Self-Organizing Resource Location and Discovery

Diego Doval
Department of Computer Science
Trinity College, Dublin

A thesis submitted to the University of Dublin, Trinity College,
in fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

September 30, 2003

THOSIS
7459.

Declaration

I, the undersigned, declare that this work has not previously been submitted at this or any other University, and that, unless otherwise stated, it is entirely my own work.

Permission to Lend and/or Copy

Trinity College Library may lend or copy this thesis on request.

Signed: [illegible] (Diego Doval)

Date: November 23, 2004

A mis padres, Luis y Ana María,
por todo lo que no se puede expresar con palabras.

Acknowledgements

When I begun my work at Trinity College at the end of September 2001 I was not only starting a degree, I had also just arrived to a new country. My family, friends and colleagues all provided support and help, making this work possible.

I would like to thank, first, my supervisor, Prof. Donal O'Mahony. Donal provided support and guidance without constraining my work, always giving me freedom to advance in the direction I thought best while helping keep my feet (relatively) close to the ground. I am also grateful for his assistance even in what were definitely non-research tasks, from finding a room to rent before I arrived in Dublin, to giving me advice on how to sort out my taxes.

Dr. Linda Doyle and Dr. Hitesh Tewari helped in many ways through encouragement, comments on my work, and proof-reading of papers as well as of this thesis. I would also like to acknowledge the help, feedback and ideas of the members of the Networks and Telecommunications Research Group, in particular Juan Flynn, Stephen Toner, Philip Mackenzie, Tim Forde and Derek Greene for their work on environments and applications that I used either directly or indirectly during the course of this work. They, and others like Brian Lehane and Patroklos Argyroudis were always willing to talk about all sorts of topics and helped me understand better some areas of networking that were outside my main focus.

I also want to thank my thesis examiners, Dr. Simon Dobson and Dr. Seif Haridi, for their comments and suggestions.

Beyond the immediate environment of College, I'd like to thank Dr. Telma Caputti, who provided unwavering support throughout the years in pursuing my ideas. She also proof-read the Mathematics sections of this work, giving me many good suggestions for improving the text and the presentation.

Paul Kenny, my friend and business partner, gave me feedback and suggestions, along with constant encouragement.

My move to Ireland from the US was made infinitely easier with the help of Dylan Parker, Tracey Mellor, Martín Traverso, Tanna Drapkin, Victor Calo and Natacha Poggio. They helped me pack, organize, and move. Through it all Marcelo Cominguez, Sergio Mirabelli and Fernando Koch provided good conversation and advice, regardless of the distance between us.

Martín helped me with other aspects of the move, and in many other ways since then. He also proof-read this thesis, providing insights, correcting typos and asking probing questions that helped me improve my work.

Tracey and Dylan went above and beyond the call of duty by not only helping me with a myriad details (and those that were more than "details"—such as taking care of my furniture!), but also spending the next two years listening patiently to my rants regularly over the phone at all hours, day or night. I am amazed at all they've done for me, and I am deeply grateful for it.

I would also like to express my gratitude to Chris Vosnidis, who has always been there, and has helped me through more than one rough patch in these last two years.

I am lucky to count them as friends.

Finally, my family, though not here with me, was always present. My parents, Luis and Ana María, supported me at every step in every way they could. I would not be where I am today without their help, and this work is dedicated to them as a small token of my appreciation. My brother Sergio and my sister Laura have done more for me than they realize, through big and small everyday things—*Gracias!*

Abstract

Networked applications were originally centered around backbone inter-host communication. Over time, communications moved to a client-server model, where inter-host communication was used mainly for routing purposes. As network nodes became more powerful and mobile, traffic and usage of networked applications has increasingly moved towards the edge of the network, where node mobility and changes in topology and network properties are the norm rather than the exception.

Distributed self-organizing systems, where every node in the network is the functional equivalent of any other, have recently seen renewed interest due to two important developments. First, the emergence on the Internet of peer-to-peer networks to exchange data has provided clear proof that large-scale deployments of these types of networks provide reliable solutions. Second, the growing need to support highly dynamic network topologies, in particular mobile ad hoc networks, has underscored the design limits of current centralized systems, in many cases creating unwieldy or inadequate infrastructure to support these these new types of networks.

Resource Location and Discovery (RLD) is a key, yet seldom-noticed, building block for networked systems. For all its importance, comparatively little research has been done to systematically improve RLD systems and protocols that adapt well to different types of network conditions. As a result, the most widely used RLD systems today (e.g., the Internet's DNS system) have evolved in ad hoc fashion, mainly through IETF Request For Comments (RFC) documents, and so require increasingly complex and unwieldy solutions to adapt to the growing variety of usage modes, topologies, and scalability requirements found in today's networked environments.

Current large-scale systems rely on centralized, hierarchical name resolution and resource location services that are not well-suited to quick updates

and changes in topology. The increasingly ad hoc nature of networks in general and of the Internet in particular is making it difficult to interact consistently with these RLD services, which in some cases were designed twenty years ago for a hard-wired Internet of a few thousand nodes.

Ideally, a resource location and discovery system for today's networked environments must be able to adapt to an evolving network topology; it should maintain correct resource location even when confronted with fast topological changes; and it should support work in an ad hoc environment, where no central server is available and the network can have a short lifetime. Needless to say, such a service should also be robust and scalable.

This thesis addresses the problem of generic, network-independent resource location and discovery through a system, *Manifold*, based on two peer-to-peer self-organizing protocols that fulfil the requirements for generic RLD services. Our Manifold design is completely distributed and highly scalable, providing local discovery of resources as well as global location of resources independent of the underlying network transport or topology. The self-organizing properties of the system simplify deployment and maintenance of RLD services by eliminating dependence on expensive, centrally managed and maintained servers.

As described, Manifold could eventually replace today's centralized, static RLD infrastructure with one that is self-organizing, scalable, reliable, and well-adapted to the requirements of modern networked applications and systems.

Contents

List of Figures

Chapter 1

Introduction

Resource location and discovery (RLD) is a key building block for networked applications and systems, since it provides abstractions between names and physical locations for machines, services, or people that correspond to that name. For users, RLD abstracts a memorable or application-specific name (such as www.tcd.ie) from its physical network location, provides a way to map from easy-to-remember names to machines, and a way to search for services or machines according to specific query terms.

In essence, *resource location* creates a level of indirection, and therefore a decoupling, between a resource[1] and its location. This decoupling can then be used to solve one or more problems: mapping human-readable names to machine names, obtaining related information, autoconfiguration, supporting mobility, load balancing, etc. *Resource discovery*, on the other hand, facilitates search for resources that match certain characteristics, allowing then to perform a location request or using the resulting data set directly.

[1]In the pages that follow, we will commonly refer to locating people, machines, and software services or agents, usually interchangeably. When the term "resource" is used, it will refer to locating/discovering items in general, and those three in particular. Similarly, the term *user* of a given RLD service is usually understood to be a person, but in general both software and people use RLD services regularly to perform different tasks. The term *user* should therefore be understood to reference any system or person that could require RLD service.

Today, we are faced with a multiplicity of application-specific RLD systems, some of them with self-organizing properties, most relying on centralized (though distributed) infrastructure to operate. Basic Internet services, such as name resolution, remain implemented as static, hierarchical, centralized systems. Additionally, recent developments in networking have given rise to heterogeneous environments where the current solutions fit awkwardly, or not at all.

1.1 Background and Requirements

During the initial stages of this work, we studied both the major current systems that perform RLD and the infrastructure on which they are based.

Our analysis of the evolution of networked systems and services clarified the constraints that had driven previous developments, and it showed wide applicability a generic solution would have. As a result of this analysis, we derived general requirements that were common in all cases:

- Nodes are often mobile, making complete dependence on fixed infrastructure difficult or impossible.
- The network transport used can vary widely between implementations, as can between fixed and mobile implementations.
- While current prototypes rarely exceed a few dozen nodes, it is expected that in the next few years it will be possible to create Mobile Ad Hoc Networks (MANETs) of size ranging from only a few nodes to several hundred or even thousands, which places widely varying degrees of scalability requirements on the protocols that must support them.
- Membership of the network is determined dynamically by location rather than by a static or server-dependent configuration.

To measure the applicability of a solution we identified the main categories of *usage* that would be given to the service, namely:

- *Inexact search*: to find resources or people that match certain values in a query. This kind of search is relatively limited in reach, since the user is looking for resources in their vicinity[2]. A typical example of this would be looking for a printer in a conference location.
- *Exact search*: to find resources that might be or not in the vicinity, and of which the user knows the full name. A typical example of this would be accessing a given Internet website.

1.2 Manifold: Generic Self-Organizing RLD

The requirements and usage patterns expected of RLD allowed us to define the elements that would be necessary for a generic solution. We focused on designing a self-organizing system that used peer-to-peer (P2P) algorithms to eliminate dependencies from centralized infrastructure, as well as providing a reliable, scalable service.

Our solution is a hybrid system, *Manifold*, that incorporates two different self-organizing P2P algorithms:

1. Manifold-b, or "Manifold-broadcast", An algorithm that supports inexact searches of (typically local) resources, and
2. Manifold-g, or "Manifold-global", An algorithm that supports exact searches on a global scale, with extreme scalability and low overhead.

The first algorithm was an extension of initial work done exclusively for MANETs [18], while the second was an original development: a self-organizing algorithm with properties that make it well-suited for resource-location problems, including node control over its data, guaranteed results,

[2]As defined in terms of network topology

and predictable time-bounds. We characterized this algorithm mathematically and later related it to an emerging body of work generally identified as Overlay Networks[3]. We compared Manifold-g with other overlay network algorithms, noting in particular that, while guaranteeing similar scalability and self-organization, they are not designed from the ground up to guarantee data location (an important element since network nodes typically require control over the location and availability of the data they publish).

Finally, we implemented Manifold for use in mobile ad hoc networks. The implementation used the two algorithms according to the requests performed: Manifold-b is invoked when an inexact query (local in nature) is performed, while Manifold-g is invoked for exact queries (potentially global in nature). Our implementation demonstrated the feasibility of the system and is currently in use in the DAWN network (a MANET currently being deployed throughout the Trinity College campus) providing RLD for various experimental applications.

Manifold, then, can be used in any type of network and will be able to bridge them, providing the basis for generic, transport- and topology-independent resource location and discovery in large-scale, globally interconnected heterogeneous networks.

1.3 Summary of Goals

The goals for this work were threefold:

- To properly define requirements and usage patterns of RLD in heterogeneous network environments and wireless ad hoc networks in particular,
- To identify a set of algorithms that can provide resource location for self-organizing networks in general and for wireless ad hoc networks in particular. To quantify the limits of these algorithms and to relate them

[3]also often referred to as Distributed Hashtables, or DHTs

to the usage patterns that will define their performance in real-world applications, and,

- To demonstrate the feasibility of the system by describing an implementation in a real-world wireless ad hoc network.

1.4 Organization of this work

There are two main elements to this work: the design of the service that provides self-organizing RLD, and the algorithms that make it possible. The sections proceed from the highest level of abstraction to the lowest, since requirements for the service affect design decisions and requirements at the algorithm level, and then work back up to a description of an implementation of the system.

Chapter 2

Background and Related Work

Manifold includes two main elements: the service that provides generic self-organizing RLD and the underlying algorithms that make the service possible. Through this chapter, we will discuss the background for each of those elements. The analysis of the evolution and current state of the art of the various components, as well as of the services themselves, presented in this chapter, will allow us to derive requirements for a generic RLD service and the usage patterns that will govern its interaction with other systems.

We will begin by outlining the evolution of networked systems from the beginnings of the Internet to the present. This outline has two purposes. First, it will show the environment in which current centralized services were deployed and used, providing context for their evolution. Second, it will show that self-organizing networks are an appropriate solution to the problem of generic RLD, given the current development of networked systems, including the Internet.

2.1 The Evolution of Networked Systems

2.1.1 The Origins of Centralized Infrastructure

The first networked systems, developed in the late 1960s, were designed to connect hosts across great distances. This was a natural outcome of the usage that existed at the time: powerful centralized machines accessed through terminals. As a result, the early Arpanet routers (IMPs) were meant to connect remote systems. However, almost immediately, as more machines came online connected to the same IMP, researchers found that more and more traffic was happening *locally*, that is, between machines connected to the same IMP that would therefore be within the same geographic location—at the time, this phenomenon was called "incestuous traffic" [54].

This "incestuous traffic" marked the appearance of what is now known as LAN traffic. Since it was first identified, LAN traffic has grown exponentially compared to the growth in Inter-LAN (or Internet) traffic, to the point where today Internet traffic is insignificant compared to LAN traffic: the majority of the traffic has been pushed to the edge of the global network. In the early 1980s, as LANs were deployed, the first Internet-wide resource location systems were being developed. Machine name to IP resolution for example, initially a simple process —the distribution to system administrators of a *hosts* file with host name to IP address mappings– had become unsuitable for the number of hosts and the speed at which the network was growing, thus requiring the creation of an automated system: DNS [57].

That is, even as LANs started their geometric growth, the first Internet Resource Location and Discovery systems (of which DNS is the best known) were being designed and deployed—but without taking into account the requirements for the dynamics imposed by LANs: more varied topologies, greater number of nodes, and, more importantly, mobility.

This did not create problems initially. The central uses of resource location and discovery are to ask the remote machine to perform a certain task (e.g., perform a calculation), to obtain data from it (e.g., obtain a name-to-address mapping, serve a file) or to store data in it (e.g., update a database record), and the dissemination of LANs happened on the backs of the early personal computers, which did not have the storage of the processing capacity to deal with anything except their own local tasks. Even some fifteen years later, in the mid 1990s, when PCs were capable of performing more tasks in "server-mode" very few applications existed that took advantage of them, consequently limiting interesting in edge-only RLD.

For a long time since the appearance of LANs then, Resource Location and Discovery was split in two: *global RLD*, exemplified by DNS, and performed either between Internet hosts or between clients and Internet hosts, and *local RLD*, such as shared printers, file servers, or contact databases using protocols such as LDAP. In a very real sense, the firewalls that protected LANs from external attack were a frontline that delimited the difference between global RLD and local RLD. There were some solutions for connecting central databases between LANs, but, with their high cost and complexity and (perhaps more importantly) lacking a standard, they remained niche solutions.

In the late 90s, peer-to-peer computing gained world-wide prominence in the form of global music-sharing systems in which individual PCs were used to serve digital music files. Almost overnight, a task that had been typically confined to the LAN (i.e., locate a particular file) could now be performed on a global scale, not between clients on the same corporate network, not between a client on a LAN and a host on the Internet, but between clients residing on completely different LANs. The dividing line between local and global RLD had finally been crossed.

At the same time, other applications began to emerge, such as peer-to-peer collaboration, and with the growth of portable devices (laptops, handhelds, cellphones) and wireless networking, the complexity of the network environment in which RLD had to operate increased even further. Applications that before required a connection to a particular server now had to provide a connection to a particular *user*, regardless of the device, or the network, the user was on.

Mobile Ad Hoc Networks (MANETs) have become an important area of innovation and research in the past few years, and most of the efforts have concentrated in solving the basic building blocks of the technology: hardware, routing protocols, and so on. MANETs are typically wireless but they can equally include "wired" components. These networks interact with the Internet through endpoints, connecting and disconnecting as they are formed, or as they move across geographical boundaries.

The Internet, which was originally designed as a network-of-networks, is fulfilling its promise. It is quickly changing from a set of static nodes connected to a high-speed backbone in "monolithic" fashion to a collection of a multitude of small networks that connect to each other depending on location and capabilities available at their entry point. In other words, the *Internet itself is becoming an ad hoc network.*

In this new context, services designed for what was originally a static network of a few thousand nodes are increasingly strained to accommodate new networked applications and systems.

2.1.2 Edge Networks and RLD

In large part, the Internet's increasingly ad hoc nature is driven by the growth of networks connected at its endpoints, commonly referred to as "edge networks."

Edge networks are becoming more powerful, more mobile, and more dynamic. As applications built on them become more distributed, edge networks will cease to be "client-only" environments, in which nodes only use the resources of the network; they will become full blown client-server environments in which all nodes request *and* provide services to the network through peer-to-peer applications deployed on a global scale.

The growth of edge networks and the variety of applications that are making use of them (from messaging systems to global file-sharing applications) underscores that a self-organizing solution to the problem of RLD is not only possible but also desirable. For example, mobility, such as that exhibited by portable devices and MANETs, presents a number of challenges for an RLD system. Devices in a mobile network might be turned on or off and might quickly move across locations with different types of infrastructure, sometimes switching to environments where a central server is not directly accessible or might not be available. Typical scenarios include a mix of devices (e.g., handhelds, embedded devices, base stations, desktop computers) that use different physical layers (such as Bluetooth, IEEE 802.11b, Ethernet) with different protocols, depending on various factors such as location and power consumption requirements.

These unique qualities mean that certain protocols and systems that work well in more static, homogeneous topologies (of which the best example is the Internet) might perform badly or not work at all in these kinds of networking scenarios. For example, a Bluetooth-enabled cell phone has no way of referencing a handheld device that might be running on a wireless network (e.g., IEEE 802.11b) even though the cell phone might have a connection path available through a desktop computer with both Bluetooth and wireless stacks. If this connection was possible, the cell phone might be able to determine that the handheld device is in range and therefore it should not display appointment reminders, since a device that is better suited for it (the handheld) is active in the vicinity.

This kind of functionality would also be useful for global communication systems, such as Internet-based VoiceIP phones or SMS (Short Messaging Systems) designed to run on mobile ad hoc networks. For example, when an ad hoc wireless network is set up for emergency crews at a disaster site, if one base station is deployed to provide connectivity within the nodes in one network (e.g., for Paramedic crews) and to the Internet, nodes from a nearby ad hoc network (e.g., Fire Department) and from the Internet (e.g., a government official trying to contact somebody on site from a remote location) would not be able to locate those nodes.

A MANET could easily function in a way similar to a "fixed" network such as the Internet, with static nodes such as PCs connected through IP-based protocols in a stable, slowly-evolving topology. Because of their nature, MANETs can also support a highly dynamic environment: mobile nodes that quickly change membership between different MANETs, nodes that quickly turn on and off, and others. In the most extreme case a MANET could be formed completely independently from any infrastructure (e.g., by the emergency crews at a disaster site). Finally, MANETs, with or without access to infrastructure, run on a variety of protocols that may not be compatible with each other, but that would still find useful to locate nodes or services between them.

If the resource location problem is solved for large-scale mobile ad hoc networks[1], the same solution would apply for networks in general, including wired/static topologies (e.g., the Internet).

2.2 Types of RLD

Based on different systems that implement RLD functions (Described below in Section 2.5) we can identify three main categories of systems, covering three different types of RLD *needs* from users. These are: Name Resolution, Directory Services, and Search Services.

[1]Throughout our work we will reference the problem of RLD on MANETs *as a test case*, while keeping in mind the larger goal of applying it on generic heterogeneous network environments.

2.2.1 Name resolution

Name resolution is the most basic and widely used RLD system. Naming is necessary to map easy-to-remember names to physical machine locations (on the Internet, a name-to-IP mapping). Naming is the core operation on any RLD system, and all types of resource location could, in general, be considered a type of name resolution problem, since they are trying to map a resource's name with its physical location.

More recently, the use of RLD for name resolution has extended to the area of *presence*, which tracks people, and sometimes software agents, across machines. Presence is a concept used by all Instant Messaging platforms [59].

2.2.2 Directory Services

Directory services are extremely common, particularly in corporate intranets. They are used to store information regarding people, machines, or even services within the organization. This differs from the concept of presence and name resolution in that it returns information about a resource or person, rather than their location.

2.2.3 Search Services

More generic and open ended search services are also common. Printing could be considered a special case of search, since a print operation on a nearby printer can be started using the appropriate search parameters. In large organizations, where employees often move between different offices or buildings, the use of search allows them to locate particular items of information, or the resources needed for particular tasks.

Search differs from name resolution and directory services in that it is more open-ended. At the end of a search query, the user will commonly perform a final operation either of name resolution, or directory service request to obtain the location or information about the resource requested, respectively. This typically happens invisibly from the user's point of view. Therefore, depending on the way it is implemented, search can be considered either a component at the same level of directory services and name resolution, or middleware built on top of them.

2.2.4 Similarities

In essence, all of these services perform a mapping of keys to values. Sometimes those key/value mappings are updated frequently, sometimes not, but that is irrelevant with respect to the location/discovery process itself. As we will see in the following chapter, the difference lies in usage patterns: whether we are trying to *locate* a resource that we already know about, or *discover* one that matches our needs.

2.3 Usage patterns

At the core of the problem of resource location and discovery is the way in which RLD services (such as those defined in the previous chapter) are typically used. In any network, users will likely try to locate and contact resources (which may or may not belong to another user) according to their *location*, which divides typical usage into distinct categories.

These categories are in a sense present in the terms "Resource Location and Discovery". *Resource Discovery* implies that the user wants to discover resources that might match certain attributes, while *Resource Location* implies that the user knows the name of the target resource, and only its physical location is unknown.

2.3.1 Local/inexact

In the first case, *local/inexact* search, the user is looking for something that is in their vicinity (as defined by the topology of the network). The search is typically performed based on *capabilities* (e.g. a printer, or a device that can act as relay to the Internet) or based on inexact queries[2] (e.g., a storage system with a given name, or another user's handheld computer). DHCP (Section 2.5.2) lookups or LDAP(Section 2.5.4) queries are examples of local/inexact requests.

In general, this means finding resources that match certain values in a query. This kind of search is limited in reach, since the user is looking for nearby resources.

2.3.2 Global/exact

When the target of the search is beyond the user's physical reach, it is commonly based on *exact names* (e.g., the printer at the user's office, a person's SIP [71] address). A DNS (Section 2.5.1) query is an example of an exact name search for a globally unique string. We define this type of search as a *global/exact* search.

Generally speaking, this type of search is performed to find resources that might be or not in the vicinity, and of which the user knows the full name. A global search, in this context, requires that a result be returned if the resource exists anywhere in the network.

Additionally, the use-case of *local/exact* searches is a trivial subcase of this one; even if the resource to be located is within physical proximity of the user, the usage case remains unchanged.

2.3.3 Global/inexact

The final category that we will consider is *inexact searches* performed on a *global scale.*

[2]Commonly, this type of use imply searching for resources that conform to a certain property or whose name matches a given substring.

This kind of generic functionality is currently provided (partially) by Internet directories or search engines. Additionally, most current file-sharing systems are global in scale; however, they provide no guarantees as to whether a resource can be found within reasonable time limits. Because of this limitation, both Internet search engines and large-scale file-sharing networks are more similar to a local search but with vastly expanded scope, rather than a truly global search that will guarantee location of a resource if it exists somewhere on the network.

Global/inexact search systems are more limited in application than the two previously described, and so its usage differs from that of resource location and discovery applications. Therefore, a system that provides this particular function in a decentralized form (and one that could match other requirements that we will specify later) is outside the scope of this work, and will not be discussed further.

2.4 Generic RLD: Requirements

The usage patterns identified above helped us enumerate a requirement that a generic RLD system must satisfy, with emphasis on generic RLD for heterogeneous networks that must interoperate on a global scale.

2.4.1 Correct and Time-Bounded

When RLD is *global/exact*, the service must guarantee that the target of a search is found as long as it exists anywhere on the network[3]. Additionally, the search operation must be time-bounded, with the goal of responding within a reasonable time period to the user's request.

[3]This is not necessary when the search is *local/inexact*

2.4.2 Coexist with Legacy Systems

Because any change for the system has to come incrementally, nodes should be able to join the network without special configuration requirements. Only running the necessary software on the node should be necessary. Further, when legacy systems, such as DNS, are available, the network should be able to revert to them if desired (since other target nodes might not be participating in the self-organizing network created by the new service).

Additionally, gateways should exist to provide two-way resolution between other network nodes (particularly Internet nodes) and nodes in the resulting RLD network.

2.4.3 Scalability

The resulting system could be used globally to connect distant ad hoc networks (e.g., to make a voice-over-data phone call between handheld devices or cell phones without using the operator's infrastructure), or locally within a large scale ad hoc network (e.g., within a group of thousands attendees to a conference).

2.4.4 Support mobility and dynamic topologies

A generic RLD solution must support constant membership changes, providing resource location for the new nodes and maintaining validity of the query results, i.e., providing results that are up to date in terms of topology. Additionally, it must provide correct results for queries for the remaining nodes of the network when one or more nodes leave the network or are switched off.

When nodes move between heterogeneous networks they shouldn't be required to change their configuration, and there should be no dependence on any particular node or underlying transport protocol to provide resource location.

2.4.5 Support low-resources

Many of the nodes participating in the network will be low-resource nodes in one or more senses: limited in processing power, or low-power, with a slow network connection, etc. As such, an RLD service must take into account this factor and either a) adapt to the capabilities of the nodes or b) have generally low resources as a minimum to join and operate within the network.

2.4.6 Support Discovery

When performing *local/inexact* searches only, users will frequently know only the general details of what they are attempting to find, that is, when they are engaging in *discovery* of a resource rather than location. Therefore, the system must provide partial attribute or string matching on local/inexact search to support this functionality.

2.4.7 Security

An important issue in resource location is security, particularly in a decentralized network such as the one described here. When resolving the physical location of resources, a node should be able to verify that the location resolved is valid and current. Without security, a malicious network user that has access to the message flow in the network could:

- impersonate other nodes and resources by answering requests for them.
- modify query results being passed along and change the values passed.

In DNS, security is largely an issue of trust between clients and servers. Both of the problems described can happen in DNS, assuming that a name server (slave or master) has been compromised (commonly called DNS "spoofing"). Additionally, if a gateway has been compromised, the DNS requests themselves can be manipulated by a malicious third-party.

We will leave the verification of the identity of the node location/discovery result to higher level application layers with enough information to make these checks (for example, by verifying signed security certificates), just like DNS does. In the future, modifications similar to those currently being discussed for DNS IETF [70] could also be applied.

2.5 RLD and RLD-based Systems: State of the Art

Throughout this section we will revisit systems that we identified as performing RLD functions either implicitly or implicitly. We will present these systems starting from the most standard and broadly used to the most specialized and research-based.

Some of the systems we will describe appear, on the surface, to have little relevance to the subject of RLD. This is not the case, however.

Solutions to specific problems, such as mobility[4], are in fact using very specialized types of RLD. If a system is designed to support, for example, mobility of network nodes, it will generally revert to some form of indirection between its physical location and its global identifier. That, in turn, creates the need for part of the system to be able to locate that global identifier, which is essentially a function performed by an RLD subsystem of some kind.

This type of specialization of RLD solutions is the norm rather than the exception. The analysis of these systems, along with their differences and similarities, leads to two important conclusions. The first is that a generic RLD system would have potential application well beyond that of resolving names. The second is that, even though some of the systems use distributed (even, in some cases, self-organizing) environments, all of them are based,

[4]We will come back to examples of solutions to the problem of mobility (primarily of devices, but also of services) below.

implicitly or explicitly, on the assumption that significant fixed infrastructure exists somewhere on the network. These two conclusions underscore, in turn, the need to maintain generality in our requirements (and, consequently, our design) and support varying degrees of scalability, as well as maintain focus on working on a solution that could function completely disconnected from fixed infrastructure.

These systems, therefore, provide not only examples of how RLD is usually approached today but also of the broad applicability that a generic RLD solution would have.

2.5.1 DNS

On the Internet, it is the Domain Name System (DNS) [55] [56] [57] that provides the lowest level of name resolution service available, and is the prototypical case of Global/Exact matching mentioned in Section 2.3.2. Name resolution creates mappings of easy-to-remember names to physical nodes. In the case of the Internet, this means mapping service names to IP [28] numbers. DNS is hierarchical and relatively static, requiring propagation of names from root name servers to a series of slaves across the network. Each IP subnetwork has a fixed static reference to the physical location of the local name server node, using it to resolve the names of all other nodes into IP numbers so that communication with them can be established. This scheme is dependent on servers; client machines are reachable only as part of the subdomain of a given server, assuming a correct setup.

The naming system on which DNS is based is a hierarchical and logical tree structure referred as the *domain namespace*. Organizations can create private networks that are not visible on the Internet, using their own domain namespaces, and each node in the DNS tree represents a single DNS name. Each organization/group is assigned authority for a portion of the domain namespace and is responsible for administering, subdividing, and naming the DNS domains and computers within that portion of the namespace, as well as for servicing queries that correspond to its particular domain.

Name-to-address mappings published through the DNS system can be resolved by clients through their Internet Service Provider's (ISP) or organization's DNS servers, as shown in Figure 2.1.

