether they exist in their subgroup or not. To find a data object, a node simply

finds the index mapping in the local subgroup and then directly contacts the node storing the data object. It is claimed that this process takes only $O(1)$ constant time.

Koorde from MIT, is an improvement on the Chord architecture. As we will see in a later section, the de Bruijn graph holds some interesting properties which can have a positive impact on the peer-to-peer architecture topology. Koorde combines the classic Chord structure with embedded de Bruijn graphs to assist with message routing and achieves $O(\log N / \log \log N)$ time for routing with only $O(\log N)$ neighbours per node.

Chapter 3

Structure of Representative Systems

In this chapter we will describe the mechanisms of existing distributed routing mechanisms. The subject of our investigation will be Pastry, Tapestry, CAN, Chord, CRN and D2B. These six have been chosen to provide us with a broad overview of the current technologies available in the realm of peer-to-peer architectures. Pastry has ties to commercial grade products in Microsoft. Tapestry is similar, but was independently developed by the University of California at Berkeley. CAN takes a unique approach to conceptualizing the namespace. Instead of using a range of numbers, CAN pictures the namespace as being d dimension Cartesian space. CAN also provides many great ideas for improvement which can be used in other peer-to-peer architectures. Chord is, by far, the most popular and most discussed peer-to-peer architecture. Most papers use Chord as a standard when discussing issues in the field

of peer-to-peer routing architectures. CRN and D2B are recently proposed architectures using unique network topologies. Both require future work to make them robust enough to be practical, but they do provide some new ideas which should be taken into account for any future designs.

There are many more than just six architectures which we could have chosen to study. We could have examined older systems such as Plaxton [23], but the ideas of those systems have been evolved and refined in current architectures. File sharing systems such as GNUTella [26] were also eliminated from our list of potential systems to study. We opted to stick with only architectures that are used for distributed file systems. Other systems such as Viceroy [19] were considered to be too complex to attempt to study, a common complaint of anyone who has tried to implement this architecture. Finally, systems such as Kelips [13] and Koorde [15] are recent additions to the peer-to-peer architecture family and were not available during the analysis phase of this study.

For each architecture that we are studying, we will examine its namespace and local structures, how to route a message from one node to another, how a node joins the network and how a node leaves the network. We will also examine the cost complexity of performing these tasks.

3.1 Pastry

Pastry [27] is a development from Microsoft and is used as a routing mechanism for their PAST [10] distributed file system, the SCRIBE distributed multicast system [5] and the Squirrel decentralized web-cache [14]. Additional distributed systems are proposed to be built on top of Pastry.

3.1.1 Name Space and Local Structures

Pastry uses a name space that ranges from 0 to $2^{128}-1$. The name space is circular in that the algorithms treat 0 as the neighbour of $2^{128} - 1$. The large exponent 128 is chosen so to allow a low probability of overlaps in either node or object IDs that get mapped into this name space. Using a *hashing function* on the node's IP address randomly generates the node's ID. This hashing function must guarantee that nodes are evenly distributed throughout the name space for optimum performance. Each node contains a routing table that holds node IDs of other nodes in the network. The size of this routing table is $O(\log N)$ where N is the number of real nodes in the network and reflects the fact that the overall architecture of the Pastry network forms a partial hypercube. The node and object IDs are considered to be a series of digits instead of a numeric value. We will cover how the elements of the routing table are formed later when we discuss how a new node joins the Pastry architecture.

For example, a possible routing table for node 10233102 is shown in figure 3.1.

Node 10233102			
Leaf Set			
Smaller	10233033	10233021	10233001
Larger	10233000		
	10233120	10233122	10233230
	10233232		
Routing Table			
	02212102	22301203	31200203
1	11301233	12230203	13021022
0	10031203	10132102	10323302
2	10200230	10211302	10222302
3	10230322	10231000	10232121
3	10233001	10233232	
1	10233120		
0			
2			

Figure 3.1: Pastry routing table for node 10233102 (base 4). IP addresses are not shown

Associated with each entry in the table is an IP address which is not shown for space considerations. The first row of the routing table represents nodes whose IDs do not match any part of the current node's ID. The second row (adjacent the "1") contains nodes that match the first digit of the current node's ID, in this case the digit 1. Similarly the third row contains node IDs that match the first two digits ("10").

3.1.2 Routing

When an object ID is requested, the node receiving the request sends it to the node in its routing table whose prefix matches at least one digit more than it is currently

matched. If this cannot be accomplished, then the message is forwarded to a node with an identical prefix but that is numerically closer to the destination. For example, if a message is being routed to node 10221230 the matching prefix is "102" (see figure 3.1). Therefore the node is routed to a node in the fourth row (beside the "2"), preferably node 10222302.

The routing table also contains a "neighbours" section known as a *leaf set*. This section of L nodes consists of $L/2$ nodes of greater numerical value, and $L/2$ nodes of lower numerical value. The value of L is predetermined at network configuration. These L nodes are neighbours of the current node, $L/2$ on one side and $L/2$ on the other. The purpose of these nodes depends on the application using this architecture. In PAST, the distributed file system designed to be built on top of Pastry, these nodes are used to redundantly store the files of each of the L neighbour nodes. Thus a request to a specific object ID only needs to be routed within $L/2$ nodes of its destination before the request can be fulfilled. This provides both data redundancy and load sharing.

3.1.3 Join

When a node wishes to join a Pastry network it must first know an existing node in the network. It can then make a JOIN request to that node which, in turn uses a hash function on the new node's IP address and sends a message to the node numerically

closest to (i.e. responsible for) that ID. Each node in a peer-to-peer architecture is responsible for the IDs (nodes and objects) that fall into their range of the namespace. Their range is determined by the peer-to-peer architecture being used. To build the routing table of the new node, each node that was touched by the join message, sends its routing table to the new node. Once the routing table has been built at this new node, it informs all the nodes in its routing table and its leaf set of its existence.

To build a routing table using the routing tables of all the nodes touched by the JOIN message, the new node takes the row associated with the number of hops it took to get to that node. If the ID of the first node contacted when joining the network did not match any of the new node's ID, then it would have forwarded the JOIN message to a node with just the first number matching. When building the new node's routing table, only the first row would be taken from the routing table of the first node and used as the new node's first row. The second row of the routing table would come from the second row of the routing table of the second node that received and forwarded the initial JOIN request. This continues until all the nodes involved in joining the new node to the network have provided their routing tables and the new node's routing table is full. The new node then sends a message to all nodes in its routing table and leaf set stating that it is now available. These nodes adjust their routing tables appropriately.

For example, to build the routing table in figure 3.1 for node 10233102, the first

row was obtained from the node that first received the request for 10233102 to join the Pastry network. The second row (adjacent to the "1") came from the second node that the join request was routed to on its way to the node responsible for ID 10233102 in the name space. In this case it would be a node with prefix "1". If the node was next routed to a node that matched the prefix "102", then the third row and the fourth row of the table would be filled.

In the case where two nodes join at the same time, it can be observed that only rarely will the joining of these two nodes conflict with each other. In cases where they do, the problem is simply dealt with by putting timestamps on the routing table update messages. When the new node notifies all the nodes in its routing table that it is available, it includes the timestamp originally sent with that nodes routing table. If the existing node has had its routing table changed since the timestamp, it informs the new node to restart the building of its routing table.

3.1.4 Leave

If a node leaves the Pastry network, either by leaving or failing, its absence is detected when a node attempts to route a message to it and there is no response. This does not slow down the routing of the message, since the Pastry routing algorithm simply attempts to connect to the next node in its routing table with the same prefix or a numerically closer node. To repair the hole left in the architecture due to the departed

node, the node that attempted to contact the missing node contacts another node in the same row of its routing table as the missing node. It requests that node's routing table entry for a node of the same prefix as the missing node. If this repeatedly fails to produce a route to a live node, the search is continued using the next row of its routing table, and so on, until a route to a live node is found.

To fix the overlapping neighbourhood sets, the node adjacent to the missing node contacts the node farthest away in its neighbourhood on the side of the failed node. This node sends its neighbourhood set to the requesting node. The node adjacent to the missing node then chooses the appropriate node from the received set (ensuring it is alive by contacting it) and makes it part of its neighbourhood and vice versa.

3.1.5 Performance

The basic structure of the Pastry architecture is an incomplete hypercube. Therefore we would expect to see the performance of the Pastry architecture to reflect this. As expected, routing a message from one node to another takes $O(\log N)$ steps since the diameter (distance between farthest nodes) of a hypercube is $O(\log N)$.

For a node to join or leave the Pastry architecture also takes $O(\log N)$ steps. This is due to a combination of having a routing table of size $O(\log N)$ and the diameter of the hypercube being $O(\log N)$.

3.2 Tapestry

In general, Tapestry is very similar to the Pastry architecture we have investigated in the previous section. Just like Pastry, Tapestry takes $O(\log N)$ messages to perform join, leave and routing operations and requires $O(\log N)$ spaces in its routing table. Because of these similarities, we will not investigate Tapestry in depth but we will discuss some of its differences with Pastry.

Tapestry [30] is the underlying routing mechanism for the Oceanstore [16] distributed archive and is based upon the Plaxton [23] mechanism. It uses a virtual hypercube as its topology. The hypercube is not complete, leaving room for new nodes to join the hypercube and take their places, but it is connected. The hypercube is formed through routing tables at each node. Null pointers are used to fill the locations of nodes that do not yet exist in the hypercube.

3.2.1 Name Space and Local Structures

The name space used by Tapestry is sufficiently large to minimize the possibility of two nodes mapping to the same ID. IDs are 160 bits in length providing a range of 0 to $2^{160} - 1$. A prefix based routing method is used which routes to a node that has at least one more matching digit than the previous node or, if that link does not exist, then to the numerically closest node. For example a route from 5432 to 7890 would follow the route $5432 \Rightarrow 7432 \Rightarrow 7832 \Rightarrow 7892 \Rightarrow 7890$.

fills the ID pattern, then the link (IP address) is stored in that part of the routing table. If that link has failed, then the second link is used.

The second item that is very different from Pastry occurs when a message is finally routed to the object ID that it is requesting. What is returned is not the actual object, as occurs in Pastry, but rather a list of nodes that contain the object. The node containing this list is called the *root node* for that object. This list of locations provides two benefits. First, if a node goes down then the object is not lost; secondly, the requesting node can choose the object located closest to itself thus balancing the load on the network.

One of the proposed improvements to Tapestry is to increase the number of root nodes for an object. This would spread out the number of nodes that know the actual locations of the object, providing better redundancy and faster search times.

3.3 Scalable CAN

A Scalable Content Addressable Network [24] (CAN) relies heavily on hashing tables to form its network. This hashing table maps the name of the object/node into coordinates that can be placed on a d dimensional mesh. Each node that is part of the CAN network is responsible for part of this mesh. Although it is commonly drawn as a mesh, it actually takes on the topology of a torus with all sides wrapping around; however, for descriptive purposes, we will refer to it as a mesh. The boundaries of

the "mesh" cover all possible results of the hashing function. For example in two dimensions, the hash could map all names within the coordinate range of $[0,0] - [4,4]$, a 4x4 mesh (torus) such as is shown in figure 3.3.

Figure 3.3: Torus used as CAN Topology

3.3.1 Name Space and Local Structures

Each node of the network is responsible for maintaining a certain range of coordinates, known as a *zone*, and for all data objects that fall in that zone. In our 4x4 example in figure 3.3, a single node may be responsible for the zone $[0,0] - [2,2]$ while another node may be responsible for the zone $[2,0] - [4,4]$. Other nodes would fill in the zones not mentioned. Each node also keeps track of its neighbours, one on each of its $2d$ sides (thus the need for the torus). This results in a routing table that is only $O(d)$ where d is the number of dimensions.

