2011

Defining Routing Policies in Peer-to-Peer Overlay Networks

Michael Pickering
University of North Florida

DEFINING ROUTING POLICIES IN PEER-TO-PEER OVERLAY NETWORKS

by

Michael Pickering

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

April, 2011

The thesis "Defining Routing Policies in Peer-to-Peer Overlay Networks" submitted by Michael Pickering in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

4/28/2011

Sherif Elfayoumy, Ph.D.
Thesis Advisor and Committee Chairperson

Signature Deleted

4/28/2011

Zornitza Prodanoff, Ph.D.

Signature Deleted

4/28/2011

Sanjay Ahuja, Ph.D.

Accepted for the School of Computing:

Signature Deleted

4/28/11

Judith L. Solano, Ph.D.
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Signature Deleted

5/2/11

Peter Braza, Ph.D.
Interim Dean of the College

Accepted for the University:

Signature Deleted

5/9/11

Len Roberson, Ph.D.
Dean of the Graduate School

ACKNOWLEDGEMENT

I wish to thank my wife for her unwavering support and understanding during the many hours I dedicated to achieving this milestone in my life and career.

CONTENTS

FIGURES

TABLES

ABSTRACT

This master's thesis involves the definition and development of a policy-based routing scheme for peer-to-peer overlay networks. Many peer-to-peer networks are in existence today and each has various methods for discovering new peers, searching for content, and overcoming connectivity problems. The addition of efficient policy-based routing enhances the ability of peers within overlay networks to make appropriate routing decisions. Policy-based routing provides a means for peers to define the types of network traffic they are willing to route and the conditions under which they will route it. The motivations for these policies are many and are described in upcoming sections.

In order to express and enforce policies, a simple policy definition language was developed. This language is sufficient for owners of overlay nodes to choose to route traffic based on their own requirements and gives node owners a means to express these requirements, such that other nodes within the overlay network can learn them. A mechanism is presented that allows these policies to be stored either in a distributed hash table or on a set of directory servers.

The effectiveness of policy-based routing was tested using a simulated network. The affect of these routing policies, in terms of both additional network traffic and requirements for client software, was also assessed. Finally, a comparison was made between storing policy information in a distributed hash table, versus on a set of directory servers.

Chapter 1

INTRODUCTION

Overlay networks are becoming more prevalent in the Internet as a means of allowing users to communicate, share content, gain connectivity, and even play games. An overlay network can be defined as a network defined atop an existing network. The "existing network" is commonly the Internet, and that is the focus of this thesis. One of the attractive qualities of overlay networks is a user can connect to any other node in the network, without regard to the other node's characteristics on the underlying network (the underlay network). Of course, matters are not this simple; logic in the overlay network's program code must account for many different aspects of the underlay network, in order to allow this level of connectivity. The good news is much work has been done in this area and today's overlay networks are capable of dealing with a variety of issues that would ordinarily prevent two hosts from communicating, such as Network Address Translation (NAT), firewalls, and even unreliable network links [Andersen01A].

In addition to consistent connectivity, overlay networks can provide other benefits to users. The Tor network, for example, provides its users with anonymity [Dingledine04]. Users who join the Tor network have their network traffic encrypted and routed between Tor nodes in such a way that makes it extremely difficult for anyone to discover actions taken by individual Tor users. The RON network provides its users with continuous access, even in the face of network problems [Andersen01A]. Experimental overlay networks like MBONE and 6bone were used by researchers who needed a functioning

multicast and IPv6 infrastructure, when these features were not yet available on the Internet [Fink04, Savetz95].

All overlay networks share a certain number of primitive functions, which allow the network to exist and function. The first of these are the *join* operation and its inverse, the *leave* operation. In order for nodes to participate in the network, they must have a way of discovering the network, and making the network aware of their presence. This implies the availability of the current state of the network, or at least enough state information to allow a new node to introduce itself to the network. There are several means of providing this state, from using a centralized "network state" store, to using a fully distributed representation of the network state, where each participant holds a portion of the overall state. When joining, a node is usually required to present some information about itself so other nodes can contact it.

The other primitives common to overlay networks involve communication: the ability of a node to both locate another node and having located it, to send messages to it. In order for nodes to communicate, there must be some path between them on the underlay network. The job of the overlay network, then, is to determine this path and the specific mechanism to allow the communication to take place, so applications running on the overlay network do not need to be concerned about the details of the path through the underlay. Communication between nodes is an interesting problem in itself and a heavily studied topic, as researchers try to connect nodes with minimal latency [Jesi06],

route overlay traffic efficiently relative to the underlay [Xu03], or group nodes based on a common interest [Zhang05].

Another feature of overlay networks, not primitive but very common, is *search*. Users of overlay networks frequently want to share resources with other users, or consume resources offered by others. Users need a means of locating these resources on the network. There have been many approaches to implementing search within overlay networks, with each alternative bringing its own advantages and tradeoffs. Many networks implementing search provide a means of improving subsequent searches for the same data and may even provide a node identifier that can be used to re-contact a node, without needing to search for it each time [Dabek03].

1.1 Peer-to-Peer Networks

Many different peer-to-peer networks are in operation today. Among these, the I2P Anonymous Network and the Tor Project have several features, which help to illustrate the multiple methods of storing and distributing information among members of the network. The Blossom network, which has been proposed but is not in existence today, is also presented. Though not implemented, Blossom introduced a policy language and a method of publishing policies using directory servers [Goodell06].

1.1.1 Tor

The Tor peer-to-peer network is designed to provide anonymous, untraceable Internet access to its users. Tor extended the Onion Routing concept first developed by the U.S. Naval Research Lab [Goldschlag96].

When an individual Tor node starts, it collects a set of attributes that describe it. These attributes include its IP address, a user-friendly name, information about the Tor software version and underlying operating system, and a public key. This information is published to a Tor directory server in the form of a *router descriptor*, allowing other Tor peers to learn about the new node [TorDir11].

In addition to these attributes, Tor nodes can specify an *exit policy,* which allows a node to define a list of IP addresses and port numbers for which it is willing to carry traffic [TorDir11]. The exit policy is published to Tor directory servers, as part of the router descriptor, at the same time attribute information is being published. Tor clients seeking access to a particular destination or service can learn which peers are willing to provide that access by parsing and evaluating the various exit policies versus the service sought.