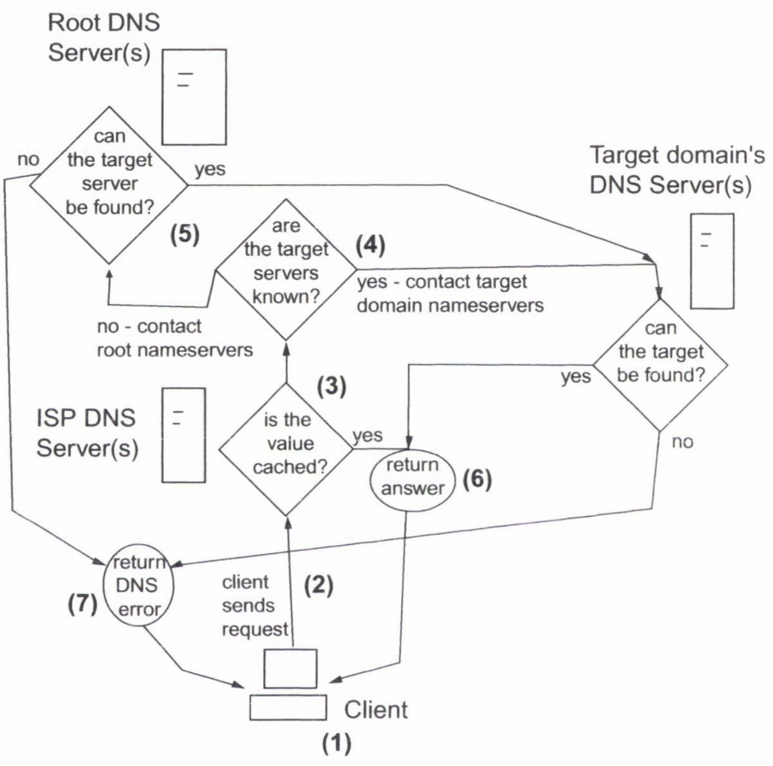

Figure 2.1: A Sample DNS Query-Reply Cycle

The cycle works as follows (Numbers at each step match correspond to those in the figure):

- Client requests a name (1), such as www.somedomain.com, in response to a user's action (e.g., navigating to a webpage).
- The client contacts the domain server (2) for its ISP or organization, requesting the address of the domain name in question.

- The client's server first tries to answer by itself using cached data (3). If the answer is cached it is returned directly. Since the client's server is not responsible for the name and is just acting as a relay, the answer is marked as non-authoritative.

- If the answer is not found in the cache, or if the value cached has expired (i.e., it is past the TTL, or Time-To-Live, of the value), then the server will try to contact the (authoritative) name servers for the domain directly (4) requesting the mapping. If the name servers are not known (i.e., not cached), the client's server requests the address of the domain's name servers from the Internet Root servers (5), which reside at a well known address.

- Once the domain's name servers are contacted, the client's server returns the result (6) or a DNS error (7) if the target name was not found.

The way in which DNS works points to a number of problems for dynamic systems. Aside from its dependency on fixed/centralized infrastructure, changes to the authoritative server for a domain (i.e., if the server is moved to a different location) have to be reported to the Root servers, which will propagate the new values to other servers as they perform new requests (as the cached results expire). This typically means a delay of several days between a move of an authoritative name server and the propagation of its new address across the internet. Increasingly, machines operate on autonomous mode, not related to any particular infrastructure, and DNS's structure prevents its use for more direct client-to-client communications.

DNS was designed to solve the problem of managing hosts tables in the original Arpanet, with requirements vastly different than those found today. In this respect, it has performed admirably, scaling well beyond its original design parameters to accommodate millions of hosts and hundreds of millions of clients. Even so, its design poses a number of problems, such as increased requirements on the domain servers (particularly on the root servers), and their increased administrative complexity, as well as their potential as points of attack or failure. Systems that solve some of the problems observed in DNS, providing better load and administration distribution, but still hierarchical and largely centralized, have been proposed in the past [10] [37] [49].

DNS resolution need not be implemented using a hierarchical system, though. Implementing DNS using a self-organizing network of servers has been proposed recently in [12]. While this would provide some improvements (most notably automatic load balancing and faster updates) there would still be dependence on fixed infrastructure, namely, the set of servers that are responsible for replying to requests. In the next chapter we will discuss some additional elements that a full solution will require that would not be satisfied in that case (for example, how to deal with ad hoc wireless networks of small devices that have no connectivity to the Internet).

A standard setup of DNS on a client machine requires that the information for the DNS server(s)'s IP addresses that serve the machine in question be entered by hand. This was appropriate initially, but as mobility and use of the Internet have become more widespread (thus bringing in less technical users) a simpler solution was required to provide that setup. This created a need for a protocol that would perform Local/Inexact searches (as defined in Section 2.3.1) for available DNS servers in the vicinity of the user.

The standard solution for this problem is provided by DHCP and NAT.

2.5.2 DHCP and NAT

The Dynamic Host Configuration Protocol [20], or DHCP, is an autoconfiguration service that provides a way to automatically assign all details about the network to a computer that wants to be a part of it. A user can simply plug a device to the network port and start working without the need for editing any settings manually. Autoconfiguration can be considered a subset of the resource location problem if resource location is used to provide a "bootstrap" mechanism that can then initialize a node's operating/connectivity parameters.

In DHCP, the client sends out a request to the DHCP server by transmitting on an IP broadcast address, essentially performing an open-ended search on the local network. The DHCP server responds to the request by providing a lease to an IP Address and other relevant details about the network, such as gateways or DNS servers.

The settings given to the client by the DHCP server are not permanent but time-bound, to solve the problem of network failures (i.e., if a node fails the server will eventually take over the IP assigned again when the lease expires). Along with the settings the server also tells the client about the period (lease-time) for which the settings are valid, and if the client needs to use the network beyond this time then it has to "renew" its lease.

For LANs, recent years have seen the growth of more dynamic network topologies, and the need to easily provide Internet access to machines in them. Again, a Local/Inexact solution is required to locate a service that provides that functionality.

NAT [79] (Network Address Translation) is a special type of autoconfiguration service that provides internet access to machines through a special type of gateway. A NAT gateway translates the clients' internal network IP Addresses into the IP Address on the NAT-enabled gateway device, making the network appear as a single device to the rest of the world. NAT gateways typically interoperate closely with DHCP to provide zero-configuration Internet access for clients in small networks.

All communications from the private network are handled by the NAT device, which will ensure all the appropriate translations are performed to maintain the "illusion" of one-to-one Internet connections for the devices connected through the NAT gateway.

When a NAT gateway receives a packet into its private interface, it strips the source IP Address from the third layer in the IP stack (e.g., 192.168.1.5) and places its own public IP address (203.154.123.223) before sending it to its target host in the Internet. (In some cases, depending on the NAT mode, the source and destination port numbers, found in layer five of the IP stack, will be changed as well). The gateway then stores this information in an in-memory table so when the expected reply arrives it will know to which workstation within its network it needs to forward it.

NAT is of note because its increasingly widespread use has given renewed importance to the issues it raises. NAT makes it impossible for current naming systems to resolve names behind the NAT server (unless another system, such as MobileIP, described in Section 2.5.5 below, is in use to bridge between the local network and the Internet). In essence, NAT turns a network of computers into an ad hoc network, with all the problems that entails for current centralized systems, and it is a situation solved by the system presented in this work (for a centralized solution to the NAT/naming problem, see the description of TRIAD in Section 2.5.6 below).

2.5.3 INS

The last few years have seen some developments in the area of name resolution specifically, in particular on the Intentional Naming System [1], or INS. INS defines a network as a set of components: clients, services, and a decentralized network composed of "Intentional Name Resolvers" or *INRs*. Clients send requests to INRs, specifying a particular name-specifier, which is matched against the services advertised in the resolver network. Services periodically advertise their intentional names (INs) to the system to describe what they provide. Intentional names are based on a set of attributes and values (i.e., key/value pairs) that allow expressing generic system information in hierarchical form.

The main activity of an INR is to resolve INs to their corresponding network locations. When a request message arrives at an INR, it performs a lookup in its name-tree. The lookup returns information which includes the IP address(es) of the destination(s) with advertisements that match the requested name as well as a set of routes to next-hop INRs to support routing when mappings change in the middle of a session.

INRs also store metric information (such as load, distance, etc) to facilitate self-managing of the different parameters the service is subjected to. Load management, for example, is thus performed by the INRs.

While providing certain desirable qualities such as automatic load balancing and self-management, INS is still dependent on network infrastructure that has to be maintained (i.e., the INRs), making it less useful in purely ad hoc environments where no central management is possible.

2.5.4 LDAP

Another example of search, but for information related to a resource rather than its location, is that of LDAP [85] [86], a commonly used directory service to provide discovery of information about resources, typically people, groups, and organizations. LDAP is a lightweight version of the X.500 Global Directory Service [84].

LDAP directories reside on a server or cluster or servers, and are typically used to store information about entities like people, offices, locations, etc., but it could equally store other types of relatively static information, essentially anything that can be described by a set of attributes. In LDAP every entry has a primary key called the Distinguished Name (DN). DNs are unique within a particular LDAP directory. An LDAP server uses a back-end datastore to store its data, but is not limited to using any particular database, achieving database-independence through the a flexible notion of a schema for the data it needs to store. The schema consists of entries organized in a hierarchy, optimized for reading rather than writing.

A typical LDAP configuration is shown in Figure 2.2.

As the figure shows, LDAP is based on a client-server model. Servers

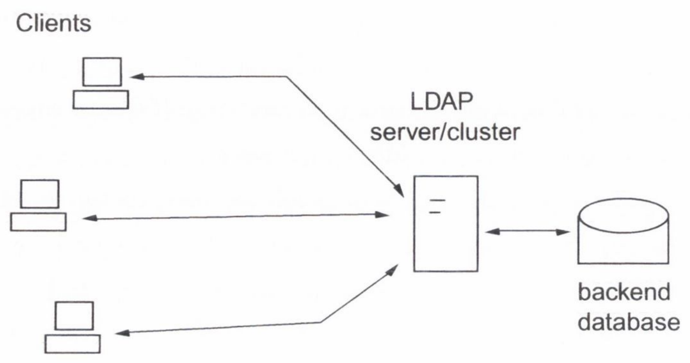

Figure 2.2: Typical LDAP Configuration

make information about resources accessible to LDAP clients, and define operations that clients can use to search and update the directory. Typical LDAP operations include:

- searching and retrieving entries from the directory
- adding new entries in the directory
- updating entries in the directory
- deleting entries in the directory
- renaming entries in the directory

For example, to update an entry in the directory, an LDAP client submits the distinguished name of the entry with updated attribute information to the LDAP server. The server uses the distinguished name to find the entry and performs a modify operation to update the entry in the directory.

LDAP can provide results both exact and inexact searches, but not globally, unless a global infrastructure of servers/clusters is deployed and maintained.

2.5.5 MobileIP

IP version 4 assumes that a node's IP address uniquely identifies its physical attachment to the Internet. Therefore, when a host tries to send a packet to another host, that packet is routed to the target host's "home network". In static environments this does not present a problem, but when a node is mobile, packets could easily end up routing to the previous location where the node was, rather than its current location. This is solved by adding one level of indirection to the process, which in effect turns the problem into one of a special type of resource location. The level of indirection establishes a fixed point that can be easily contacted and that will be notified of changes of the target node, thus adapting to mobility. The fixed point, whatever its form, thus becomes a dynamic table that keeps the position of the mobile node updated as it moves, providing in essence Global/Exact resource location for the target machine, based on an identifier.

One such solution is MobileIP [64]. In MobileIP, when the target host is on its home network and a another host sends packets to it, those packets are handled normally. However, if the target is mobile and away from its home network, it uses *agents*, to work on behalf of it. The *home agent* must be able to communicate with both the home network *and* with the mobile host when it is online, independently of the current position of the mobile host. The home agent then becomes a permanent relay for that mobile host. The *foreign agent*, meanwhile, is responsible for relaying requests to the mobile host in its foreign network.

An example of a MobileIP node in a foreign network is shown in Figure 2.3.

As we can see in the figure, the MobileIP host uses the foreign agent and the home agent to receive information from other clients, essentially letting the agents act as resolvers for its location.

To determine a change in its network environment, the mobile host detects its current location (and therefore the need to register with the agent) based

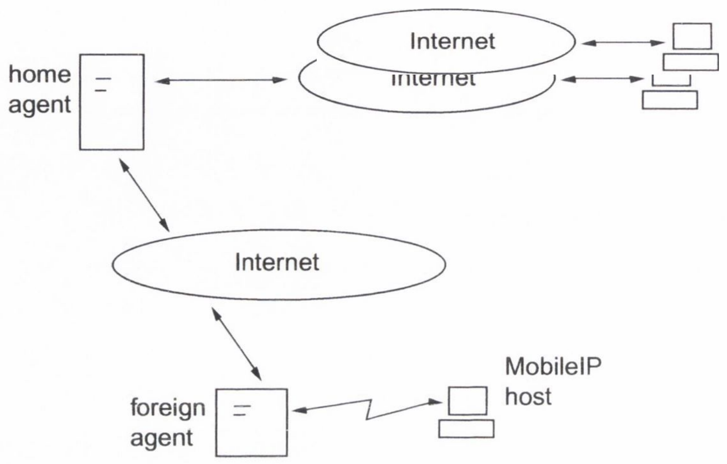

Figure 2.3: A MobileIP Node in a Foreign Network

on looking at periodic adverts of the foreign agent and home agents.

When the mobile host returns to its home network, it does not require mobility capabilities, so it sends a deregistration request to the home agent to deactivate tunnelling and to remove previous care-of address(es).

At this point, the mobile host does not have to (de)register again, until it moves away from its network. The detection of the movement is based on the same method explained before.

While MobileIP's solution appears to solve the problem of mobility, it has one important drawback: since the agent must be running constantly at a fixed location, it must be managed from a centralized location as well. So mobility in Mobile IP is obtained at the cost of further centralization of the infrastructure, with all the subsequent disadvantages.

2.5.6 TRIAD

TRIAD [13] (which stands for Translating Relaying Internet Architecture integrating Active Directories) is a content layer on top of IP that provides scalable content routing, caching, content transformation and load balancing, integrating naming, routing and transport connection setup.

To perform its functions, TRIAD creates a set of *address realms* that are interconnected through a hierarchy. At the leaf level, an address realm corresponds to an certain network owned by an organization, or a set of nodes organized within a network. The router that connects this level to the WAN acts as a TRIAD relay agent between realms, translating addresses as it relays packets between the realms that it interconnects. Higher-level address realms correspond to local and global Internet service providers (ISPs). Backbone or wide-area ISPs can connect at peering points, as it happens today, but through high-speed relay agent routers. Within a realm, the operation of naming, addressing and routing operates the same as currently with IPv4. Thus, TRIAD doesn't require changes in the host or the router.

End-to-end Internet-wide identification of a host interface or multicast channel is provided using a hierarchical naming scheme based on DNS naming. Names are the basis for all end-to-end identification, authentication, and reference passing in TRIAD, since there is no other identifier for the host interface that is global and persistent, unlike addresses in IPv6 and in the original Internet architecture. In particular, IPv4 addresses have no end-to-end significance since they can change depending on network configurations, dynamic hosting, and so on, and are reduced to the role of transient routing tags.

TRIAD supports name resolution, wide area relaying and content lookup. Name resolution in particular is supported by TRIAD's Internet Name Resolution Protocol (INRP), a protocol that is backward-compatible with DNS. Clients initiate a content request by contacting a local TRIAD content router, just as they would contact a DNS server. Their requests may include just the "server" portion of a URL, although TRIAD supports looking up the entire URL, allowing for fine-grained load-balancing and higher availability. Because names are distributed based on a distance-vector routing algorithm with path information, TRIAD can also reduce latency and round-trip times by routing information along more optimal paths (compared to DNS).

A name resolution cycle for TRIAD looks very much like the one presented for DNS in Figure 2.1, with the following important exceptions:

- The DNS servers are replaced by *Content Routers*, which allow publishing and propagation of the information.
- The round-trip times are shorter, because of INRP's characteristics.
- The Content Routers support resolution of generic names, not only of the server name portion of them, allowing resolution of server names as well as of individual URLs.

TRIAD is, then, an indirection infrastructure that improves in several respects on previous systems, such as DNS, but that is again dependent on fixed infrastructure (as well as modification/additions on the current Internet infrastructure).

2.5.7 i3

Finally, a solution proposed recently in a research context to essentially the same problem (but with wider applicability) is the Internet Indirection Infrastructure [80], or i3. i3 provides a layer of indirection that decouples sender from receiver: sources send packets to a logical identifier and receivers express interest in packets sent to an identitier. Delivery is "best effort" like in today's Internet, with no guarantees about packet delivery. The system is similar to IP multicast, but in IP multicast the infrastructure must build efficient delivery trees; in i3 these are managed by a trigger inserted into an overlay network (Overlay Networks are discussed in detail in Sections 2.9 and 2.9.1) which routes queries efficiently.

At its core, i3 is relatively simple. The i3 services are provided by a self-organizing "cloud" of servers that provide the mapping functions and routing. When a node (Receiver) wants to receive information, it can register with the cloud by inserting a trigger that effectively states "Send all packets with identifier *id* to address *R*," as shown in Figure 2.4.

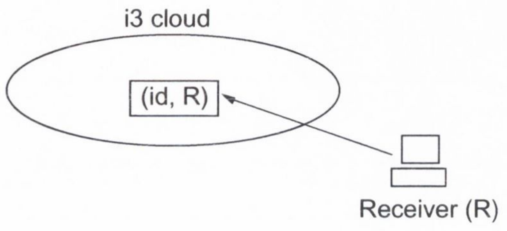

Figure 2.4: An node publishing its location to the i3 cloud

After a trigger is added, another machine (Sender) can send packets into the i3 cloud using the *id* necessary to reach *R*. The i3 cloud then matches the id to the trigger and derives the address *R*, subsequently forwarding the packets, as shown in Figure 2.5.

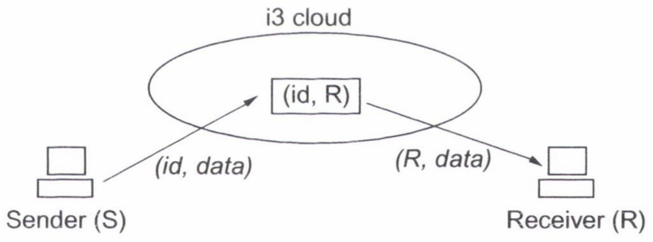

Figure 2.5: A node sending data through the i3 cloud

This process describes the unicast functionality of i3. Note however that more than one Receiver can register for a given identifier, thus allowing i3 to support other modes of operation, such as anycast and multicast.

In i3, the "fixed point" is a logical identifier, completely removed from physical location. Its operation depends on a self-organizing network of hosts that take responsibility for performing the necessary mappings. Thus, as other systems that we have described, i3 depends on infrastructure that has to be deployed and maintained.

2.5.8 State of the Art: Implications

We have covered a variety of systems that perform RLD either explicitly or implicitly. Even though some of them exhibit desirable features for a generic, dynamic RLD solution, all of them rely to different degrees on fixed infrastructure. Furthermore, none of them is specifically designed to deal with all of the requirements that a generic solution to the RLD problem must exhibit. Most use specialized RLD solutions that are not interoperable or that are not suitable for use in other applications.

For these systems, support for both highly dynamic topologies at the edge of the network coupled with extreme scalability requirements becomes a problem. None of those systems satisfies *both* the local/inexact requirement *and* the global/exact requirement in a single solution, increasing the complexity for applications that require both of these services (as many typically do). Finally, none of the systems can operate completely independent of centralized infrastructure, making them unsuitable for operation in wireless ad hoc networks.

Many systems designed for specific purposes (e.g., systems that support mobility in IP networks) use RLD concepts or subsystems to achieve their results. Therefore, if a generic self-organizing resource location and discovery system is scalable enough, and resilient enough, its applicability would not be limited to mobile ad hoc networks alone, or even to name resolution or resource discovery alone. Such a system would be useful for a variety of networked infrastructure and applications, from name resolution, to mobility applications, to presence functions provided directly to end-users.

2.6 The Manifold Algorithms: Evolution, and Related Work

In the preceding sections we discussed different types of RLD systems and presented various examples, deriving the usage patterns that govern them. We discussed requirements for our solution from those usage patterns in Section 2.4.

Based on the requirements and usage patterns, we concluded that Manifold would require two different self-organizing P2P algorithms to operate. The first of the algorithms, Manifold-b , is based on a TTL-based P2P algorithm (see Section 2.8 below). The design of the second algorithm, Manifold-g, was based on the idea of creating a virtual topology using the values of the names stored by each node that would allow efficient traversal and provide guaranteed results.

After the work on Manifold-g was completed, we related it to a category of algorithms known as Overlay Network algorithms (also referred to as Distributed Hashtables, or DHTs—see Section 2.9 below). In the sections that follow we will review P2P systems in general and these two types of P2P algorithms in particular, pointing out some of their differences, advantages and disadvantages.

2.7 Introduction to P2P systems

Peer-to-peer (P2P) systems are distributed systems that operate without the need of centralized control or organization. They achieve this by running the same software on each node (even if the software differs between nodes, they can also interoperate by conforming to the same set of network-based application programming interfaces, or APIs). Network-wide behavior thus *emerges* from the action of the local algorithms in each of the nodes. All P2P systems, regardless of their application, provide a mechanism to find the location of a given piece of data within a network. All P2P systems provide a lookup function that, given a certain string to be found returns a set of nodes that match it. We will refer to this function as the *locator function.*

Because of their decentralized nature and relatively low consistency requirements, P2P networks have no single point of failure, continuing to function even with multiple node failures.

The network structure defined by P2P networks is, regardless of their type, *independent of the underlying physical topology*; 'neighbors' can exist across the Internet or within the same subnetwork. While some types of networks adapt to the underlying physical topology, this kind of optimization is not in general required for the proper operation of the P2P algorithm.

Recent years have seen research and development on the field of P2P systems of all kinds grow steadily, as the many examples presented in [63] show. In large part, this has been propelled by the massive success (if not necessarily in commercial terms, certainly in exposure and number of users) of Internet file-sharing applications. Another factor is that these systems proved that self-organizing protocols can effectively function in a global scale, something that is, at first glance, slightly counter-intuitive.

The idea of peer-to-peer systems however, is not new. As we noted in the Introduction, the Internet (then Arpanet) was originally designed to provide host-to-host connectivity where remote hosts were treated as equals, creating something that was, both in principle and in practice, a network of peers. There is one crucial difference, however, between the original Arpanet design and today's P2P networks: self-organization.

Arpanet, though decentralized by nature, required extensive human intervention: for example, a host newly connected to the network could not, on its own, advertise its existence to other hosts. Additionally, the first popular applications on Arpanet quickly tilted the field towards client-server models (e.g., Telnet [15]), resulting in diminished interest in true self-organizing P2P models.

Earlier, we cited DNS as an example of how name resolution today happens in centralized, hierarchical fashion. However, it is important to note that DNS provided one of the first examples of self-organization found on the Internet. To the clients that resolve queries, DNS servers provide only one function, i.e., resolving Internet names. But within the DNS tree, a DNS server operates, in some sense, as a node in a P2P network: at times, it acts as a client, requesting information from servers higher in the tree, at times, it acts as a server, returning information to other DNS servers that might require it. During the 1990s, self-organization started to emerge as a standard feature in different systems, such as distributed databases and operating systems, but most of them re-created the idea of a server as a network of self-organizing servers that acted as a (sometimes loosely) coordinated cluster.

To be truly self-organizing, P2P systems must support a set of basic functions: *Join*, to connect to the P2P network, *Insert*, to add keys to the network, *Search*, to find keys, and *Leave*, to disconnect from the network while keeping its structure intact. In the case of failures, the network must also include functions to recover, which are typically built using subsets or modified implementations of the Join and Search algorithms. All of the functions must work—in principle—without dependence on centralized infrastructure, though they will benefit from it, particularly for initial Join operations to the network, where the entry point to the network must be located.

As the power of desktop computers (and, lately, mobile devices) increased, the final stage was reached: true self-organizing P2P, deployed on a global scale, applied to any type of device.

2.7.1 Bootstrapping Self-Organizing P2P

Before going into more detail for each of the types of P2P network, it is useful to note the alternatives available for a crucial element for any self-organizing

P2P algorithm: bootstrapping.

The first step in joining a P2P self-organizing network requires identifying at least one node that is already connected to the network. The challenge with self-organizing networks is that they lack fixed points of entry, and that the network as a whole has no global identifying address and no permanent nodes that can act as gateways. This makes unpredictable the context in which a node joins the network.

In current systems, bootstrapping is usually done by providing a fixed, well-known server that serves a list of some of the P2P peers. If a node has connected to the network once, it can rely on previous information about the network to try to contact nodes that were known, in the past, to exist connected to the network. If a node is connected to multiple P2P networks, a search in one network can be used to find nodes in the other. Similarly, another possibility is to employ connections to random IP addresses in certain address ranges, or to perform IP broadcasts in the local subnet to locate a connection.

We can therefore characterize bootstrapping processes as belonging to one of three generic types. The three types of bootstrapping are:

- *by fixed-point,* typically residing on the Internet. Fixed-points of P2P networks keep track of sets of previously existing nodes (by registering nodes that have already joined the network) and return addresses of nodes in the network on request. These fixed points may or may not be nodes in the network themselves—in many cases they are the equivalent of DNS servers, deployed at multiple points in the network to minimize single-point-of-failure problems.

- *by previously-known network nodes,* in which the incoming node has already connected previously and has stored a list of previous neighbors. The incoming node uses the list to attempt connections to those nodes, that will be in the target network with higher probability than randomly selected nodes.

- *by intersection with another network,* where the incoming node performs a search in another network (to which it is already connected) for nodes that that belong to the target network. The origin network can be another P2P network, or, more commonly, an IP network on which a random search or broadcast request (at class-D subnet level) is performed.

Self-organizing P2P networks use all three bootstrapping methods, typically in sequence, from most accurate (fixed-point bootstrapping) to the least (intersection).

2.8 TTL-based P2P Systems

The most common method for providing P2P locator functions on Internet applications involves a form of Time-To-Live (TTL) controlled flooding mechanism. With this approach, the querying node wraps the query into a single message, then sends that message to all the neighbors it knows about. The neighbors verify if they can reply to the query (by matching the query to the strings stored in its internal database), in which case they send back a reply, or otherwise they forward the query to their own neighbors incrementing the TTL value of the message in the process (if the TTL value is past a certain threshold the message is not forwarded again). The TTL value in effect defines a "horizon" for the query: a boundary that will control its propagation.

Figure 2.6 illustrates an example of a P2P search with TTL-controlled flooding. In the figure, node *Nq* is requesting the value associated with a string located in *Nr* (and for which only *Nr* can provide the value). The query is transmitted across the network while the result returns directly to the node requesting the information (the number of concentric circles indicate the number of hops the message has performed up to that point). In a sense, the network itself resolves the resource mapping requested. Once the lookup

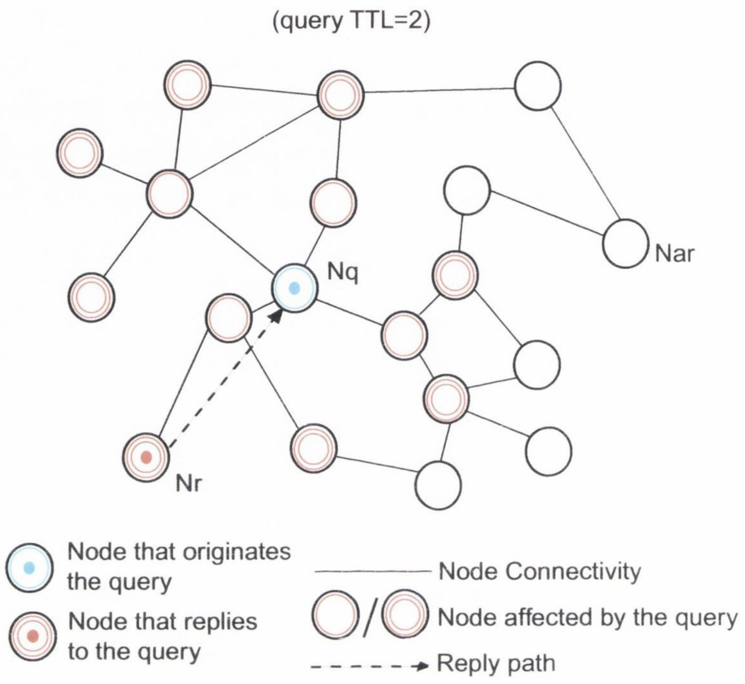
(query TTL=2)
Nq
Nar
Nr
Node that originates the query
Node that replies to the query
Node Connectivity
Node affected by the query
Reply path

Figure 2.6: A Sample TTL-based P2P Network and Query

has been solved, it can be cached by the requesting node so future requests will resolve faster.