3.3.2 Routing

To find an object in the network, the first thing that is done is to hash the name into the coordinate range using the globally known hashing function. This hashing produces a point p where the data can be located. A request is then passed from neighbour to neighbour using a *greedy routing* technique which takes $O(dN^{1/d})$. Each node checks to see whether point p falls within its own zone of responsibility. If not, then it passes the request message along to its neighbour that is responsible for the zone of coordinates closest to the requested point p . If a node does not respond to the message (after a timeout), the message is forwarded to another neighbour in an attempt to circumvent the faulty node.

3.3.3 Join

Adding a node to the network follows three simple steps and only affects $O(d)$ other nodes. First the new node must find a node in the existing network. This is done through either out of band communications, or attempting to use a list of "potentially available" addresses. Step two involves randomly picking a point p within the coordinate space and routing a JOIN message to that point p using the above routing technique. When the node that is responsible for the zone containing point p receives the JOIN message, it splits its zone of responsibility in half, keeping half for itself and assigning the other half to the new node. This split is performed across the longest

side, or vertical (x axis in two dimensions) if the sides are equal. The part of the hash table containing points in the new nodes zone of responsibility, is handed over to the new node along with the data objects in that new zone.

The last step involves updating the neighbour data. The node that split its zone sends a subset of its neighbours table, plus itself, to the new node. It also updates its own neighbour table to remove the nodes that are no longer neighbours and add the new node. Finally, messages are sent to all the neighbours, telling them to update their tables accordingly.

3.3.4 Leave

When a node leaves the network, the event falls into one of two cases depending on whether the node leaves gracefully or harshly (i.e. due to the detection of a fault.)

A graceful departure involves having the node hand over its coordinate zone and all associated data objects to a neighbouring node. This neighbouring node is picked so that either a valid zone is created, in which case the zones are merged, or the neighbouring node responsible for the smallest zone will temporarily manage both zones. A background maintenance process is used to reassign and balance zones that are managed by the same node. Similar to the Join process, this only affects $O(d)$ other nodes.

The other case of nodes departing the network is when there is some sort of fault,

either with the network or with the node itself. If a neighbouring node does not respond to messages, the requesting node (and potentially all the neighbours of the failed node at the same time) start a timer based on the size of the zone for which it is responsible. This means that the smallest zone will start the next step sooner than that of larger zones. The next step, once the timer has expired, is to send a TAKEOVER message containing its zone size to all of the neighbours of the failed node. If a node receives a TAKEOVER message from a smaller zone, than it abandons its attempt to takeover for the failed node. If it is the smaller zone, then it replies with its own TAKEOVER message.

The node that wins the TAKEOVER election merges with, or takes over responsibility for the zone. In this case, however, the objects that were held at the failed node are lost. Solutions to this problem will be discussed in the following paragraphs covering improvements to CAN.

3.3.5 Improvements

Although the basic CAN mechanism is relatively simple to describe, it does leave some large holes that need to be improved. These improvements include multi-dimensional coordinate space, realities, overloaded zones, multiple hash functions and topology sensitive construction. Some of these ideas are still being worked out in current research.

Multi-dimensional coordinate space comes from the observation that the design of the CAN algorithms does not limit it to 2-dimensional space. With a small increase in the size of the routing table, the distance between nodes can be reduced by "folding" the network from 2 dimensions into 3 dimensions or more. This would change each coordinate zone from being a rectangle to being cubic, thus increasing the number of neighbours to each node and thus shortening the routes between any two nodes.

Realities, also known as multiple coordinate spaces, goes one step beyond the concept of multiple dimensions, and explores the idea of having multiple CAN networks mapped over the same nodes. When a node is added to the CAN network and a random point p is generated, the node is added, zones split, etc. in the area of the point p . Now consider if multiple random points were generated for the node, each point for a different reality. The node would be added to multiple CAN networks that cover the same nodes. Each reality would have its own routing table and storage space.

For example, we can see that data object (not node) B exists in two different CAN networks in figure 3.4. Although B 's object ID may be the same, it falls into two different coordinate zones which would be the responsibility of two different nodes. This improves redundancy and provides alternate routing alternatives should a node fail.

Overloaded coordinate zones are formed by setting a global variable MAXPEERS

Figure 3.4: CAN Network improvement examples. A. Overloading B. Multiple Realities C. Multiple hashing

that represents the maximum overload of nodes in a zone. As a node is added to the CAN network and the random point p directs it to where it should join, the node that would normally have split its zone with the new node first checks to see if its MAXPEERS value has been reached. If not, then the node is added as a peer to the zone, copying the same routing table, objects and neighbours. If the number of peer nodes in the zone has reached MAXPEERS, then the new node causes the

zone to split, taking a number of the nodes with it if they fall into the new zone. This replication creates fault tolerant redundancies that help preserve the integrity of the network and reduces the amount of change forced upon the network topology as nodes join and leave.

For example, the nodes denoted as A in figure 3.4 are both responsible for the same coordinate zone and provide both load sharing and redundancy. If MAXPEERS was set to 2, then the next node added to this coordinate zone (top left quarter) would cause a division of the zone to occur. One of the new zones would be overloaded with two nodes and the other zone would only have one node.

Multiple hashing functions work on the data object side of the CAN architecture. When a data object is added to the network, its name is used to hash it into the coordinate space. If this hashing function produced multiple points in the coordinate space, then multiple copies of the same data would be stored in different zones. The redundancy would create stability and fault tolerance. It also means that when a request is made for that object, the multiple locations returned by the hashing function would allow the requestor to retrieve the closest copy thus minimizing network traffic and search time.

We find data objects C in figure 3.4 using a multiple hashing function to appear in two places in the same CAN network. Since the multiple hashing function would assign two object IDs to object C , it makes object C appear in two different coordinate

zones. This increases redundancy since two different nodes are responsible for each of the objects. It also increases the chance of a request for object C finding it sooner than it would have if there had only been one copy of object C .

Topology sensitive construction is an attempt to solve the problem that adding nodes to the CAN network do not take into account the fact that neighbours may be a great distance apart (in view of the network). For example, a route from zones 1 to 2 to 3 may travel from Ottawa, Canada to Paris, France and then back to Toronto, Canada. The random location of nodes into the coordinate space is good for minimizing faults, but may be bad for network efficiency. By attempting to determine where a node exists within the physical network space, the random positioning of that node in the CAN coordinate space can be skewed towards adding the node at a point where neighbouring zones are a minimal distance away in the IP network. This location of a node in IP network could be accomplished by pinging nodes of known location and measuring the latency and number of hops required to reach each.

Uniform partitioning deals with the fact that a network that has nodes evenly spread throughout will perform better than one that has most nodes concentrated in one area. A good hashing function should provide this although relying on randomness to provide load balancing will not produce the most optimum solution. It should be possible to produce additional algorithms to force nodes to be added to the network more evenly while not having global knowledge of the network layout as a

whole.

3.4 Chord

Chord [28] appears to be the most popular and most talked about of the peer to peer distributed routing architectures. The namespace used by Chord ranges from 0 to 2^m-1 where m is large enough that two node or two object names will have very little probability of hashing to the same *key value*. The architecture is most commonly drawn as a circle since the namespace logically treats keys 0 and 2^m-1 as neighbours in the ring.

3.4.1 Name Space, Local Structures and Routing

Objects and nodes share the same namespace in Chord. Therefore, to find an object, you would need to find the node with the same key value. However, not every key value will have a real node associated with it. A real node is one that is an actual physical computer in the network. To deal with this problem, the concept of successors is introduced. A successor of an object is the next node clockwise in the circular namespace from the objects key value. More precisely, this means the *successor* is the node with key value greater or equal to the object value (taking into account the wraparound at node $0 = \text{node } 2^m$). Thus, if there is a request for a certain object, the message must be routed to the next real node, clockwise in the circle, which has

a key value greater than or equal to that of the object.

Each real node in the Chord circle only needs to keep track of the next real node in the circle. Messages may be routed around the circle clockwise from each node to its successor until it reaches the node that is immediately greater (or equal) to the key value of the object being requested. While this method works, it is very inefficient since a single request may, theoretically, take N hops (where N is the number of nodes in the network) to find the correct node.

To resolve this problem, each node in Chord contains a table known as a *finger table*. This finger table contains at most m entries (recall there are $2^m - 1$ entries in the namespace). The i^{th} entry in the finger table at node A is the successor node (first real clockwise node with greater or equal key value) of $A + 2^{i-1}$ ($1 \leq i \leq m, \text{ mod } m$). Each entry in the finger table stores the Chord key value of the object along with its IP address, the range of the interval to the next finger table entry, and the successor node that should be contacted for that range. This results in a routing mechanism which uses $O(\log N)$ messages.

For example, figure 3.5 shows a Chord network that contains three real nodes, 0, 1 and 3. Node 3 is responsible for the range 3 to 2, node 1 is responsible for the range 1 to 1 and node 0 is responsible for the range 4 to 0. Thus data keys 6, 1 and 2 are stored at 0, 1 and 3 respectively.

From figure 3.5 we can also see an example of the finger tables. Examining the

Figure 3.5: Chord network with Finger Tables. Taken from [28]

finger table of node 0, we see in the first row refers to node 1 as the start of the next zone, the second row refers to node 2 and the third row refers to node 4. The third column lists the successors of each of these nodes. For example, if node 0 were trying to find a data key at node 5, it would pass the message directly to the node in the finger table responsible for the range were node 5 falls, in this case node 4. However, node 4 does not exist as a real node so the successor of node 4 is used. The last column states that the successor of node 4 is node 0. Thus the request can be fulfilled locally at node 0. If node 4 did physically exist as a real node, then the request message would be sent directly from node 0 to node 4. Node 4 may not be able to fulfil the request, but at least it would have a better idea as to who is responsible for the data key 5.

It is worthwhile to note that the first entry in the finger table of node A is the

next successor of A in the circle. These shortcuts are not required to make the Chord architecture work, but they do make it significantly more efficient. This does add additional complexity when a node wishes to join or leave the network since these tables must be kept up to date to be useful.

In addition to the successor node and finger table stored at each node, the address of the *predecessor node* is also stored. The predecessor of a node is the first real node counter-clockwise in the ring. Therefore it is possible to traverse the ring clockwise by following successor nodes and counter-clockwise traversing predecessor nodes. This will make it easier to add a new node to the Chord network.

3.4.2 Join

Three things must be accomplished to properly add a node to the Chord network. First the new node must have its predecessor node and finger tables initiated. Secondly, the finger tables and predecessor nodes of existing nodes must be updated. Finally, the application software using the Chord routing mechanism must be notified so it can reorganize its data using the new node. While all this is going on, every existing node's successor node data must be maintained and all data object keys must not be lost.

For a new node, B , to join Chord, it must first know an existing node, C , in the Chord network. This node hashes the IP address of the joining node to produce a key

value in the namespace. Next, to initialize the predecessor and finger table of node B , it asks an immediate neighbour for its predecessor node and finger tables and use those as hints to calculate its own predecessor node and finger tables.

For the second step of updating the predecessor node and finger tables of other existing nodes, the new node B uses the predecessor nodes to traverse the ring counter-clockwise. As it traverses the ring counter-clockwise, it updates the finger tables until it reaches the node that has node B as its predecessor. This particular algorithm takes $O(\log^2 N)$ steps but a more complex algorithm claims to be able to do this in $O(\log N)$ steps. [28]

The final step to adding a node to the Chord network is to inform the higher level application using Chord that the new node has been properly added to the routing architecture. This encapsulation of the routing layer is necessary so that it may be reused for various purposes, not just for a distributed file system. The higher layer application will probably move (or copy) data objects associated with certain keys to the new node. Since only the successor node of new node B will have its responsibilities reduced, the data transfer will occur between only these two nodes.