This information is published to Tor *directory authorities*. Directory authorities work together to create a consensus about the condition of the network, thus providing a unified picture of the state of the routers (nodes) within the Tor network, along with their attributes, capabilities, and exit policies [TorDir11]. Tor has a small number of directory authorities (fewer than 20), but the shared network state information is provided to a much larger set of directory caches, which respond to client queries seeking information about other nodes on the network [TorDir11].

Since a Tor router descriptor contains information about a node's attribute information, capabilities, and exit policy, Tor uses a single mechanism to store all types of node information.

1.1.2 I2P

I2P is a peer-to-peer network allowing peers to communicate anonymously. An I2P user can communicate with another user by using the pseudonym of the other user. The I2P software routes this communication through the network in such a way that neither user is aware of the IP address or other underlay details of the other user. To allow this, each I2P node serves as a tunnel for other I2P nodes, as the network makes these anonymous connections [I2P11A].

When an I2P node starts, it registers itself with the I2P *netDb*, a distributed network database [I2P11B]. I2P clients publish a "RouterInfo" item with a list of attributes, including their software version, uptime, and a list of capabilities. The netDb is maintained by a set of *floodfill routers*, a special type of I2P node similar in function to Tor's directory servers [I2P11B]. However, the storage mechanism and record propagation method is very different from Tor's. Instead of attempting to replicate the complete set of node descriptors, I2P uses the Kademlia algorithm to break up the node set into portions and assign responsibility for these portions to different floodfill servers [I2P11B, Maymounkov02]. For redundancy, the floodfill server responsible for any particular descriptor will also replicate the descriptor to the nearest seven floodfill servers [I2P11B]. Nearness, in this context, refers to distance as defined by Kademlia, applying

the XOR operator to node identifiers and counting the number of differing bits; nearer nodes have fewer differing bits [Maymounkov02]. Because I2P uses nearness to determine which floodfill servers should be responsible for a given data element, it can selectively distribute data to a subset of floodfill servers. This also allows I2P queries to be issued against both the floodfill server responsible for a certain data element and a "nearby" server, providing redundancy in case of node failure.

1.1.3 Blossom

Geoffrey Goodell's thesis, *Perspective Access Networks*, describes a new use for overlay networks: that of providing access to network perspectives [Goodell06]. Network *perspectives* are defined here to mean the view a certain host sees of the Internet, which can be different for different hosts, for various reasons. Some web sites, for example, offer different content to visitors based on the geographic location of the visitor. Goodell's experimental implementation of a perspective access network client is called "Blossom." It allows users of the network to offer, search for, and utilize perspectives offered by other network users.

Blossom is built as a part of Tor, an existing peer-to-peer overlay built for purposes of anonymity. Blossom does not have a specific requirement for Tor, but the Tor network provides a level of maturity and functionality of which Blossom can take advantage. For example, Tor provides means of building pathways (*circuits* in Tor's terminology) between nodes, which Blossom can make use of when constructing routes [Goodell06]. As an existing, mature overlay network, Tor has already implemented all the primitive

functions discussed earlier, such as *join* and *leave*. Blossom, as an application running on that network, can take advantage of these functions, without regard to their implementation.

Figure 1 illustrates a simple Blossom network. In this example, a user in Japan (Node_3) has built a Tor circuit (shown with a heavy line) to a node in the United States (Node_1), allowing this user to use the network perspective of Node_1 and, for instance, surf the web from the perspective of a user located in the United States. Node_3 learned of this perspective by querying its directory server Directory_2, which learned it from a relationship with Directory_1 (shown with a dotted line).

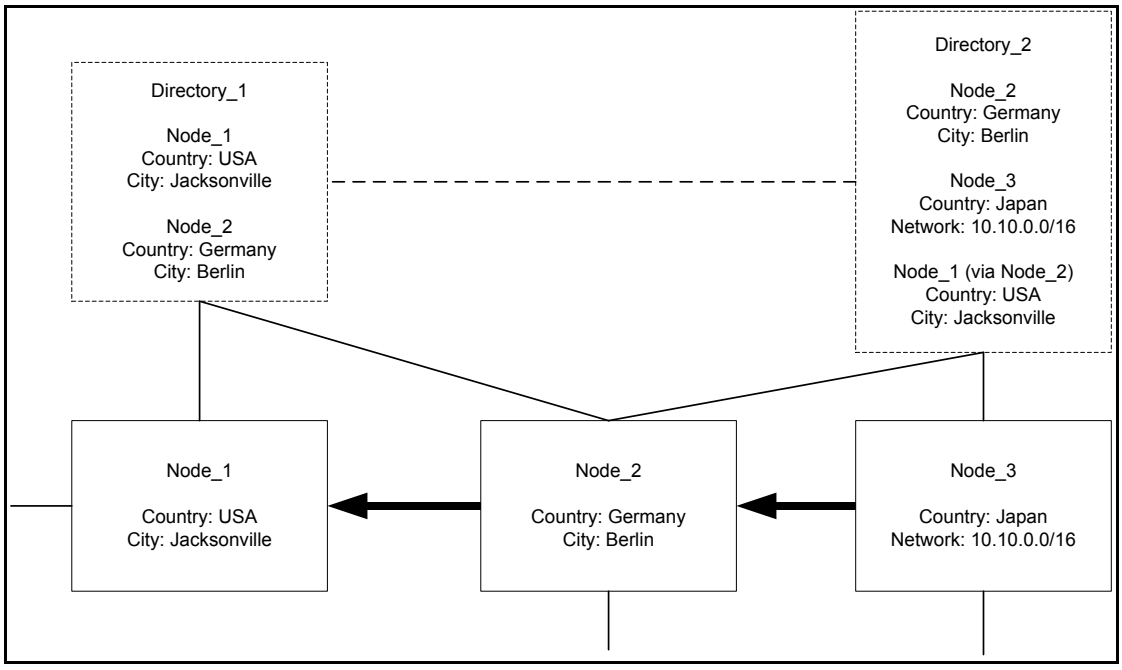

Figure 1: A Sample Blossom Overlay Network

In Blossom, each node must register with at least one directory server. When registering, nodes send information about their perspectives to the directory server, thereby advertising them to the peer-to-peer network. These perspectives can contain a variety of information, such as the country in which the node resides, a particular network that node has access to, or a service the node could provide [Goodell06]. Blossom's directory servers store this information and allow other users to learn of these perspectives by issuing directory queries. A Blossom user who learned of an interesting perspective could contact the provider of the perspective by building a circuit through the Tor network, as Node_3 has done in this example.