The example underscores the problems of flooding-style P2P networks: even though the query can only be answered by Nr, all the nodes within TTL-range (with being 2 in this case) of the query have to process it. Additionally, if the value had been stored in node Nar the result would not have been found unless the TTL of the message was set to a higher value, potentially requiring flooding the entire network.

TTL-Controlled networks are *unstructured* in the sense that nodes attach themselves to the network according to measures unrelated to the content itself, such as join-order, connection speed or even physical proximity, resulting in a randomly created connection topology. This approach makes it simpler to maintain connections but has two problems:

- Since there is no correlation between content location and network topology, search within the network is essentially open-ended, forcing the protocols to use TTL measures to control propagation of the messages and thus avoid flooding the entire network. This results in the possibility that even though content might be available, it might not be reachable by all nodes in the network. In other words, a result cannot be guaranteed *even if the target exists somewhere on the network.*
- Because the network is built randomly, search for a particular element within the "horizon" has a theoretical limit of N steps, where N is the number of nodes within the reach of the query (In practice, different sections of the graph are usually traversed in parallel, reducing lookup times). Strictly speaking, queries on an unstructured P2P network have a maximum number of steps of the order of N, or $O(N)$.

Our initial work on name resolution on ad hoc networks, the *Nom* [18] system, was based on the two most important self organizing systems at the time: Gnutella [33], the first true P2P system to be widely deployed (and still popular today), and Freenet [11], a system that focused on P2P networks as a way to provide anonymity and defeat censorship.

2.8.1 Gnutella

Gnutella is a fully decentralized P2P application layer protocol that is designed to provide file sharing on the Internet, implemented as an open protocol that runs on top of HTTP [25] and that supports host discovery, search, and file transfer. The set of all Gnutella-protocol-enabled applications on the Internet constitutes what is commonly referred to as the Gnutella Network.

Nodes in Gnutella communicate with their peers by receiving, processing, and forwarding standardized messages. The reach of messages within the network is controlled by a Time-To-Live (TTL) field embedded in the message, which is decreased at every step. When the TTL field reaches zero, the message is not forwarded.

To join the network, an incoming node must know about a node already connected to the network (see Section 2.7.1). Once connected to the network, a node participates in it as follows:

- by performing discovery for other nodes,
- by propagating the messages that it receives from its peers,
- by initiating queries,
- by replying to queries,
- by retrieving files (or, more generally, content), and
- by giving access to files requested from it.

Nodes in Gnutella communicate with their peers by receiving, processing, and forwarding messages. Messages can be one of the following types: *Ping* and *Pong* for discovery-requests and replies, respectively, *Query* and *Query-Hit* for queries and replies, and *Push*, used when the client that is publishing the file is behind a firewall or NAT server and thus has to initiate the connection itself (instead of the connection being initiated by the requester). A Gnutella message consists of the following:

- GUID (Globally Unique Identifier), which provides a unique identifier for a message on the network.
- TTL (Time-To-Live), the maximum number of hops that this message is allowed to perform.
- Type, which indicates which type of message is being communicated (e.g., Query, QueryHit, etc.)

- Hops, a count of the hops this message has performed.
- Payload Size (in bytes), the size of the data expected to follow the message.

The procedure to limit the lifetime of a message is simple:

- Before forwarding a message, a node will decrement its TTL field and increment its Hops field[5]. If the TTL field is zero following this action, the message is not forwarded.
- If a node receives a message with the same GUID and Type fields as a message already forwarded, the new message is treated as a duplicate and consequently discarded.

In summary, the node that is initiating a query sends the query message to its neighbors, which in turn forward it until the TTL limit is reached. When processing a query, each host will try to match the query with its local content, and respond with a set of URLs [27] pointing to the corresponding files if there are matches.

The process can be seen in Figure 2.7, where node Nq is requesting a certain key. The *Query* message is propagated until its TTL-limit (of 2 in the example) and nodes $Nr1$ and $Nr2$ reply with a *QueryHit* message. Nq then chooses to request the content only from $Nr1$. $Nr3$, which also contains the key, cannot reply since the TTL of the message does not allow it to reach the node.

Because of its widespread use, Gnutella was also the first self-organizing P2P system whose traffic patterns where systematically studied [74] (along with those of Napster, the first Internet music-sharing application that while

[5]At all times, the TTL and Hops fields must satisfy: $\text{TTL}_0 = \text{TTL}_i + \text{Hops}_i$ where TTL_i and Hops_i are the values of the TTL and Hops fields (respectively) at the ith hop, for $i >= 0$.

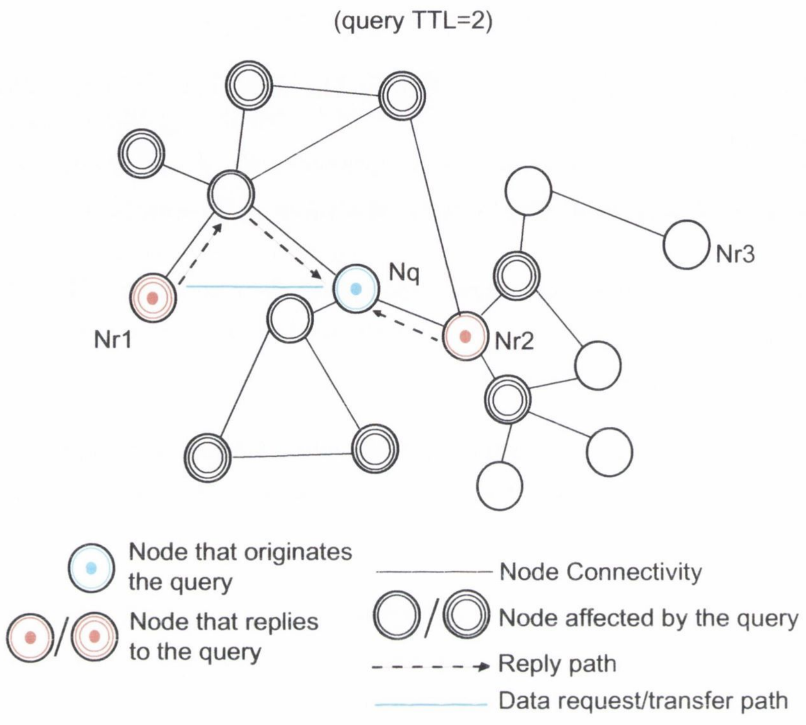
(query TTL=2)
Nq
Nr1
Nr2
Nr3
Node that originates the query
Node that replies to the query
Node Connectivity
Node affected by the query
Reply path
Data request/transfer path

Figure 2.7: A Sample Lookup in a Small Gnutella Network

performing peer-to-peer data transfers, relied on a centralized directory to function).

Pure self-organizing systems sparked a renewed interest in applications to facilitate free speech (or, more accurately, defeat censorship). Freenet was the first of those.

2.8.2 Freenet

Freenet is a distributed information storage and retrieval system designed primarily to:

- address privacy concerns in other P2P systems, and

- guarantee maximum availability of content.

Specifically, Freenet has five main design goals:

- anonymity for both producers and consumers of information
- deniability for producers of information
- resistance to attempts by third parties to deny access to information
- efficient dynamic storage and routing information, and
- decentralization of all network functions.

Freenet's design allows the network to adapt to usage/load patterns, transparently moving, replicating, and deleting files as necessary to provide efficient service without constantly resorting to generalized broadcast search, or using centralized indexes. Because of the way in which it handles replication, however, Freenet is not intended to guarantee permanent storage, since the survivability of the content depends on how often it is accessed.

Users contribute to the network by giving bandwidth and a portion of their hard drive (a local "data store") for storing data. Unlike other peer-to-peer file sharing networks, a Freenet user can't control what is stored in the data store. Instead, files are maintained or removed depending on how often they are accessed, with the least popular being discarded to make way for newer or more popular content. Data is stored encrypted in the data store, to resist local attacks.

In Freenet queries are passed from node to node in a chain of proxy requests, with each node making a routing decision based on the key that is searched. To limit propagation, each query is given a TTL ("hops-to-live" in Freenet terminology), and queries are assigned pseudo-random identifiers to prevent loops (letting nodes reject queries that they have seen before). This process is continued until the query is matched or the TTL is exceeded, in which case the query fails. The query (or failure notice) returns through the same node-to-node path established by the request, therefore guaranteeing local anonymity (i.e., each node only knows the next/previous node in the chain, though access to the complete network traffic would still expose the origin and destination of the information).

2.9 Overlay Networks

As we identified the usage patterns that applied to generic RLD, we concluded that a TTL-based algorithm (Manifold-b) was an appropriate solution to serve queries based on the *local/inexact* search use case for RLD systems. TTL-Controlled systems such as that defined by Manifold-b are ideal for this function: they cover all the nodes in a certain area; and they can easily match a regular expression or substring against a set of strings for which they can respond.

However, the usage patterns pointed to a bigger challenge: providing global/exact location of a resource. The qualities of TTL-controlled algorithms make them well suited to provide local/inexact searches (our first usage pattern). However, global/exact searches require that content is guaranteed to be found whenever the information is available in the network. This implies that all nodes have to be reachable by the content-location service, and that search times are bound by a predictable limit, to avoid having to wait an arbitrary amount of time for a reply. In other words, search in this case has to be *deterministic.* This must be achieved with zero-dependency on centralized services, self-organization and, consequently, automatic load balancing to avoid overloading any particular network node.

Based on these requirements, we designed a second algorithm, Manifold-g. Manifold-g was based on the idea that the values of the names stored on each node could be used to create a virtual topology with a specific structure (derived mathematically from those values). Since the structure was predictable, it could also be navigated predictably, allowing both guaranteed results and time limits on the search process. As we have mentioned above, after the work on Manifold-g had been completed we related it to a category of self-organizing network algorithms that has been under development in recent years: Overlay Networks.

Overlay networks [19] involve a level of network semantics above that of basic routing that create a structure that can be navigated predictably; these extended semantics are achieved by organizing the overlay topology based on some of the content exposed by the nodes rather than using their more immediately available physical organization, thus creating a virtual topology on top of the physical topology[6]. These networks in effect create special purpose routing abstractions to optimize the process of searching for particular data items by performing distributed lookup.

In essence, overlay networks ask the question "what is the search space?" and instead of answering "a graph," they consider the answer to be "*a set of strings*". Overlay network algorithms then use the values of the *bit-sequences* defined by the strings. The strings are generally transformed using consistent one-way hash functions similar to those presented by Karger et. al. on their work on consistent hashing and random trees [39] (which improved on systems such as that defined for a Distributed Dynamic Hashing Algorithm [17]), reducing the size of the string space to be manipulated with the additional, important side-effect of providing good load balancing by evenly distributing the values that exist in the network.

Overlays were proposed as a way of creating topologies according to *content* rather than other parameters (node-dependent or arbitrary) of the nodes. By creating a structure based on content, overlays turn the problem of search from a standard graph-traversal problem into a set of steps that can be calculated according to a mathematical function, reducing the

[6]In terms of creating a virtual topology on top of the physical topology, TTL-based P2P networks are also a type of Overlay. However, we will use the term "Overlay Network" only in relation to networks that create virtual topologies based on node content attributes.

overall load on the network and making the query process deterministic. In abstract terms, an overlay network operates like a hashtable by allowing insertion, querying, and removal of strings. Those strings are derived in some way from the content exposed by the nodes, for example, by using a consistent hashing algorithm. In fact, overlay network algorithms are also referred to as *distributed hashtables* or DHTs—throughout the rest of this work, we will prefer the more generic term Overlay Network to refer to these types of content-based networks.

Overlay network systems include Chord [81], CAN [66], Viceroy [48], Pastry [72], and Kademlia [53]. We will now briefly consider some of their generic qualities, and then look more specifically at how these systems implement content-based networks.

Overlay networks have the following common qualities:

- Guaranteed results. If the data item exists in the network, it will be found regardless of its location.
- Provable bounds on lookup times (typically of $O(\log n)$ with n the number of nodes in the network),
- automatic load balancing, and
- self organization.

Figure 2.8 shows a hypothetical overlay network and a query propagating through it. While the algorithm used to construct the overlay varies according to the network type the end result is generally similar to the one shown in the figure in terms of topology: symmetrical in nature and fairly consistent on a node-per-node basis.

Since nodes in the overlay are connected according to a consistent mathematical function applied to the content stored, a query for a particular key can be routed directly to the node that is storing the desired *key, value* pair, resulting in the minimal query path shown in the diagram.

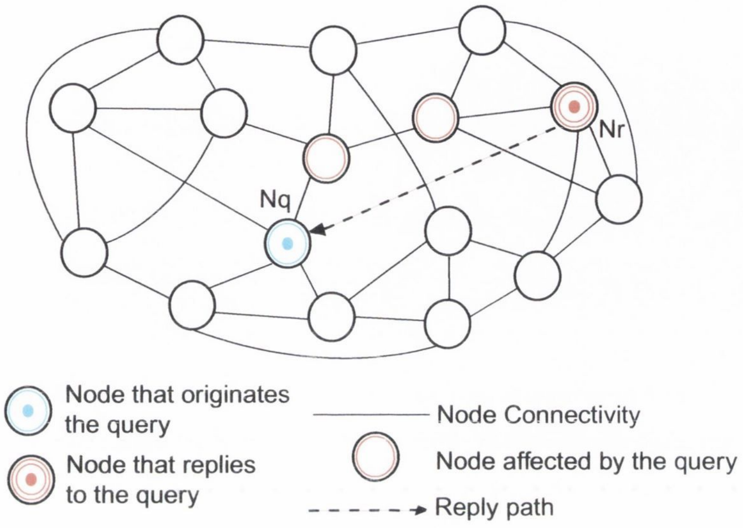
Nq
Nr
Node that originates the query
Node that replies to the query
Node Connectivity
Node affected by the query
Reply path

Figure 2.8: A Sample Overlay Network and Query

Overlay networks can be understood as one-to-many mapping functions of a set (that contains all the possible bit strings of a certain size) onto itself while preserving its relations and operations [7]. The resulting subset of results of the mapping function defines the appropriate "neighbors" for that string (and therefore for the node that stores that string [8]). The original set of strings then becomes an algebra (since it will include both the values and a group of operations that satisfy the conditions for an algebra) that maps the set of strings onto itself.

The set of all possible strings is clearly bigger than the set of any real world network. To solve this problem, overlay networks always define, implicitly or explicitly, a way to "complete" the space, so, as far as the mapping function is concerned, the space will be complete, or they establish procedures to jump across sections in the space in ways that maintain the characteristics of the

[7]in mathematical terms, and *endomorphism*

[8]We are assuming each node exposes a single string. The other case, where multiple strings are stored in a single node (whether they belong to that node or not) is a specific application of the generalized case.

search process.

Any overlay network algorithm contains, at its core, at least one such mapping function that, given a string to be found (and knowledge of the current location on the network) returns the next step in the path to that result string. Thus, the *theoretical path* between two nodes is fixed by that mapping function; the only thing that changes is how this path is mapped using the nodes/values present on the network, which makes the search through the overlay deterministic (bounded).

The mapping function is at the core of the algorithm, and it is used not only to calculate neighbors to connect to the network, but also to as part of the locator function that is at the core of every P2P algorithm. To connect to the network, a node requires only to know the address of a node already in the network, obtained through bootstrapping (see Section 2.7.1. Once connected to the network, any node can calculate the path from itself to a target node. This calculation is a completely abstract exercise: a node's neighbors can be calculated without any information other than the string value itself, and the only step necessary is to obtain the actual network locations of each node along the way (to account for missing strings in the actual mapping of the theoretical topology).

In other words, the theoretical path between two nodes is fixed by the mapping function, and the only thing that changes is how this path is mapped using the nodes/values present on the network.

The mapping function has an additional condition to fulfill. If a given neighbor i of a node [9] N is M, then the neighbor i of node M should be N:

$$N_i = M \qquad \Longleftrightarrow \qquad M_i = N$$

In mathematical terms, the mapping functions used in overlay networks

[9]Since a node has several neighbors, the mapping function maps a single string onto a subset of the original set of strings. Neighbor i then represents value i in the subset of results provided by that function

satisfy the conditions for a *metric* in the algebra defined by the set of strings plus the operations that allow their manipulations. Therefore, all overlay networks define (implicitly or explicitly) a *metric space* on the string set considered. The fact that overlay network functions are metrics on the set of strings is not a coincidence: it is a necessary condition for the overlay network structure to be valid. (The mapping function has an additional requirement, that of symmetry).

As an example, let's consider a network where nodes want to "publish" a given name (for example, the nodes in a distributed database system). In this example the content "exposed" by a node is also its identifier, which we define to be a positive integer value (this will allow us to skip the step of having to hash the identifier/content, which would be necessary if we were using strings). If the identifiers are *universally unique* then we can, for illustrative purposes only, define a few simple rules to create an overlay topology:

- The "neighbors" for each node (in the overlay) will be two: the node whose value is the next available (higher) integer, and the node whose value is the previous available (lower) integer.
- At the limits (if the current node is either the lowest identifier in the network, or the highest), the neighbor will be the opposite maximum in the range of nodes available.
- To join the network, a node needs to locate another node already existing in the network as described in Section 2.7.1. Then, the incoming node can use the *search* function to find the "slot" in the network where it should insert itself by locating its two neighbors. If one (or both) of the neighbors is not present, the incoming node will "cover" for the missing node(s) by assuming their responsibilities until the node(s) arrive. If they are present the incoming node simply connects to them while taking over functions from the node(s) that were "covering" in the incoming node's absence.

- The first node to join the network will "cover" for its neighbors itself.

These rules define a *linear* overlay topology, through the use of a linear mapping function. Searching in this topology follows a simple algorithm: the search can start at any node, and the node will determine the relation between its own value and the "target" value: if the target value is higher than its own, it will pass on the request to its neighbor of higher content value, and the process will continue making decisions locally until it reaches the destination node, which can then reply directly to the requester with its physical network address for additional operations.

The example is unrealistic in that the search time will be bound, but linear, making lookup times unacceptable. Typical Overlay Network algorithms use more complex rules to organize the nodes (resulting in logarithmic lookup upper bounds), as we'll see in the sections that follow.

2.9.1 Chord

Chord operates like a distributed hashtable by allowing insertion, querying, and removal of strings on a virtual data structure maintained in a set of participating nodes. The strings used as keys are derived in some way from the content exposed by the nodes by using a consistent hashing algorithm, so everything in Chord centers around a distributed hash lookup primitive. Chord can find data using only $\log(N)$ messages, where N is the number of nodes in the system, and its lookup mechanism is provably robust in the face of frequent node failures and re-joins.

As shown in Figure 2.9, Chord defines a basic "ring" topology; its basic algorithm implies one connection per node and is thus inherently resilient to node joins, leaves or failures. Each nodes store key/value mappings according to the values of the keys and the identifiers given to the nodes, redistributing key/value mappings as new nodes come into the network. For the example in Figure 2.9, which shows a network of eight nodes with 4 keys, if a node

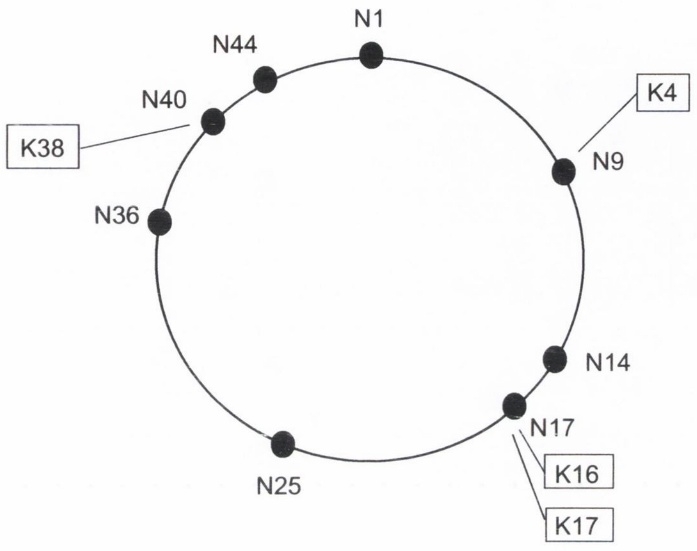

Figure 2.9: The Basic Chord Ring Topology

with identifier 16 were to enter the network, key 16 would migrate from node 17 to the incoming node.

Chord then extends the basic successor node with a set of *fingers* according to powers of two, so for a value n the neighbors will be those nodes that match a rule that evolves according to $n + 2^0, n + 2^1, n + 2^2 ... n + 2^{m-1}$, with m the number of bits in the identifier[10], as shown in Figure 2.10.

In this case, several of the fingers that N9 would connect to are not present in the network. Therefore the responsibilities for the missing nodes are assumed by the next node that is present in the topology. As new nodes come in, they would take control of the connections that they are responsible for, just as they will take control of the keys that "belong" in their section of the space. Chord stores multiple key/value pairs in each node, automatically

[10]m is the number of bits in the identifier that can be assigned to each node, and thus defines the maximum number of nodes supported to be 2^m. Formally speaking, the fingers for a node in Chord are defined as each of the successors in the ring for node $(n + 2^{k-1}) mod 2^m \quad 1 < k < m$

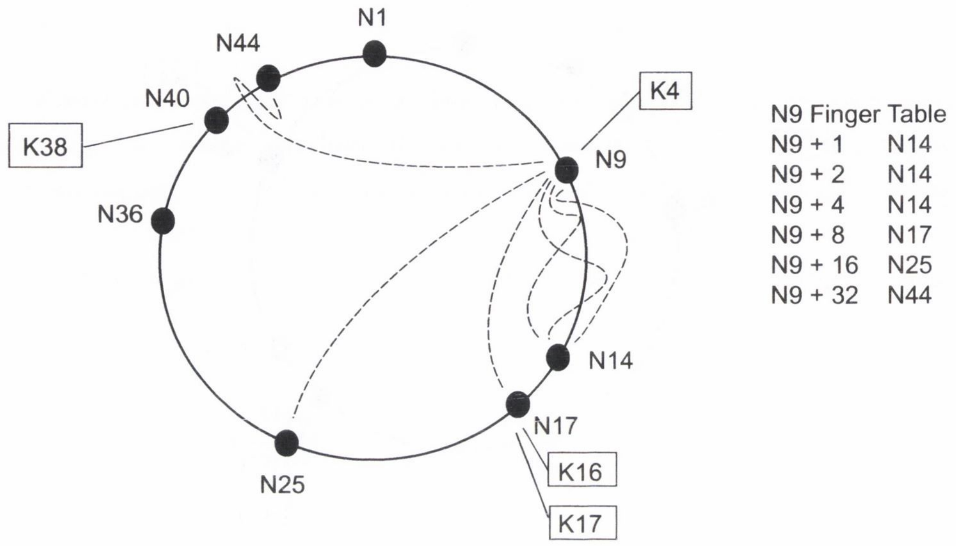

Figure 2.10: Chord Fingers for One Node

balancing the load of the nodes as new entrants to the network arrive.

When the fingers are used, the lookup time becomes of the order of the logarithm of the number of nodes, $O(\log N)$. Note that, to maintain correctness, Chord only requires that the *successor* node pointer be correct, and it can revert to use the basic scheme when, at any given step, the fingers table has been damaged (e.g., by node failures). Since the fingers table is small on each node, it can be maintained valid (along with the successor pointer) through a periodic "stabilization" algorithm. In Chord, lookup times of $O(\log N)$ are maintained even when faced with probability of node failure of 1/2, as long as the initial network is stable.

Figure 2.11 shows a sample Chord lookup operation, in which N9 requests the value associated with K38. N9 first finds the finger connection that gets it closer to the target (N25). The process is then repeated at each node, moving closer to the target based only on information local to each node. This sample lookup shows the maximum number of steps for a network of 8

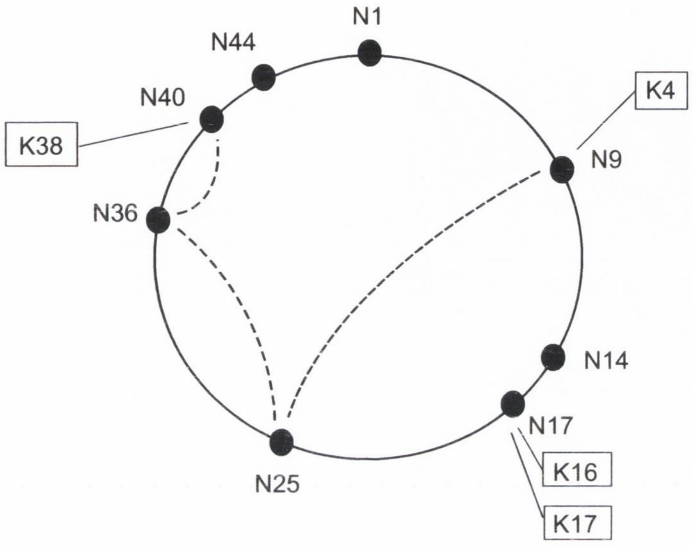

Figure 2.11: A Sample Chord Lookup

nodes, which is $\log 8 = 3$.

Chord has been used as the basis of several self-organizing systems and services, such as CFS [14] and i3 [80].

2.9.2 CAN

CAN uses a d-dimensional Cartesian coordinate space to implement the distributed hashtable abstraction. The coordinate space is partitioned into sections of dimension $d-1$ called *zones*. Each node in the system is responsible for a zone and identified by its boundaries. Keys stored in the system are mapped onto a point in the coordinate space, and stored at the node whose zone contains the point's coordinates. Each node contains a table of all its neighbors, defined as the nodes whose zones share a $d-1$ dimensional hyperplane. The lookup operation in CAN is implemented by forwarding a query message along the path that approximates a straight line in the coordinate space from the querying node to the node storing the target key.

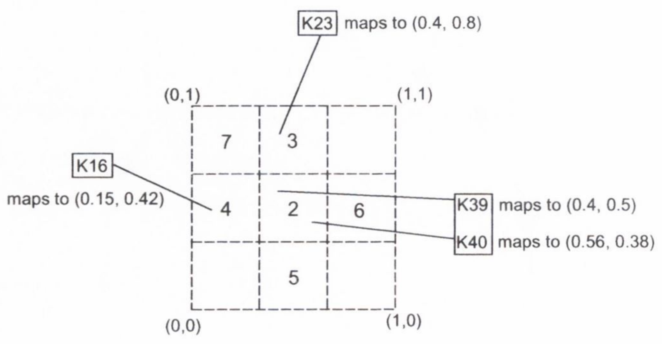

Figure 2.12: A Two Dimensional CAN

As an example, consider the small CAN network shown in Figure 2.12, organized is organized on a two dimensional space. In the example, connection points are defined along edges of dimension one. Each node is therefore connected to four other nodes at most (and one at a minimum). In the figure, we can also see keys assigned to different nodes based on their calculated coordinates (derived through a hash of the key). The node responsible for holding a (set) of given key(s) is the one that is covering the coordinate space to which the key belongs.

To join the network, a new node first chooses a random point in the coordinate space, and asks a node already in the network to find the node n whose zone contains that point. Node n splits its zone in two and assigns one of the halves to the new node. The new node can easily initialize its routing table, since all its neighbors, except n itself, are among n's neighbors.

Figure 2.13 shows the same network after a new node (node 8) has entered the network, partitioning the space that previously was covered by node 2 alone.

Once the new node has joined, the new node announces itself to its neighbors. This allows the neighbors to update their routing tables with the new node, as well as re-assigning keys that correspond to the space covered by the new node (In Figure 2.13, node 8 has taken over key K40 from node 2).

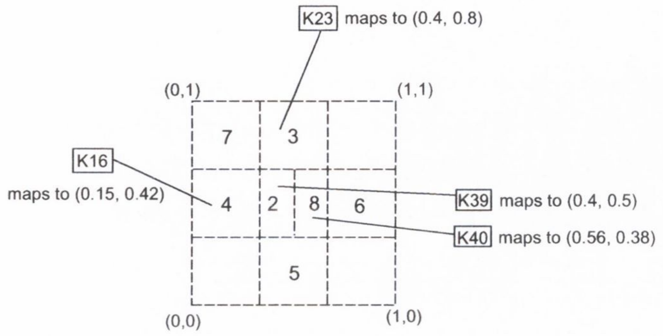

Figure 2.13: A Two Dimensional CAN After a Node Join

When a node departs, it hands its zone to one of its neighbors. If merging the two zones creates a new valid zone, the two zones are combined into a larger zone. If not, the neighbor node will temporarily handle both zones. To handle node failures, CAN allows the neighbor of a failed node with the smallest zone to take over. One potential problem is that multiple failures will result in the fragmentation of the coordinate space, with some nodes handling a large number of zones. To address this problem, CAN runs a node-reassignment algorithm in the background. This algorithm tries to assign zones that can be merged into a valid zone to the same node, and then combine them.