3.4.3 Leave

When a node, B , either leaves the network or fails, those nodes who contain B in their finger tables must get in touch with the successor of B . An additional

requirement is that no data object query may be lost while the system re-establishes a stable environment. To accomplish this, the concept of a successor node needs to be extended to be a list of the $O(\log N)$ nearest successor nodes in the Chord ring. When a node C detects that its successor has failed, it simply uses the next nearest live successor in the list. If another node is trying to contact the failed node directly from a finger table entry, then, after a timeout period, the requesting node will use the previous entry in the finger table and allow that node to use its successor list to find the requested data object. A separate background routine runs occasionally to repair and update the finger tables of all the nodes that refer to the missing node.

A side benefit of the successor list versus a single successor node is provided for the higher level application using Chord. This successor list becomes a natural replication mechanism. Data at node B can be replicated at each of the nodes in its successor list. That way, a node failure does not result in lost data. The setup and maintenance of such a mechanism is left to the implementation of the higher level application.

3.4.4 Performance

Chord is another of these architectures which form a partial hypercube. Thus, we would expect to see performance reflective of this topology. The finger tables at each node are of size $O(\log N)$. Likewise the process of filling the finger table of a new node takes, with high probability, $O(\log N)$ steps. Recall that the join method

described in Section 3.4.2 takes $O(\log^2 N)$ steps while other methods described in [28] take only $O(\log N)$ steps. Similarly, when a node leaves the Chord architecture, it takes $O(\log^2 N)$ steps.

3.5 CRN

Another network architecture has been introduced recently called CRN [11] or Censorship Resistant Network. It still has some issues open for resolution. This system makes use of a *butterfly network* topology and has the unique feature in that its topology architecture is *static*. Each node in the overall butterfly topology is referred to as a *supernode*, and each supernode is made up of a set of real nodes.

3.5.1 Name Space and Local Structures

Figure 3.6: Example of a CRN Butterfly Network

The butterfly network architecture (figure 3.6) is broken into three distinct segments, the top row, the bottom row and the middle rows. All data in the network is

stored in the bottom row. As we will see later, the data is actually spread throughout the network, but it is logically organized and replicated in the supernodes of the bottom row. All real nodes in the network are in K supernodes in the bottom row, where K is a predetermined constant in the system and N is the number of real nodes in the network. The real nodes are also in K different supernodes in the top row. The top row is used to initiate messages. The real nodes are also in $K \log N$ supernodes in the middle rows. These rows are used for message routing.

Figure 3.7: Example of a Supernode

Let us examine what exactly is a supernode. A supernode is a set of real nodes. As nodes join the network the set that makes up each node grows to accommodate, and similarly as nodes leave the set shrinks. Supernodes cannot communicate between themselves since they are only logical entities. The real nodes that are in the set of the supernode have a direct connection to a node in each of the supernodes that have a logical connection to their supernode. (See figure 3.7)

3.5.2 Join

When a node wishes to join this network, it contacts a node that already exists in the network. This node randomly assigns it to K supernodes in the top row, K supernodes in the bottom row and $K \log N$ supernodes in the middle rows. In each of the assigned supernodes, the node could copy the routing details from the other nodes in the respective supernode sets, although this mechanism has been left as an open design issue.

Figure 3.8: Example of adding nodes to a CRN

For example (figure 3.8), let's add nodes A and B to the CRN. If $K = 2$, then we need node A to be randomly assigned to two supernodes in the top row, two in the bottom row and $2 * \log(12) = 2$ in the middle row(s). Similarly node B would also need to be assigned to two top, bottom and middle supernodes. Therefore it is obvious that every node in the network will be assigned to at least one supernode in the top row, bottom row and middle row(s).

3.5.3 Leave

How a node leaves the network is also left as an open issue. However the easiest method is for the node to simply disappear. Other nodes in each of the supernodes where it was a member will contain the same data and have the same links. When a new node joins the network and is added to these supernodes, the data and links will be restored.

3.5.4 Routing

Routing in the CRN takes a guaranteed minimum number of messages ($O(\log N)$ time with $O(\log^2 N)$ messages) due to the high organization of the network. First we recognize that any node which wishes to make a request (i.e. all nodes in the network) are contained within K supernodes in the top row. Therefore all requests start in the top row with the requesting node sending the request to the rest of the nodes in the supernode set. The request is then sent down the butterfly structure from top to bottom, from one supernode to the next, routing towards the ID of the data item requested. In reality, since there is a set of nodes in each supernode, and the requesting node is in K supernodes in the top layer, the number of request messages moving down the butterfly structure is sufficiently large that it is very probable that at least one will reach its destination.

3.5.5 Performance

The CRN architecture uses a butterfly topology of supernodes. Because the Join process must contact at least $K \log N$ nodes, it takes $O(\log N)$ steps. At present, for a node to leave the network, it simply disconnects, so it is understandable that this takes only $O(1)$ constant time to complete this process. A more robust method needs to be developed which may affect the performance of this function.

Sending a message to another node makes use of the supernode structures. In a butterfly network of normal nodes, it would take $O(\log N)$ time. Since we are dealing with supernodes each containing a set of nodes, it is accomplished in the same amount of time using $O(\log^2 N)$ messages.

3.5.6 Improvements

Some of the open design issues include the fact that they have designed the algorithms assuming that the size of N (number of real nodes) is fixed. They have also assumed that the number of data items in the network is also fixed at N . Although this may make their proofs easier, it is not practical.

As mentioned previously, the details around how a node inherits its routing information is also an open issue. Additionally, the details about how a node leaves the network in a nice way is left for future work. In fact, the initial design of this architecture assumed that all N nodes existed and were placed in the network by

hand, meaning that all the supernodes were balanced having the same number of nodes in each.

This also brings up the issue of how to grow or shrink the overall butterfly architecture when a large number of nodes have either been added, or when enough nodes have left that the size of the supernodes is at risk to the overall stability of the network. Additionally, the issue of what happens when a single supernode is reduced to an empty node is also left open.

3.6 D2B

A very recent technical report by P. Fraigniaud and P. Gauron [12] details an up and coming architecture which makes use of the de Bruijn graph. The benefit of this new architecture is that it is very simple to understand and implement. In addition it is very efficient having constant degree, requires only $O(\log N)$ hops to route to any key and a constant number of messages between neighbouring nodes when a node joins or leaves the network.

3.6.1 Name Space and Local Structures

It is easiest to describe a de Bruijn graph using a binary name space. However, this by no means limits the architecture to only two dimensions. De Bruijn graphs are defined by the property $N = 2^d$, a d dimensional graph will have 2^d nodes and each

node has constant degree as defined in [9]. The name space of D2B ranges from 0 to $2^m - 1$ where m is sufficiently large enough to guarantee with high probability that node or object keys are not duplicated. Suggested possible values of m are 128 or 256. Each of the keys must be padded on the left with '0' digits so they are of length m . For example, if $m = 4$, the value 1 would be thought of as 0001.

The simplest form of a de Bruijn graph is a one dimensional graph containing only two nodes, 0 and 1 (see figure 3.10(a)). All data objects whose keys start with 0 are stored at node 0 and all data objects whose keys start with 1 are stored at node 1. Node 0 has a pointer to node 1 and node 1 has a pointer to node 0.

3.6.2 Routing

Figure 3.9: Example of a D2B network

To route from one node to the other, you remove the leftmost digit of the key and add the j^{th} leftmost digit of the target key to the righthand side, where j is the hop number. For example, in figure 3.9, if a message needs to be routed from node 001 to node 100 it would follow the route $001 \Rightarrow 011 \Rightarrow 110 \Rightarrow 100$. On the first hop

from node 001, we drop the leftmost digit (0 of 001) and add the j^{th} leftmost digit of the target key. In this case, on the first hop it is the digit 1 of 100, resulting in a hop to node 011. On hop number two, we drop the leftmost digit (0 of 011) and add the second leftmost digit of the target key (0 of 100) resulting in a hop to node 110. Finally on hop number three, we drop the leftmost digit (1 of 110) and add the third leftmost digit of the target key (0 of 100) to result in 100.

3.6.3 Join

The building of a de Bruijn graph uses similar rules to the routing mechanism. First, a node wishing to join is assigned a temporary ID from a node that is already a member of the network. This ID is an m bit binary number that falls within the namespace. A JOIN message is routed towards this temporary ID from the existing node providing the IP address of the node that wishes to join.

For example, in figure 3.10(b), if we go back to our simple graph with only node 0 and node 1, each with one pointer to the other, we can easily see how a node joins the network. Let us say that the temporary ID assigned to the joining node is 1. Upon receipt of the JOIN message (either from node 0 or from node 1 itself), node 1 will split itself into two. The original node will receive a new label of 10, and the new node will be labelled 11. A data object will be split between the two nodes depending on its key value.

Figure 3.10: Building a D2B network

To fix up the pointers, we introduce the following rule. A node, x , with ID $x_1x_2\dots x_m$ will have a pointer to nodes $x_2\dots x_m0$ and $x_2\dots x_m1$. In our example with nodes 0, 10 and 11 we determine that node 10 will have pointers to 00 and 01. This is because we start with node ID 10, drop the leftmost 1 digit which leaves us with the digit 0, we then append both a 0 and 1 to the right of 0 giving up 00 and 01 respectively. We do not have nodes 00 or 01, but we do have node 0 which is responsible for all keys with a prefix of 0, so the 00 and 01 pointers will point from node 10 to 0.

Similarly node 11 will drop its first 1 digit leaving us with only the righthand digit 1. Appending 0 and 1 to this produces pointers to nodes 10 and 11. Clearly we can point from node 11 to 10 easily and the pointer from 11 to 11 is redundant so it can be ignored.

Finally we can do the calculations for pointers from node 0. As noted before, node 0 is acting as node 00 and 01. It continues this behaviour while calculating

the pointers. Node 00 requires pointers to 00 and 01, both of which are redundant pointers pointing back to node 0. Node 01 requires pointers to 10 and 11 which are now easy to create.

3.6.4 Leave

If a node wishes to leave the network it must first do three things. First, it must find a substitute, second it must redistribute its data objects and finally it must update the pointers. Note that a node leaving the network is different from detecting and repairing due to a crashed node.

In some cases finding a substitute is simple. For example in our previous work with the graph 00, 01, 10 and 11 in figure 3.10 (c), if node 11 wishes to leave then it would use 10 as a substitute, transfer its object data and help 10 set its pointers to become node 1. One way to picture it is that when a node joins the network, a node splits in two. When a node leaves, the same pair join back together to produce a single node.

3.6.5 Performance

The D2B architecture makes use of the de Bruijn graph. As we have seen, it is very efficient, requiring only $O(\log N)$ steps to route a message from one node to another. What is even more interesting is the constant $O(1)$ time required for a node to join or leave the network. Unfortunately, the leave method requires refining to get around

a serious problem.

When two nodes can easily join together to form one node, all is well. However, there are other cases where the answer is uncertain. For example, if we have a network containing only three nodes, 0, 10 and 11, and node 0 needs to leave. There will be no node responsible for any name that starts with 0, and it cannot merge with 10 or 11. How to deal with this and other similar circumstances is still under investigation. The solution to this problem may take longer than $O(1)$ constant time.

Chapter 4

Commonalities of Representative Systems

Each of the peer to peer routing architectures we have reviewed has some features in common with the others. The commonalities could be in their logical architectures, their methods for having nodes join the network, their namespace, their mapping functions, performance measurements, or their operations, or any combination of those.