In the Blossom network, nodes store information about themselves on a set of directory servers, which have explicitly configured connections to each other [Goodell06]. This requires manual intervention on the part of directory operators to function and, depending on configuration choices made by directory operators, could result in a disjoint network. That is, certain peers could advertise resources and routing policies to a network of directory servers, but other peers may never be able to learn of these attributes or policies. Figure 2 depicts a disjoint network. Even though Directory_1 and Directory_2 can communicate, neither of them can reach Directory_3 or Directory_4. Users of Directory_3 or Directory_4 can never learn of resources advertised on Directory_1 or Directory_2.

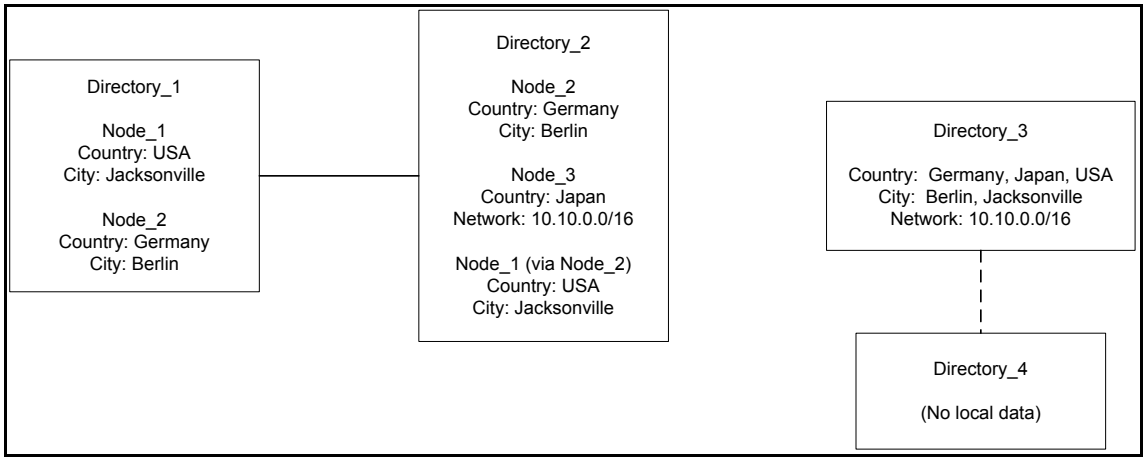

Figure 2: A Disjoint Blossom Network

Even if all directory servers were directly or indirectly reachable, queries involving iterative or recursive lookups may force clients to wait as the overlay infrastructure propagates the query. Blossom supports directory replication, which could improve clients' lookup times, at the expense of added replication traffic across the network and the increased storage requirement of the directory servers.

Blossom directory servers support the aggregation of attributes, in an effort to reduce the amount of data sent between peers. Aggregation allows route attributes to be expressed in a compact manner, but ultimately results in some information being lost. In computing a route based on aggregated route information, multiple peers within the overlay may satisfy a route, but one or more possible routes are ultimately not usable, due to policies enforced within some of the peers. This limitation cannot be discovered without attempting to create the route and if a routing limitation is discovered using one potential path, another must be selected.

Figure 3 illustrates attribute aggregation. In this example, Directory_3 aggregates information learned from Directory_2. If Directory_4 queried Directory_3 for the "Country" attribute "USA," Directory_3 would respond, but further queries would be necessary to locate the attribute.

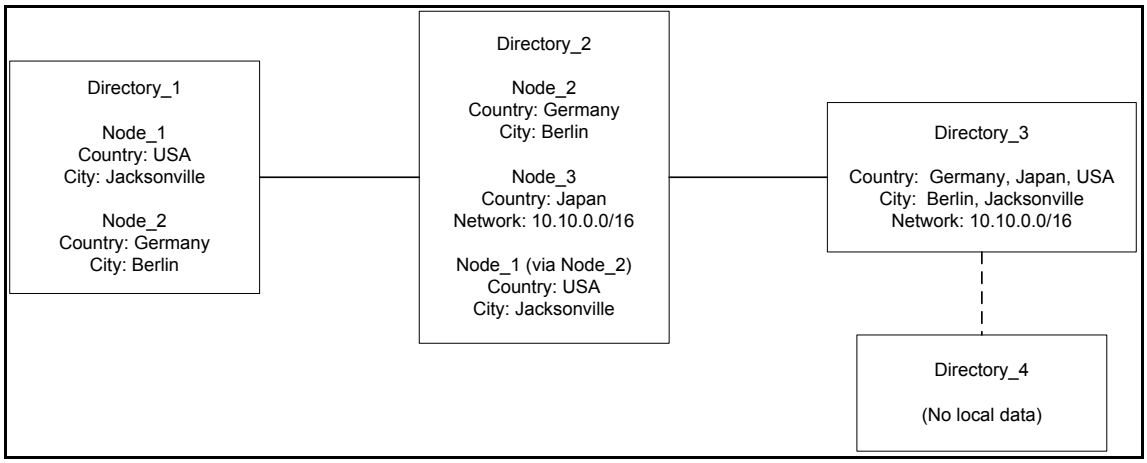

Figure 3: Attribute Aggregation

1.2 Internet Routing

The idea of using directory servers to store perspective and routing information, as described above, is based on the mechanisms used to exchange routing information on the Internet.

The Internet is, as the name suggests, a group of networks. These networks are operated by various entities, each with their own priorities in mind. There are also financial interests at play; high-speed communication equipment and the expertise to operate it cost money. For this reason, network operators often have contractual agreements with

other operators to provide the communication that makes the Internet possible [Huitema95].

A network managed by a single entity and with a consistent routing policy is called an *autonomous system* (AS). An autonomous system can consist of any number of computers and networks, but in order to interact with other parts of the Internet, it needs a way to connect to other ASes. The routers responsible for connecting the AS to other ASes are called *border routers* and are responsible for enforcing the previously mentioned contractual agreements among various parties [Huitema95]. Border routers exchange this information using the *Border Gateway Protocol* (BGP). BGP allows routers to communicate information to each other about networks for which they are responsible, and about networks to which they can provide connectivity [Huitema95]. The directory server arrangement used by Blossom is based heavily on BGP's architecture.

These routing policies, in addition to being stored within the border routers themselves, can be stored in a globally available database. There are several databases, which maintain this information, such as RIPE (Réseaux IP Européens) [RIPE11]. It is important to note, while these databases store routing information and make it available to interested parties; enforcement always takes place at the border routers themselves.