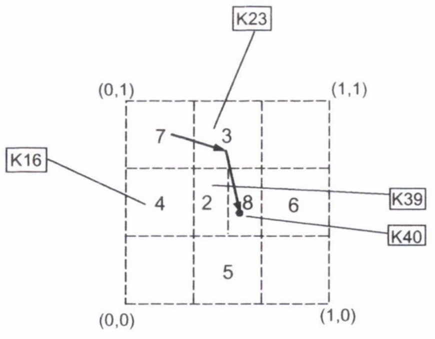

Figure 2.14: A Sample Lookup in a Two Dimensional CAN

Lookups in CAN proceed (as mentioned above) essentially by approximating a straight line between the requesting node and the node that is responsible for the target area in which the key resides. Figure 2.14 is an example of this process. To lookup K40, node 7 calculates the expected coordinates of the key in the space. With that information, node 7 routes the request to the neighbor closest to the target section of the space, in this case node 3, which repeats the process, reaching node 8, creating the search path shown in the figure with a solid line. Node 8 then returns the value associated with K40 to node 7 and the search cycle is completed.

2.9.3 Oceanstore and Tapestry

In the mid-90s, Plaxton et. al. proposed [65] a randomized hierarchical distributed data structure that gave rise to systems such as Tapestry [36] and Oceanstore [42]. This data structure yields routing locality with balanced storage and computational load, but does not provide dynamic maintenance of membership. That is, the Plaxton algorithm requires a static set of participating nodes as well as significant work to pre-process this set to generate a routing infrastructure, which complicates coping with node failures.

Tapestry assigns unique identifiers to nodes and objects in the distributed system, uniformly distributed in the namespace. Each tapestry node contains pointers to other nodes as well as mapping between object identifiers and the Node-identifiers of storage servers. Queries are routed from node to node along neighbor links until an appropriate object pointer is discovered, at which point the query is forwarded along neighbor links to the destination node.

OceanStore is a "utility infrastructure" designed to provide continuous access to persistent information in a global scale whose routing mechanism is another variation on Plaxton et. al's data structure. Since this infrastructure is comprised of untrusted servers, data is protected through redundancy and cryptographic techniques. To improve performance, data is allowed to be cached at any node in the network; monitoring of usage patterns allows adaptation to regional outages and denial of service attacks, and it can enhance performance through pro-active movement of data to more active locations.

Entities in OceanStore are free to reside on any of the OceanStore servers. This freedom provides maximum flexibility in selecting policies for replication, caching, and migration. Addressable entities[11] in OceanStore are identified by one or more GUIDs (Entities that are functionally equivalent, such as different replicas for the same object, are identified by the same GUID). Clients interact with these entities with a series of protocol messages. The role of the OceanStore routing layer is to route messages directly to the closest node that matches the query. In order to support this routing process, OceanStore creates a distributed, fault-tolerant data structure that explicitly tracks the location of all objects. Routing is thus a two-phase process. Messages begin by routing from node to node along the distributed data structure until a destination is discovered. At that point, they route directly to the destination. Thus, the OceanStore routing layer does not replace IP routing, but is built on top of it as an overlay.

2.9.4 Other Systems

HyperCast

In the late 90s, HyperCast [46] [47] was put forward as a protocol based on an overlay that used a hypercube topology for efficiently performing multicast on a network, and it has more recently been used for peer-to-peer networking of very large groups. HyperCast embeds spanning trees into incomplete hypercubes through the use of an algorithm that uses a Gray Code. More recently, HyperCast has been used to perform point-to-point communications between nodes.

In HyperCast, all data is transmitted along trees that are embedded in

[11]An entity in OceanStore can be a replica, an archival fragment, or a client

the overlay network topology. For each node in the network, there is an embedded spanning tree in the overlay network with that member as the root of the tree. Given the root of an embedded tree, any node can locally determine its children and parent with respect to that tree. Each member forwards data to its children or parent in an embedded tree with respect to a specific node. The embedding of trees (and other data structures) in abstract topologies such as hypercubes as butterflies was also studied in relation to massively parallel machines, for example, in [43].

Viceroy

A Viceroy network is similar to a butterfly network [41] combined with a set of predecessor and successor links like those defined by Chord. In addition to predecessor and successor links, each server includes five outgoing links to chosen "long range" contacts. First, each node chooses a "level" at random in such a way that when n servers are operational, one of $\log n$ levels is selected with nearly equal probability. Edges connecting a node at level l to other nodes are selected according to the following steps:

- a "down-right" edge is added to a long-range contact at level $l + 1$ and distance roughly $1/2^l$ away
- a "down-left" edge at a close distance on the ring to level $l + 1$.
- An "up" edge to a nearby node at level $l - 1$ is included if $l > 1$.
- "level-ring" links are added to the next and previous nodes of the same level l.

Routing proceeds in three stages: the first one consists of a "climb" using connections to a level-1 node. In the second stage, routing proceeds down the levels of the tree using the down links; moving from level l to level $l + 1$ one follows either the edge to the close-by down link or the far-away down link, depending on whether the target v is at a distance greater than $1/2^l$ or not. This continues until a node is reached with no down links, which presumably is in the vicinity of the target, at which point a “vicinity” search is performed using the level-ring links until the target is reached. This process requires $O(\log N)$ steps with high probability for randomly built networks.

Pastry

Pastry gives each node a randomly chosen identifier, indicating its position on an identifier circle. It routes messages requesting (or inserting) a given key to the node with identifier that is numerically closest to the key, using 128-bit identifiers in base 2^p where p is an algorithm parameter typically set to 4. Each node maintains a leaf set, composed of nodes closest to its own identifier and larger than it, and those closest to its own identifier and smaller than it. These heuristics allow Pastry route queries according to a network-proximity metric. Each node is likely to forward a query to the nearest one of k possible nodes, using a neighborhood set of other nearby nodes. As long as a failure doesn't involve an entire half of the leaf set, correctness in the algorithm is guaranteed.

To optimize forwarding performance, Pastry maintains a routing table of pointers to other nodes spread in the identifier space. This can be viewed as $\log_2 pN$ rows, each with $2p-1$ entries each (where N is the number of nodes in the network). Each entry in row i of the table at node n points to a node whose identifier shares the first i digits with node n, and whose $i+1$st digit is different (there are at most $2p-1$ such possibilities). Given the leaf set and the routing table, each node n implements the forwarding step as follows:

- If the key that is being looked up is covered by n's leaf set, then the query is forwarded to that node.
- In general this will not be the case until the query reaches a point close to the key's identifier. In this case, the request is forwarded to a node

from the routing table that has a longer shared prefix (than n) with the sought key.

- If the entry for the target node is missing from the routing table because the node doesn't exist, or because that node is unreachable, n forwards the query to a node whose shared prefix with the key is at least as long as n's shared prefix with the key, and whose identifier is numerically closer to the key. Such a node must clearly be in n's leaf set unless the query has already arrived at the node with numerically closest identifier to the key, or at its immediate neighbor.

If the routing tables and leaf sets are correct, the expected number of hops taken by Pastry to route a key to the correct node is at most $\log pN$. Pastry has a join protocol that builds the routing tables and leaf sets by obtaining information from nodes along the path from the bootstrapping node and the node closest in identifier space to the new node. It may be simplified by maintaining the correctness of the leaf set for the new node, and building the routing tables in the background. This approach is used in Pastry when a node leaves; only the leaf sets of nodes are immediately updated, and routing-table information is corrected only on demand when a node tries to reach a nonexistent one and detects that it is unavailable.

Kademlia

Kademlia uses an XOR-metric to dynamically route messages/keys to the node with the identifier numerically closest to the key, similar to Pastry. Each Kademlia node has a 160-bit node identifier (Node identifiers are constructed as in Chord). Every message transmitted by a node includes its identifier, thus allowing other nodes to record the originator node's existence if necessary. Keys stored in Kademlia are also 160-bit identifiers. For finding and publishing keys, Kademlia relies on the distance between two identifiers, defined as their bitwise exclusive or (XOR) interpreted as an integer,

$d(x, y) = x \oplus y$.

Kademlia nodes store contact information about each other to route messages. Every node keeps a list of (IP address, Port, Node identifier) triples for nodes of distance between 2^i and 2^{i+1} and itself. These lists are maintained through a least-recently seen eviction policy, and live nodes are never removed from the list, thus maximizing the probability that nodes contained in the list are valid.

The system differentiates between finding nodes and values stored in them; one node might contain one or more values. When a node is being looked up, the search procedure in a node returns information for a number of nodes it knows about that are closest to the target identifier, which are then queried for the result (the procedure stops when a value is found). In this sense, Kademlia performs a search across a likely set of neighbor nodes of a target value which will typically result in search times of $O(\log N)$, where N is the number of nodes.

To find a key/value pair, a node starts by performing a lookup to find the k nodes with identifiers closest to the key, and the procedure halts immediately when a node returns the value (this response might come from a holding node, or from a node that is actually caching the result). Once the node is found, a similar recursive procedure for the key is performed in it.

Early Applications of Virtual Topologies

While searching the literature for references on work similar to Manifold-g, we found similarities beyond those of Overlay Networks, first on the work done on parallel computing architectures and on predictable (i.e., mathematically derived) virtual topologies in different areas. In all cases studying these systems helped us understand the qualities and properties of Manifold-g.

In the late 70s and during the 80s, new types of massively parallel computer systems were developed that used the concept of a network organized according to some well-known parameter to create a topology that could be navigated predictably. The network, in that case, was one of processors that would divide a particular task in several sub-components, achieving faster processing times (for example, Tree Machines [3], the Cosmic Cube [76], the Connection Machine [44], the nCube [60] and Butterfly Networks [77]). Processor networks were static in their topology, and were not self-organizing (although they had capabilities to route around failures, see, for example, [35]), but many of their routing concepts are similar to those used in content-based overlay networks today. Even at that time, solutions that went beyond processing that used virtual topologies for other purposes were proposed, such as [32].

Similar alternatives have been explored in the realm of physical network transports. One of the best examples of these is the the Manhattan Street Network [50], where nodes are connected in a two dimensional grid (mesh) with alternating rows and columns, where the wraparound links between nodes place the resulting two dimensional grid on the surface of a logical torus. A node is represented by a simple 2x2 switch and, at the beginning of each time-slice, it switches slots from its two incoming links to its two outgoing links. The regular topology of a Manhattan Street Network makes it well suited for self-routing algorithms such as those described in [6] [51] [52] [69] .

2.10 Comparison of P2P Systems

The central difference between TTL-controlled algorithms and overlay algorithms is that overlays guarantee that a result will be found if present in the network. Overlays also guarantee that the lookup time (either with a result or a failure) will be bounded; TTL-controlled algorithms can make no such guarantee.

Compared to TTL-controlled algorithms, overlay algorithms require a much smaller number of steps to reach the desired node. However, this is done at the cost of a higher number of *physical hops* the message has to perform on the network. That is, for a given hop between nodes in an overlay network the physical distance covered might usually be more than physical hops between nodes for a flooding-based network. Overlay networks make up for this deficiency by significantly reducing the total number of hops at the overlay level, by operating as a highly optimized point-to-point communication mechanism, rather than using the "brute force" approach of flooding-based networks.

Note that the connectivity pattern in an overlay network is different from the one obtained using a TTL-based algorithm: the number of connections to and from each node is constant[12]. This is an important feature because the structure created by the overlay has to be, either implicitly or explicitly, *symmetrical*, and it is the structural symmetry of the topology that allows nodes to forward queries optimally from any node with the certainty that a path to the target will always be found.

The symmetry of the network (either implicit or explicit) contributes to the short-path lookup times of overlays. Nodes use the lookup function to define which content they are responsible for, therefore ensuring the integrity of the structure.

Overlays structure their content based on the exact values of the keys stored, making them unsuitable for inexact (or substring) searches. TTL-controlled algorithms can deal properly with both exact and inexact queries alike.

In case of node failures, overlay network algorithms provide mechanisms for the network to recover and re-create or maintain an appropriate network structure. TTL-Controlled networks, on the other hand, have lower consistency requirements, allowing them to operate even if only one connection to one node in the network is available and making them more resilient to

[12]Some overlay networks might obtain this result implicitly (i.e., not necessarily through direct connections between nodes, but by adding traversal rules to the space)

large-scale, disruptive changes in network topology.

We will now look at how these specific implementations of these two types of P2P algorithms can be combined to provide a self-organizing solution to the problem of generic RLD.

Chapter 3

Manifold, an Overview

3.1 Introduction

In Section 2.3 we noted that a user will typically have two different requirements for resource location: *local/inexact* search, and *global/exact* search. We will now briefly discuss the *Manifold* algorithms, how they differ from the algorithms discussed in the previous chapter, and their interaction and design, to provide context to the following chapters, in which each algorithm is described, and analyzed, in detail.

Manifold uses a hybrid, dual-algorithm system. One algorithm, *Manifold-b*, handles inexact queries; the other, *Manifold-g* deals with exact queries in a global scale. The first algorithm is based on a TTL-Controlled P2P algorithm, and the second is based on an overlay network algorithm. Manifold doesn't force the application to make a choice of which algorithm to use; it receives an inexact query in the form of a regular expression [16] [40] [82] (or, alternatively, with a string and an additional parameter specifying that the string is actually a substring to be matched, which creates the regular expression on the fly). If the string to be matched is a regular expression, Manifold-b is used. If not, Manifold-g is used.

3.2 Resolving Inexact Queries: Manifold-b

Manifold-b uses TTL-Controlled P2P to serve queries based on the *local/inexact* search use case for RLD systems. TTL-Controlled systems are ideal for this function: they cover all the nodes in a certain area, they have proven scalability up to tens of thousands of nodes; and they can easily match a regular expression or substring against a set of strings for which they are responsible.

The Manifold-b algorithm is, at its core, relatively simple, and similar to the algorithm found in well-known P2P systems, Gnutella in particular. Differences for these types of algorithms appear in specific implementation details, such as caching, or usage of underlying network features, such as increased use of nodes that have access to fast network connections (e.g., applying concepts described in [30]), which are also implementation dependent.

3.3 Exact Search in a Global Scale: Manifold-g

For the second use case, of *global/exact* search, Manifold uses an overlay network to guarantee results in bounded time if the target value exists on the network, something not possible with TTL-limited P2P networks, in which the diameter of the search is usually less than the diameter of the network itself.

Manifold's overlay network algorithm was designed with the requirements of resource location in mind, in particular with support for content-locality. In current overlay networks content can be said to be *non-local*: published content may or may not reside on the node that publishes it [1] (a consequence of their design goal of achieving good load balancing) through some kind of

[1]It should be noted some systems (CAN and Chord in particular) could be forced to maintain locality, or, in other words, a strict correspondence of a single content value to its originating node. However using them in this fashion could create unnecessary strain in the systems; as we have already seen both CAN and Chord are designed primarily as distributed self-organizing hashtables that maintain several key/value pairs per node, and pairs might require redistribution among nodes after a certain number of insertions.

mapping of contents to nodes. Additionally, they all rely on an implicit or explicit mechanism to obtain bounded search times: in some cases (as in the case of CAN) the content mapping function has certain built-in qualities that guarantee a given search speed, in others (such as in Chord) the content is stored according to a hash function but then an additional mechanism (Chord's "fingers") is used to improve the search efficiency. In all cases, the search time can be improved by modifying certain parameters within the network, such as number of neighbors per node.

This *non-locality* of content, while useful for various reasons in contexts such as distributed network storage [14] [42], is not a desirable feature in other contexts, most notably name resolution in particular and RLD in general, where it is important that the node that serves the content is the same that publishes it.

Additionally, current designs place little emphasis on minimizing resource usage on the nodes while increasing performance, since they generally require a node in the network to store multiple keys. The common solution to increasing performance usually results in an increase of the number of connections (or initial lookups, to maintain state on neighbors) a node has to perform. While this is acceptable for PCs, it might be more of a problem for mobile devices or devices with slow or low-quality wireless connections, or with limited processing power.

The problem of *global/exact* search in RLD has certain specific requirements beyond RLD in general:

- *Content locality.* As mentioned before, name resolution ideally requires a one-to-one mapping between content and the node that publishes it. For example, if *CompanyX* is publishing its name/address mapping, it is safe to assume that *CompanyX* would like a certain degree of control over response times; for example exposing the content through a cluster of servers rather than through a single machine. If the content is not local to the node that publishes it, the name for *CompanyX* might be served by *CompanyY*'s servers or machines, which might be overloaded or might be *CompanyX*'s competitor, therefore making it an unacceptable choice for *CompanyX* for political, rather than technical reasons. For an in-depth analysis of the political, social, and technical issues raised by name resolution in a global scale see [58].

- *Low resource usage.* A self-organizing network has the potential to provide global name resolution even between client devices, without any need for server intervention. Consider the following scenario: *UserX* is travelling with a wireless device, and she connects to the Internet, therefore "publishing" her new address to the network, so other users can find her address and contact her directly to send a file. Current solutions to this problem would involve specific server configuration of systems like DHCP at potentially large cost, while using an overlay network would provide this functionality as part of the same system used to look up server (rather than client) addresses. However, small devices are usually severely limited in connection speed, memory, storage, and/or processing power. In this case, the ability to off-load management of a node's responsibilities to a proxy is an important element.
- *Speed with extremely large scalability requirements.* Name resolution is an application that, if deployed in today's systems, would include hundreds of millions of devices, with this number reaching the billions of devices in the near future.

A global, self-organizing RLD system requires locality of content, extreme scalability, and the ability to provide speed/resource usage tradeoffs for deployment in devices of different connection and performance capabilities. While many other applications that use overlay networks might not have these specific requirements, any of them would benefit from a solution to the problems they present.

3.4 Design

We will go into more detail on the design of a real-world implementation of the Manifold system in Chapter 7, but it's useful to introduce some of the design concepts at this point, to clarify the basis of the discussion for this chapter as well as the next part of this work where we discuss the theoretical basis of the algorithms that compose Manifold.

In keeping with the end-to-end principle [73], Manifold does not require modifications on the underlying network layer addressing and routing; this allows Manifold operate on different platforms, network types, and transport layers. Manifold achieves this level of abstraction through the use of *messages*. Messages can be received locally[2] or through the network interface. Once a message is received there is one component that chooses which algorithm is responsible for that message, and passes it on. The query process is itself stateless, that is, both algorithms operate by receiving a message (which carry their own state, such as nodes visited in the query path, etc), processing it, and continue appropriately, either replying to the query, forwarding the message, or ignoring it. The only state maintained is query-independent, having to do with the structure of the P2P networks used.

The core of Manifold is controlled by two independent algorithms that process messages received from an internal message manager. Through the manager, they have the ability to send messages to the local (requester) application as well, or use the network interface to send messages to other Manifold nodes (specifying which algorithm on the target node will be responsible for processing the received message).

[2]through Inter-Process Communication for example, or function calls, or local message passing within a single process.

3.4.1 Algorithm Use and Interaction

The two algorithms coexist within the same environment, and the algorithm to be used is chosen based on the type of query requested by the user. If the query involves a substring search (e.g., all the names that begin with the term "printer") then the system automatically uses the Controlled Flooding algorithm. If the query is exact (e.g. a specific name like "printer01") the query will be routed using the overlay algorithm.

The two algorithms don't require interaction; they essentially create two parallel networks to resolve the different types of query: the Manifold-b TTL-based network is limited in reach (from the point of view of each node); the Manifold-g network is global in nature. That is, while every node belongs to the same global overlay, each node might belong to different Manifold-b networks of limited reach.

While it's not *required* that the algorithms interact, there are some cases where interaction is important in providing *best-effort resolution*. For example, if a small section of the overlay has been suddenly been cut off from the global network (e.g., if its Internet access point failed) then reverting to use the local Manifold-b network automatically would be a useful feature; since it's possible that the target node is actually local. If not, a failure will be reported, but switching to the Manifold-b network has the potential to provide a response even in the face of catastrophic failures for the overlay.

Chapter 4

Manifold-b

4.1 Introduction

As we have mentioned earlier, our initial work on RLD was *Nom* [18], a system that was applied to mobile ad hoc networks exclusively. This initial work led us to explore the generic RLD problem in more detail. As we further identified the requirements of generic RLD, the *Nom* algorithm emerged as an appropriate solution for the first half of generic RLD, and it was used as a basis for the Manifold-b algorithm, presented in this chapter.

4.2 The Algorithm

Manifold-b[1] is a TTL-based self-organizing algorithm. At the core of Manifold-b is a loop that monitors messages coming both from the network and the application level code (i.e., *query*/*query-reply* messages from the network and *query-initiate* messages from the application) and reacts according to the message type received, either forwarding the message received if it doesn't apply to the current node, creating and forwarding an appropriate *query-reply* message if the node should respond to the query, or creating a *query*

[1] "Manifold-broadcast"

message and inserting it into the network.

We will now consider the basic functions of the algorithm, as mentioned at the beginning of Chapter 2.6: *Join*, *Leave*, and *Search* as well as Key insertion and removal.

4.2.1 Node Join

The Join process in Manifold-b starts with the P2P bootstrapping rules discussed in Section 2.7.1. Once a node has found one or more nodes in the network, it will request connections to them using a special type of message, *connection-request*, which will be either accepted or denied. Additionally, once connected, a node can expand its reach (ie., expand its neighbor list) by sending out *query-bind* messages into its known peers (see Section 4.2.3).

4.2.2 Node Leave

A node will, if possible, inform its neighbors that it is disengaging from the network. This is done by sending a *leave-notify* message to its peers. The *leave-notify* message is one-way; no replies are required or expected. If a target node is not reachable for some reason, then no further steps are attempted for notification. In some scenarios, it is possible that the target might not be reachable only temporarily (e.g., in a wireless network, if the target has moved temporarily out of range). In that case, the next connection the target attempts with the leaving node will fail, and the rules for dealing with node failures will be followed (see next section).

4.2.3 Key Search

The basic search algorithm of Manifold-b operates in a continuous loop, as follows:

1. Receive message. If the message has already been processed (i.e., its

Message identifier is found in the internal Message identifier list[2]) list, ignore it.

2. Retrieve a list of neighbors (obtained from the underlying routing protocol). Alternatively, in a mobile environment a direct broadcast can be attempted.

3. If message is a *query-initiate* message, build the query message and send it to the neighbors.

4. If message is a *query* message, check whether this node contains the information requested on the query. If so, create a *query-reply* message and send it to the neighbors so it can go back to its destination [3]. If the message's number of hops is past the TTL limit, ignore it. If the information requested is not in the current node, increment the number of hops in the message and re-send the query message to the node's neighbors. After re-sending the *query* message, store the Message identifier in the identifier list (for use of the list see step 1).

5. If the message is a *query-reply* message, check whether the request was sent by this node (see Step 3). If the *query* was sent by this node, return the result to the application layer that made the resource-location request. If the query was not sent by this node, increment the number of hops in the *query-reply* message and re-send it to the node's neighbors, storing the Message identifier in the identifier list (for use of the list see step 1). Note: the query-reply contains the original query information, making storage of queries unnecessary. Since the query also has a timestamp, the node is able to determine that a timeout on

[2]Message identifiers must be globally unique. Universally unique identifiers can be obtained by concatenating a variety of data including current system time, node identifier, and other elements such as ethernet address. Some operating systems (such as Microsoft Windows) allow creation of globally unique identifiers via API calls.

[3]In most cases, the reply could be sent directly to the requester as an optimization. The basic algorithm, however, makes no assumptions in that regard.

the query has already occurred (and only cache the query for future use instead of passing the result to the application).

An additional type of message handled is *query-bind* and *query-reply-bind.* They are processed just like *query* and *query-reply* messages, but they are intended to be an aid in the *Join* process described above. Nodes answer *query-bind* requests assuming that they have not passed their set maximum number of neighbors in the network.

Once the algorithm is implemented for a particular platform, several optimizations are possible, including caching of neighbors' physical addresses (depending to the dynamics of the network), return of the query messages directly to the requester using the underlying routing protocol.

4.2.4 Key Insert, Key Remove

Because Manifold is designed to expose keys locally from a node (for example, names with which the node will be identified), key-insertion operations will normally happen locally. However, there is no limitation in the system that mandates that a key can't be inserted from a remote node, and in some cases it will be useful to provide self-organizing key-storage mechanisms that will exist as services available to the network.

Manifold-b supports a *key-insert* message, with parameters that identify the key being inserted and its origin (as well as whether the key is a cache copy or not). This type of message can be received either locally or remotely. After a *key-insert* message the key in question will be available for resolution by the Key Search function.

A *key-remove* message operates in a similar way, but deleting the key in question from the store.

4.3 Dealing with Node Failure

Manifold-b has extremely low consistency requirements. Quite literally, all that's required is that a single connection into the network be valid for a search query to proceed. Manifold-b nodes should have, although they're not required to by the algorithm, a minimum number of neighbors to ensure fast query propagation (We will come back to the issue of performance and number of neighbors in the next section). If a connection is attempted, either to initiate a query or propagate it, and the neighbor to whom the query must be forwarded is not online, the node should engage in re-discovery (i.e., perform a *Join*) to rebuild its list of neighbors. This ensures that the network maintains connectivity based on activity (that is, re-discovery requests are performed on the basis that new queries have to be forwarded) without requiring potentially expensive keep-alive measures.

4.4 Analysis

While it is clear that flooding-based schemes could present scalability problems, the size of networks such as Gnutella, with millions of nodes operating concurrently, makes it clear that they *can* work, even in large scale deployments (although their inability to guarantee results remains unchanged). Broadcast-related issues have been studied in the past, in particular for wireless networks in [2] [23] [26] [75].

It would be possible to use different broadcast schemes, and even use limited-broadcast within structured overlays such as that described in [21]. However:

- in highly dynamic or failure-prone environments such as ad hoc networks, the low consistency requirements of Manifold-b provide an advantage in terms of reliability, overhead, and performance, particularly when the network is in the process of forming or it only has a few nodes. Wireless ad hoc networks in particular are by nature broadcast environments, and thus naturally suited to broadcast operations.

- finally, Manifold-b can assist Manifold-g as an out-of-band method for nodes attempting to join the Manifold-g overlay.

Since we are not concerned with the specific physical transport used, we will show the results of our analysis of Manifold-b, performed on a simulation. The three main factors to take into account when analyzing the algorithm are:

- The total traffic it generates depending on the number of nodes in the network.
- The traffic it generates within a node's range, which could potentially limit the bandwidth available to each node if the local (i.e., in-range) traffic generated by the algorithm is too high.
- The speed with which an answer can be received. This *query-reply* speed is directly related to the average path length for the network. As mentioned in [29] the average path length (number of hops required) for a given transmission between nodes is expected to grow with the spatial diameter of the network, that is, the square root of the area (s), or $O(\sqrt{s})$ for a fixed transmission range capacity per node.

Our Manifold-b simulator creates a "world" consisting of several nodes, each running the protocol, each with a unique physical identifier and node name. The simulator is built on top of the Swarm Simulation System [24], a software package widely used for multi-agent simulation of complex systems.

The simulator allowed us to both obtain data on the messages exchanged between the nodes as well as visualizing in real time the propagation of the messages throughout the network. Different types of messages (e.g., a query request, or a query reply) could be visualized differently by changing the color of the nodes that carried them. The system also ensured that even when all nodes where running on the same machine (and therefore sharing a single processor) their actions could be synchronized through time, to represent a simplified network[4], were actions happen simultaneously. The nodes were located on a virtual two-dimensional grid, and the system allowed us to place nodes in different configurations (e.g., grid, random, etc).

Each run of the experiment created a configuration of a random network with 100 nodes. For each run, a node chosen at random inserted a query for a randomly chosen string from the list of strings that are known to be available in the network. In each simulation cycle, every node processed the messages that arrived in the previous cycle. As we mentioned in the previous paragraph, this simultaneous processing of messages is a simplification of the real world case, but it allowed us to find the upper bound of messages set by maximizing the number of simultaneous messages that could be theoretically be sent at any given instant.

Our measurements indicate that the main factor conditioning performance for a network running the algorithm is not the total number of nodes, but rather the average number of neighbors for a given node[5].

Based on the results shown in Table 4.1 and Figures 4.1 and 4.2, it is possible to find the maximum bandwidth that will be allocated to resource location by using the following function[6] :

[4]In a real network actions might happen simultaneously or not, but this was not a factor in our simulation since we were interested in measuring traffic and propagation patterns, rather than response-time related issues.

[5]In Table 4.1 and Figure 4.2 the value *Peak Messages In Range* represents the average number of messages within a node's range, and it can therefore be used to calculate the cost, in bandwidth terms, of the service. This value is obtained by averaging the peak number of messages per simulation cycle.

[6]Note that this bandwidth peak usage happens in bursts, since resource location is not typically requested constantly, but rather as users perform functions that are meant to initiate sustained data traffic. The impact of resource-location traffic is therefore limited when compared to typical traffic.