In this section, we will discuss each of these commonalities listed above. We will then summarize what is needed to form a basic, generic, peer-to-peer architecture.

4.1 Structures

There are three basic topologies that we have found in the peer-to-peer architectures studied. Those topologies are the hypercube, the butterfly and the de Bruijn. All three fall under the same family of graphs known as the cube connected family.

Hypercubes are known as a very strong topology that is popular in many fields of study. One of their best properties is that the diameter of a hypercube is at most $\log N$, where N is the number of nodes in the hypercube. Another property that is popular is the ability of the hypercube to easily support routing by binary node name. A route from node 000 to 111 can be easily accomplished by changing one bit at a time from 0 to 1. Thus the route would follow $000 \Rightarrow 100 \Rightarrow 110 \Rightarrow 111$.

Butterfly topologies were originally designed to support routing and have directional links from the top to the bottom of the graph. The height of the butterfly graph is at most $\log N$, given that N is the number of (super)nodes in the graph. Another unique feature of the butterfly topology is that each (super)node has a constant degree, independent of the number of (super)nodes in the graph.

Although butterfly topologies have an efficient routing performance, their biggest downfall is their lack of scalability. For a butterfly topology to grow by only one node, requires that more than double the number of existing nodes be created to complete the necessary links. To solve the problem of scalability in the butterfly topology and yet mostly keep the performance efficiencies, the concept of supernodes was created. The supernode allows for some scalability since new nodes are added to the set of nodes contained in the supernode. If all the supernodes get sufficiently full, it is possible to double the number of supernodes and grow the butterfly topology.

The de Bruijn graph was first created in 1946 by N. de Bruijn [9]. When it grows,

the names of the nodes change as they split. For example, node 0 splits into nodes 00 and 01. De Bruijn graphs also have constant degree per node and a diameter of $\log N$.

Some of the logical architectures end up forming partial hypercubes. The Pastry and Tapestry architectures have purposely used partial hypercubes. The Chord architecture appears to use a circle, however, with the addition of the finger tables at each node, a hypercube is partially formed. Hypercubes have always been popular architectures. Their logarithmic diameter means that messages are guaranteed to have a relatively short distance to travel to reach their destination. There are also multiple paths to each point in a hypercube, allowing for alternate routing paths should a failure occur.

The two other unique architectures we have come across are the Butterfly network, in the CRN architecture, and the deBruijn network, in the D2B architecture. The use of a static supernode architecture in CRN makes it difficult to compare it to the dynamic architectures of the others. Although the process of adding or removing supernodes is not covered in [11], it is obvious from the basic CRN structure in figure 3.6 that adding a single supernode will be difficult. To keep the symmetry of the network, it is necessary to add at least double the number of existing supernodes.

The deBruijn network used in the D2B architecture does allow relative ease when adding nodes. However there are some difficulties when removing nodes. In some

cases, this is easy and straight forward by simple merging two nodes back together, but other cases it is much more difficult, and still an open area of research.

Of course all of these peer-to-peer networks we are discussing are actually *virtual networks*. These virtual networks exist solely through the routing information stored at each node. The actual physical/electrical/internet distance between the closest two nodes in any P2P network may be incredibly far. This is the same concept as VPNs (Virtual Private Networks) which are used as commercial network security devices over the internet.

4.2 Namespace

4.2.1 Global Namespace

For a Content Addressable Peer to Peer Network to function, it must have the ability to map all of the nodes and data objects into the same range of names. We will discuss this mapping problem next, but first we must investigate this range of names that we wish to use. What is commonly referred to as the *namespace* of a P2P architecture is usually a range of numbers. Typically the numbers range from 0 to $2^m - 1$, where m is chosen beforehand to make the range exceptionally large compared to the number of node and data objects that will be stored within it. The purpose of such a large range is to ensure with high probability that no two data objects or node objects will map to exactly the same value in the namespace.

Of course, there are exceptions to the idea that a namespace is a range of numbers. As we have seen in the Scalable CAN architecture, a d -dimensional Cartesian space could also be used. We still use ranges of numbers along the x , y , z , etc. axis to form the overall space. Ranges are defined at the near and far corners points (i.e. (2,2) to (4,4)) instead of actual ranges (i.e. 2 to 4), but the concept is still the same.

Figure 4.1: Example of a Namespace Ring

Although we have talked about a namespace as being a range of numbers, it is really circular. After $2^m - 1$, the next point is 2^m which is equivalent to zero. This forms a ring, such as is found in figure 4.1. In Scalable CAN a torus is formed.

Node and data objects need to share the same namespace in a peer-to-peer architecture. This allows the nodes and the data objects to be addressed using the same naming convention. If we wish to find a specific data object, we only need to find a node that is close to it within the namespace. Since they are using the same naming convention in the namespace, we simply have to send a message to the name of the the data object to find the node that is responsible.

4.2.2 Area of Responsibility

Each node's *area of responsibility* for storing and serving data is defined as a subset of the namespace. All data and node objects must fall within the bounds of the namespace range defined by the architecture. A node will store in local storage any data object that is mapped into its area of responsibility. The node must also serve up this data object whenever it receives a request for that object.

In addition, the cumulative area of responsibilities for all nodes in the network must cover the entire global namespace. If they did not, then it is possible for a data object to be mapped into a portion of the namespace which does not have a node responsible for its storage and the data object would be lost.

On the other hand, areas of responsibility cannot overlap. No two nodes can be responsible for the same range of the namespace. Although there may be redundant storage of data objects between neighbouring nodes, there must still be one and only one node that is primarily responsible for any area of responsibility.

Recall in figure 4.1 that each node was responsible for a portion of the ring. Also recall that the areas of responsibility are not forced to begin or end at point zero. We can clearly see that the cumulative areas of responsibility entirely cover the range of the namespace without any overlaps.

4.3 Namespace Mapping

Another area of commonality between these different peer-to-peer routing systems is their requirement to map an item (node or data) into the architecture's namespace. This mapping must take the item's real-world name and map it into a unique number that falls within the bounds of the namespace defined by the architecture. The mapping function must be able to map into any point in the namespace, not just a subset. Typically this mapping is performed by a hashing function, but we cannot limit ourselves to exclusively this technology.

As an example, the IP address of a node is put through a hashing function to create a unique address in the namespace of the system, in most cases a single number. Hashing to a number is quite logical since computers deal with number data much easier than any other format. Some other suggestions of data to hash, instead of the node's IP address are its private key (assuming it has one) or the system/processor serial number. With the internet running out of IP addresses, more and more machines are beginning to share IP addresses (i.e. behind different routers). This means that using IP addresses may not always produce a unique address in the namespace for two separate nodes. Therefore other information, or combination of information, that makes that specific node unique in the entire universe of the internet is required. This will guarantee with very high probability that the hash function will produce a unique address in the namespace of the peer to peer network.

4.4 Operations

Finally, we must define the operations that a peer-to-peer architecture must be able to perform. All the architectures we have studied have supported the same set of basic operations: node join, node leave and data update/search.

Node joins allow a new node that has contacted the existing network to be provided with a name in the namespace. Further, it informs the new node's neighbours of its existence, so they can update their routing information, and it informs the new node of its neighbours.

Nodes leaving the network do so either by simply disappearing or by informing the network that they are leaving. The first method is harsh and requires the architecture to first discover the missing node and then compensate for its loss. The second, more graceful approach allows the network to recover more quickly in most cases. The CRN architecture does not care which method is used. In fact, any architecture making use of "supernodes" or overloading will not be greatly affected by either a node joining or leaving. This will be discussed further in the next chapter.

The final common operation to be discussed is that of data object update/search. No matter what operation is being done to the data, whether adding new data, updating existing data or searching for data, the routing operation is the same.

- A search for the data must be done in the network. This is accomplished by

mapping the data's name into the namespace of the architecture.

- Secondly, the node responsible for maintaining the portion of the namespace containing the data must be found and notified.
- Finally the data operation can be performed.

4.4.1 Problems for Node Joins

All of the systems we have reviewed have required that, to join their network, a node must first know another node in that network. This knowledge is explained as being from either outside information channels, or preexisting in the node. Outside information channels are usually defined as a user typing in an address provided by other communications (email, phone, web, etc). Information that is preexisting in the node is either hardcoded to a site that has a high probability of existing (i.e. a website URL) or could be a leftover routing table from that nodes previous membership in the network. This routing table is completely out of date and cannot be used once the node joins the network, but it can be used as a list of addresses to contact to make the initial JOIN request.

Some of the work-a-rounds for this problem are to use internet search engines to find membership, or using a family of algorithms known as expanding broadcasts. Either solution will help the node that wishes to join to find the network without having to have previous knowledge or external intervention to inform it of an existing

node in the network. Problems arise if there is more than one network that the node is eligible to join. This entire area is still very open for discovery.

4.5 Performance

When comparing the efficiency of each of these architectures in table 4.1, one thing is evident. No matter the architecture, it takes $O(\log N)$ time to route a message to a specific point in the network. Even Scalable CAN, which does it in $O(dN^{1/d})$, actually works out to $O(\log N)$ time when d is equal to $\log N$. This consistent $O(\log N)$ time is because all the architectures use a topology that has maximum diameter (distance between farthest points) of $\log N$. Each architecture uses a routing method that, given a stable network, guarantees that the message will always travel at least one step towards its destination. Thus a message can only take at most $O(\log N)$ steps to cross the entire diameter of the network. Let us observe that, although the CRN architecture claims only $O(\log N)$ time for routing, it does so using $O(\log^2 N)$ messages due to its parallel design with all the nodes in each supernode forwarding each message.

When it comes to adding or removing a node, there is a large range of performance times, from $O(\log^2 N)$ in Chord to $O(1)$ in CRN and D2B. This is due to the complexity of the architectures and the amount of routing information that must be changed. Given this fact, Tapestry even states that it is not designed for very fluid

Arch	Join	Leave	Route
Pastry	$O(\log N)$	$O(\log N)$	$O(\log N)$
Tapestry	$O(\log N)$	$O(\log N)$	$O(\log N)$
CAN	$O(d)$	$O(d)$	$O(dN^{1/d})$
Chord	$O(\log N)$	$O(\log^2 N)$	$O(\log N)$
CRN	$O(\log N)$	$O(1)$	$O(\log N)$ using $O(\log^2 N)$ messages
D2B	$O(1)$	$O(1)$	$O(\log N)$

Table 4.1: Timing Comparisons of Studied Architectures

networks where large numbers of nodes are constantly joining and leaving.

The most efficient is the D2B architecture which claims $O(1)$ constant time for adding nodes. This is because, when a D2B node splits to add a new node, it only needs to inform the nodes from which it is receiving messages.

When it comes to removing nodes, both CRN and D2B claim $O(1)$ constant time. However, both have problems with this mechanism which they state needs further work. CRN simply allows a node to disappear since other nodes in each of the supernodes where it is a member will pick up the slack. As other nodes are added to the network, they will fill in the "holes". There is no mechanism for changing the overall structure of the network should too many nodes leave a supernode rendering it empty. D2B has a constant number of steps for a node to leave but only under certain circumstances. When two nodes can easily join together to form one node, all is well. However, there are other cases where the answer is uncertain. For example, if we have a network containing only three nodes, 0, 10 and 11, and node 0 needs to leave. There will be no node responsible for any name that starts with 0, and it

cannot merge with 10 or 11. How to deal with this and other similar circumstances is still under investigation.