Several proposals have been made that would move routing policies into centralized route servers, which are responsible for maintaining routing policies in a central

authoritative location, as opposed to being simply a searchable repository [Feamster04, Govindan98]. These route servers could then drive the configuration of routers, which would ensure policies were both described and applied consistently. Parties wishing to establish routes would always be able to rely on the routing information learned from the route servers to be correct and consistent with the policies enforced within border routers.

1.3 Overlay Routing

Content location in peer-to-peer networks has been heavily studied. A variety of contemporary algorithms exists to allow requestors to locate content, which could represent a file, the location of another node, or anything else. In the past, networks like Napster used centralized content location, which made a single entity (i.e., Napster) responsible for tracking the location of content and answering queries for that content. The Napster organization was the single point of failure in this design. When Napster was shut down, the network ceased to exist [BBC00]. This single point of failure was eliminated in Gnutella's design, though it ultimately proved not to be scalable [Jovanovic01]. Current efforts like Chord and Kademlia use distributed hash tables to locate content [Stoica01, Maymounkov02].

Once a requestor locates the desired content, the peer-to-peer network's next task is to allow the user to connect to the content provider. In its most simple form, this is done using the underlay network, which is generally the Internet. The requestor could make a standard TCP/IP connection across the Internet and request the content directly from the provider. However, this is not always possible, due to firewalls, NAT, or other obstacles

between the requestor and provider. Therefore, overlay networks add a layer on top of the Internet and provide the requestor with an abstraction of the provider. This abstraction allows the requestor and provider to communicate, not necessarily using TCP/IP to make a direct connection, but by using the communication services of the overlay network. In addition to simply making the connection, the overlay could also provide enhanced capabilities, including anonymity, security, robustness, and connectivity that would not otherwise be possible [Andersen01B, Stoica02]. These additional capabilities come at a price. In order to provide this abstraction, overlay networks must often take on the additional task of routing traffic. That is, many of the low-level routing functions that move traffic through the internet are re-implemented to move traffic through the overlay.

1.4 Shortcomings in Overlay Routing

In both standard Internet routing and in overlay routing, the primary goal is to move data closer to its destination. In addition to this, there can be other secondary tasks as well. For example, some peer-to-peer networks attempt to select peers that have close proximity, to allow for lower latency in the network [Xu03]. Others try to meet a different objective, such as reducing the number of crossings between provider networks [Jesi06].

Goodell introduced the concept of policy-based routing in his work on Blossom [Goodell06]. Policy-based routing would allow nodes to offer services to their peers selectively, based on individual characteristics of that peer and on the services the peer

was seeking. These policies could be made available to the network, allowing peers to make intelligent decisions on whether to carry the network traffic of other peers.

There are several potential benefits to providing new policy-based routing features within overlay networks. First, peers may have the ability to forward traffic between parts of the overlay or out of the overlay, but might want to reserve some of their bandwidth for specific purposes. A routing policy would provide the means to express this to the network and could be used by the node itself in enforcing the limit. A flexible enough policy could enforce this limit only in the presence of higher-priority traffic or only at certain times.

Second, peers may want to provide priority services to certain types of users; for example, a peer may want to give preference to users who would otherwise have problems reaching certain content. Some governments, for example, block access to certain web sites for ideological reasons. Requestors specifically seeking access to these services could be given priority over users seeking unrelated services.

Third, peers may be unwilling to route traffic originating from certain other peers or groups of peers, within the network. Alternatively, peers may be willing to route traffic *only when* it originates from certain other peers or groups of peers. Peers may also wish to restrict the locations to which they are willing to forward messages. A robust routing policy could express all these limitations, so other peers could learn it before attempting to use these forwarding services. This could prevent clients from attempting to forward

traffic through nodes unwilling to carry the traffic, because clients could learn the policies ahead of time.

The Tor network offers a basic set of features that begin to address routing policies. Tor nodes may specify their willingness to serve as *exit nodes* (i.e., their willingness to allow traffic to leave the Tor overlay and exit onto the Internet). Furthermore, they may specify what types of traffic they want to exit and to which destinations. This information is known as an *exit policy* [TorDir11]. Exit nodes register this information with Tor's directory servers, when they join the network [TorDir11]. When a Tor client wishes to build a route through the network, it uses information learned from directory servers to decide which exit node is able to provide the desired services.

Tor's exit policy definition allows for a single, global set of rules that apply equally to all traffic handled by a node. There is no capability to enforce rules based on characteristics of the requestor. Therefore, Tor is able to address peer-to-peer routing policies up to a point, but lacks the complete policy-based routing capabilities described in section 2.1.3 [Dingledine04]. This is expected, of course, since the entire purpose of Tor is to allow users to access the network anonymously. If an exit node were able to determine characteristics of other nodes requesting its services, it would compromise Tor's purpose.

The I2P network has an even more limited set of options. I2P does not share Tor's purpose of providing anonymous access to Internet resources, so it does not need a way to specify which types of traffic it is willing to carry. Instead, I2P focuses on

pseudonymous communication within the I2P network itself and on the ability to host services within the I2P network. I2P does not have a counterpart to Tor's exit nodes.

Goodell outlines a modification to the internet standard RPSL (Routing Policy Specification Language) that allows some of these policies to be expressed [Goodell06]. These routing policies are expressed as a series of attributes peers could publish to the overlay network, so they could be available as routes are being built by the overlay. These attributes would be aggregated among directory servers in Goodell's proposed approach. Route determination would be an iterative process of querying directory servers and following references, until one or more peers offering the service is located. Then, a route to one of the peers must be established and in the event the route could not be completed (e.g., if one of the intermediate peers was unwilling to route traffic), other routes could be tried iteratively [Goodell06]. Though policies and query methods are described in [Goodell06], they are not actually implemented in Blossom.

1.5 Areas of Possible Improvement

The shortcomings described in the previous section fall into two categories. First, a peer-to-peer node needs a generalized way to describe it, the services it wishes to offer to the network, and the conditions under which it will grant access to those services. Second, a node needs a way to make this information available to the rest of the peer-to-peer network, where it can be discovered and used by other nodes.

1.5.1 Routing Policy Definition Language

Goodell proposed a modification to RPSL, Perspective Routing Policy Perspective Language (PRPSL), which meets Blossom's design goals but proved too limited for all the cases described above [Goodell06]. RPSL is designed to allow expression of Internet autonomous system policies and PRPSL modifies RPSL, to express Blossom directory server peering policies and perspective configuration [Goodell06].

PRPSL has some directives to offer or hide routes from individual clients who have authenticated with the directory server. However, the main purpose of PRPSL is to configure directory server peering and to define selective perspective advertisement among directory servers [Goodell06].