Average Neighbors	Average Cycles Until Done	Average Cycles Until Reply Received	Total Messages To Resolve Query	Peak Messages In Range
3	22	13	715	16
6	14	9	1356	67
10	12	7	1990	145
15	9	5	3130	415
21	7	4	4191	791
26	7	4	5189	1304
33	6	3	6552	2315
52	5	3	10330	6724
55	5	3	10962	7396
69	4	3	13672	9604
79	4	2	15740	9801
95	4	2	18955	9801
99	3	2	19604	9801

Table 4.1: Simulation results

$$\text{number of messages} = \frac{BP}{S}$$

Where B is the total bandwidth (in bytes per second), P is the proportion of bandwidth to allocate to resource location (which can be configured by an end-user application or operating system setting) and S is the average message size. As an example, assuming resource location was to be confined to a peak of 20% of bandwidth, for a 2Mbps system, it would mean a bandwidth usage limit of ~45 KBytes. This value translates into ~1800 messages per second. Therefore according to Table 4.1 the algorithm would support at most ~20 average number of neighbors for the system at the desired 20% bandwidth, allowing one query per second at a constant rate. This result is based on a query-resolution cycle of a duration of ~250 milliseconds. If the value is smaller, more queries can be resolved per second at a constant

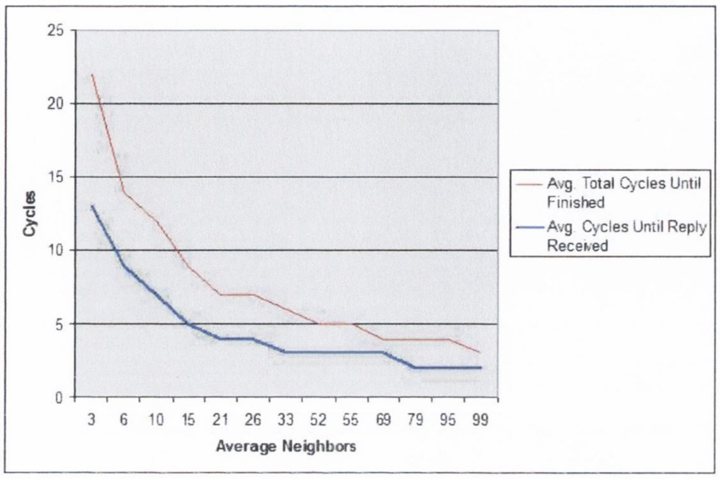

Figure 4.1: Influence of the average number of neighbors on the network-wide cycles necessary to resolve a request

rate for that bandwidth usage proportion. The number-of-neighbors limit therefore points to parameter that has to be controlled (for example, in the case of a wireless ad hoc network, by reducing radio range until an appropriate number of nodes is within range, or in the case of wired networks by simply requesting less neighbor connections) for Manifold-b to maintain its usefulness in high-density wireless networks, and to a number that must be appropriately limited when used in "wired" networks.

4.5 Summary

Manifold-b is an appropriate solution for RLD for problems of inexact search, or of exact search that is not required to be global or time-bounded. Applications for this algorithm include in particular those that imply the discovery portion of RLD, or location without guarantee that the results will be found. We will now discuss the Manifold-g algorithm, which solves the second part of the problem by providing global search with guaranteed results.

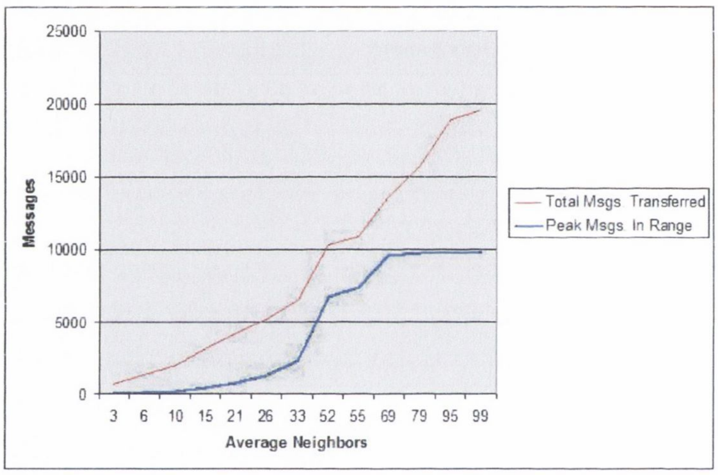

Figure 4.2: Influence of the average number of neighbors on messages transferred

Chapter 5

Manifold-g

5.1 Restating the Problem of Search

The limitations of TTL-based algorithms, particularly with regard to guarantees on results and lookup times, led us to look for an alternative solution.

We began by recognizing that TTL-based self-organizing algorithms consider the search algorithm a distributed version of a local search algorithm, which is in the end based on fast string matching of the requested string against a database of the strings stored by the node. The focus is usually, therefore, on the graph traversal techniques used to move between nodes in the graph, while the properties of the search space itself (i.e., the space defined by all the possible strings to be searched) is rarely if ever considered a factor.

We then realized that if the connection between nodes was determined based on the values exposed by them, rather than through other measures, the structure of the topology would be predictable, allowing us to guarantee that if a node existed then it would be placed at a certain location that could be determined dynamically with respect to the other nodes. Furthermore, if the structure had the appropriate characteristics, navigating the space of values could be fast enough to apply to large scale networks.

Therefore, we recast the search problem as one of predictable traversal through a mesh of finite boolean sequences (the strings published by the nodes in the network), understood as an l-dimensional hypercube, or l-cube. We were thus able to derive a an algorithm with complexity $O(\log N)$ where N is the total number of possible boolean strings in the space. We called this algorithm Manifold-g[1].

Throughout this chapter we will show how this algorithm maps into a self-organizing overlay network, and then show how the problem of "holes" in the hypercube can be solved by an appropriate node-insertion algorithm, and how that leads to a maximum complexity of $O(\log n)$ where n is the *actual number of nodes* in the network. In the next chapter, we will consider additional improvements on the algorithm, such as optimizations to the basic topology, and how Manifold-g can adapt to the topology of the underlying physical network through the use of *proxies*, that can also be used by low-power nodes to off-load some of their tasks.

5.2 A Short Description

The Manifold-g algorithm uses finite boolean sequences (in other words, strings of 0's and 1's), organizing them in an l-dimensional space (where l is the maximum length of the binary string of particular characteristics, which are then used to traverse the space efficiently.

One of the key elements of the algorithm is the fact that any node in the network will not only search for data in the network, but also provide data to the network itself. For example, when the network is used for name resolution (i.e., name to IP address mapping) each node in the network will can map the name itself. This means that if the algorithm can provide a way to map from a particular value (node in the network) to another in a predictable way, the search problem would be solved.

[1] "Manifold-global"

For this, we consider the boolean space X (see Appendix A) as an l-dimensional space, with l being the maximum length of the boolean string. We are thus able to calculate different points that "connect" elements of the linear boolean space to different dimensions by using an XOR function recursively applied starting with the bit-string derived from the name of the node that initiates the query. The way in which we use the function will guarantee that each point will only appear once and will always be mapped into the same values in this new function space relative to all its neighboring values. Once we show that the the space can be navigated deterministically, we will show how it is a matter of applying the same function recursively to traverse from one point to another in the original l-dimensional space, therefore completing the query.

5.2.1 The Neighbor Function

We're interested in being able to define (calculate) our neighbors locally, ie., based on a node's local information *only*. Using the theory from Appendix A, and in particular its definition of Hamming distance δ, we define the neighbors S_i of a node S as the set of those nodes that satisfy

$$\delta(S, S_i) = 1 \quad \forall \; 0 \leq i < l$$

To calculate those neighbors using only local information (i.e., the value of the string in question) we can apply the boolean XOR operator ($\oplus$) as follows:

$$S \oplus P_i \quad 0 \leq i < l$$

With l the length of the string (ie., the dimensions of the l-cube), and P_i defined as a boolean string in which

$$P_{i_{(j)}} = 0 \quad \forall \ j \neq i$$

and

$$P_{i_{(j)}} = 1 \quad \forall \ j = i$$

which implies l neighbors for any string S in space X (X defined in Appendix A).

In terms of base ten integer values, this means the neighbors of a node are calculated by performing an XOR operation between S's own value against each power of 2 between 0 and $l - 1$.

It is important to note that, according to this definition, neighbors could be (and indeed will be) lower- as well higher-value strings (in integer terms) than the value being considered. The number of neighbors of lower value will be the number of powers of 2 below the number, while the number of neighbors of lower value will be the number of powers of 2 above it.

We should also point out that:

- the neighbor function $N(X)$ is *symmetric*, that is to say that

  $$N_i(X) = Y \quad \Longleftrightarrow \quad N_i(Y) = X \quad \forall \ (X, Y) \in S$$

  Where S is the set of all possible strings, X and Y are node names and N_i is the neighbor i of the node to which the function is applied.

- the neighbor function maps a set onto itself:

  $$N_i(X) = Y \quad \Longleftrightarrow \quad (X, Y) \in S$$

  Where S is the set of all possible node names.

Value	Neighbor 1	Neighbor 2	Neighbor 2
000	001	010	100
001	000	011	101
010	011	000	110
011	010	001	111
100	101	110	000
101	100	111	001
110	111	100	010
111	110	101	011

Table 5.1: Values and Neighbors for a String Length of 3

5.2.2 Space Defined by the Neighbor Function

As mentioned in Appendix A, we can look at the original boolean set as the set of vertices that define an l-dimensional hypercube, where l is the fixed length of the boolean strings used. Each dimension j of that hypercube will contain 2^j strings (or vertices). However, we are only concerned by the actual number of neighbors, which remains at l due to the qualities of the neighbor function described in the previous section.

As an example, let us consider a fixed string length of 3. Table 5.1 shows all the possible string values (nodes) and the corresponding 3 neighbors per node, obtained by applying the boolean neighbor algorithm described above:

Now, to visualize the l-cube as a 3-dimensional euclidean space, we define a point in euclidean space $P = (x, y, z)$ for a string S as given by $x = S_{(0)}$, $y = S_{(1)}$ and $z = S_{(2)}$. If we also specify that when any two strings are neighbors (that is, S and T satisfy $\delta(S, T) = 1$) they are connected by an edge, we obtain the graph shown in Figure 5.1.

Viewing the space in this form helps to intuitively understand how the neighbors are connected by a distance of 1 in the virtual topology, and later it will also facilitate visualization of the search process, particularly for string lengths $n = 2$ and $n = 3$.

In the discussion that follows, we will refer to a *complete hypercube* when

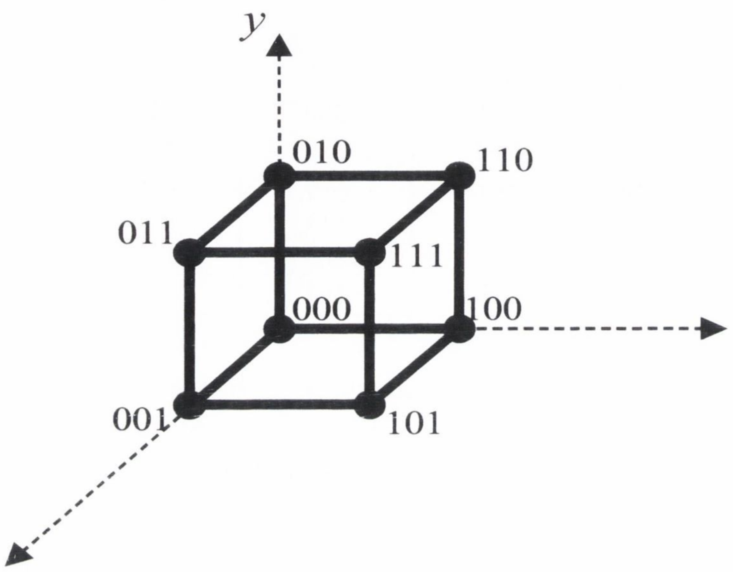

Figure 5.1: A Length-3 String Space Mapped into an Euclidean Space with the Neighbor Function

discussing an l-cube with all its vertices. If one or more of the vertices are not present, as would be the case if the node responsible for that string was not present, we will refer to it as an *incomplete hypercube.*

5.3 Search in a Complete Hypercube

As a first step, we will define a local function to traverse a complete, or fully connected, hypercube. The assumption of a complete hypercube is, however, unrealistic, at any time the number of strings inserted in the network will be less than the maximum possible given a string length l (since not all the possible combinations of names will be present). We will define a search function that takes into account incomplete hypercubes in Section 5.5.

The complete-search function uses the topology of the hypercube to traverse the graph one string at a time through local operations (that are, globally, viewed as a single function applied recursively).

On each string (a node might be responding for one or more strings), the search function reduces the Hamming distance δ between the current string, S_C and the target T by 1, until the target is reached. We will consider the search as originating from a particular *origin string* S_O and try to find a path from it to a *target string* T. In the first step, $S_C = S_O$.

We therefore define a *local* minimal path function[2] σ as follows:

$$\sigma(S_O, T) = \{\Delta_1, \Delta_2, \ldots, \Delta_j\} \quad \text{with } j = \delta(S_O, T)$$

Where each Δ_j is the string returned at each step for the recursive function Δ:

$$\Delta(S_C, T, i) = \left\{ \begin{array}{rcl} S_C \oplus P_i & \Longleftrightarrow & \delta(T, S_C \oplus P_i) - \delta(T, S_C) = 1 \\ S_C & \Longleftrightarrow & \delta(T, S_C \oplus P_i) - \delta(T, S_C) = 0 \end{array} \right\} \forall \; 0 \leq i < l$$

With P_i as defined above in the neighbor function. To avoid backtracking on a path, we add the condition

$$\forall \quad \delta(T, \Delta(S, T, i)) = \delta(S_C, T) - 1$$

which guarantees that the distance will be reduced at every step, eventually reaching the target. Note that for the final step $\Delta(S, T, i) = T$. *This function returns, with each iteration, a set of strings that will allow us to "move" one step closer to the target*, halving the distance to it from the current position in the hypercube. Any of the strings can be chosen as the S_C to be used in the next iteration. While it is quite clear that on a complete hypercube the full search is completely predictable, as we will see shortly when defining traversal it is important to define a local function as a basis.

In other words, the local minimal path function σ returns

[2]on the fully connected hypercube

$$\sigma(S_O, T) = \{S_{C1}, S_{C2}, \ldots, S_{Cj}\} \quad \text{with} \quad j = \delta(S_O, T)$$

The set of S_C strings found between S_O and T is the path P between the origin and the target, and the number of strings in that subset is the *path length*, or P_l.

There are some important elements of the algorithm that should be noted:

- As we mentioned earlier, the algorithm is purely local. Only information regarding the current position (and the previous position) is needed to proceed to the next step in the process.

- Because of this, the algorithm can be implemented either as a recursive process (where each node is responsible for forwarding the query to the next node) or as an iterative process (where the origin node queries each node in succession for the next node to be contacted according to the algorithm). The theoretical definition is recursive, however, and implementing the algorithm as an iterative process is related to how the load is to be distributed on the network. We will consider the differences in implementation later in this work.

- Since the path moves along the edges of the hypercube bridging differences of 1 between bit strings, it is obvious that the maximum path length for σ is $\lceil l \rceil = \lceil \log N \rceil$ where N is the number of nodes in the hypercube, as

  $$N = 2^l$$

  And so σ has a complexity of $O(\log N)$.

- If we consider the binary strings as base ten integers, the algorithm can be viewed as moving between one node and the target by the power of 2 that will take the search closer to its target. This is possible because the space is already organized with connections between nodes according to powers of 2.

5.4 An Example

As an example, let us consider a search from an origin $S_O = 0, 0, 1$ for a target $T = 1, 1, 1$. The first call to the search function is the set returned by calls to $\Delta(S_C, T, i)$ with $0 \leq i < 3$ since $l = 3$ and $S_C = S_O =' 001'$. That is:

$$\begin{aligned}\Delta(S_C, T, 0) &= S_C \\ \Delta(S_C, T, 1) &= S_C \oplus P_1 =' 011' \\ \Delta(S_C, T, 2) &= S_C \oplus P_2 =' 101'\end{aligned}$$

Therefore both $'011'$ and $'101'$ takes us closer to the target. We choose $'011'$, set it as the new S_C, and repeat the process. At the next step, we directly obtain T as the result of the iteration, and the search process is completed.

From the point of view of the euclidean representation we described earlier, the algorithm is simply moving across edges in the direction of the target string. This example can thus be viewed graphically as shown in Figure 5.2.

5.5 Operations in an Incomplete Hypercube

A central assumption in the algorithm presented in the previous section is that the network will be a fully connected l-cube, i.e., that every string in the space will be available for searching and, more importantly, for providing paths to other strings. This is clearly unrealistic for most cases: while some types of self-organizing networks (such as sensor networks) could be designed to "fill" a search space completely, most cases (e.g., name resolution on the Internet) will not. So a central problem is how to provide full connectivity without affecting the algorithm's behavior and its search complexity of

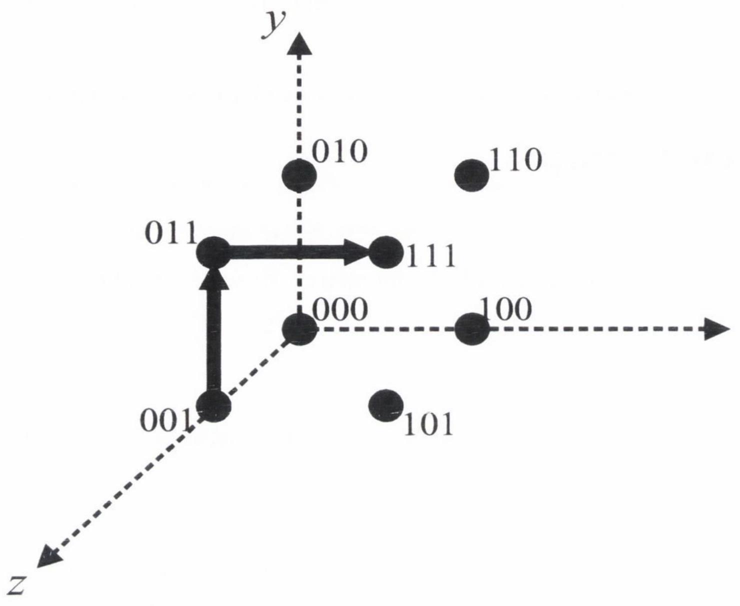

Figure 5.2: A Sample Search for Length-3 Strings Viewed in an Euclidean Space

$O(\log_2 N)$.

A solution would be to organize the space as it grows according to certain rules that guarantee full connectivity. This has to be guaranteed at the point when a node joins the network, which results in a relatively expensive operation compared to the cost of a search (as will be for the neighbors of a node leaving the network, which would have to rearrange their connections to adapt to the new space topology). However, the value of this operation will far outweigh its cost if it is kept within certain bounds.

We will use the search capabilities of the network to maintain its full connectivity as new nodes join or leave. The join/leave operations will then be dependent on the speed of the search, which we already know to be extremely fast even for large networks.

It is important to note that while what follows in this chapter is an accurate mathematical formalization of the process, it is not an exact representation of the actual implementation of the algorithm. Later on we will describe such an implementation to create a self-organizing name resolution system.

5.6 The Shadow Mapping: Definition

Given X a boolean algebra as defined in Appendix A (a set of boolean values B plus its operations), we will define X' to be another boolean algebra formed by combining a set B', such that $B' \subseteq B$ with the same operations. We also define B'' to be a subset (or partially ordered set[3]) of B', such that $S_x < S_y \ \forall \ x < y$ and where x and y define the position of the string in the poset. In other words, B' is the algebra that represents a network as it exists, while B'' is a partially ordered set of B' used within the shadow mapping definition.

Boolean strings in B' can be of two kinds:

- Actual strings, denoted as before as S, T, S_n, etc., or
- What we will call *shadow* strings, noted as S', T', S'_n, etc. A shadow string is a string that does not exist in B' but exists in B, that is S' is a shadow string iff $S' \in B$ and $S' \notin B'$.

Based on the concept of shadow strings, we will define a *shadow string mapping* SSM such that

$$SSM(S') = F(S', B'', 1)$$

Where S' is a shadow string and $S \in B'$. The shadow string mapping F is defined as a recursive function, as follows:

[3]A partial order is an order defined for some, but not necessarily all, items. For instance, the sets $M = \{000, 001\}$ and $N = \{000, 010, 011\}$ are subsets of $P = \{000, 001, 010, 011\}$ but neither M or N is a subset of the other, so "subset of" is a partial order on sets.

$$F(S', B'', m) = S_m \quad \text{if} \quad S' \leq S_m \quad \text{and} \quad 0 < m \leq l$$

where l is the number of elements in B'', the poset of B', and S_m is the element at position m in B''.

$$F(S', B'', m) = F(S', B'', m+1) \quad \text{if} \quad S' > S_m \quad \text{and} \quad 0 < m \leq l$$

and finally

$$SSM(S', B'', m) = S_{m-1} \quad \text{if} \quad S' > S_{m-1} \quad \text{and} \quad m > l$$

Which covers the ceiling value of the set. In this last case, of course, it will still hold that $S' < sup(B)$.

The shadow mapping, then, specifies which node in the network (understood as the string value it represents) is responsible for "covering" which shadow strings. To visualize the shadow mapping more easily, it's useful to think just in terms of values, that is, A string S is responsible for a given shadow string S' if:

- $S > S' > S_n$ where S_n is any of the nodes connected to S, either as neighbors or shadow neighbors.
- $S' > S > S_n$, which covers the remaining case, i.e., when a shadow string is of higher value than any of the strings present.

In practical terms, the inherent load-balancing effected by the shadow mapping means that the overhead (in terms of state maintained) created by shadow strings on the nodes that are covering for them is not significant.

5.7 The Space-Complete Join Function

The next step is to consider how to complete the hypercube, even if values are missing, to maintain the properties of the search, by combining the basic properties of the hypercube with the shadow mapping definition defined above in Section 5.6. We will therefore define the *Join* function for our space $B' \subset B$. Before that, however, we should make a note of how new strings are added into B'.

While at its most basic this process relies on basic hypercube search, it can be modified to take advantage of the information available in an incomplete hypercube, as explained in Section 5.9 below.

Because there is no central or hierarchical organization to the set, we need a 'starting point'. In actual peer-to-peer networks, finding this initial node to connect to the network is done using bootstrapping techniques as those discussed in Section 2.7.1. From the algorithmic point of view, however, we can only assume that such a node exists, and we will refer to it as S_R, a reference node that is already connected. We will use this node to determine whether the new string to be inserted S_N exists in B':

$$\cup(B', S_N) = B' \cup S_N \quad \Longleftrightarrow \quad \sigma(S_R, S_N) = S'_N$$

With S'_N following the definition of shadow string given in Appendix A. Given that strings are unique, a string can only be added once to B', so a string is added to the set if and only if a search for itself has returned a shadow string, that is, an empty space not yet filled. Since the space B'' is dependent on B', it will grow as B' grows.

Now, the basic search algorithm in this new case remains unchanged. We should keep in mind that the result of the search function $\sigma(S_O, T)$ will be a set of strings of the form $P = \{S_{C1}, S_{C2}, \dots, S_{Cj}\}$, defining a path between S_O and T. However, this result applies only to B, and not necessarily to B', since some of the strings of P might be shadow strings, and not valid strings.

To map the shadow strings to an actual string (and thus be able to provide a path through strings that actually exist), we must use the shadow string mapping function, selectively on the shadow strings, by applying

$$SSM(R_i) \ \forall \ 0 < i \leq l \quad \Longleftrightarrow \quad R_i \notin B'$$

where R_i is the string at position i in the solution path R. Applying the shadow string mapping function in this way will give us a modified result path, R', with only valid values (i.e., values such that $S \in B'$) and therefore a valid path for the subset B'.

When a new string is inserted into the network, the following algorithm is applied:

- Consider, as before, that S_R is the reference node (already in the network) and S_N is the node that is joining the network.
- Calculate the list of neighbors $S_{N_i} \forall 0 \leq i < l$ for S_N. Now for each of S_{N_i}, $\sigma(S_R, S_{N_i}) = S_i$, we obtain a set of neighbors that are either a) the actual string and the node responsible for it or b) a shadow string.
- If the "real string" (i.e., the node responsible for that string) replies, then assign S_i as its neighbor. In the process, the node/string that was "covering" for S_N must be notified that S_N has arrived. Those that were using it as shadow neighbor must also be notified.
- If a node covering for that string (as shadow string) replies, check if we need to take over for that shadow string, and if so, do it. Otherwise, we'll register S_i as shadow neighbor and update the covering node.

The *Node Leave* inverts the process to maintain connectivity in the hypercube.

Given this, the Join operation has a ceiling is $\lceil l \log N \rceil$ steps, with a resulting complexity for the node operation of $O(l \log N)$. We will now show the impact of shadow nodes in the search process, and how it then effectively reduces the complexity of the search and join to $O(\log n)$ and $O(l \log n)$ respectively, with n being the actual number of strings inserted in the network, rather than the maximum possible (i.e., 2^l).

5.8 Building a network: A step-by-step example

This section is a step-by-step example that shows how network topology evolves as new nodes enter it, and how the nodes retain full connectivity as the space is completed. For this purpose, we'll use a 3-bit namespace, and assume that all eight possible nodes will join the network (i.e., self-insert). The following is a summary of the join process used by nodes as they come online and connect to the network:

- a node with string value S calculates its neighbors $S_0, S_1, \ldots, S_m - 1$ (with m string length).
- the node locates one node already in the network through out-of-band methods.
- the node searches for each of its neighbors, obtaining either the node itself or a shadow (along with the node that is currently responsible for that shadow).
- the node joins the network, notifying its neighbors, and in the process "taking over" the shadow nodes that it is responsible for, according to the shadow mapping.

This example will show, for each new node added to the network, the results of the searches for its neighbors and the network diagram that exists after the node has completed the join process, as well, as which nodes have to be reassigned according to the shadow mapping. Figure 5.3 is a guide for the network diagrams in this example.

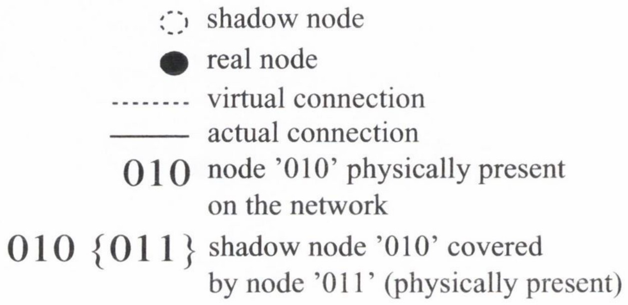
shadow node
real node
virtual connection
actual connection
010 node '010' physically present on the network
010 {011} shadow node '010' covered by node '011' (physically present)

Figure 5.3: Incomplete Hypercube Example: Diagram Guide

The initial network is empty. The diagram in Figure 5.4 represents the positions of the nodes as they exist in the theoretical space.

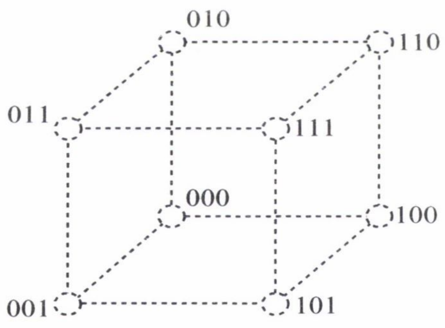

Figure 5.4: Incomplete Hypercube Example: Initial Network

The nodes will be inserted in the following arbitrary order: 100, 111, 011, 010, 000, 001, 101.

Each node that joins must first find an entry point, i.e., a node that already belongs to the overlay. Once this is done, the incoming node uses that entry point to locate its neighbors. It then proceeds to notify them and take over shadow nodes as appropriate.

The first node added has a value of '100'. Through out-of-band methods, the node determines that there are no other nodes in the network and sets itself in control of the entire string space, creating shadow nodes (mapped to itself) for its own neighbors.

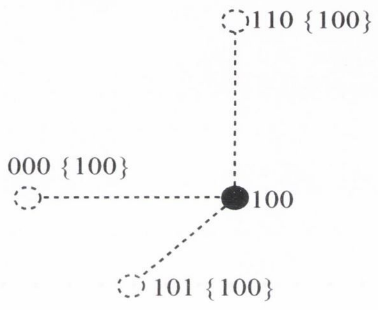

Figure 5.5: Incomplete Hypercube Example: First Node

The resulting network is shown in Figure 5.5.

When '111' is added, it looks for its own neighbors, 011, 110, and 101. It obtains:

- 011 $\rightarrow$ 100 (shadow node, covered by 100)
- 101 $\rightarrow$ 100 (shadow node)
- 110 $\rightarrow$ 100 (shadow node)

Through the shadow string function, node 111 determines that it should take over coverage for 110 and 101, resulting in the new network topology.

The resulting network after node 111 is added is shown in Figure 5.6.