Two cases are possible when two or more nodes wish to join a peer-to-peer network at the same time. In the first case, the two nodes are added at opposite sides of the network and the processes never cross each other. These additions can happen in parallel and therefore the performance of these operations is unaffected.

The second case happens when the two new nodes are going to share an area of responsibility. Their requests will be sent in serial, due to the nature of computer networks, to the node responsible for the area of responsibility. This node must deal fully with the addition of the first node before it can deal with the addition of the second node. It is possible that the join request of the second node can no longer be handled by the receiving node since the contested area of responsibility may have moved under control of the first new node.

The performance of the join operation for the first node would be unaffected by the pending join request of the second node. Only the second node's join request would be delayed at most $O(\log N)$ time while the first operation completes. One additional step may be required to reposition the second join request at the node now responsible for the area of responsibility it wishes to split.

When multiple nodes wish to leave at the same time a similar situation occurs. The first case is the same, two nodes wish to leave that are on opposite sides of the

network and do not affect each others operations. The performance of these two leave operations would remain unaffected by the fact that they both occur at the same time.

The other case is when the two leaving nodes do affect each other. When one node wishes to leave it must tell it neighbour and transfer its area of knowledge to that neighbour. A second node wishing to leave may inform its neighbour, possibly the node already wishing to leave. The second node wishing to leave should be informed of a different node that can take over its area of responsibility.

The performance of the first node leaving is not affected by the second node wishing to leave. The second node will only take one step longer to determine which node is taking over its area of responsibility. Thus it can be said that the performance of the second node performing the leave operation is also unaffected.

Chapter 5

Towards a General P2P Architecture

Now that we have highlighted some of the commonalities between the peer-to-peer architectures we have studied, we can start to describe a *generic* peer-to-peer architecture. Each node in the architecture holds certain properties and responsibilities, and the architecture as a whole has properties which we will discuss. Since peer-to-peer networks are dynamic, any description of them must be done at a point in time t . Thus we assume that any symbols used should also contain a subscript " t " to denote they represent a value at a set point in time.

A peer-to-peer architecture has a *namespace*. Each node is responsible for a small portion of that namespace, known as a node's *area of responsibility*. To be able to use this namespace, a *mapping function*, $h(x)$, must consistently map the names of nodes

and data objects into the namespace. A node also has an *area of knowledge* that allows it to know about some of its neighbours in the network. Finally, each node must have some *operations* which it can perform. The operations that are typically available to a node are join, leave and data search/update. These operations are given to the node when it first joins the peer-to-peer network.

5.1 Namespace

A generic peer-to-peer architecture must have a namespace. A namespace, using a mapping function, $h(x)$, is used to map all the names of nodes and data objects into a common naming domain. This namespace can be up to d -dimensions, and for each dimension is bounded by a range of values starting at 0 with an upper limit of $2^m - 1$. The value of m does not need to be equal in each dimension, thus the namespace may form a rectangular rather than a square in 2 dimensional space. In most cases, the architecture defines this namespace to be circular in each dimension, thus 2^m is equated to 0. Therefore we can describe a d -dimension namespace, $\mathcal{S}$, as being made up of a set of d boundaries $\{b_1 \dots b_d\}$ where, for each $b_i = \{0 \dots 2^m - 1\}$

5.2 Mapping Function

To be able to map names between the real world and the namespace of a peer-to-peer routing architecture, we require a mapping function $h(x)$. This mapping function is

used for both the nodes and the data object, as can be observed in figure 5.1, since both must belong to the same namespace. The entire namespace must be within the range of the mapping function, any point in the namespace has the same probability of being used.

Figure 5.1: Mapping Nodes and Data Objects into a Namespace

In addition, a mapping function must be able to consistently produce the same point in the namespace given the same inputs. Given two different real world names, the mapping function must, with high probability, produce different points in the namespace. Otherwise there would be a conflict when trying to search for a particular data object's name. The names of nodes and data objects may overlap, but not the names of two data objects or the names of two nodes.

Figure 5.2: Generic P2P Node

5.3 Area of Responsibility

Given this namespace for the overall architecture, we can now examine an individual node, P . In figure 5.2, we see that node P is assigned a set of ranges, $R(P)$, of the namespace $\mathcal{S}$ known as node P 's local area of responsibility. Note that in the figure, the size of the area of responsibility is not necessarily to scale compared to the size of the namespace. The data objects that are mapped using $h(x)$ into this area of responsibility are physically stored in the local storage area of node P . Each member of $R(P)$ represents a range between 0 and $2^m - 1$ for each dimension in $\mathcal{S}$. This set of ranges $R(P)$ must fall somewhere within the overall namespace $\mathcal{S}$, of the peer-to-peer architecture. Therefore, $R(P) = \{R_1 \dots R_d\}$; $R_i = \{r_{b_i}, r_{e_i}\}$, $0 \leq r_{b_i} \leq 2^m - 1$ and $0 \leq r_{e_i} \leq 2^m - 1$. At any point in time t , there are some nodes in the system $N(t)$, each corresponding to a point. $P(t) = h(n) : n \in N(t)$, that is, a point named P at time t can be assigned to a node n using the hashing function $h(n)$ where n

is a member of the set of nodes in the system, $N(t)$ at time t . For each dimension, the ranges of all nodes in the entire peer-to-peer network must completely cover all points in the bounds of the namespace for that dimension; that is, at any time t ,

$$p \in P(t) \cup R(P) = \mathcal{S}.$$

5.4 Area of Knowledge

Our generic peer-to-peer node P , in addition to its area of responsibility, also has a local *area of knowledge*. The set $W(P)$, is comprised of all knowledge that node P has of the rest of the peer-to-peer network and forms part of the overall structure of the peer-to-peer network. The area of knowledge includes a set of node addresses, $W(P) \supseteq \{W_0 \dots W_w\}$, outside of node P 's area of responsibility. The value of w may be in the range of $\log N$ due to the size of the routing table being affected by the fact that topologies using the hypercube topology tend to have $\log N$ connections from each node. It is possible that $w = N - 1$, that is, a node knows every other node in the network, but this is impractical. In the following, for simplicity we will assume that $W(P) = \{W_0 \dots W_{\log N}\}$.

If a node P needs to route a message elsewhere in the network, it must rely on its area of knowledge to provide the best address. The area of knowledge usually takes on the form of a table to assist in organizing known neighbour's addresses. This is done in a way that makes it easy for a node to determine the best address to forward

a message to that is closer to the final destination of the message in some way.

If we drew a graph of a peer-to-peer network, we would use nodes for points. For links, we would take the entries in each node's area of knowledge and draw a link between the node and each node listed in their area of knowledge table. Viewing the overall graph would show us the topology of the peer-to-peer network. The most popular topology is the from the cube family, which includes the hypercube, the de Bruijn and the butterfly topologies.

5.5 Operations

The operations available to be performed at node P are: node join, node leave and data search/update. These three basic operations are all that is required to keep a simple peer-to-peer network operating. The join operation allows node P to assist a new node in acquiring its own area of responsibility and area of knowledge. The leave operation for node P requires it to assist other nodes in the takeover of its area of responsibility and to adjust other node's area of knowledge for all other nodes that contain P as part of their area of knowledge set. The data object search/update operation causes node P to send data to a requestor if the requested data falls into node P 's area of responsibility. If it does not, then node P forwards the message to one of the nodes in its area of knowledge that it decides would be better suited to handle the request.

Given a network of N identical peer-to-peer nodes that form a connected network topology, as described above, we will now investigate how our generic peer-to-peer architecture will operate. The operations we must detail are node join, node leave and data search/update.

To route a message, such as data search/update, from one node to another, a node n must first determine where it needs to send this message. It must use the *mapping function* $h(x)$ to map the name of the data into the peer-to-peer architecture's namespace. Once a point in the namespace has been determined to be the message's destination, the node will use its area of knowledge to determine the closest node that it should forward the message to. Who is "closest" depends on how the area of knowledge is organized and the rules used to form the topology. Its choices could be:

- the numerically closest to the destination
- the electronically closest to the destination
- the geographically closest to the destination
- the closest to the destination by digit. (left to right or right to left, ex. 023 is closer than 033 to 123, right to left digit match)

When this step is completed, if the message reaches a node that does not contain the data object within its area of responsibility, the process is repeated. This continues

until the message reaches a node where the data items does fall into its area of responsibility.

When a node wishes to join the peer-to-peer network, it must know an existing node in the network which it can contact to begin the join process. This existing node must provide a name in the namespace for the new node. It accomplishes this by either using a mapping function $h(x)$, or by assigning it a name through some other mechanism. Once the new node has a name, the existing node must send a message to the new node's neighbours informing them of the existence of this new node. The new node will negotiate with its neighbours to determine its area of responsibility and its area of knowledge. Once these are calculated and the new node's neighbours have incorporated it into that portion of the namespace, the new node is ready to act as part of the overall peer-to-peer network.

For a node to leave the peer-to-peer architecture, it must inform its neighbours that it is leaving by using its area of knowledge. It must interact with its neighbours to transfer its area of responsibility to other nodes before it can leave. The neighbouring nodes must adjust their areas of knowledge to remove references to the departing node.

5.6 Performance

The performance of the operations depends on the diameter of the topology of the peer-to-peer architecture. This will give us the worst case for routing a message from

one end of the topology to the other end. The average distance between nodes will give us a better picture of the performance we can expect. For example, topologies in the cube family of graphs tend to have performances in the $O(\log N)$ range due to the fact that the diameter of the topology tends to be $O(\log N)$.

In some cases, the dimension of the topology also affects the performance metric. The higher the degree of a node, the higher the number of links which must be maintained. This will also affect the size of the routing table (a node's area of knowledge), which affects the performance of the join and leave operations.

The leave operation can sometimes be performed in constant time in some architectures. This is due to the fact that for a node to leave these architectures is equivalent to simply turning off. There is no work to be performed to transfer the area of responsibility of the departing node since the design of the architecture itself provides enough redundancy to recover naturally.

Chapter 6

Desirable Features: Towards A New Generation of P2P Architectures

Two areas will be covered in this chapter. First we will look at some problems that are common to all peer-to-peer architectures. Secondly, we will look towards the future at a new generation of peer-to-peer architectures which combine most of the unique features we have discovered in the previous chapter and incorporates solutions to the problems discussed there.

6.1 Current Problems and Possible Solutions

This section will examine some of the current problems that are still being investigated. We will study different approaches and directions for possible solutions. Some of these problems include high churn networks, geographic awareness during both construction and routing, hotspots, node capacity and load balancing.

6.1.1 High Churn Networks

One of the open questions in [25] is how peer to peer architectures react to a large number of nodes joining and leaving rapidly. This is a valid question for networks hoping to work with the general public over the internet. This high churn of nodes entering and leaving the network can put a great strain on the logical architecture as it has to constantly grow and shrink to accommodate this churn. One of the best proposed solutions is called "supernodes" in CRN and "overloading" in Scalable CAN.

The idea for overloading is that the network architecture is either static or changing much slower than the rate at which nodes enter and leave the network. Each node in the architecture is actually a set of real nodes. These sets can grow and shrink as nodes join and leave the network without affecting the overall logical architecture of the network. Any node that joins the set only has to learn who the other set members are, information that can be passed on from any existing member. Redundancy of data can naturally happen among set members. Any set can decide when it becomes too large or too small, and the architecture will either grow or shrink depending on a predetermined threshold level.

For example, lets say the upper threshold limit of a set in an architecture is predetermined to be 25 and the lower threshold is predetermined to be 5. If there is a set that already has 25 members, then the next node that joins that set, assuming no

nodes have left, will push the number of members in the set to 26. Since this exceeds the threshold, the supernode containing the set of 26 nodes would go through the process of adding a new supernode to the overall architecture. The set of 26 would then be split into two sets of 13 each, one in the original supernode and one in the new supernode.