Ideally, a policy definition language would provide the benefits described in section 1.4, and be extensible enough to meet unanticipated needs. This language would give individual nodes, rather than directory servers, the ability to define their capabilities and policies. Such a language is described in the subsequent sections.

1.5.2 Attribute and Routing Policy Storage

Though routing policies are ultimately enforced by the nodes, which provide access to routes, there are various possibilities for the storage and distribution of routing policy information, as well as for the attributes used to evaluate other nodes against the policy.

Routing policies could be stored on directory servers, as is currently done in Tor and I2P, and discussed in Blossom. This method of storage is familiar and well understood, and provides many advantages and capabilities, since directory servers are capable of understanding client requests and tailoring their responses to individual clients.

On the other hand, if routing information were stored centrally, clients could select peers offering a particular resource by querying the central route store, rather than by iteratively following a series of references. This should improve the performance of route determination by allowing clients to choose peers they know are willing to carry traffic, based on the routing policy. The use of centralized routing information has been proposed by Feamster et al. [Feamster04] and Govindan et al. [Govindan98], though in these cases, the proposal applies to internetwork routing, rather than routing in peer-to-peer networks.

In the past few years, much work has taken place in the area of distributed hash tables. Distributed hash tables (DHTs) provide the typical benefits of hash tables familiar to the computing community, but have their storage spread across a number of nodes within a peer-to-peer network. DHTs do not provide traditional hash tables' O(1) lookup times, because their data is diffused over many nodes in the network, but they are still capable of O(log n) lookup times in most cases [Androutsellis-Theotokis04], where n represents the number of nodes. It should be possible for distributed hash tables to serve as a centralized store of node attributes and routing policies.

It is not currently clear whether directory servers or distributed hash tables would provide the best method of storing policy data. Goodell entertains distributed hash table storage, but dismisses DHTs on the basis they are not functionally suitable for the Blossom network, which cannot be guaranteed to have full connectivity [Goodell06]. I2P has experimented with a Kademlia DHT, in the past, and is considering moving back toward DHT storage, in the future [I2P11C]. A comparison between directory server and distributed hash table storage is needed to determine whether one method is more suitable for storing routing policy data.

Chapter 2

A NEW METHOD FOR POLICY-BASED ROUTING

2.1 Defining Attributes, Capabilities, and Policies

The peer-to-peer networks described in the previous section provide peers a way to define and publish attributes about themselves, their capabilities, and to some extent policies defining the type of traffic they are willing to route. This section describes a method of expressing these three things in detail, and of encoding and representing them in a consistent manner, in a way that they can be stored and discovered within the network. The section will conclude with a description of how both directory servers and distributed hash tables can provide this storage and discovery, and some of the advantages and trade-offs of each approach.

2.1.1 Attributes

A peer's *attributes* are properties about itself it wishes to make available to other peers in the network. Tor provides a minimal number of attributes, since it is focused on the anonymity of its peers; similarly, I2P's few attributes relate to software versions. As we will see, making policy decisions based on a peer's attributes will require peers to present more detailed information, therefore, we allow arbitrary attributes to be defined in a simple attribute_name = attribute_value format. Multiple values for a single attribute are not supported. The collection of a node's attributes make up an *attribute set*, an example of which is shown in Figure 4.

```
country = us
publickey = 123456
nodename = Saturn
```

Figure 4: An Example of an Attribute Set

The attribute set can then be made available to any nodes in the network that wish to inspect it. This is done at routing policy enforcement time, as discussed below. A list of example attribute fields and their purpose is shown in Table 1.

Attribute name	**Purpose**
country	the country a node resides in
publickey	a public key representing the node, which could be useful in a reputation system
nodename	a user-defined name for the node, which could be useful in a reputation system
network	a CIDR-style network address where the node resides
jointime	a date/time field representing the time the node joined the network
hostos	the host operating system
netversion	the version of the P2P network software
netname	a network name of significance to the P2P network

Table 1: Example Attribute Names

The precise meaning of the values is not defined as part of this work. For example, "country" might be a descriptive name, or might be an ISO 3166-style country code. These details can be specified apart from the work here and do not affect the ability to evaluate the suitability of attributes and their storage against the goals of this work or the

performance of the network. Furthermore, additional attributes can be defined in the same manner to allow new features to be added to the network.

2.1.2 Capabilities

Nodes wishing to provide services within the network will have a set of capabilities they can offer. For example, a node may be willing to provide e-mail forwarding to certain users, or to provide web proxying, or access to a particular network perspective. These capabilities are grouped into *capability sets*. A node can use policies to make capability sets selectively available to other peers based on their attributes, as shown in the next section. An example showing two different capability sets is shown in Figure 5.

```
capset basicweb {
    content = news, discussion;
    port = 80, 443;
}
capset mail {
    content = email, anonymity;
    port = 25, 587;
    maxbw = 50;
}
```

Figure 5: An Example of a Capability Set

In this example, a node wishes to offer a capability set called "basicweb" containing content related to "news" and "discussion" on ports 80 and 443, and another capability set called "mail" offering different content and ports, and specifying a bandwidth limit for those capabilities. Table 2 shows a list of possible capability types, and Figure 6 contains the capability set grammar.

Capability name	Purpose
content	a content type (such as news, discussion, or anonymity) for which a peer is willing to provide services. These concepts are taken from [Goodell06]
service	a service a node is willing to perform, such as web proxying, e-mail relaying, etc.
minbw	the minimum bandwidth a node can provide to a given service (kB/s)
maxbw	the maximum bandwidth a node will provide to a given service (kB/s)
port	TCP ports to which a node is willing to forward
perspective	a perspective a node can provide

Table 2: Example Capability Types

```
S ::= CapSet+ <EOF>

CapSet ::= "capset" String "{" (CapLine ";")+ "}"

CapLine ::= String "=" ValueList

ValueList ::= Value [ ("," Value)+ ]

Value ::= [ "not" ] String

String ::= ["A"-"Z", "a"-"z", "0"-"9"]+
```

Figure 6: Capability Language Grammar

It is conceivable that many other capabilities could be represented, such as a minimum or maximum allowed throughput, connectivity to a different network, or the use of encryption. The language is extensible in that it allows additional capabilities to be added to the definition and to be expressed, using the same form. The set defined above

is enough to illustrate the idea of capability sets and to allow peers to search for specific capabilities.