'011' looks for its neighbors, 010, 111 and 001. It obtains:

- 010 $\rightarrow$ 100 (shadow node)
- 001 $\rightarrow$ 100 (shadow node)
- 111 $\rightarrow$ 111 (real node)

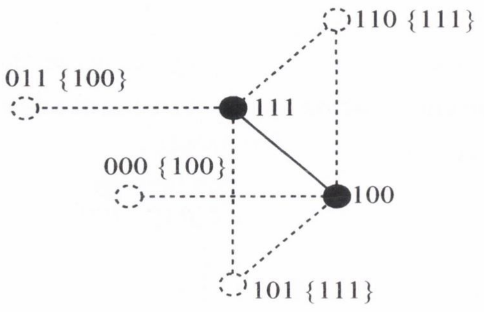

Figure 5.6: Incomplete Hypercube Example: Second Node

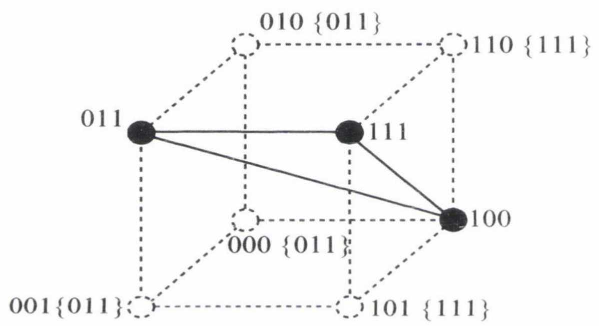

Figure 5.7: Incomplete Hypercube Example: Third Node

011 determines that it must take over coverage of 010, 001, and 000 from 100. This results in a 'physical' connection between 011 and 100, which in a complete network would not exist, to connect 100 to its shadow neighbor 000, and the neighbor of a shadow 100 is covering for, 110.

The resulting network after node 011 is added is shown in Figure 5.7.

'110' looks for its neighbors, 111, 100 and 010. It obtains:

- 010 → 011 (shadow node)
- 100 → 100 (real node)
- 111 → 111 (real node)

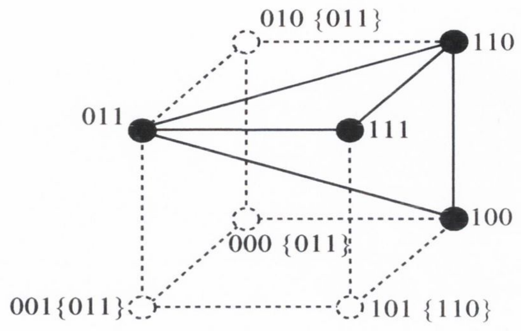

Figure 5.8: Incomplete Hypercube Example: Fourth Node

110 must take over its place, replacing the shadow previously being covered by 111, and it also takes over coverage of the shadow of 101 from 111, removing the physical connection between 111 and 100 for that shadow node.

The resulting network after node 110 is added is shown in Figure 5.8.

'010' looks for its neighbors, 000, 011 and 110. It obtains:

- 000 → 011 (shadow node)
- 011 → 011 (real node)
- 110 → 110 (real node)

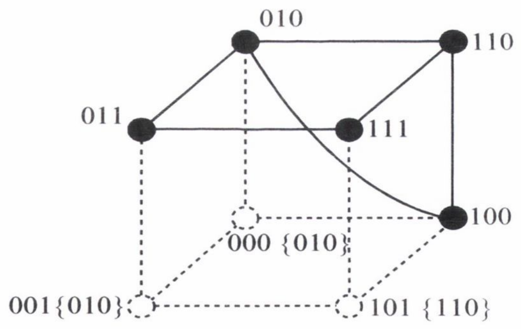

Figure 5.9: Incomplete Hypercube Example: Fifth Node

010 takes over coverage of shadows 001 and 000 from 011. Also, because it replaces the 010 shadow covered by 011, it replaces the physical connection from 011 to 110 with one to itself, also using that connection for the 000 shadow it has taken over.

The resulting network after node 010 is added is shown in Figure 5.9.

'000' looks for its neighbors, 010, 001 and 100. It obtains:

- 001 → 010 (shadow node)
- 010 → 010 (real node)
- 100 → 100 (real node)

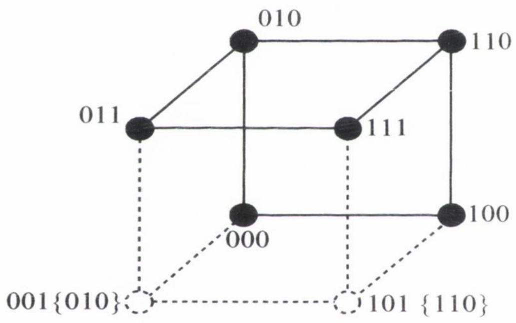

Figure 5.10: Incomplete Hypercube Example: Sixth Node

000 inserts itself without taking over coverage of any shadow nodes. It does, however, remove the connection that 010 had with 100 which existed to connect 100 to the 000 shadow covered by 010.

The resulting network after node 000 is added is shown in Figure 5.10.

Then 001 comes online, looking for its neighbors, 000, 011, 101. It obtains:

- 000 → 000 (real node)
- 101 → 110 (shadow node)

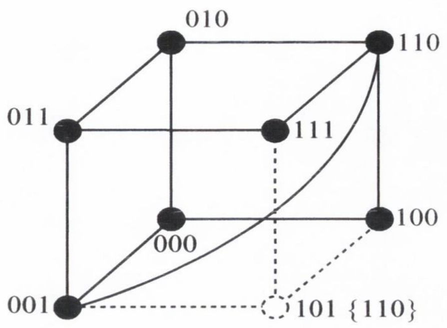

Figure 5.11: Incomplete Hypercube Example: Seventh Node

- 011 → 011 (real node)

001 inserts itself without taking over coverage of any shadow nodes. It does, however, create a connection to 110, since 110 is covering for shadow node 101.

The resulting network after node 001 is added is shown in Figure 5.11.

Neighbors, 001, 100, 111. It obtains:

- 000 → 000 (real node)
- 100 → 100 (real node)
- 111 → 111 (real node)

101 inserts itself, removing the connection that existed from 001 to 110 (because 110 was covering for its shadow), and completed the network.

The final resulting network (the complete hypercube) is shown in Figure 5.12.

5.9 Search in an Incomplete Hypercube

Performing a search in an incomplete hypercube where the missing strings are "covered" can be modified to take advantage of the fact that certain nodes

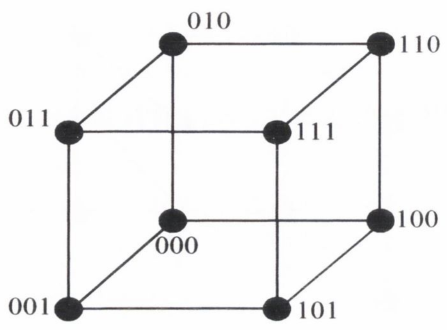

Figure 5.12: Incomplete Hypercube Example: Eighth Node

actually hold more information than nodes in the fully connected hypercube. First, S_O is the origin string/node (the one that starts the query). As before, S_C is the *current* value and we add S_{CC_i} as the i values currently covered by S_C as shadow strings. T is the target value. The function σ can be modified as follows:

At the start, $S_C = S_O$. We calculate $\delta(S_C, T) = \delta_{CT}$ as well as $\delta(S_{CC_i}, T) = \delta_{CCT_i}$, for each of the i values currently covered as shadow strings.

Now, if $\delta_{CT} = 0$ or any of $\delta_{CCT_i} = 0$, we have reached the target.

If $\delta_{CT} = 1$ or any of $\delta_{CCT_i} = 1$ then we are one step away from the target, and the result will be found in the next iteration.

If $\delta_{CT} > 1$ and $S_{CCT_i} > 1$, then choose all neighbors SN_i of S_C that satisfy

$$\delta(SN_i, T) < \delta_{CT}$$

and all shadow neighbors SN_{CCT_i} neighbors for shadow nodes S_{CCT_i} that satisfy

$$\delta(SN_{CCT_i}, T) < \delta_{CCT_i}$$

from both sets, choose the node for which the distance between it and the target is the *smallest.* That node is the new S_C; assign it, and repeat the process.

5.10 Performance in an Incomplete Hypercube

Searching in an incomplete hypercube actually has the advantage that the number of steps required to reach the target is reduced.

As we have mentioned, the *maximum* number of steps to get from any vertex of the l-cube to any other is l.

$$j = \log 2^j = \log N$$

Again, N is the number of vertices in the hypercube.

Now, consider a network with n actual nodes, such that $n < N$. We must keep in mind that when there are less nodes than the maximum, some nodes will be "covered" by others, as shadow nodes. Essentially when a node is covering another it's acting as *two* nodes, itself and the shadow—note that potentially it could be more, but one is the minimum, and in any case it would average out since a node covered there would not be covered elsewhere. Because it's acting as two nodes, then the number of connections out of the node is twice the "normal", so the distance to the target can be divided by four, rather than halved, in a single step. This results in that step being skipped (or rather, in the possibility of taking a step that takes us much closer to the target than normal).

It is clear that, for any path, there will be a maximum of

$$(\log N) - (\log n)$$

steps that will *not* be real nodes, but shadow nodes. That is, these are steps that will be in practice skipped since another node is covering for them (potentially more than one step could be skipped, but to get the upper bound we use only one skip per shadow—the minimum).

So, if $\log N$ is the maximum number of steps and $(\log N) - (\log n)$ is the maximum number of steps that will be skipped, then the actual number of steps can be obtained by subtracting both maximums:

$$(\log N) - ((\log N) - (\log n)) = (\log n)$$

And so the upper bound is actually $\lceil \log n \rceil$ which gives us search path lengths of complexity $O(\log n)$ for n actual nodes in the network, and join complexity of $O(l \log n)$.

5.11 Analysis

To analyze the behavior of the Manifold-g algorithm we implemented a system that considered nodes as objects with pointers to each other (to simulate network connections). We used string lengths of 14, giving us a maximum of $2^{14} = 16384$ nodes. Our simulation ran in steps, incrementing the size of the network (with ten new nodes joining at every step), with a starting node chosen at random from the list of nodes already in the network. After each join step, we performed 100 queries between randomly chosen origin and target strings (with a probability of 1/2 of the target string being a shadow string in the network, to verify behavior of searches on strings that have not been inserted). The results of the simulation can be seen in Figure 5.13.

As the results show, the maximum path length remained below or at its expected theoretical limit of $\log n$ throughout, as new nodes entered the network, with the average number of steps for a search remaining substantially lower, since random searches might at times be requested on strings that are "close" in the 14-cube topology.

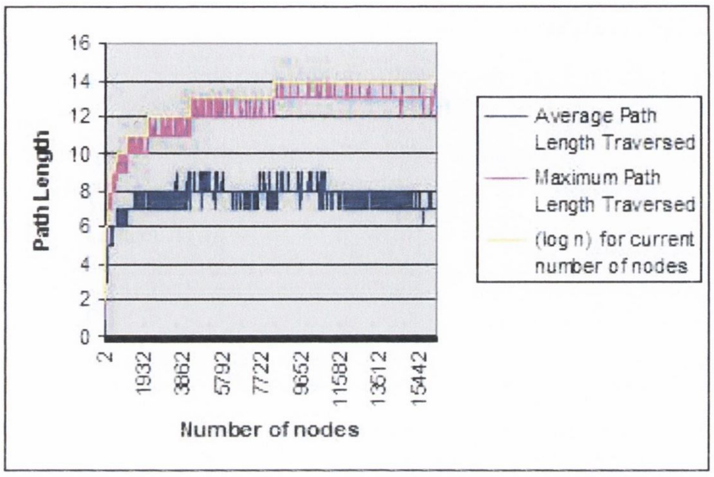

Figure 5.13: Average and Maximum Path Lengths compared to their theoretical maximum on a Manifold-g network of increasing numbers of n nodes

As a final note, while at its peak the join operation requires a maximum total of (in this case) 196 messages, we should note that this actually represents $l = 14$ paths that are being traversed in parallel when the node is gathering the information to join the network, with each parallel track being 14 messages. If each step takes, for example, 100 msec, the total time required for a join operation in this network would be 1.4 seconds at its maximum, when the network is almost complete. This time decreases accordingly with the observed maximum number of messages required (seen in the graph) when the network is incomplete.

Although in our examples we focused on functions that assume a one-to-one mapping between nodes and their content (the node's name), the functions, and their respective results, apply equally well when aggregation of several key/value pairs occurs in a single node (thus trivially supporting the mapping of multiple names to a single node).

5.12 Summary

The properties of the Manifold-g algorithm make it well-suited for large populations of nodes, while guaranteeing data location and setting an upper limit on the time necessary to complete an operation. Together with Manifold-b, the two algorithms satisfy the requirements outlined earlier in this work for generic, network-independent RLD.

Chapter 6

Manifold-g: Extensions and Improvements

6.1 Introduction

The initial Manifold-g overlay design was based on a view of the overlay as a hypercube obtained using the binary values of the strings in question. With each node connected to l neighbors (with l the number of bits of the string) we create a hypercube of l dimensions. This allowed us to create a search function with an upper bound of $O(\log n)$ on the number of steps per search, requiring $O(\log l)$ initial connections.

Given the conditions under which a mapping function creates a valid overlay network, it is possible optimize its structure according to different parameters. As long as the modifications chosen maintain structural symmetry, it will remain a valid overlay mapping function for Manifold-g.

We will now consider several optional optimizations that could be made to the basic Manifold-g algorithm, improving different operating parameters.

6.2 Hashing

We should note that, unlike other overlays, Manifold-g does not depend on hashing to function, only on a name-to-binary mapping. The simplest way to do this mapping is to directly take the binary value of a name string. Since each alphanumeric character has a unique ASCII code, this would result in unique strings. However, the algorithm can define its own mapping because using the ASCII mapping would be wasteful. In general a large number of character combinations will produce names that are difficult to remember and will therefore not be used at all.

Since the binary strings can be derived in any form as long as they maintain relative uniqueness in the space, an immediate optimization is the use hashing on the names, to reduce string size (reduce the space dimensionality) and to balance string distribution. If the hash function is properly defined, collisions in the resulting string space should happen with low probability.

6.3 Search In a Meta-Hypercube

6.3.1 Increased Number of Connections

One way to improve the basic algorithm is to increase the number of connections, forming what is known as a *variant hypercube* [83]. Doubling the number of connections would mean half the number of steps are required to reach one node from any other. This improvement, however, has a limit. For example, in an 8-bit space a search would take 8 steps. Increasing the number of connections to 16 would halve the maximum number of steps, to 4. Increasing the connections to 32 halves the steps again to 2, and then further increases only mean that there is less probability that the maximum of 2 steps will be reached. After 32 connections, the only way to achieve a one-step search always is to create 256 connections.

For 64-bit strings this is still not useful: already 64 connections are necessary. Another alternative has to be considered: partitioning.

6.3.2 Partitioning

Partitioning divides the hypercube into a set of multiple hypercubes, which are connected through a single node each. Every sub-hypercube is thus connected to every other hypercube; searches then involve a search in the local hypercube (for a jump point into the target's sub-hypercube) and then another search in the target's sub-hypercube for the target node. This separation allows us to partially "linearize" the number of connections, freeing up a number of connections per node to be used in the more standard fashion of increasing the number of connections in each sub-hypercube to increase the local speed. We can thus cut by half the upper bound of both the number of connections required, and the maximum number of steps per search. An additional side-effect of partitioning is that failures become less problematic, since they affect a lower number of nodes.

For this, we will use the fact that any string is actually a combination of a number of others. So for example, the set of all possible 8-bit strings is the combination of the set of 4-bit strings with itself. Table 6.1 illustrates this partitioning.

If for a string of l bits we get a set with 2^l values, then partitioning the string by half creates $2^{l/2}$ subsets of $2^{l/2}$ values.

Now, given that each subset has the same number of values as there are subsets, we can use a formula to connect a single value uniquely to another subset. Essentially, we would be subdividing the hypercube into a set of smaller hypercubes, with every hypercube connected to each other by a single connection, as shown in Figure 6.1 (with the connections between hypercubes in blue).

In the figure, we have only used three connections per hypercube, dividing a 5-bit space into 4 3-bit spaces. The most efficient use of partitioning in this case would be dividing a 6-bit space in the same way, thus creating 8

0000	0000
0000	0001
0000	0010
0000	0011
⋮	⋮
0000	1111
0001	0000
⋮	⋮
0001	1111
1111	0000
⋮	⋮
1111	1111

Table 6.1: Decomposition of an 8-bit String Set into Combinations of 4-bit Sets

smaller degree-3 cubes where each node connects to three other nodes in its own degree-3 cube and to a single node in another cube.

Once we have this partitioning, searching for any value in the full space requires the following steps:

- Find the node in the "source" subspace that connects to the "target" subspace (which can be calculated by partitioning the name of the target string).
- use that node to connect to the target subspace.
- once in the target subspace, perform another search internally for the actual target node.

Using this search algorithm, the number of connections required per node becomes $l/2 + 1$ instead of l. However, the maximum number of steps in the search becomes $2(l/2) + 1$. If we partitioned a 64-bit string we would need

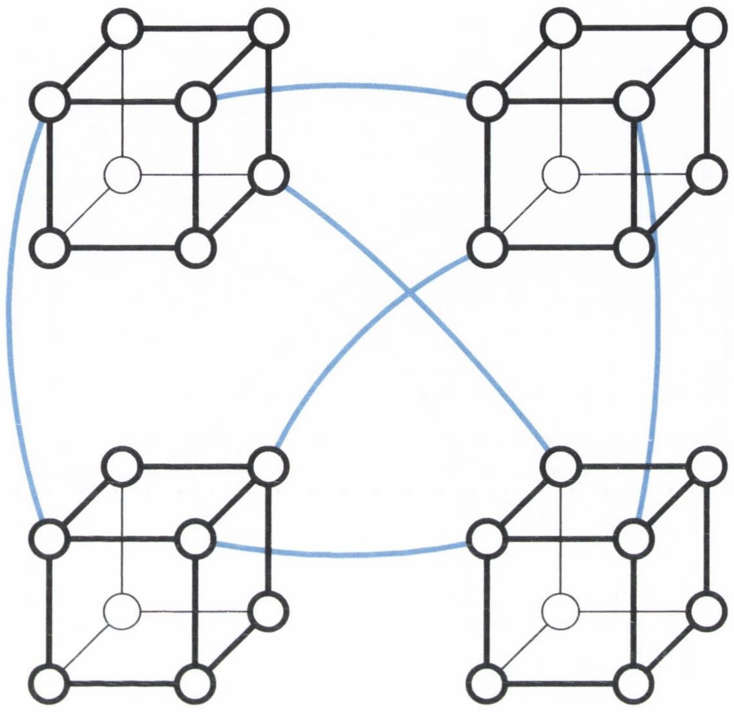

Figure 6.1: A Graphical Representation of Partitioning

33 connections, but search could take a maximum of 65 steps on a complete hypercube (i.e., with 2^64 nodes).

Figure 6.2 shows the path a particular search would take. In the figure, node S is looking for the target T (The search path is marked in red).

Recursive Partitioning

Partitioning can be done recursively. Adding another level of recursion means that every node would connect to:

- All necessary nodes in its subspace
- One node for the level-1 aggregation

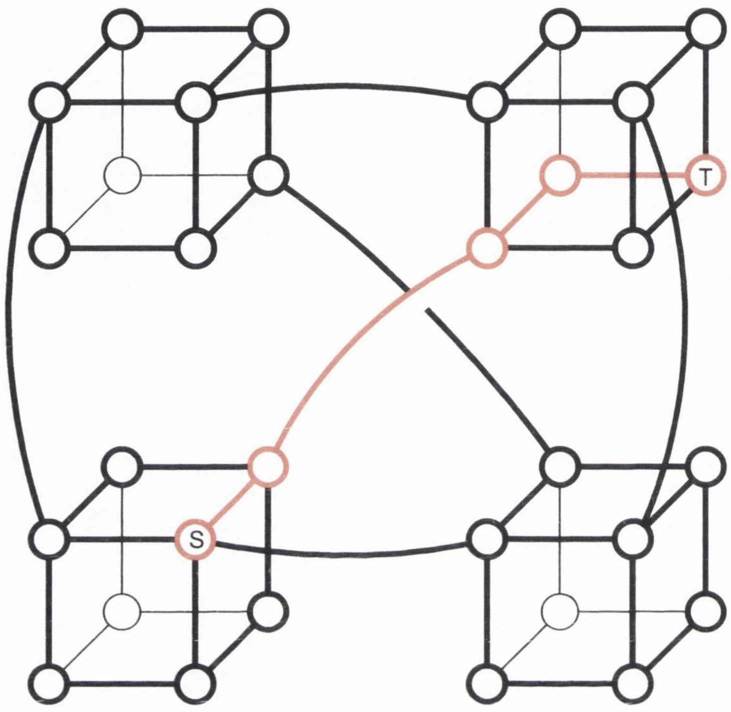

Figure 6.2: A Search Path in a Partitioned Space

- One node for the level-2 aggregation.

Figure 6.3 shows the connections of the first node (without showing all connections into its 4-bit). We can see how that node connects completely to its own subspace, and then to the next level-2 subspace, and so on. Therefore if any node in the first 4-bit subspace wanted to connect to a node starting with '000000000001' it would look for node '0000000000000000' which would then route the query appropriately to the subspace.

In the example shown, partitioning the 16-bit string twice (creating a set of 2^8 8-bit subspaces, each of which is divided into a set of 2^4 4-bit, we'd need 6 connections: $4 + 1 + 1$. Any search would then have the following steps:

- Find node within the 4-bit space (max 4 steps) that connects to the

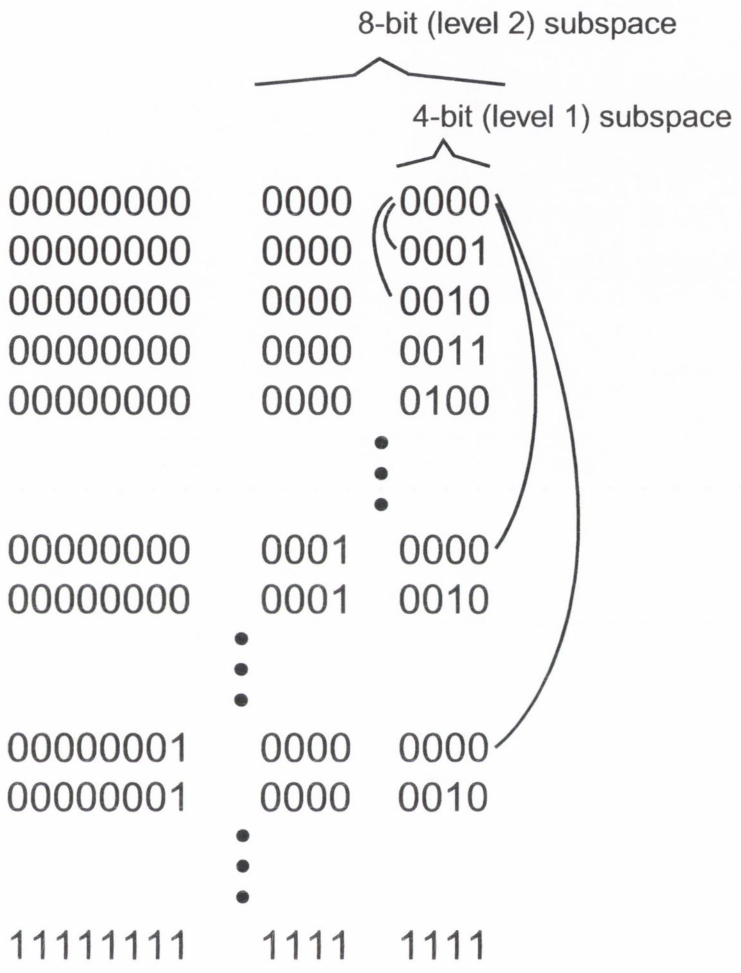

Figure 6.3: Applying Partitioning Recursively

node in the local 8-bit space that will allow us to connect to the target 8-bit space.

- Jump to the new 4-bit subspace, still within our local 16-bit space, and perform another search. (4+1 max steps).

- Within the target 8-bit space, perform the same two-step process.

This gives a total maximum of steps of $4 + 1 + 4 + 1 + 4 + 1 + 4 = 19$. While this is more than the number of steps required for a purely logarithmic approach (16) only 6 connections are required per node, compared to 16 for the logarithmic approach.

Subspace Jump Function

The subspace jump function (that maps a single value from one subspace to another subspace) is the key of the partitioning algorithm. Like the standard function, it must be symmetric, so that based on a number we can calculate the target value, and its inverse must exist, so that nodes can use it to derive the jump-node based on the target subspace desired. This requires a new neighbor function. The following is a function that fulfills these requirements:

$$N(M) = mod(M, 2^l) * 2^l + ((v - mod(v, 2^l))/2^l)$$

Where M is the decimal value of the node whose neighbor we are looking for, l is the string length for the subspace (the length of the complete string divided by 2) and $mod(x, y)$ is the remainder of x/y.

Given the function defined above, which maintains symmetry, the same algorithms used for the standard hypercube can be used for the partitioned hypercube.

6.3.3 Partitioning With Multiple Connections

In addition to simple partitioning, we can further reduce the number of steps we need to find a value within each partition by "targeting" the jump into the next partition more accurately. Using 4 connections between each partition (if the connections are properly placed) ensures that any node, at any position within those partitions, will have to do no more than 1/2 steps necessary to reach the appropriate node.

6.3.4 Combining the Techniques

Now if we combine multiple-connection partitioning with increased number of connections in each partition, we can reduce the number of steps. For example, if we divide a 64-bit string by 2 we'd need 65 connections but now a search would take $16 + 1 + 16$, or 33 steps. With this approach, while the number of active connections has remained basically unchanged, we have cut the maximum number of steps for any search by half.

We can thus combine recursive partitioning with increased number of connections to substantially improve on the upper bounds of the logarithmic approach. In particular, we divide the 64-bit space (resulting from hashing name strings into 64-bit strings) twice, giving us a base subspace of 16 bits.

We create 32 connections in the level-1 subspace of 16 bits, giving us a maximum of 8 steps. Then we use 2 sets of 4 connections (one set for each of the higher-dimensional subspaces) giving us a total of 40. A search would then take, at most $4+1+4+1+4+1+4 = 19$ steps. With this approach, we have achieved less than 2/3 the number of connections and almost 1/4 the upper bound of the number of steps compared to the logarithmic approach discussed in the basic algorithm.

6.4 Adapting the Overlay Network to the Physical Topology

The Manifold-g algorithm defined so far is efficient, but it doesn't take into account actual network topology, since in the overlay network a node's neighbors could potentially be located on the opposite end of the physical network. In fast networks this wouldn't be a problem, but when slow connections are involved (such as low-speed modem connections, or, as in the example presented in the next chapter with wireless connections, where the number of collisions will increase if traffic is constantly crossing the entire network) this issue can't be ignored, since it can result in search times much slower than those implied by the number of steps. For example, in a network with 64 nodes, the algorithm defines at most 6 steps for the search ($2^6 = 64$). But if by chance each node's neighbors are located in distant points of the physical network (for example, requiring 500 msec to get from one point to the other) then a query could potentially take 3000 msec to resolve, which is not acceptable from the user's point of view in such a small network.

An algorithm that ignores network topology is also forgoing the potential speed improvement that highly connected nodes can bring. As it has been pointed out in [30], real-world networks tend to create so-called power-law topologies, where a few nodes connect to many more nodes than the average number of connections per node, and most nodes have less connections than the average. If we could achieve a closer mapping to the physical network topology, we would not only improve the response time by reducing the physical distance between a node and its neighbors, but we would also be exploiting more efficiently the power-law qualities of the physical network topology. Similar work has been done for other overlay topologies, for example [9] [34] [67].

6.4.1 An Abstraction of Physical Distance

The first step towards adapting the overlay network topology to the physical topology is to be able to quantify the physical "distance" between nodes. This has to be done in terms relative to the network at hand, since different networks will have different parameters to define the distance between nodes. The distance function will then have to be implemented in particular for each network type, according to certain constraints.

6.4.2 The Distance Function

We define the physical distance function γ as

$$\gamma(N, M) = V$$

Where N and M are two physical nodes on the network that are connected to the overlay network, and V is the resulting distance value. This distance function must also satisfy:

- The distance V of a node to itself must be zero:

$$\gamma(N, M) = 0 \iff N = M$$

- The distance between two different nodes must be positive and non-zero:

$$\gamma(N, M) > 0 \iff N \neq M$$

- If a node N is connected to another node M through a unique path that includes node J, then their relative distances must satisfy:

$$\gamma(N, M) = \gamma(N, J) + \gamma(J, M)$$

  with both $\gamma(N, J) > 0$ and $\gamma(J, M) > 0$. The constraint of this previous equation must satisfy an equality relationship ($=$) rather than one less-constrained of lower-or-equal ($\leq$) because the distances are part of a discrete domain, i.e., arbitrarily-defined distances between nodes.