If another new node joins the network and is assigned to one of these supernodes containing a set of 13 nodes, then it is simply added to the set. Since the number of members of the set is now 14, well below the upper threshold of 25, the supernode does not need to make any architectural changes. The new node can simply be informed of its routing and data information by other members of the set.

If, for some reason, 9 members of the set of 14 leave the network, the set can continue operating at the lower threshold with only 5 nodes. If one more node leave the set, dropping its membership to 4, then the supernode must leave the architecture and the set of 4 nodes and their data must be merged into another set. The data from both sets must be spread around to all the nodes of the new set.

This, of course, poses another problem. What if the set that results from merging the two sets is larger than the upper threshold. This would cause the supernode to create a new supernode and split its set. Since we just removed a supernode because its set was too small, this seems like a waste. All that needs to be done is to move nodes from set to set. This is still an open area of research.

6.1.2 Geographic Awareness - Construction

Another open question is that of the construction and maintenance of a *geographically aware topology*. This means that neighbouring nodes in the architecture are relatively close together geographically. There are pros and cons to this. Geographically close nodes will minimize the time required to pass messages and data between themselves, but they risk network integrity if a geographic event happens (earthquake, router failure) that removes them from the network.

To determine where two nodes are geographically cannot be done simply by counting network hops between neighbouring nodes. This is known as electronic proximity awareness. For geographic information to be used in the placement of a node, it could be placed into a hashing function which takes location into account. For example the country and province/state or postal/zip code could be used as part of the node's name when it is provided to the hashing function.

Another suggestion for determining the geographic location of nodes is made in [24]. Using landmark nodes in the internet, such as DNS servers, a node's location can be triangulated by measuring the round trip time of a message to a number of these landmarks. Although this may not reflect exact physical geographic locations, it does help determine a geographic location within the domain of the internet.

The CRN network described in [11] potentially provides a unique solution to this problem. Recall that the supernodes in the bottom layer of the architecture

contain all network nodes (and all network data distributed in a redundant way), and all the supernodes in the top layer contains all the same networks nodes, but in different supernode sets. Therefore, if we make all the nodes in a top layer supernode geographically close, we ensure that they are not geographically local in the bottom layer where the data is stored. However, since the top layer is the one to start the network traffic for searches, this traffic would be geographically local thus minimizing message latency.

6.1.3 Geographic Awareness - Routing

Another similar topic is called *proximity routing*. There are several ways we can define the term "proximity". We can say nodes are in proximity by measuring how close they are geographically, electronically, or even by putting a communications cost on each link between nodes. The cost could be a time cost, a dollar amount (satellite versus land line) or a combination of both.

Previously we tried to build the network to account for geographical distances. Proximity routing allows random building of the network, but when a message is routed, it then takes geography into account. Since there are usually multiple choices as to which node to forward a message, proximity routing would allow the choice of forwarding the message to the closest geographical neighbour out of the list of logical ones.

Another similar choice would be to take the neighbour with the lowest communications cost. The geographically closest may or may not be the closest electronically. This method addresses the problem of having geographically close logical neighbours, risking large outages due to local disasters, but still takes on some advantages to using the geographically closest possible logical neighbour.

6.1.4 Hotspots and Bottlenecks

An issue that can plague any peer-to-peer network is popular data. A piece of popular data can cause a *hotspot*, a point in the network that gets completely bogged down with data requests while other nodes in the network receive few or none. Obviously we need more copies of the popular data; this brings up the choice of temporary caching or having more permanent copies. The choice would be easy if we could predict if the data would only be popular for a period of time, or if it would always be in high demand. However, since we cannot predict the future, we must come up with another method.

Caching the data would involve having each node request and store the data of the top T requests that it routes. Note that it cannot simply store the data, it must request it. This is due to the fact that the request for the data passes through the routing mechanism, but the data is transferred directly from the source to the requestor. Although this additional request for the data would only increase the load

on the hotspot, if done early enough it may not cause as many problems as it solves.

The permanent copy solution can be found in some of the systems we have investigated. Scalable CAN uses multiple hashing to place separate copies of the same data under the control of multiple nodes. The hashing function simply produces multiple points in the Scalable CAN namespace. A node requesting the data would also be given the multiple points from the hashing function, of which it could choose the closest.

For a higher probability of a speedy return if there are a lot of faults in the network, the requesting node could also send its requests to all the points returned by the multiple hashing function. It would then be a race between the closest and the least busy nodes to return the data. Although this does nothing for load balancing and removing hot spots, if the data's fast return is more important than network health, then this is a possible solution. This technique will also allow the receiving node to compare the returned results to ensure they are the same. This validates the accuracy of the data when multiple copies are stored separately.

Tapestry also uses the permanent copy solution. When a request reaches the point specified by the hashing function, what is returned is not the actual data, but rather a list of sources. Using this list, the requestor can then directly contact any or all of these nodes. If it contacts one at a time, it could contact either the closest, the one with the lowest communications latency or the one with the smallest load. It

could also use the shotgun approach and attempt to contact them all, using the one that replies the quickest and using the others to validate the accuracy of the data by comparing multiple copies.

6.2 Towards a New Generation of P2P Architectures

In this section we will look at a new generation of peer-to-peer routing architectures. This next generation must take into account all the problems encountered by previous architecture designs and combine the solutions proposed by others. We will first discuss the properties of a peer-to-peer architecture. Finally we will investigate potential structures, namespaces, mappings and responsibilities.

6.2.1 Overview

The design of the next generation of peer-to-peer routing architecture must take into account the features and best practices we have found. Such an architecture would need to be able to:

- route a message to any node in at least $O(\log N)$ time
- be resilient to large numbers of nodes joining and leaving
- redundantly store data so that, with high probability, it could be retrieved even

with a large number of node failures.

- be always available and be able to deal with hotspots caused by a large number of requests for popular data.
- contain a namespace and a mapping mechanism to place data and node objects into that namespace with high probability that no two objects would be mapped to the same name.
- support of the three basic operations required for a P2P network: node join, node leave, data update/search.

Some of these goals can be achieved immediately by reusing ideas proposed in the architectures already investigated.

6.2.2 Multi-Layers

Instead of building an entirely new architecture, a new generation of peer-to-peer routing architecture should use the existing architectures and simply layer them. Scalable CAN proposed the idea of "multiple realities" where the same nodes of the network actually exist in multiple, entirely separate CANs at the same time. A new peer-to-peer architecture should take that idea one step farther. Instead of just using multiple CANs, use some D2Bs, some Chords, some CRNs, etc. A study of an architecture using multiple Chord layers is found in [29].

All the nodes in the network are members of each architecture all at the same time. This means that, for each architecture layer, a node will have a completely independent area of responsibility and area of knowledge. We will define k as being the number of architecture layers in the network. If a node is responsible for a certain area of responsibility in one architecture, it will be responsible for a completely different, unrelated, independent area of responsibility in another architecture. It will also be responsible for yet another completely different, unrelated and independent area of responsibility in a third architecture. The size, location and number of stored objects is also separate, independent and unrelated for each of the k different architecture layers.

In a similar manner, the node's area of knowledge for one architecture layer, will be completely different, unrelated and independent from any of the other $k - 1$ areas of knowledge. The routing/finger tables for each architecture are totally separated from the tables of any other architecture on the same node.

Why would we want to build such a system? Multiple architecture layers will provide a much greater level of reliability. With the data stored by so many nodes in the network, it will be much more difficult to lose any due to failed node. We will also be able to use the multiple architecture layers to balance the load on the overall network. A hotspot or bottleneck in one layer can be easily bypassed by simply switching to another layer to continue the routing process.

Unfortunately such an architecture comes with a cost. The amount of network traffic will increase at least k times. We will also require much larger storage capacities on each node. Thirdly, the complexity of maintaining k routing tables will also be a large hurdle.

6.2.3 Namespace and Mapping

To be able to achieve true independence of the different k layers, each layer will be required to have its own independent hashing/mapping function for nodes. We will discuss the hashing function for data objects later. If there were not separate hashing functions for nodes in each layer, then the nodes would map to the same place in each of the k architecture layers. A node would have similar areas of responsibility and similar areas of knowledge in each layer, defeating the purpose of using k layers of architecture.

Note that we only want to use a separate hashing function for nodes. For data objects we want to use the same hashing function throughout all k layers. Both of these hashing functions must cover exactly the same namespace. If we use the same hashing function, then we are obviously using the same namespace. We can do this because all the architectures we have studied (except Scalable CAN) use a namespace ranging from 0 to $2^m - 1$. Therefore the data items will appear at the same name throughout all k layers, however the nodes in each layer will be widely scattered so

no node will be responsible for the same area of responsibility in any two layers.

Figure 6.1: K Layers of P2P Architectures

Let us consider the example in figure 6.1. Each layer could be a different type of architecture although there is nothing to say that we cannot have more than one of the same type of architecture. Some layers could be Tapestry, some CRN, some D2B, etc. The rings represent the namespace of each architecture layer. We can see node P appearing in each ring. In addition, node Q demonstrates that two nodes are not in the same relative position in any two architectures. The "0" marks the beginning of the namespace in each ring. We assume that all namespaces are the same size ranging between 0 and $2^m - 1$. Note that each ring starts in a different place signifying that node P falls under a different name in each layer and thus has a different area of responsibility and area of knowledge. This statement also holds true for node Q .

We can use the concept of *virtual nodes* to assist in explaining the k layers of peer-to-peer architecture on each node. Each node contains k virtual nodes, each

one a member of a different, independent peer-to-peer architecture. Each of the k virtual nodes act as an independent node in each of the k peer-to-peer architecture layers, one virtual node per architecture. The only thing these k virtual nodes have in common is the fact that they exist on the same real node.

6.2.4 Operations

The operations of join, leave and search/update are repeated for every architecture layer at each node. This is required because every architecture layer may use different methods to perform each operation, depending on the architecture's design. For example, a join operation in the Pastry architecture uses different methods when compared to the join operation from the Chord architecture.

Although the join operation is implemented separately for each layer, the join operations must be run together. If a node wishes to join the network, it must first contact a node already in the network. This new node must be added into each of the k different layers. Therefore the node that is the current member of the network would have to initiate the join process within each of the different k architecture layers. In a similar manner, if a node wishes to leave the overall network, it would need to initiate the leave operation in each of the k architecture layers.

The search operation can go either way. A node may initiate the search operation in only one layer, or it may choose to use a "shotgun" approach and initiate a data

search operation in all k layers, taking the result that is returned first. If the node needs to add or update data, it must initiate the update operation in all k architecture layers so that all copies of the data object are kept in sync.

6.2.5 Routing in Multi-Layers

Using a shared namespace for the data objects will make the job of routing much easier. If a node wants to find a data object, it only needs to perform the hash function once to know where within the namespace the data object lies, for all k layers. If a node compares all the virtual nodes it represents in all the layers, it will find one virtual node that is closer to the data object's name than the others. It should use that virtual node in the architecture layer where it appears to start the routing method. This is because it will probably take the least number of hops to get to the data object compared to starting with any of the other virtual nodes in the other architectures.

For example, let's say a node had virtual nodes (in the same namespace) named, 10, 100, 1000, 10000 and 100000. If the node wanted to find a data object named 110, it would have a higher probability of finding it faster if it started in the layer where its virtual node is named 100, than if it started in the layer where its virtual node is named 100000.

Recall when we were discussing the search operation in a previous section, we

noted that a node may wish to use a "shotgun" approach to perform a search for a data object. Also, during the join and leave operations, it is also required for a node to join/leave every architecture layer. Therefore we must have the ability to lock some messages into a layer and not allow the routing of these messages to switch layers.