Unlike attributes, capability sets can contain multiple values for a given capability name. As shown in Figure 5, the "basicweb" capability is willing to provide forwarding on two different TCP ports. It is implied that the capability set excludes values not specified. Capabilities could also be excluded by using the keyword "not." A node wishing to offer access to all defined content types except "news" could specify "content = not news, *." A peer providing only news access and no access to other content types, would specify "content = news." The allowable values, such as "news," "discussion," "anonymity," are not defined here; in an actual network implementation, these values and their meanings would be specified in a universally available document.

The information within capability sets is made available to any peers in the network. Peers seeking access to a particular capability can search the network for this capability and determine which nodes were candidates for providing them with the desired service.

2.1.3 Policies

Policies tie capability sets to attributes and allow nodes offering services to decide whether they should provide these services to requestors, based on requestors' attributes. A sample list of policies is shown in Figure 7. The policy language grammar is shown in Figure 8.

```
for country = us, network = 192.168.0.0/16
  deny basicmail, web;

for country = ca
  allow web;

for country = cn
  allow web, basicmail, enhancedmail;
```

Figure 7: An Example of a Policy

```
S ::= PolicyItem+ <EOF>

PolicyItem ::= "for" AttrMatch Action CapsetList

AttrMatch ::= AttrExpr [ ("," AttrExpr)+ ]

AttrExpr ::= [ "not" ] String "=" (String | Wildcard)

Action ::= "allow"
         | "deny"

CapsetList ::= Capset [ ("," Capset)+ ]

Capset ::= String

String ::= ["A"-"Z", "a"-"z", "0"-"9"]+
```

Figure 8: Policy Language Grammar

The first part of the policy, the “for” portion, allows the specification of one or more attributes that must be matched in order to apply the policy. The second part lists the specific capability sets granted, or denied, to peers matching the attributes in the first part. In the example shown in Figure 7, a peer with an attribute set containing “country = ca” (i.e., a peer located in Canada) would match the second policy line. Such a peer

would be granted access, via the "allow" keyword, to the "web" capability set, as defined on the node containing the policy.

When attempting to determine a node's access via the policy, evaluation continues through all defined policy lines until a match against the "for" criteria is found. Evaluation stops at that point and further policy lines are not considered. If no match is found, the requestor is denied access to all capability sets. Since the "deny" keyword explicitly denies access to a requestor matching certain attributes, it can be useful if a later policy rule grants access to a broader group of attributes. This is illustrated in Figure 9.

```
for country = us
  deny web;

for country = *
  allow web;
```

Figure 9: A Policy with Precedence

Since policies and capability sets are available to both the requesting peer and the providing peer, both are equally capable of making a decision about whether the requesting peer can access a given capability. This allows the requesting peer to evaluate itself against the policy and avoid making a request to a peer that will ultimately deny it access. In fact, any peer on the network could perform this evaluation, which is a helpful characteristic discussed in the subsequent sections.

2.1.4 Security Implications

It is useful to allow requestors to see a node's policy before requesting access to a capability, because they can determine for themselves whether they would be granted access and therefore avoid the need to request it. However, access to the policy would seem to give a requestor exactly what they need to try access a service illicitly. A requestor could seemingly construct an attribute set that exactly matches the criteria to access a desired capability.

A node providing a service has two methods for determining whether a requestor has presented a correct attribute set. Some attributes can be directly verified. For example, "country" could be validated by using a geolocation service. With the requesting peer, "publickey" could be validated cryptographically. Another option is to use a reputation scheme, perhaps provided by the peer-to-peer network. Reputation systems for peer-to-peer networks are an active area of research, covered in detail in [Marti05].

2.2 Storing Attributes, Capabilities, and Policies

Both Tor and Blossom use a directory server approach to store attributes, capabilities, and policies (though attributes, capabilities, and policies are not treated as distinct in these networks). I2P uses a floodfill server approach, which resembles a directory server approach, but introduces elements of distributed hash tables to spread the data among the directories. In this thesis, a comparison is made between storing attribute, capability, and policy information in directory servers, versus using a distributed hash table for the same purpose. The implementation of each approach is discussed in the following sections.

2.2.1 Attribute Sets

A requesting node's attribute set can always be provided to the requestor at the time a request is made, so it may not seem necessary to provide for attribute storage at all. Nevertheless, we provide a means to store an attribute set in both a directory server and a DHT, because this prevents peers from needing to send their attributes every time they request a service. An attribute set's value is stored in its entirety, using a lookup key consisting of a node identifier specific to the overlay network. This node identifier is a network-specific abstraction that allows for direct communication with a node, thus bypassing the routing services of the P2P network [Dabek03]. Using the attribute set given as an example in Figure 4, a node with an identifier of 1190683980 would store the data as shown in Table 3. Any other node could easily look up the attribute set of another node via the node identifier. Note this approach works for both directory server and DHT storage.

Key	**Value**

1190683980	`country = us`
	`publickey = 123456`
	`nodename = Saturn`

Table 3: Attribute Set Storage

2.2.2 Capability Sets

The most straightforward way for a node to publish its capability sets to the network is to send the entire set as a unit, along with its node identifier. While this could work in a directory server-based network, using a node identifier as the sole key would preclude

nodes in a DHT-based network from querying for specific capabilities. With this in mind, it makes more sense to break capability sets into individual capability definitions and to store these definitions. For example, a node offering a "content=discussion, news" capability would store two separate entries: one with a key of "content=discussion" and another with "content=news." The value of each is the node identifier of the node offering the service. A node with identifier 382719848 would store this data, as shown in Table 4. The capability set name, such as "basicweb" or "mail," as given in the example in Figure 5, must also be stored, since the querying node must know the capability set name in order to evaluate the policy.

Key	**Value**
`content=discussion`	382719848, basicweb
`content=news`	382719848, basicweb

Table 4: Capability Set Storage

When a peer wishes to search for a capability, such as "content=news," it sends this query to the directory server or DHT and gets back one or more identifiers of nodes it should contact. The peer can then use the identifiers to contact these nodes and request the capability. As we will see in the next section, a peer could also look up the policies of the nodes offering the capability and determine which node, if any, is willing to provide it service.

2.2.3 Policies

Like attribute sets, a policy can always be obtained directly from the nodes enforcing the policy. Even so, we still provide for storage of policies in directory servers, as well as a DHT. Similar to attribute sets, policies are stored with a lookup key consisting of the node identifier and with a value containing the text of the policy. Therefore, a node with the identifier 382719848 and the example policy shown in Figure 7 would store the entry as shown in Table 5.