Sample Distance Functions

The distance function γ is by nature network dependent, and it can be implemented in different ways, as long as it satisfies the conditions specified in the previous section. Some examples are:

- for IP networks the distance could define whether the nodes are in different class A, B, C or D networks, a value that is immediately available between two nodes.
- for ad hoc wireless mobile networks, the distance could be the number of hops. Example: if to get to node X node A requires 3 hops then the distance is 3. So any physical neighbor of the node would be at distance 1.
- in general, distance can be calculated not physically (number of hops) or "geographically" (IP network value differences) but by testing the connection speed between the two nodes in question. This value is more representative, since two nodes might be distant but connected through high speed lines, which might make it more effective to use that node as a neighbor rather than a node that is closer physically but connected through a much slower channel.

For mobile networks, and wireless networks in particular, the definition of the distance function becomes important. The definition for distance as number of hops is more appropriate for mobile networks because as the nodes' positions change relative to each other, so will the distance, since the hops required will also change. The first distance function, defined for a fixed IP network, would not be useful in this case since the distance would remain fixed even if the nodes changed relative positions, and therefore routing paths.

6.4.3 The Distance-Based Algorithm

With the distance function defined, it is possible to improve the connection algorithm described earlier. We should keep in mind that this improvement is more related to implementation (and real-world constraints) rather than theoretical need, and in fact it is completely separate from the original algorithm, which can choose whether or not to implement it. The distance-based algorithm can be applied to both Manifold-b and Manifold-g, although we will focus our discussion on its use for Manifold-g as it is the overlay that will benefit the most from the advantages it provides.

We are looking for certain qualities in improving the original algorithm, namely:

- We want to maintain *locality*: the distance-based algorithm should not rely on global information, or on server-based approaches.
- Since the distance-based algorithm is not intrinsically tied to the basic search algorithm, the search algorithm should not need any modification to support this improvement.

Applying the Distance Function

We define the following rule to adapt the overlay topology to the underlying physical network:

1. When a node M contacts a neighbor N in the process of a search (which include searches related to *join* operations), it will request the neighbor's N distance to its own neighbors, as well as their neighbor's addresses $N_1, N_2, \ldots, N_i$. With this information, it can calculate the distances to those nodes. If the distance to one of those nodes is such that

   $$\gamma(N, M) < \gamma(N, N_i) + \gamma(M, N_i)$$

   (where N_i is neighbor i of node N) node M will define N_i as a *node proxy* for its connection with N. N_i will then cache N's information, and M will contact N_i whenever a search's next step would be N. The proxy is *symmetrical*, so it will proxy requests both for M in relation to N, and viceversa.

The proxy node, as defined, is loosely coupled with the nodes it is serving: it only needs to be informed of changes in the neighbors state either N or

M. When a query is answered by a proxy rather than by the actual target node, an identifier will be added to the reply to specify that that is the case. This is since the calculations for distance-based optimization have to be done against the original node rather than the proxy (although the proxy's values could also be taken into account).

To prevent excessive proxy-load, a node can deny the request to become a proxy, forcing the requesting node to look for another suitable node to use as proxy, or to skip using a proxy altogether for that particular neighbor.

To better understand how this distance-based algorithm works, let's consider the sample (physical) network topology from Figure 6.4.

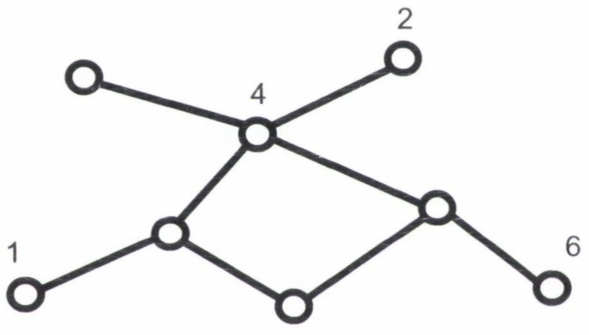

Figure 6.4: A Sample Physical Topology

In Figure 6.4, we have several nodes of which only 4 are enabled to join the overlay network, with values 1, 2, 4 and 6. (Note that the numbers represent the "names" of the nodes, and not their physical address).

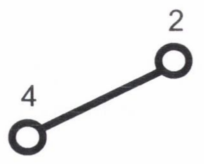

Figure 6.5: A Two-Node Overlay Network

Initially only nodes 2 and 4 are active, creating the overlay topology seen in Figure 6.5. We should note that the nodes are "covering" for other nodes to create the complete boolean space.

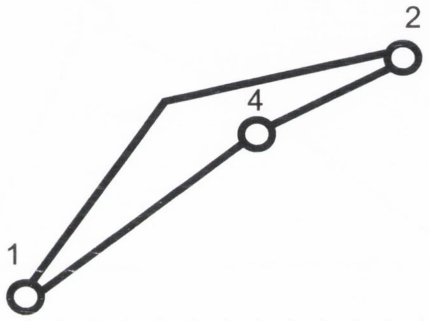

Figure 6.6: Node 1 Joins the Network

Then, node 1 joins the network (Figure 6.6), connecting to 4 (since 4 is responsible for shadowing nodes 3 and 5, both of which are neighbors of 1) and connecting to 2 as its existing neighbor.

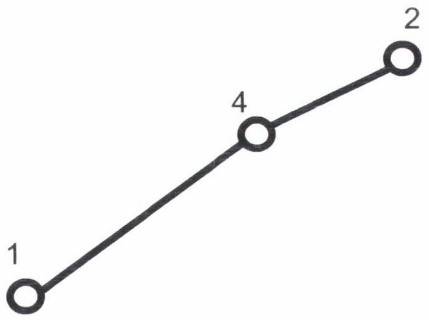

Figure 6.7: Node 4 Joins the Network

After the initial connection happens, node 1 requests information regarding 2's neighbors to calculate its distance in relation to them. 2 will relay 4's information and 1 will determine that 4 satisfies the proxy-distance condition, and thus request that 4 operate as a proxy, creating the final overlay topology seen in figure Figure 6.7.

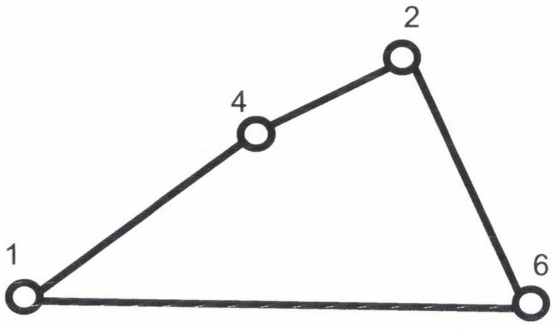

Figure 6.8: Optimized Overlay Topology

Finally, if Node 6 joins the network, it will require one connection to each node. Node 2 will be able to act as proxy for the connections between node 6 and node 4, since it is physically closer. Node 1 will require a direct connection to 6. The final result is shown in Figure 6.8.

6.4.4 Distance and Network Adaptability

It is important to note that while this improvement does not affect the algorithm's complexity, it does change the balance of the load in the network. In particular, adapting the topology to connection speeds shifts load into "hub" nodes that are connected to many others through high speed lines. This is desirable, since those nodes are the most likely to be powerful machines that are already performing server- or proxy-like functions. In essence, a properly applied distance function will map the overlay topology not just to the physical network, but to a power-law network, allowing the algorithm to use qualities found both in structured overlay networks and power-law networks.The adaptability achieved by this algorithm is clearly sub-optimal. Over time, however, connections will adapt better to the underlying network topology.

Additionally, the use of proxies is important for low-power nodes. Regardless of the distance function, low-powered nodes could use proxies to off-load some or all of their responsibilities to other nodes that are better able to handle them, improving network performance and resilience.

Chapter 7

Manifold: An Implementation

7.1 Introduction

The initial implementation of the Manifold system was targeted to run on network conditions where speed and self-organization are the most critical: on a mobile ad hoc network. Since all types of networks are increasingly acquiring properties found, until now, only on MANETs, using a MANET allowed us to provide a proof-of-concept environment that tests the various requirements found for general RLD.

We implemented the algorithms to run on a system based on the NTRG Stack [62], which provided us with a pre-existing set of components that supported mobile ad hoc networks both in wireless and wired configurations, and allowed us to focus exclusively on the Manifold implementation rather than solving other problems related to MANETs, such as routing. The NTRG stack is simple and extensible, and it can incorporate different routing algorithm implementations as well as physical connectivity layers, including software radio and 802.11. The stack is also supported by the JEmu [22] radio emulator, greatly simplifying creating and running tests scenarios of several devices for debugging and evaluation.

The NTRG stack uses the concept of *layers* to define abstraction boundaries that simplify implementing different elements of a complete ad hoc stack can be implemented, including low-level connectivity, routing and security, among others, and we implemented the Manifold algorithm to match those requirements.

7.2 Considerations for Mobile Ad Hoc Networks

We will now cover some of the special considerations required for a wireless ad hoc implementation of Manifold. Special considerations might not be necessary if the MANET is emulating a protocol such as TCP/IP, but it is necessary in other cases, for example, when the ad hoc routing protocol in the MANET does not ensure reliability (a common occurrence). Additionally, MANETs, because of their "broadcast" nature, benefit from reduced message traffic and at the same time enable certain optimizations that wouldn't be possible with other physical environments.

7.2.1 Routing and Location

Routing in MANETs has been an active area of research in recent years. Interestingly, elements of the resource location problem surface on ad hoc routing algorithms since the concept of neighbors has a physical counterpart in wireless networks. In particular dynamic ad hoc routing protocols such as Dynamic Source Routing, or DSR [38] perform several functions that bear a strong resemblance to some of those performed by location services. Another example is the Grid Location Service, GLS [45], in which location services can interact closely with the routing protocol, providing solutions that adapt better to certain circumstances. Our Manifold implementation, however, makes no assumptions regarding the type of routing used in the MANET.

7.2.2 Scalability

Because of the broadcast nature of MANETs, Manifold-g is at a disadvantage with respect to Manifold-b, which is a broadcast algorithm and so naturally suited to take advantage of this particular physical transport environment. This means that for groups of nodes *that are operating autonomously* and that are concentrated in a small area (which can be defined as an area small enough so that any node is within two hops of any other), Manifold-b will always be faster, and require less network resources, than Manifold-g. Because of this, applications operating in that kind of environment would be better off by always sending queries to be processed by Manifold-b, even in cases when an exact match is required.

7.2.3 Security

As we have mentioned in Section 2.4.7, Manifold currently leaves the verification of the identity of the node location result to higher level application layers with enough information to make these checks (for example, by verifying signed security certificates), just like DNS does. In the future, Manifold could be extended so that initial verifications can be made directly at the resource-location level by using cryptographic digital signatures, using decentralized trust systems such as the PGP Web of Trust [5] and SDSI [68]. (Security and trust-based modifications to DNS are currently under consideration at the IETF [70]).

7.2.4 The Namespace

The *namespace* of a resource location protocol is the standard used to identify resources in the network. In DNS [55] [56], the namespace is defined by case-insensitive alphanumeric combinations of a maximum length of 255 bytes, by convention ending in a particular string that identifies root domain, such as "net" or "com" and they are part of the protocol definition, i.e., DNS can only resolve queries for resources whose names exist in its namespace. Achieving a single, consistent namespace definition was actually one of the primary goals in the development of DNS.

Since Manifold is designed to operate in heterogeneous networks and support different applications including presence and machine location, enforcing a DNS-style namespace is not acceptable. Manifold resolves any string stored as a mapping in its internal database for a particular node. Applications can then make use of it in different ways, for example, storing DNS-style names or using SIP [71]. The application can simply store the mapping desired and other applications running on different nodes will be able to resolve it regardless of the type.

Mapping a String Space to a Boolean Space

For Manifold-g, we will use hashing (as discussed in Section 6.2) to normalize the namespace and reduce the string size.

Although not likely, it is possible that the hashed value of two different strings return the same value. If two names (corresponding to different nodes) are hashed to the same value, the nodes evenly divide the task of managing that particular vertex in the hypercube; if one of the nodes leaves the remaining node will take over all of its tasks.

For this application, we will then hash name strings into a 16-bit value, resulting in a hypercube of degree[1] 16.

7.2.5 Prerequisites and assumptions

All nodes are named

One of the algorithm implicit assumptions is that a search has to occur with a node that is already on the network as a starting point. In our implementation, it follows then that a node has to insert itself into the network before

[1]dimensions

it's able to run a query in it. Nodes that are not inserted into the network will be able to use a node that is in the network as a *query proxy* to obtain results. This means that for a node to be able to initiate a search without a query-proxy, it has to be connected to the hypercube, and therefore has to have a name assigned. Furthermore, the name has to be *unique*, as specified next.

All names are unique

In the example of name resolution, having a unique name per node not a problem. Name resolution depends precisely on every node having a unique name. For the example at hand, of name resolution in an autonomous ad hoc network, we can specify that names are chosen by the machine's owner, who will be notified if the name already exists in the network, therefore requesting that the owner choose another name.

Bootstrapping

To be able to join the network, a node must also have access to a bootstrapping mechanism as defined in Section 2.7.1. The out-of-band method must provide the node with one or more active nodes in the network. It is important to node that *any* node in the network will do, since the network will maintain full connectivity as it grows, all nodes already in the network will be able to provide the new node joining with enough information to insert itself in the appropriate position.

For this example, we will choose broadcast as bootstrapping method.

Full node connectivity

In the case of the implementation, full node connectivity implies that any node has to be able to establish a direct connection to any other node by using the target's physical network name (which is the IP number in IP-based networks). This implies an unbroken routing chain between all the nodes. Later we will consider how networks with partial routing might still be able to resolve queries by routing the reply through the hypercube network itself. However, this solution problem of name resolution assumes that a direct connection will be established between the nodes after the name is resolved, so the requirement of full routing between any two nodes is not spurious.

7.3 The Manifold Layer

The current version of the Manifold library is built for Windows operating systems. It uses the lowest common denominator of Windows-based API functions (Win32 base) so it is portable across a variety of Windows-based systems, including Windows 98, Windows 2000 and Windows-CE based platforms such as Microsoft PocketPC. Porting the library to other operating systems or platforms should not be difficult due to the relatively simple nature of the operations required (i.e., thread management, local and network I/O) and because most of the platform-dependent functions are abstracted into a set of function calls that are implemented as necessary for each platform on which Manifold is deployed.

The implementation has been tested both on real-world ad hoc wireless networks with a few nodes and on large networks with the JEmu [22] ad hoc simulator, both concentrated on small areas and spread over large distances.

7.3.1 A High Level View

From a high-level perspective, the Manifold design is relatively simple:

Figure 7.1 shows the basic components: a *Local message manager* that connects Manifold with applications in the local device, the *Algorithm Manager* (AM) that controls message flow between local/remote calls and the specific algorithms, each of the algorithm cores, and the *Network Interface Library*, which abstracts network calls so that Manifold remains independent

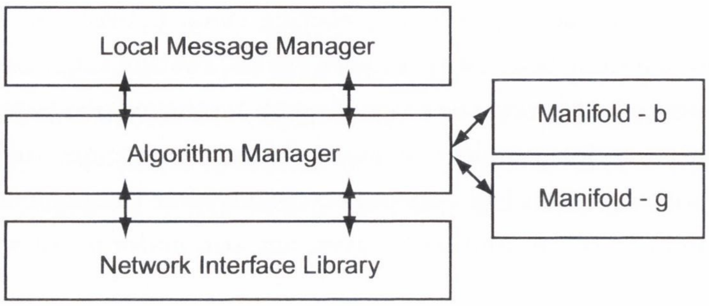

Figure 7.1: High Level Block Diagram of Manifold

of any particular underlying physical network implementation. Each block (including the algorithms) runs independent of the others. The algorithms themselves interact with the local application and the network (that is, with other Manifold nodes) only through the AM, keeping the algorithm code abstract and ensuring that in the future new algorithms (or variations of the current algorithms) can be plugged in for testing and deployment without requiring modifications either on the internal Manifold code, or on the applications that use it. Additionally, since the AM manages the algorithms, it can sometimes bridge between them. For example, if there was a catastrophic failure on the Manifold-g overlay structure, while it is being rebuilt the AM can route requests through Manifold-b, which has lower consistency requirements, to provide best-effort results even if the overlay is temporarily not available.

Manifold is intended to be used by local applications in a "black-box" fashion: the application needs to resolve a name or pattern to a machine or series of machines that can match that name, and it calls the Manifold library which eventually returns the result, or a failure. While some of the parameters of the Manifold-g and Manifold-b topologies might be available for configuration by applications, behavior of Manifold is largely opaque.

Furthermore, each Manifold node is opaque to all others, operating independently except for maintaining certain lists of known neighbors for each of the networks.

One of the important elements of implementing Manifold-g in ad hoc networks (wireless ad hoc networks in particular) is to make the operations *atomic*. This is necessary to avoid relying on the concept of connections, which might not be available in particular ad hoc network implementations. Everything is done in a single atomic step on a per-node basis.

When a Manifold node initializes, it sends out requests for neighbors including its local identifier. The neighbors then update their tables accordingly and reply to the incoming node with their updated tables (including, in the case of Manifold-g, shadow nodes for which the incoming neighbor must assume coverage). This reduces node-joins to an efficient two-step process that minimizes the probability of disruption of connectivity.

The core of Manifold is controlled by two independent algorithms that process messages, one at a time, received from an internal message manager. Through the manager, they have the ability to send messages to the local (requester) application as well, or use the network interface to send messages to other Manifold nodes (specifying which algorithm on the target node will be responsible for processing the received message).

Manifold nodes connect to each other through the use of *messages* that can be received through any of the standard message-passing mechanisms used in software today: Inter-Process Communication (IPC), API method calls, RPC [78], local message passing, and so on. Distinctions between messages that arrive from the network and messages that arrive from local applications are only made when a result has been found, since network messages must be routed through one path (to return the reply to the query originator) and local requests must be replied directly through the local communication mechanisms.

7.3.2 Messages

Messages passed between nodes contain information relevant to the task at hand (i.e., Join, Query, Leave, etc), for example:

- The string being searched for, both in binary and alphanumeric form.
- Physical address of the node that originated the query.
- Physical address and name of the first node in the network to receive the query (useful when that node acted as query-proxy for another node outside the network).
- List of all the nodes that the query has "visited" in order.

Once a message is received the algorithm responsible for the message is chosen dynamically (through polymorphism), and processed. The behavior of the algorithms is largely stateless, that is, the algorithms receives a message (which carries its own state, such as nodes visited in the query path, etc), processes it, and continues appropriately, either replying to the query, forwarding the message, or ignoring it. The only state maintained, having to do with the structure of the P2P networks used, is query-independent.

Message Format

For messages, Manifold uses an XML [7] based format. XML was chosen because of its platform independence and ease of parsing while maintaining readability, which aids in debugging and in creating additional implementations. Additionally, XML is easily extensible, while maintaining backward compatibility.

Details on the specific XML message formats used by Manifold can be found in Appendix B.

7.4 Operations

7.4.1 Operations: Node Join

When joining the network, a node first has to collect the necessary information, i.e., find the location in the networks of its would-be neighbors. Once the information is collected, it can initiate a join process. The join process has to be almost atomic from the point of view of the network, since a node in the middle of a join process essentially disrupts the connectivity structure and query paths have to be halted while a join is in progress. Therefore a node first obtains all the information and then performs all necessary changes to the network.

Node initialization steps:

- do a broadcast of neighbors in range for the incoming node S_N, looking for Manifold nodes, including a search for each of the neighbors (for Manifold-g), which essentially uses the receiving node as query-proxy to perform queries for S_N's neighbors.
- those nodes that receive the request will reply to S_N with their information. The information comes separately for both Manifold-b and Manifold-g.

For Manifold-g,

- On each node that receives the request, add the node to the its neighbors list modifying shadows as necessary.
- On the node that receives the replies, S_N, for each reply, modify S_N's shadow list, and neighbor list according to the replies received from each neighbor S_{N_i}, as follows:
  - If S_{N_i} is the actual node that S_N searched for, assign it as a neighbor and assign ourselves as neighbor of that node. Because

    S_{N_i} had to have full connectivity, there had to be a node T covering in S_N's place. So S_N must also notify other nodes that were connected to S_N's "shadow" in T and take control of its name

  - If the node obtained is a shadow node, assign it as a shadow neighbor and add ourselves to the list of neighbors for S_{N_i}'s shadow name on the responding node, T.
  - Negotiate the takeover of additional shadow nodes with the responding node T (whether its the target node, or a shadow node), since that node might be covering for shadow nodes that belong to L's "shadow space." This can be determined simply, as per the space-complete algorithm described in the previous chapter, by verifying the decimal value of each shadow node in that neighbor. If the numeric value of a shadow node P is such that $D(T) < D(P) \leq D(S_N)$ then S_N should take control of that shadow node, which involves contacting the neighbors for that shadow node and updating their information.

- On each algorithm, once all neighbors have been gathered, switch state to initialized=true.

At the beginning the shadow node list of any node will be only its neighbors mapped to itself. As new nodes enter the network, it will be re-balanced. Shadow node update operations decrease as new nodes come online and the hypercube topology is completed.

For Manifold-b, nodes are simply added to the Manifold-b neighbor list. Note that the neighbor lists for Manifold-b and Manifold-g are different and will likely have little overlap.

7.4.2 Operations: Node Leave

The algorithm for a node leaving the network reverses the process of joining the network. Essentially, a node leaving the network will notify its neighbors of the fact, and with each of them to return the shadow nodes they were covering for (if any). Also, each node that is being disconnected from the node leaving the network will have to perform a "partial join," i.e., it will have to search for the node that is leaving after it's left and connect to the shadow node that responds, thus maintaining full connectivity.

Node Failure

When a node or its network connection fail, its neighbors might not be notified of the failure. P2P algorithms thus have to be aware of potential gaps gaps in the topology, and be ready to rebuild it when necessary.

Manifold-b treats node failures in a "lazy" fashion. No active polling is made to ensure that neighbors exist. However, if a query has to be forwarded and the target node does not reply, Manifold-b will perform a partial or complete *Join* operation to restore its connectivity, and continue the transfer after the operation is complete and connectivity is restored.
Manifold-g

operates in a similar way (by initiating a partial *Join process*), which is appropriate for non-simultaneous node failures. However, if multiple node failures have occurred simultaneously and the network has not had time to recover, it is possible that the appropriate value might not be found on a query. While the Manifold-g structure is rebuilding, Manifold-g attempts best-effort resolution by forwarding the queries through Manifold-b.

7.4.3 Operations: Search

Search on Manifold chooses the algorithm based on the type of query received, exact, or inexact. A query into a Manifold node can be received in two ways, irrespective of which algorithm will process it:

- If software running in the node has made the query

- If the query has been received (i.e., forwarded) from another node as part of the query routing process.

As far as the node receiving the query is concerned, these two events are indistinguishable, and are processed in the same fashion.

For inexact queries Manifold-b is used.

For exact queries Manifold-g is used. The algorithm for a node N receiving a query Q contains the following steps:

1. add N to the list of nodes traversed by Q.

2. if the query matches N's name, reply to the origin node with N as a response. Otherwise, continue with the next step.

3. if the query matches one of the shadow nodes N is covering for, reply with N as a response. Otherwise, continue with the next step.

4. if the query matches one of N's neighbors, forward the query to that neighbor; the local loop will be finished and it will start in that neighbor. Otherwise, continue with the next step.

5. iterate through N's neighbors and N's shadow neighbors (i.e., nodes that are neighbors of a node that N is covering for) searching for the minimum distance between each neighbor/shadow neighbor N_x and the target string $\lceil \delta(N_x, T) \rceil = \delta_x \quad \forall \ \delta(N_x, T) < \delta(N, T)$, that is for all the distances that are less than the current distance. If a node is found, forward the query to that node; the local loop will be finished and it will start in that node. Otherwise, continue with the next step.

6. If this step has been reached, it means that a path can't be found to the target node of the query, and so that target is not on the network. However, if the query message has reached N, it means that either N or one of its neighbors is the "closest" topologically to the target node, and therefore should reply for that node (therefore specifying that the

target doesn't exist and they are covering for them). For this, N will check with each of its neighbors (and shadow neighbors) to see which of them has authority over the target name's value by comparing the decimal value of each neighbor of the current neighbor in control C (which initially is N) to each of the neighbors of N such that in the end $dec(N_i) < dec(Q) \leq dec(C)$ for i any of the positions in a list that includes N and all of its neighbors. Finally, N will reply to the origin node with C as a response, while C will take over for covering that node's value.

These basic steps force the algorithm to navigate the hypercube topology as if it was complete even though it is not, thus satisfying the space-complete search algorithm defined above.

7.5 Class/Flow Diagram

Figure 7.2 shows the relationship between the main Manifold C++ classes, along with the main data flow paths[2].

The core of Manifold is the message processing loop, which detects incoming messages (both from applications and from the network interface) and processes them if appropriate. When a message is received, the main message loops detects whether it is a Manifold message, and if so it parses it (i.e., "deserializes" it from its platform-independent XML representation) and creates an internal Manifold *Message* subclass that is passed on to the *ConnectionManager*. The *ConnectionManager* routes the *Message* to the appropriate *Algorithm* subclass, based on its type[3]. The *Algorithm* subclass

[2]Only the main classes used in the implementation are shown in the figure. The full implementation consists of 50 C++ classes comprising approximately 6,000 lines of code. Both the Manifold-b and Manifold-g simulators are—although built on completely different code-bases due to their different purpose— each on its own roughly similar in complexity to the implementation.

[3]Messages can be either standard messages or "control" messages, used for Join/Leave

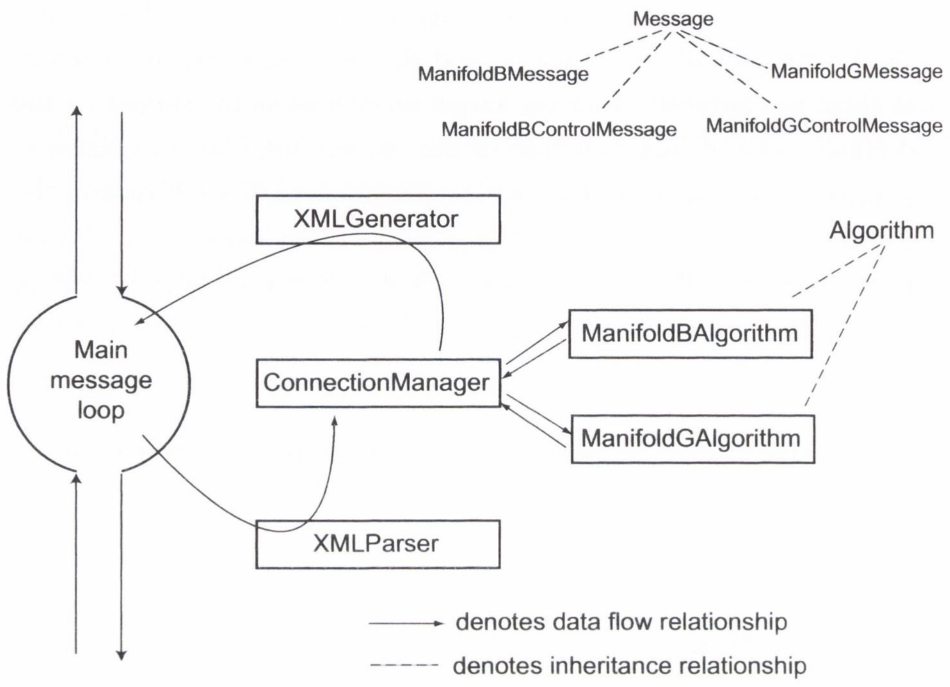

Figure 7.2: Manifold Class/FlowDiagram

then decides whether to ignore, forward, or reply to the message, and generates a new internal message that is passed back to the *ConnectionManager*, which will "serialize" it back into platform-independent XML and forward it to the main message loop so that it can be sent to either the network interface or back to the application.

7.6 Implementation Results

The main objective of our Manifold implementation was to prove that the system could work, beyond the simulations we had already performed, in a

operations. Control messages differ from standard messages in that they don't contain query-related data real-world environment.

The test environment presented by our ad hoc network was that of a few nodes distributed over a relatively small geographic area (the Trinity College Campus). While both Manifold-g and Manifold-b were tested and used, the size of the network prevented large-scale performance testing of the algorithms[4]. We could, however, prove that it delivered timely responses to user requests, which is an important subjective test any technology must pass. Tests were run both with "pure" networks composed only of devices and with networks that mixed devices and simulated nodes, to further verify these subjective results.

The Manifold implementation was thus successful in proving that the system could work. Not only basic Manifold tests worked correctly, but the Manifold layer was quickly used as part of two higher-level applications, as described below.

7.7 Applications

The two main applications that currently make use of the Manifold layer on the NTRG Stack are the 4GPhone [61] and an Instant Messaging (IM) application.