6.2.6 Addressing the Current Problems

As we observed in the previous section, there are a number of current problems in the field of peer-to-peer routing. Some of these problems are hotspots, high churn networks and proximity construction and routing. We will now discuss how a k layered peer-to-peer architecture can deal with these problems.

Hotspots and Redundancy

There are many benefits behind storing the data separately in each of the k layers and using separate hashing functions in each layer too. It provides a great amount of redundancy and various forms of data distribution.

If a hotspot occurs in one layer, or a node suddenly leaves the network, the k layered architecture will allow for easy recovery due its redundancy. Redundancy occurs not only because the data is stored k times, once for each architecture layer, but each layer may have its own data redundancy mechanism built into the specific architecture used by that layer. If any kind of blockage occurs, then a node now has a

choice of $k-1$ other architectures is can switch to, to move closer to the destination. In fact, after each routing step, it would be beneficial for a node to compare a message's destination against all its k areas of knowledge to determine which layer probably has the shortest path. The node may also already have the data object in one of its k areas of responsibility.

High Churn Networks

Another concept that was suggested in both CRN and CAN is the idea of supernodes or overloading. Having multiple real nodes responsible for the exact same spot in the network provides both redundancy and load sharing. It also reduces the stress on the network caused by large numbers of nodes joining and leaving. This means that joins and leaves in the architecture we are using are actually joins and leaves of supernodes as they fill and empty. These would be triggered by preset threshold limits for how full or empty we allow a supernode to be before forcing it to split or merge.

By using supernodes, we can get routing tables for the new node simply by copying one from another node in the set and informing the rest of the set about the new member. Similarly, any data needed for storage by the new node can be copied from an existing set member. A single message would need to be sent from the supernode to neighbouring supernodes who can send it messages, with the address of the new node.

Proximity Construction and Routing

With so many areas of knowledge available at each node, a message has a great deal of choice when it reaches each node. This amount of choice gives us the ability to switch between layers as we route, allowing us to route to the closest location both electronically and geographically. When constructing the architectures, we could take geographic proximity into account when constructing one layer and electronic proximity into account when constructing another layer. Each layer could be built with a different proximity rule in mind. Some layers would keep nodes in close proximity while other layers would force a far distance between nodes for security against geographic disasters.

6.2.7 Performance

The performance of a k layered peer-to-peer routing architecture is not that much different from any other peer-to-peer architecture we have previously studied. The join operation will take only as long as the slowest architecture in the k layers. Since the steps of the join operation happen in parallel between layers, the times will not be greater than the slowest algorithm. Of the architectures studied, the slowest join operation took $O(\log N)$ steps. Even if the join operations were performed in series for each layer separately, it would still take $O(k \log N)$ steps. Since k is a constant, this simplifies to $O(\log N)$ steps.

The leave operation will also happen in parallel between the layers. The worst time in the previously studied architectures was in Chord at $O(\log^2 N)$. Using the same logic as we did to calculate the performance of the join operation, we can say that the leave operation will take at more $O(\log^2 N)$ steps.

Similarly, to route a message will take $O(\log N)$ steps. We know we can do better than this in most cases. The node that begins the data request will be, in the worst case, $\log N$ steps away from the data object in one layer. However, if the node switches the routing of the message to another layer, it should find a layer where it is less than $\log N$ steps away from the data object. Therefore the worst case for finding a data object is $O(\log N)$, but in practice we should be able to improve upon this time.

6.3 Summary

In this chapter we have investigated a number of current problems and some possible solutions. We have also described a future generation of peer-to-peer architectures.

6.3.1 High Churn Networks

Peer-to-peer architectures, especially those available to the general public over the internet, have to contend with a large number of nodes joining and leaving the network. This puts a great deal of stress on the architecture as it must constantly grow and shrink to meet the demand. One of the proposed solutions was to make each

node in the network a member of a set. As nodes join and leave, these sets would grow and shrink much easier than an entire peer-to-peer architecture. An upper and lower boundary on the size of the set would determine when the sets split or merge. If we then treat each set of nodes as a single node in the peer-to-peer architecture, we will have slowed down the rate of change in the peer-to-peer architecture to the rate at which the sets split or merge.

6.3.2 Geographic Awareness

Secondly we discussed the concepts of geographic awareness during construction and routing. A geographically aware topology is constructed so that neighbouring nodes in the peer-to-peer architecture are also close geographically. This should minimize the communication times between neighbouring nodes, thus increasing the speed of the peer-to-peer network. However, this also leaves the peer-to-peer architecture vulnerable to geographic disasters such as earthquakes or power failures.

Geographically aware routing allows random construction of the peer-to-peer network, but takes location of nodes into account when routing a message. The "closest" node is used whenever a choice must be made by the routing mechanism. The term "close" can be defined as either geographically close or electronically close.

6.3.3 Hotspots

Hotspots are caused by a large number of requests for a popular piece of data. This causes the node that holds the data in its area of responsibility to become so busy that other requests are delayed. Caching can be used to alleviate the problem for a short amount of time. Another solution is to use a hashing function that produces multiple names and store the data at multiple locations in the peer-to-peer network. The nodes searching for the data can then direct their request to the closest copy.

6.3.4 Future Peer-to-Peer Architectures

The final section of this chapter explained a new generation of peer-to-peer architectures. These new architectures take advantage of the strengths of the architectures previously investigated by using them all at the same time by layering them one on top of the other over the same namespace. Each architecture stores all the data in the namespace and is responsible for all the data in the namespace. However, since each node is located at a different place in each architecture layer, it will be responsible for totally separate data for each layer.

This layered approach produces a great deal of redundancy which is useful for overcoming problems such as hotspots and data recovery due to failed nodes. It also provides for fast routing since a message can be switched between architectures as

it is routed. This allows it to take the shortest, unobstructed path to its destination. Because of this, the performance cost for routing a message is as good as the performance of the slowest architecture layer with no additional overhead.

Chapter 7

Summary and Looking to the Future

As we have seen, the field of distributed peer-to-peer routing architectures have continued to grow in popularity. In this section we will review the components that make up a peer-to-peer architecture, some existing architectures we have studied, a new generation of peer-to-peer architecture and some current problems and proposed solutions.

7.1 Overview of P2P Architecture Components

Although we have studied a number of different peer-to-peer architectures, we have noticed that there are basic components that make up a peer-to-peer routing architecture. They all require a structure, a namespace, a mapping function and support for the basic operations join, leave and search/update.

7.1.1 Structure

The structure is formed by the routing tables stored at each node. We have referred to these tables as a node's *area of knowledge*. The area of knowledge holds a node's knowledge of some of the other nodes in the peer-to-peer network. It is organized by the peer-to-peer architecture in such a way that the node is able to determine the best node to forward a message to that is a least one step closer to the message's destination.

From the structures we have investigated, we recognized that the structures were all from the cube family of graphs. The most popular topology was the hypercube, but the butterfly and the de Bruijn graphs were also being considered by some more recently proposed peer-to-peer architectures.

The benefit of using a topology from the cube family of graphs is their $\log N$ diameter, where N is the number of nodes in the network. The diameter is the distance between the farthest points in the graph. with a maximum distance of $\log N$, we know that routing a message in this family of graph will take at most $O(\log N)$ steps.

7.1.2 Namespace

When we talked about the namespace of a peer-to-peer architecture, we refer to a range of numbers. In a single dimension the range is typically from 0 to $2^m - 1$ where

m is chosen to be sufficiently large (128 or 256) so that no two nodes or data objects will be mapped to the same point in the namespace. All nodes and data objects must be mapped into this same namespace because we want to be able to find a data object by finding the closest node in the namespace.

Each individual node is responsible for a portion of the namespace known as a node's *area of responsibility*. Collectively, all the areas of responsibility must completely cover the entire namespace of the architecture without any gaps or overlaps. A node's area of responsibility is the portion of the range that a node is assigned. The node is responsible for storing and serving the data objects that are mapped into its area of responsibility.

7.1.3 Namespace Mapping

Several times we have referred to the process of mapping a data object or node into the peer-to-peer architecture's namespace. The *mapping function* used is typically a hashing function that maps the name of a data object or node into a number contained in the namespace. Both nodes and data objects can use the same hashing function, although this is not necessary. A mapping function must be able to place a name anywhere within the entire namespace range and it must be able to consistently reproduce the same name for the a node or data object. It must also have a high probability of not producing the same name for two different nodes or data objects.

7.1.4 Operations

Each peer-to-peer routing architecture must be able to support basic operations such as node join, node leave and data object search/update. Depending of the complexity of the architecture, some of these operations produce performance metrics that are better than others.

The join operation happens when a node wishes to join the peer-to-peer network. The new node must first know another node that is already a member of the peer-to-peer network it wishes to join. Once the new node has contacted this existing node, the existing node must provide the new node with a name in the namespace and initiate the join process by sending a message to the new node's neighbours. The new node can then begin to negotiate its area of responsibility with its new neighbours and can begin to form its area of knowledge using information gathered from the initial existing node and from its new neighbours.

When a node wishes to leave the peer-to-peer network, it should inform its neighbours of its intentions. The neighbouring nodes must decide how to take over the parting node's area of responsibility so that no data objects are lost. Sometimes the parting node simply leaves the network without informing its neighbours. Once the missing node is discovered, its neighbours must decide how to take over the missing node's area of responsibility and retrieve the missing data from redundant copies stored through the peer-to-peer network.

The final basic operation that must be supported by all peer-to-peer architectures is that of data search/update. When a node wishes to find a data object, it must first use the mapping function to find the name of the data object within the namespace. The node then sends a request message to the node responsible for that data object. The requesting node may not know who is responsible for the data object, so it uses its area of knowledge to find a node that is at least one step closer to the destination. This process is repeated until the message reaches its destination.

7.2 Advanced Issues

Although there has been much work done in the area of peer-to-peer routing architectures, there are still some current problems. Some that we have discussed and have presented potential solutions for are high churn networks, hotspots and geographic awareness.

7.2.1 High Churn Networks

Peer-to-peer architectures that wish to work with the general public over the internet are faced with a unique problem that may not face corporate implementations as severely. Users of the internet tend to be very dynamic, joining and leaving randomly and with high frequency. This produces a very high rate of churn in the peer-to-peer architecture. Such a high rate of churn will put a great deal of stress on the

architecture as it has to constantly grow and shrink to attempt to keep up with this churn. The number of messages required to keep up with the constant change will also become a factor in slowing down data request, the real reason for having such a peer-to-peer architecture in the first place.

To overcome the affect of large numbers of joining and leaving node making the architecture constantly changing, it has been proposed that "supernodes" be created. Each supernode takes over the place of a node in the peer-to-peer architectures as we know them. Instead of nodes forming the topology of the architecture, supernodes would take their place. These supernode are nothing more than sets of real nodes. As nodes join and leave the network, they are assigned into different supernodes. Each supernode is responsible for its own area of responsibility and has its own area of knowledge as well as keeping track of the nodes in its set. Each set is provided with predetermined upper and lower thresholds on its size. Should the number of nodes in the set pass the upper threshold, then the supernode would split in two and the peer-to-peer architecture would perform a join operation for the new supernode. Should the size of the supernode pass the lower threshold, then the supernode would perform the leave operation to hand over its area of responsibility and then merge its set of nodes with another supernode.

There is still a great deal of research required in the area of supernode. Some of the problems include optimal message passing between supernodes, more robust

methods for nodes to join and leave supernodes while informing the neighbouring supernodes of the change, supernode split and merge techniques and communications between the nodes within a supernode's set.