Key	**Value**

382719848	`for country = ca`
	`  allow web;`
	`for country = cn`
	`  allow web, basicmail, enhancedmail;`

Table 5: Policy Storage

2.3 Applicability of a Directory Server Approach

The use of directory servers to store attributes, capabilities, and policies is straightforward. Individual nodes can register their attributes, capabilities, and policies when they join the network and other nodes can query the directory servers, using the keys as described above. Because of their flexibility in storing the data they receive, directory servers can provide lookup methods in addition to the simple key-based lookups described in the previous section. For example, a node wishing to find a server willing to provide a web proxy capability from a British perspective could specify "country=gb, service=webproxy." Unlike a distributed hash table, which can only map

from single capabilities to node handles, a directory server has access to complete capability sets and is able to provide a list of all known nodes providing that specific combination of service. Directory servers can also evaluate more complex queries, such as queries using an inequality operator. For example, a query like "minbw > 50" could be answered by the directory server, since it has complete information about all the "minbw" entries it stores. A directory server could even evaluate a requestor's attributes against the policies of all nodes satisfying a capability query and only provide the requestor with a list of nodes willing to perform the service for the requestor.

2.4 Applicability of a DHT Approach

A distributed hash table can easily be used to store attribute sets and policies as described above. A node seeking attribute information about another node simply looks it up using the other node's handle as a key. Likewise, a node requesting another node's policy could look it up using the other node's handle.

A peer querying for a given capability, such as "content=news," is able to get a listing of nodes offering this capability. In addition to the node handle, the peer learns the capability set name, as defined on each node. It uses this when it subsequently evaluates the node's policy, to determine if it can access the capability, based on its attributes.

Implementing the "not" operation for capabilities proves difficult. For example, nodes wishing to exclude a certain perspective from the results of a capability query would implicitly be searching for all other possible values. The only way to search for this,

given the encoding described in section 2.2.2 would be to specify all values except the one to exclude. This would only be possible where a fixed, predefined number of capability values existed and even then would be inefficient. In addition, since hash tables work based on exact matches, a node wishing to express a comparative match would not be able to do so. Queries such as "minbw > 50," which can be answered by a directory server, could not be answered by a distributed hash table. Some possible workarounds using a hybrid approach are mentioned in Chapter 4, but are beyond the scope of consideration here.

Finally, searching for combinations of capabilities is more difficult with distributed hash tables. Since capabilities are stored separately, applications must make a separate search for each capability desired and combine the results to determine which nodes offer all of the desired capabilities.

Chapter 3

EXPERIMENTATION AND EVALUATION

3.1 Network Simulators

Several network simulation products exist, each of which can model networks at various levels of abstraction. Some of these platforms are intended for simulating the lower layers of the network and are useful for network protocol testing. Other platforms dispense with the lowest network layers and instead focus on providing a framework on which overlay networks can be built and tested. Several requirements were considered in selecting a simulator for experimentation. First, the simulator needed to be extensible, meaning additional features could be added if they did not exist. Second, the simulator needed to provide an implementation of a distributed hash table, since the goal was to experiment with a distributed hash table, not to implement one. Finally, the simulator needed to allow applications to be written and to execute using the services of the simulated network. The ability to record the use of the network's resources was critical for understanding the behavior of the approaches being studied.

OMNeT++ is a popular low-level simulator, capable of simulating sensor networks, lower-layer protocols, such as Ethernet, and higher-layer Internet protocols, such as IP and TCP [OMNeT11]. Since OMNeT++ operates at such a low level, it is not well suited for testing overlay networks. Because of this, the OverSim framework was written to build on OMNeT++ and to provide many features specific to overlay testing

[OverSim11]. Both OMNeT++ and OverSim are written in C++ and considerable difficulty was encountered in compiling them and their dependencies. Because of these difficulties, the complexity involved in C++ development, and the potential for portability problems in running simulations on different computer systems, OverSim was eliminated from consideration.

P2Psim is intended for simulating overlay networks and is cited several times in the peer-to-peer literature [P2Psim11]. It provides several distributed hash table implementations, including Chord. It has not been updated since 2005, and because it was written in C++ and provided little documentation, it was not considered further.

PeerSim is a popular Java-based simulator [PeerSim11]. PeerSim allows new peer-to-peer protocols to be implemented and tested and provides a good separation between the protocol and network code. It is written in Java and is kept up-to-date. The separation between the protocol code and application code is not as well defined and its documentation is not as complete as some of the other simulators considered.

Overlay Weaver is a relatively new simulator, written in Java and providing a good separation between application, protocol, and network code [OverlayWeaver11]. It provides several distributed hash table implementations and allows simulations to run across multiple computers simultaneously. It also provides visualization of running simulations, though this is best used with small network sizes.

Ultimately, the PlanetSim simulator was chosen for all the experiments performed in this work [García05]. It is written in Java, allowing it to be easily extended and allowing simulations to run on a variety of platforms. PlanetSim provides several documents with examples of its use and provides complete JavaDoc documentation of its internal components. Of the simulators considered, PlanetSim provided the best separation between applications and the overlay network, which was required to experiment with the DHT-versus-directory server approaches.

Using PlanetSim, it is possible to configure many nodes in any configuration, to schedule events, and to measure the performance of the network. PlanetSim is implemented using a model defined in [Dabek03], which provides a common way to implement overlay networks and to run applications against them. PlanetSim's lowest-level object is the *Network* object. This contains a collection of *Node* objects, which would most commonly represent end-users' personal computers on the Internet. PlanetSim nodes can host one or more *Application* objects, which represent executable code running on a computer system [García05]. During the execution of a simulation, applications make use of the lower layers to send and receive messages. The handling of these messages, including their routing and delivery, is handled by the lower layers. PlanetSim's architecture is shown in Figure 10. Note application objects exist on each node, but are only shown for one node.