The applications initialize the Manifold layer to map the name of the owner of the device to the physical ID of the node in the ad hoc network. The application can then make a function call to the resource location layer with the requested username as a query, and waits asynchronously for Manifold to return the result. In this way, the requested name (the owner of the machine) is dynamically mapped to the physical node ID for the device, in this case D4. With the physical address resolved, the ad hoc routing protocol (e.e.,

[4]While it was possible to simulate nodes in the network, that was equivalent to running simulated networks of a large size, which defeated the purpose of testing in a real-world environment. Further testing in large-scale real-world networks would be a key element of any future work on Manifold, as outlined below in the conclusions

DSR) is now able to establish a route to that destination.

Applications also use the Manifold layer directly or indirectly. The 4GPhone, for example, allows a user to directly input the name of the person to be called and keep a list of recently-called users, while the IM application creates a *presence* layer that uses Manifold to maintain state on the users by periodically updating their location on the network. One location is established, the actual communication can then begin without the need for centralized management of mappings or configurations.

Subjective tests of several nodes running on both the JEmu emulator and actual devices (both PCs and handheld devices) have shown that Manifold is an appropriate solution for this type of usage, particularly because of the "disconnected" nature of the networks formed, usually without access to centralized infrastructure, which would preclude the use of other solutions.

The implications of the types of applications enabled by Manifold are profound. For example, voice or message communications are currently routed through centralized services within a particular network (e.g., the POTS[5], or a celular network). Interoperability between these networks is complex and expensive, and users are generally limited to using one type of device (e.g., a cell phone, or several phones connected to a single landline). A generic RLD system such as Manifold would allow networks of different types to locate users and establish communications independent of network provider or device used, allowing to separate the directory functions performed by, for example, phone service providers, from their functions as data-carriers. Additionally, as we have seen in Section 2.5, RLD also has applicability to several other areas, such as mobility or routing. New types of applications, such as collaboration systems that operate globally between different types of devices and over various networks, would also be possible.

[5]"Plain Old Telephone System"

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Resource Location and Discovery is a key service of networked systems. The evolution of networked systems however, in particular of the Internet, has occurred at a faster pace than many of the services that must support them. This is placing increased strain on a largely centralized infrastructure that finds it difficult to adapt to the new conditions of mobility, rapidly changing topological dynamics, and multiplicity of physical transports and network conditions. At the same time, the increasingly ad hoc nature of the Internet is making RLD systems more critical than ever, with the result that applications have to resort to deploying specialized RLD solutions that are not compatible and that are still dependent to various degrees on centralized infrastructure.

Additionally, there hadn't been to date an analysis of the usage patterns and requirements that drive generic RLD across all types of networks. These requirements pointed to a system that maintained scalability, was able to provide results for both inexact (local) and exact (global) queries, could function equally well in small ad hoc groups and on the Internet, without depending on expensive, complex centralized infrastructure.

Our solution to this problem is Manifold, a generic, network-independent RLD service based on two self-organizing algorithms that allowed us to provide resource location for networked systems in general (and self-organizing networks, such as MANETs, in particular) based on those requirements. The first algorithm, Manifold-b, provides local/inexact RLD, while the second, Manifold-g provides global/exact RLD, supporting guaranteed results to have an upper bound of $O(\log n)$ steps (with n the number of nodes in the network). We further proposed a number of optional improvements to the algorithms (in particular for Manifold-g) that could, under certain circumstances, provide better performance and adaptability to the topologies created by the Manifold algorithms.

Our analysis of Manifold and subsequent proof-of-concept implementation for mobile ad hoc networks shows that those requirements were not only desirable but also achievable, and that the resulting generic system is useful for a variety of different applications, such as voice communication and instant messaging.

Manifold and the concepts on which it is based have the potential to create a uniform, global infrastructure that provides scalable, self-organizing RLD services at all levels of the network stack, simplifying support for newly dynamic networked systems that potentially cover the entire global network, and enabling new types of applications for the future.

8.2 Summary of Contributions

The main contributions of this thesis are:

- an analysis of the problem of Resource Location and Discovery in the context of heterogeneous networks, current systems, and a clear identification of the usage patterns and basic requirements generic RLD must satisfy, regardless of design or implementation,

- a system, *Manifold*, that satisfies those requirements, providing RLD for heterogeneous network environments in general and for wireless ad hoc networks in particular, and a set of self-organizing algorithms to support it (including analysis and quantification of their limits), and, finally,

- a demonstration of the feasibility of the system by implementing it in a real-world wireless ad hoc network, and verifying its applicability by actual use in real-world applications.

8.3 Future Work

Manifold holds great potential for future work, both theoretical and practical. Specifically, the next step should be its deployment on an actual large-scale network of millions of nodes, both mobile and fixed, to provide final measurements beyond those described in this work (which include the theoretical proof, simulations, and small-scale real-world experiments). This would present a number of questions to be resolved that we discuss in the paragraphs that follow.

One important design element of Manifold is its transport independence. While this potentially allows Manifold to bridge different networks, an additional element, not described in this work, would be necessary: a routing bridge. A routing bridge is a software component or device that is connected to two or more different types of networks and can translate requests between them. Since one of the main uses of Manifold is to enable device location to perform certain tasks, such as data transfer or processing requests, it is desirable that this bridge allow not only for the routing of Manifold messages, but also for the transfer of generic packets between the different networks. Even so, this is not required, and only the ability to locate someone or something (even if a direct connection cannot be established) is still useful for presence and other pervasive computing applications that depend to a large degree on "environment awareness."

Ideally, the design of such a bridge would be based on self-organizing concepts as those used for the design of Manifold, allowing the creation of autonomous bridge-clusters that can provide bridge functionality reliably for certain network or networks within range, allowing location of the distributed service through Manifold-b requests. While a bridge is, in fact, a piece of fixed infrastructure, such a connection point would clearly have to be preexisting before its deployment (otherwise there would be no need for it), so no additional requirements would be necessary on the physical network infrastructure to support it.

This concept opens up a number of interesting possibilities, like its potential to also perform some proxy functions such as those defined for the distance-based algorithm discussed in Section 6.4, allowing better adaptation of the virtual topology to highly dynamic physical topologies.

While it is clear that network topologies are increasingly dynamic, the *degree* to which they are dynamic is not obvious. That is, it is not readily apparent how quickly membership changes when using networked devices with both ad hoc- and Internet-capabilities in real-world environments. While we might be tempted to assume that there is no limit to how fast membership can change on a network (e.g., by nodes activating or coming within range of others) the dynamics of network topologies are not completely random or unpredictable: they are closely related to the users' behavior and the operation of the devices that conform them. A deeper understanding of not only the network dynamics but also of the *social* dynamics that drive them would be useful to improve how Manifold (and other applications) are applied in particular contexts.

For example, if the network is highly dynamic and reduced to small numbers of nodes without access to fixed infrastructure, Manifold-b, rather than Manifold-g with appropriate time-to-live parameters on the queries might well be used as the default mechanism for name resolution even when exact resolution is necessary, since the higher consistency requirements of Manifold-g might unfavorably affect performance. On the other hand, if the network is largely centered around Internet connectivity, rather than ad hoc connectivity, Manifold-g will usually be a better option.

Another case to consider is when only reduced groups of nodes with broadband connectivity (i.e., within range of fixed infrastructure) have highly dynamic membership requirements, in which case the use of the previously mentioned proxy might be a mandatory requirement rather than an optional feature.

These questions point to a larger problem. While the theory of network dynamics has been an active topic of research in recent years, we still haven't reached an in-depth understanding of the social and technological factors that affect their operation and behavior. These gaps in knowledge has led us more than once to underestimate the resilience, scalability, and capabilities of systems, such as global music-sharing platforms, and even of the Internet itself.

A better understanding of the governing dynamics for networked systems, both from a theoretical as well as from an empirical sense, will open up new avenues of research and allow us to define solutions that fit different problems, allowing the creation of applications that adapt dynamically and autonomously to the increasingly complex demands of our connected world.

Appendix A

Hypercubes: Theory and Properties

A.1 Definitions: the Boolean Space, and the Concept of Distance

The algorithm also depends on the maximum length available to a string that will serve as a search key. For the moment, we will make certain definitions that will later be mapped to a real-world situation. We will make these definitions in the subsections that follow.

A.1.1 Initial Definitions

The algorithm makes use of a Boolean Algebra [4] defined as a set B of elements $a, b, \ldots$ such that:

- B has two boolean operations, AND (symbolized by $\wedge$) and OR (symbolized by $\vee$). These operations:

- Satisfy the *idempotent* laws:

$$a \wedge a = a \vee a = a$$

- Satisfy the *commutative* laws:

$$a \wedge b = b \wedge a$$

$$a \vee b = b \vee a$$

- Satisfy the *associative* laws:

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c$$

$$a \vee (b \vee c) = (a \vee b) \vee c$$

- Satisfy the *absorption* law:

$$a \wedge (a \vee b) = a \vee (a \wedge b) = a$$

- They are *mutually distributive*:

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$$

$$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$$

• B contains universal bounds $\emptyset$ and I which satisfy:

$$\emptyset \wedge a = \emptyset$$

$$\emptyset \vee a = a$$

$$I \wedge a = a$$

$$I \vee a = I$$

- B has a unary operation $a \to a'$ of complementation which obeys the laws:

$$a \wedge a' = \emptyset$$

$$a \vee a' = I$$

For this operation we will use the notation '!'. So $!a = a'$ and $a =!a'$.

An element of such an algebra has the general form of $a_1, a_2, ...a_n$, where n is the maximum length of the elements, and components a_i are one of $0, 1$. Throughout this work, unless otherwise noted, one such element will be from here on referred as a *boolean string*, *boolean value* or simply a *value*, using the notation S or S_n. The notation S_n refers then to a particular string, such as S_1 or S_2, and it should not be mistaken by a reference to a particular value in the string, which will always be referred to as $S_{(i)}$.

In this context, we will define the XOR function (which we will denote as $\oplus$), also known as the *exclusive disjunction*, which yields true if exactly one (but not both) of the two values are true. Formally, the XOR function can be defined as:

$$S_1 \oplus S_2 = (S_1 \wedge !S_2) \vee (!S_1 \wedge S_2) = (S_1 \vee S_2) \wedge (!S_1 \vee !S_2)$$

Strings considered in this work will have a *fixed length* of n. Below, we will see that this n is also the dimensionality of the space on which we will map our complete set of boolean values. Given that we are dealing with boolean numbers, a string of length n will span all the positive integers between 0 and $2^n - 1$.

The algebra defined also has an absolute upper bound, $sup(B)$, such that $sup(B) \geq S \; \forall \; S \in B$ and an absolute lower bound $inf(B)$ such that $inf(B) \leq S \; \forall \; S \in B$. In the case of our finite boolean algebra, these values can be readily calculated , they are: $inf(B) = 0$ and $sup(B) = 2^n - 1$.

Finally, we will sometimes use the decimal mapping (i.e., the boolean string considered as a number and mapped to a decimal value), which we will refer to as $dec(S)$. Formally, $D(S)$ is defined as:

$$D(S) = \sum_{i=0}^{n}(2^i S_{(i)})$$

Where n is, as mentioned before, the length of the boolean string in question, i.e., the number of boolean characters in the string.

A.1.2 Hamming Distance

The total number of positions at which these strings differ is referred to as the *Hamming distance* between the strings. The Hamming distance δ can be calculated as follows:

$$\delta(S,T) = \sum_{i=0}^{n} |S_{(i)} - T_{(i)}|$$

where n is the length of the bit-string, ie., the dimensionality of the hypercube (diameter?)

S and T are two bit strings (that define a position in the hypercube).

Note that:

$$\delta(S,T) = \delta(T,S) = 0 \quad \Longleftrightarrow \quad T = S$$

$$\delta(S,T) = \delta(T,S) \quad \forall \ S,T \in X$$

$$\delta(S,T) \leq \delta(S,Z) + \delta(T,Z) \quad \forall \ S,T,Z \in X$$

and so $\delta(S,T)$ can be considered a *metric* [8] on the set of binary strings X. $X + \delta$ defines, therefore, a metric space.

A.1.3 The Boolean Algebra as Hypercube

Now let the labels in vertices in a hypercube be bit-strings, as directed by their euclidean coordinates, defined as a set of points C_l in R_l whose coordinates are all 0 or 1, i.e., the set of vertices of the unit l-cube (a hypercube of l dimensions). These labels form a sequence of bit-strings that define a *set* of bit-strings that matches that of the algebra previously defined.

The Hamming distance δ between two vertices of a hypercube is the number of coordinates at which the two vertices differ [31]. $\delta(S, T)$ is also equal to the shortest path in the hypercube (minimal path) between any two strings, S and T.

Incidentally, δ is called a *manhattan distance* in Euclidean metric spaces.

Throughout the text, we will refer to l interchangeably as the length of the string or the dimensionality of the hypercube in question.

A.1.4 String Uniqueness

For the purposes of the algorithm, we will consider each string unique. That is to say, if a certain boolean string S_1 is composed of a particular subset of elements $a_1, a_2, ...a_n$ (where n is the maximum length of the boolean string, and elements are either value of $0, 1)$ and another string S_2 is composed of elements $b_1, b_2, ...b_n$, if S_1 and S_2 are equal term by term such that $a_i = b_i$ for $n > i > 0$ then we say that $S_1 = S_2$ and they will represent a single position in our search space. Conversely, if $a_i \neq b_i$ for $n > i > 0$ then $S_1 \neq S_2$ and therefore they represent different positions on the search space. In other words, the function that maps our basic boolean space into the search space is *injective* since $f(x) \neq f(y)$ for any $x \neq y$.

Appendix B

Manifold Message Format

B.1 Introduction

In this appendix we will briefly present the message templates used by Manifold for both Manifold-b and Manifold-g queries, as examples of how the messages contain the state necessary to perform operations and of our usage of XML.

Both of the message templates presented here apply to queries. While the examples deal with final replies to a successful query, the same template is used as a query traverses the network in both cases (without including the *results* tag).

B.2 Manifold-b Message Template

```
<manifold-message>
    <mainParameters
        guid="[message guid]"
        type="[identifier for the message type]"/>
    <originNode
        name="[origin node's name]"
        networkID="[physical network id of the origin node]"/>
    <query value="[substring to be matched in the query]"/>
    <results found="true">
        <result
            networkID="[node that replied]"
            matchingString="[first string that matched]"/>
        <result
            networkID="[node that replied]"
            matchingString="[second string that matched]"/>
        ...
        <result
            networkID="[node that replied]"
            matchingString="[nth string that matched]"/>
    </results>
    <hopParameters
        ttl="[this message's time-to-live]"
        hops="[number of hops performed]"/>
</manifold-message>
```

Note that this implies that Manifold-b results for a given query arrive separately from different nodes, as the query is propagated through the network, providing more results back to the caller application on the origin node.

B.3 Manifold-g Message Template

Manifold-g messages are similar to those of Manifold-b, but they add information related to the path traversed so far.

```
<manifold-message>
    <mainParameters
        guid="[message guid]"
        type="[identifier for the message type]"/>
    <originNode
        name="[origin node's name]"
        networkID="[physical network id of the origin node]"/>
    <query value="[string to be matched in the query]"/>
    <visitedNodes>
        <node
            name="[first node name]"
            networkID="[first node id]"/>
        <node
            name="[second node name]"
            networkID="[second node id]"/>
        ...
        <node
            name="[nth node name]"
            networkID="[nth node id]"/>
    </visitedNodes>
    <results found="true">
        <result
            networkID="[node that replied]" matchingString="101"/>
    </results>
</manifold-message>
```

If Manifold-g determines that the path has already been traversed fully but a reply hasn't been found, it replies by setting the *results* tag as follows:

```
<manifold-message>
    ...
    <results found="false"/>
    ...
</manifold-message>
```

Bibliography

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, *The design and implementation of an intentional naming system*, 17th ACM SOSP, December 1999.

[2] R. Bejar B. Krishnamachari, S. B. Wicker, *Phase transition phenomena in wireless ad-hoc networks*, Proceedings of the Symposium ib Ad-hoc Wireless Networks (Globecom), 2001.

[3] S. N. Bhatt and C. E. Leiserson, *How to assemble tree machines*, Proceedings of the fourteenth annual ACM symposium on Theory of computing, ACM Press, 1982, pp. 77–84.

[4] G. Birkhoff and S. Mac Lane, *A survey of modern algebra*, 5th ed., p. 317, Macmillian, New York, 1996.

[5] M. Blaze, J. Feigenbaum, and J. Lacy, *Decentralized trust management*, In Proceedings 1996 IEEE Symposium on Security and Privacy, May 1996, pp. 164–173.

[6] J. Brassil, A. Choudhury, and N. Maxemchuk, *The manhattan street network: a high performance, highly reliable metropolitan area network*, Computer.Networks and ISDN Systems (1994), no. 26, 841–858.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, *Extensible markup language (xml) 1.0 (second edition), w3c recommendation*, World Wide Web Consortium (W3C), October 2000.

[8] V. Bryant, *Metric spaces: Iteration and application*, Cambridge University Press, 1985.

[9] M. Castro, P. Druschel, Y. Hu, and A. Rowstron, *Exploiting network proximity in distributed hash tables*, In Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo), 2002.

[10] D. R. Cheriton and T. P. Mann, *Decentralizing a global naming service for improved performance and fault tolerance*, ACM Transactions on Computer Systems **7** (1989), no. 2, 147–183.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, *Freenet: A distributed anonymous information storage and retrieval system*, ICSI Workshop on Design Issues in Anonymity and Unobservability, June 2000.

[12] R. Cox, A. Muthitacharoen, and R. Morris, *Serving dns using chord*, Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS) (Cambridge, MA), March 2002.

[13] *Triad: a scalable deployable nat-based internet architecture.*, 2001, http://www.dsg.stanford.edu/triad/.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, *Wide-area cooperative storage with cfs*, ACM SOSP, October 2001.

[15] J. Davidson, W. Hathaway, J. Postel, N. Mimno, R. Thomas, and D. Walden, *The arpanet telnet protocol: Its purpose, principles, implementation, and impact on host operating system design*, Proceedings of the fifth data communications symposium, ACM Press, 1977, pp. 4.10–4.18.

[16] M. R. Dempsey and L. H. Goldsmith, *A regular expression pattern matching processor for apl*, Proceedings of the international conference on APL, ACM Press, 1981, pp. 94–100.

[17] R. Devine, *Design and implementation of DDH: A distributed dynamic hashing algorithm*, FODO, 1993, pp. 101–114.

[18] D. Doval and D. O'Mahony, *Nom: Resource location and discovery for ad hoc mobile networks*, Proceedings of the Mediterranean Ad Hoc Networking Workshop, Med-hoc-Net 2002 (Sardegna, Italy), September 2002.

[19] ______, *Overlay networks: A scalable alternative for p2p*, IEEE Internet Computing **7** (2003), no. 4, 79–82.

[20] R. Droms, *Dynamic host configuration protocol, rfc 1541*, IETF Network Working Group, November 1997.

[21] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi, *Efficient broadcast in structured p2p networks*, Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03) (Berkeley, CA), February 2003.

[22] J. Flynn et. al., *A real time emulation system for ad-hoc networks*, Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference 2002 (CNDS '02) (San Antonio, Texas), January 2002.

[23] M. Kosuga et. al., *Analysis of wireless message broadcast in large ad hoc networks of pdas*, In proceedings of Fourth IEEE conference on Mobile and Wireless Communications Networks, 2002, pp. 299–303.

[24] N. Minar et. al., *The swarm simulation system: A toolkit for building multi-agent systems*, Tech. Report 96-06-042, The Santa Fe Institute, June 1996.

[25] R. Fielding et. al., *Hypertext transfer protocol – http/1.1, rfc 2616*, IETF Network Working Group, June 1999.

[26] S. Ni et. al., *The broadcast storm problem in a mobile ad-hoc network*, Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking, August 1999.

[27] T. Berners-Lee et. al., *Uniform resource locators (url), rfc 1738*, IETF Network Working Group, December 1994.

[28] V. Cerf et. al., *Internet protocol, rfc 791*, IETF, September 1981.

[29] J. Li et.al., *Capacity of ad hoc wireless networks*, Proceedings of ACM Sigmobile, ACM Press New York, NY, USA, 2001, pp. 61–69.

[30] L. A. Adamic et.al., *Search in power-law networks*, Physical Review E (Statistical, Nonlinear, and Soft Matter Physics) **64** (2001), no. Issue 4.

[31] G. Exoo, *A euclidean ramsey problem*, Discrete and Computational Geometry **29** (2003), no. 2, 223–227.

[32] R. J. Flynn and H. Hadimioglu, *A distributed hypercube file system*, Proceedings of the third conference on Hypercube concurrent computers and applications, ACM Press, 1988, pp. 1375–1381.

[33] *Gnutella protocol v0.4*, 2001, http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[34] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman, *Skipnet: A scalable overlay network with practical locality properties*, In Proceedings of USITS (Seattle, WA, March 2003), USENIX, 2003.

[35] J. Hastad and T. Leighton, *Fast computation using faulty hypercubes*, Proceedings of the twenty-first annual ACM symposium on Theory of computing, ACM Press, 1989, pp. 251–263.

[36] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao, *Distributed object location in a dynamic network*, 14th ACM Symp. on Parallel Algorithms and Architectures (SPAA), August 2002.

[37] Y. Hu, D. Rodney, , and P. Druschel, *Design and scalability of nls, a scalable naming and location service*, In Proc. of INFOCOMM, June 2002.

[38] D. B. Johnson and D. A. Maltz, *Mobile computing*, ch. 5 - Dynamic Source Routing in Ad Hoc Wireless Networks, pp. 153–181, Kluwer Academic Publishers, 1996.

[39] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web*, ACM Symposium on Theory of Computing, May 1997, pp. 654–663.

[40] S.C. Kleene, *Representation of events in nerve nets and finite automata*, Automata Studies, Ann. Math. Stu., no. 34, Princeton U. Press, 1956, pp. 3–41.

[41] J. Kleinberg, *The small world phenomenon: an algorithmic perspective*, 32nd ACM Symposium on Theory of Computing, May 2000.

[42] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, *Oceanstore: An architecture for global-scale persistent storage*, Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.

[43] T. Leighton, M. Newman, A. G. Ranade, and E. Schwabe, *Dynamic tree embeddings in butterflies and hypercubes*, Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, ACM Press, 1989, pp. 224–234.

[44] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S.

Wells, M. C. Wong, S. Yang, and R. Zak, *The network architecture of the connection machine cm-5 (extended abstract)*, Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, ACM Press, 1992, pp. 272–285.

[45] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris, *A scalable location service for geographic ad-hoc routing*, Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00), August 2000, pp. 120–130.

[46] J. Liebeherr and Tyler K. Beam, *Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology*, First International Workshop on Networked Group Communication (NGC '99) in: Lecture Notes in Computer Science, vol. 1736, July 1999, pp. 72–89.

[47] J. Liebeherr and B. S. Sethi, *Towards super-scalable multicast*, Tech. report, Polytechnic University CATT 98-121, January 1998.

[48] D. Malkhi, M. Naor, and D. Ratajczak, *Viceroy, a scalable and dynamic emulation of the butterfly*, ACM Principles of Distributed Computing (PODC), July 2002.

[49] T. P. Mann, *Decentralized Naming in Distributed Computer Systems*, Tech. Report STAN-CS-87-1179, Stanford University, Stanford, California, 1987.

[50] N. F. Maxemchuk, *The manhattan street network*, Proc. IEEE GLOBECOM, December 1985, pp. 255–261.

[51] ———, *Routing in the manhattan street network*, IEEE Trans. Comm. **35** (1987), no. 5, 503–512.

[52] ———, *Comparison of deflection and store-and-forward techniques in the manhattan street and shuffle-exchange networks*, Proc. of INFOCOM 89, vol. 3, April 1989, pp. 800–809.

[53] P. Maymounkov and D. Mazieres, *Kademlia: A peer-to-peer information system based on the xor metric*, 1st International Workshop on Peer-to-Peer Systems (IPTPS'02), 2002.

[54] R. M. Metcalfe, *Packet communication,*, Computer Classics Revisited, vol. I, Peer-to-Peer Communications, 1996.

[55] P. Mockapetris, *Domain names–concepts and facilities, rfc 1035*, IETF Network Working Group, November 1987.

[56] ______, *Domain names–implementation and specification, rfc 1035*, IETF Network Working Group, November 1987.

[57] P. Mockapetris and K. J. Dunlap, *Development of the domain name system*, ACM SIGCOMM Computer Communication Review **18** (1988), no. 4, 123–133.

[58] M. L. Mueller, *Ruling the root: Internet governance and the taming of cyberspace*, The MIT Press, March 2002.

[59] B. Nardi, S. Whittaker, and E. Bradner, *Interaction and outeraction: Instant messaging in action*, In Proceedings CSCW'2000, ACM Press, 2000.

[60] (Corporate) Ncube, *The ncube family of high-performance parallel computer systems*, Proceedings of the third conference on Hypercube concurrent computers and applications, ACM Press, 1988, pp. 847–851.

[61] D. O'Mahony and L. Doyle, *Architectural imperatives for 4th generation ip-based mobile networks*, Fourth international symposium on wireless personal multimedia communications, Aalborg, Denmark, September 2001.

[62] ______, *Mobile computing: Implementing pervasive information and communication technologies*, ch. An Adaptable Node Architecture for Future Wireless Networks, Kluwer Academic Publishers, August 2001.

[63] A. Oram (ed.), *Peer-to-peer: Harnessing the power of disruptive technologies*, O'Reilly and Associates, 2001.

[64] C. Perkins, *Ip mobility support for ipv4, rfc 3220*, IETF Network Working Group, January 2002.

[65] C. Plaxton, R. Rajamaran, and A. Richa, *Accessing nearby copies of replicated objects in a distributed environment*, ACM SPAA, June 1997, pp. 311–320.

[66] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, *A scalable content-addressable network*, ACM SIGCOMM, August 2001.

[67] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, *Topologically-aware overlay construction and server selection*, Proceedings of IEEE INFOCOM'02, 6 2002.

[68] R. Rivest and B. Lampson, *Sdsi—a simple distributed security infrastructure*, Tech. report, Massachusetts Institute of Technology, Cambridge, MA, 1996.

[69] T. Robertazzi and A. Lazar, *Deflection strategies for the manhattan street network*, IEEE ICC 91, vol. 3, June 1991, pp. 1652–1658.

[70] S. Rose, *Dns security document roadmap, rfc 1541*, IETF DNEXT Working Group, November 2001.

[71] J. Rosenberg, H. Schulzrinne, Columbia U., G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *Sip: Session initiation protocol, rfc 3261*, IETF Network Working Group, June 2002.

[72] A. Rowstron and P. Druschel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms, November 2001.

[73] J. Saltzer, D. Reed, and D. Clark, *End-to-end arguments in system design*, ACM Transactions on Computer Systems (1984), no. 2, 277–288.

[74] S. Saroiu, P. Krishna Gummadi, and S. D. Gribble, *A measurement study of peer-to-peer file sharing systems*, Proceedings of Multimedia Computing and Networking 2002 (MMCN '02) (San Jose, CA, USA), January 2002.

[75] Y. Sasson, D. Cavin, and A. Schiper, *Probabilistic broadcast for flooding in wireless mobile ad hoc networks*, Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2003), March 2003.

[76] C. L. Seitz, *The cosmic cube*, Communications of the ACM **28** (1985), no. 1, 22–33.

[77] H. J. Siegel, *Interconnection networks for smid machines*, Computer, vol. 12, 57–65, no. 6, 1979.

[78] R. Srinivasan, *Rpc: Remote procedure call protocol specification version 2, rfc 1831*, IETF Network Working Group, August 1995.

[79] P. Srisuresh and K. Egevang, *Traditional ip network address translator (traditional nat), rfc 3022*, IETF Network Working Group, January 2001.

[80] I. Stoica, D. Adkins, S. Ratnasamy, S. Shenker, S. Surana, and S. Zhuang, *Internet indirection infrastructure*, First International Workshop on Peer-to-Peer Systems, March 2002.

[81] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, Proceedings of ACM SIGCOMM, August 2001.

[82] K. Thompson, *Programming techniques: Regular expression search algorithm*, Communications of the ACM **11** (1968), no. 6, 419–422.

[83] N. Tzeng, *Analysis of a variant hypercube topology*, Proceedings of the 4th international conference on Supercomputing, ACM Press, 1990, pp. 60–70.

[84] Various, *The directory — overview of concepts, models and services*, CCITT X.500 Series Recommendations, December 1988.

[85] M. Whal, A. Coulbeck, T. Howes, and S. Kille, *Lightweight directory access protocol (v3): Attribute syntax definitions, and others, rfc 2252*, IETF Network Working Group, December 1997.

[86] M. Whal, T. Howes, and S. Kille, *Lightweight directory access protocol (v3), rfc 2251*, IETF Network Working Group, December 1997.