7.2.2 Hotspots

Hotspots are formed within a peer-to-peer network when there is a very popular data object. The number of requests for that data object produce so much message traffic that the node responsible for the popular data and its neighbours get choked, delaying responses to all requests and messages. The best method to alleviate hotspots is to prevent them in the first place.

Data redundancy and caching have been presented as possible solutions for the hotspot problem. Caching involves allowing neighbouring nodes to cache the results of data requests and reply to them as required. This causes a problem with data freshness since it may not be known which node have cached the data.

A more popular method for reducing hotspots is data redundancy. To accomplish this, the mapping function used to place the data object into the namespace needs to provide multiple points rather than just one. If the data is stored at multiple points spread across then entire namespace, then when a node wishes to retrieve the data it may contact the closest point from the list produced by the mapping function. This will spread the requests across the network, balancing the load on the entire

peer-to-peer network.

7.2.3 Geographic Awareness

Geographic awareness can happen during the construction and the routing phases. During construction, the peer-to-peer architectures attempts to place geographically close nodes as logical neighbours within the topology. Instead of using geographical distances, we may also consider using electronic distances (ie. number of internet hops) between nodes as our measure of closeness. In either case, this method attempts to minimize the distance between neighbouring nodes which will also result in minimized communications cost. The biggest downfall of this method is the threat of geographical disasters such as earthquakes or power failures causing more damage in the network than would have normally happened without geographically aware construction.

Instead of building the geographic awareness into the topology of the peer-to-peer network, geographic awareness during routing allows for normal, random construction of the network. During routing it is normal of the area of knowledge of a node to produce a choice of several nodes which are all potential routes to the destination. At this point of choice, geographic routing would pick the node that is closest geographically or electronically. This eliminates the threat caused by geographically aware construction but still has the advantage of minimizing the communications cost

between neighbours.

7.3 Future

As we have seen, the field of routing in peer-to-peer networks contains many varied solutions to the problem. However, there are still large areas which need further research to produce totally robust and efficient architectures which are commercially viable.

7.3.1 Virtual Nodes

Another property which can be taken into account when building a network is the size of the machine at a node. It seems unfair to treat a low end desktop computer the same as a high end, multiprocessor, large capacity, high bandwidth server. If the capacity of a joining node can be determined, then possibly it could be treated as multiple virtual nodes of smaller capacities. This would make better use of the higher capacity of that particular machine. However should that machine fail, it does mean that more recovery must take place to deal with the multiple nodes that the machine represents.

To assist in determining what is a large capacity machine versus a small or medium capacity machine, the architecture could have predetermined minimum requirements for a node. It would determine that the minimum requirements would be a certain

amount of memory, local storage space, network bandwidth and/or processing power. Each node that joins the network would be evaluated against these requirements. Each full set of minimum requirements met would be a separate virtual machine.

For example, let's say the minimum requirements of an architecture were 200 MHz processor, 300 MB memory, 1 GB storage space and 1 MB network bandwidth. Let's also assume that a node wants to join the network with a 1 GHz processor, 1 GB memory, 10 GB storage space and 10 MB network bandwidth. The machine has 5 times the processor requirements, 3 times the memory requirements, 10 times the storage requirements and 10 times the network bandwidth requirements. This machine could be, at most, 3 virtual nodes since it is limited by the amount of memory. Adding at least 500 MB of memory to the machine would allow it to become 5 virtual nodes. It could not become more than that since the processor capacity would then become the limiting factor.

A similar idea to virtual nodes is to put a weight on each machine. Instead of dividing machines into virtual nodes, a weighted node would remain as a single node in the network. However, the architecture would assign it a larger area of responsibility within the namespace. A larger area of responsibility means more data for the node to store and serve.

7.3.2 Load Balancing

The peer-to-peer architectures we have studied have relied on their hashing functions to provide a random distribution of nodes and data objects across the namespace. Randomness is by no means a reliable guarantee that nodes or data will not become concentrated into one area of the namespace. This has a negative effect on the architecture, making a few nodes work much harder than others. To solve this problem, load balancing is required.

Load balancing can be achieved by removing the randomness from the hashing function. As new nodes join the network, their name can be assigned so that they appear in a sparsely populated area of the namespace. This will reduce the size of the area of responsibility for nodes already in this area. As data objects are placed in the namespace, they can be assigned names that place them in areas of responsibility that do not have as many data objects as other areas of responsibility. Both of these methods will balance the load across the architecture as a whole.

Unfortunately, such assigning of names requires global knowledge of the distribution of both nodes and data objects across the entire network. Perhaps future research will show that local load balancing between a node and its neighbours will result in global load balancing across the whole network.

7.4 A New Generation of P2P Architectures

We have described a new generation of peer-to-peer routing architectures which combine the features of the previously studied architectures. We have expanded the idea presented by Scalable CAN where a node exists in multiple networks at the same time. Instead of using multiple CAN architectures, we have shown that we can use any number of architectures, including multiples of the same type.

Each node acts as a virtual node in each architecture at the same time. A node has a different name in each architecture, but each data object has the same name across all the architecture layers. This means that all the architectures must share the same namespace dimensions. Each data object is located in the exact same place within the namespace in each layer. The nodes that are supporting each layer are in a completely random order. Therefore with high probability, no data object is stored twice on the same physical node.

We can also use the layers to organize the nodes in different manners. One layer could organize the nodes in geographic proximity, another layer could be in electronic proximity and another in random order.

One of the huge benefits is the redundancy that is naturally created. The data objects are stored by the appropriate node(s) in each layer. Therefore each layer has a full copy of all the data objects that are in the system.

For a data object to be searched out, any node in any architecture can begin the

search operation. As the message is being passed towards the destination, it can be switched between layers. This allows it to move around hotspots, bottlenecks and failed nodes with ease.

7.5 Conclusion

This thesis has shown that there are many different peer-to-peer routing architectures available, each with their own special features. We have determined which of these features can be reused in other architectures and have described a new generation of peer-to-peer routing architecture that reuses many of these features.

With this summary of the current state of technology in the peer-to-peer routing architecture field, future research will be better able to address outstanding problems and produce a more robust, reliable peer-to-peer routing architecture. In turn, this will increase the number, variety and useability of peer-to-peer applications, opening still other fields for research.

Bibliography

- [1] *Napster*, <http://www.napster.com>.
- [2] *Ohaha smart decentralized p2p sharing*, <http://www.ohaha.com/design.html>.
- [3] Arno Bakker, E. Amade, Gerco Ballintijn, Ihor Kuz, P. Verkaik, I. van der Wijk, Maarten van Steen, and Andrew S. Tanenbaum, *The globe distribution network*, Proceedings of the USENIX Annual Conference, June 2000, pp. 141–152.
- [4] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, *Splitstream: High-bandwidth content distribution in a cooperative environment*, In Proceedings of 2nd International Workshop on Peer-to-Peer Systems(IPTPS'03) (Berkeley, CA), February 2003.
- [5] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, *SCRIBE: A large-scale and decentralized application-level multicast infrastructure*, IEEE Journal on Selected Areas in communications (JSAC) **20** (2002), no. 8, 1489–1499.
- [6] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos, *A prototype implementation of archival intermemory*, Proceedings of the Fourth ACM International Conference on Digital Libraries, 1999, pp. 28–37.

- [7] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong, *Freenet: A distributed anonymous information storage and retrieval system*, Lecture Notes in Computer Science **2009** (2001), 46–66.
- [8] M. Datar, *Butterflies and peer-to-peer networks*, In European Symp. on Algorithms, vol. 2461, 2002, p. 310.
- [9] N. de Bruijn, *A combinatorial problem*, Koninklijke Nederlandse Academie van Wetenschappen Proc. **46** (1946), 758–764.
- [10] P. Druschel and A. Rowstron, *PAST: A large-scale, persistent peer-to-peer storage utility*, HotOS VIII (Schloss Elmau, Germany), May 2001, pp. 75–80.
- [11] Amos Fiat and Jared Saia, *Censorship resistant peer-to-peer content addressable networks*, Proceedings of Symposium on Discrete Algorithms, 2002.
- [12] P. Fraigniaud and P. Gauron, *The content-addressable network d2b*, Technical report lri 1349, LRI, Universit Paris-Sud, Jan 2003.
- [13] Indranil Gupta, Kenneth Birman, Prakash Linga, Al Demers, and Robbert Van Renese, *Kelips: building an efficient and stable p2p dht through increased memory and background overhead*, In 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03) (Berkeley, CA, USA), February 2003.
- [14] Sitaram Iyer, *Squirrel: A decentralized peer-to-peer web cache*, In Proceedings of Principles of Distributed Computing (PODC 2002), July 2002.
- [15] Frans Kaashoek and David R. Karger, *Koorde: A simple degree-optimal hash table*, In Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03) (Berkeley, CA, USA), February 2003.
- [16] John Kubiawicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer,

- Christopher Wells, and Ben Zhao, *Oceanstore: An architecture for global-scale persistent storage*, In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) **35** (2000), no. 11, 190–201.
- [17] N. Leibowitz, M. Ripeanu, and A. Wierzbicki, *Deconstructing the kazaa network*, In 3rd IEEE Workshop on Internet Applications (WIAPP'03)., 2003.
 - [18] D. Liben-nowell, H. Balakrishnan, and D Krager, *Analysis of the evolution of peer-to-peer systems*, In ACM Conf. on Principles of Distributed Computing (PODC) (Monterey, CA), July 2002.
 - [19] D. Malkhi, M. Naor, and D. Ratajczak, *Viceroy: A scalable and dynamic emulation of the butterfly*, In Proceedings of Principles of Distributed Computing (PODC 2002), July 2002, pp. 183–192.
 - [20] Alan Mislove, Ansley Post, Charles Reis Paul Willmann, Peter Druschel, Dan S. Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra, *Post: A secure, resilient, cooperative messaging system*, Submitted.
 - [21] D. Malkhi N. Lynch and D. Ratajczak, *Atomic data access in content addressable networks: A position paper*, First International Workshop on Peer-to-Peer Computing (IPTPS 2002), IPTPS, March 2002.
 - [22] T.-W. J. Ngan, D. S. Wallach, and P. Druschel, *Enforcing fair sharing of peer-to-peer resources*, 2nd International Workshop on Peer-to-Peer Systems (IPTPS) (Berkeley, California), February 2003.
 - [23] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa, *Accessing nearby copies of replicated objects in a distributed environment*, ACM Symposium on Parallel Algorithms and Architectures, 1997, pp. 311–320.

- [24] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, *A scalable content-addressable network*, Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, ACM Press, 2001, pp. 161–172.
- [25] Sylvia Ratnasamy, Scott Shenker, and Ion Stoica, *Routing algorithms for dhts: Some open questions*, In Proceedings of 1st International Workshop on Peer-to-Peer Systems, March 2002.
- [26] M. Ripeanu, I. Foster, and A. Iamnitchi, *Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design*, IEEE Internet Computing Journal 6 (2002), no. 1, 50–57.
- [27] Antony Rowstron and Peter Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), November 2001, pp. 329–350.
- [28] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable Peer-To-Peer lookup service for internet applications*, Proceedings of the 2001 ACM SIGCOMM Conference, August 2001, pp. 149–160.
- [29] Ming Xie, *A decentralized redundant peer-to-peer system based on chord: Routing, scalability, robustness*, Master's thesis, Ottawa-Carleton Institute for Computer Science, December 2003.
- [30] Ben Zhao, John Kubiatowicz, and Anthony Joseph, *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*, Technical Report UCB/CSD-01-1141, 55, Computer Science Division, U. C. Berkeley, April 2001.