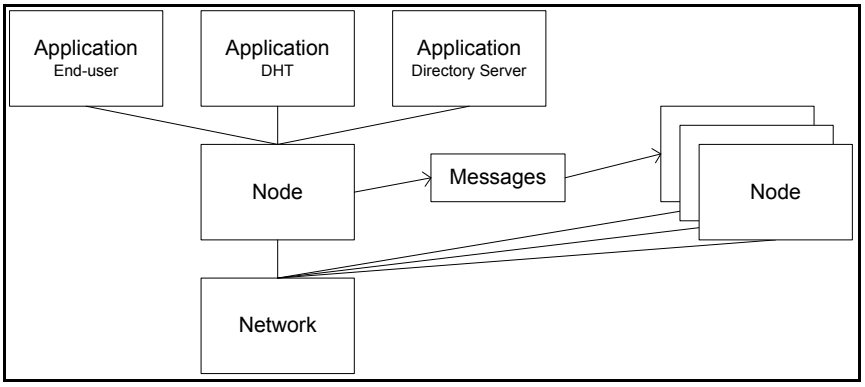

Figure 10: An Illustration of PlanetSim's Architecture

The PlanetSim model implements routing in the overlay layer, though it is possible for applications to provide "hints" to the overlay, if they know a node's exact location. This would be the case, for example, if nodes have previously communicated or a node has learned another node's address via a lookup. These hints, or "NodeHandles" in PlanetSim's terminology, allow the network to send messages directly to their destination, bypassing the routing logic in the overlay. This accurately models applications using the routing services provided by the overlay network, which are subject to the performance limitations of the overlay (e.g., Chord's O(log n) performance [Stoica01]) and applications acting directly on information learned from queries.

During a simulation run, PlanetSim can gather response times and compute the average time taken for applications to discover and use routes. The results can be compared between simulation runs using distributed hash table storage and runs using directory server storage.

3.2 Application Architecture

PlanetSim's application objects allow application logic to be implemented. During the simulation, applications are responsible for generating and consuming messages within the network. The lower layers of the architecture are important for ensuring messages reach their destinations, but applications are responsible for producing and consuming all messages. Applications represent actions taken by end-users and ultimately drive the simulation. This section describes three PlanetSim applications implemented to test the proposed methods of storing routing policies in the network.

3.2.1 DHT Application

PlanetSim provides an implementation of the Chord [Stoica01] and Symphony [Manku03] distributed hash tables. Chord is frequently cited within the DHT literature. Because of its wide acceptance and familiarity, it was used for the DHT platform in this thesis. Note, however, no Chord-specific features were used, so any DHT implementation should be equally functional. The DHT application provided by PlanetSim allows multiple values to be stored for a given key, so if multiple nodes registered a particular capability, a querying application would receive a response containing information about all nodes offering the capability.

3.2.2 Directory Application

To experiment with a directory server implementation of routing policy distribution, a Blossom-like directory application was written that provided similar functionality to the directory servers described in [Goodell06]. As specified in Goodell's paper, only a

subset of nodes functioned as directory servers. The proportion of directory nodes to non-directory nodes could be varied to measure the effect on the resulting lookup times and network traffic, but a fixed proportion was used for all experiments. This is further described in section 3.3.1.

Directory servers were used to store application-level data, including attributes, capability sets, and routing policies. When applications were started, they registered themselves with a particular directory server and sent it any attributes, capabilities, and routing policies they wished to advertise to the network.

As in Blossom, the directory servers were configured with explicit peering relationships to each other. There are several types of possible peering relationships; the most important types, and the ones implemented in the directory application, are *full* peering and *proxy* peering. These are the peering types implemented in the PlanetSim directory application. Full peering involves a full exchange of directory records, so a server *B* peered with server *A* would have a complete copy of *A*'s records, as well as its own local record store. Proxy peering allows servers to make recursive lookups to satisfy queries. In this case, if server *B* had server *A* as a proxy peer, it could forward any requests it was unable to satisfy to server *A*. Figure 11 uses example data to illustrate the proxy and full peering relationships.

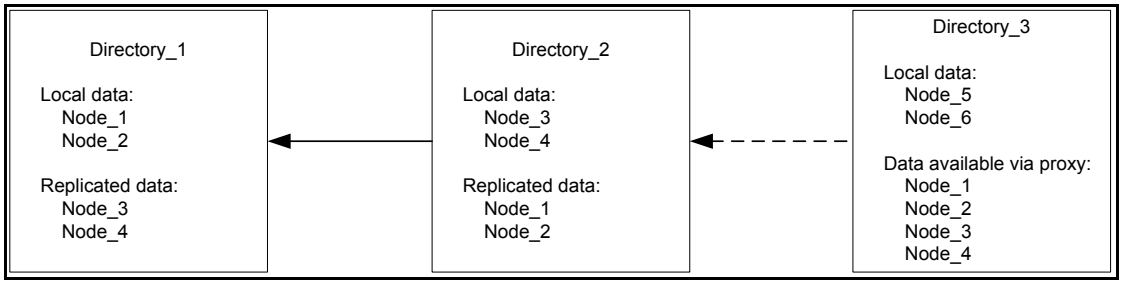

Figure 11: Peering Relationships between Directory Servers

There are six client nodes (not shown), two registered with each of the directory servers shown. Directory_1 and Directory_2 are full peers of each other, represented by the solid line. Each has a complete copy of the other's data; Directory_2 learned of Node_1 and Node_2 through its peering relationship with Directory_1. Both Directory_1 and Directory_2 are able to answer queries with the data from Node_1 through Node_4.

Directory_3 has Directory_2 as a proxy peer, represented by the dashed line. It has access to its own data, and through its proxy relationship with Directory_2, has indirect access to Node_1 through Node_4. Directory_3 is, therefore, able to answer queries with data from all six nodes.

These peering relationships were critical to the performance of the network, as well as the amount of storage space required. The higher the degree of connectivity between directory servers, the faster lookup requests could be satisfied. However, in the case of full replication, higher connectivity increased the amount of replication traffic, since every user application registered with a directory server resulted in application data being sent to all that server's full peers. For proxy peering, higher connectivity meant

directories would try to satisfy requests by contacting all their proxy peers, increasing the amount of network traffic generated by a single query. Even though some peers may not have been able to satisfy a query, they would receive it and possibly forward it to their proxy peers. Directory servers did refuse to forward proxy requests to peers who had already seen the request, to prevent loops. They also refused to forward proxy requests more than a configurable number of times, as described in section 3.3.1. Even with these safeguards, proxy queries increased traffic on the network. Furthermore, if multiple peers could satisfy a query, the network was forced to carry redundant responses.

3.2.3 End-User Application

Every node in the simulated network ran an "end-user application," which represented functions being performed by an end-user. This could represent any action taken by a user of a peer-to-peer application that involved actual interaction with the network, including issuing queries and attempting to make use of network services. The end-user application registered a variety of attributes, capability sets, and routing policies on behalf of the user. In addition, the end-user application performed work, attempting to access capabilities offered by other applications on the network; these capabilities were discovered by querying either the distributed hash table or directory server, depending on the experiment's configuration.

The end-user application also served as the enforcement point of routing policies. Peers attempting to make use of routing services offered by other peers could only be allowed to do so when the routing policy of the target peer allowed it. The target peer enforced or

denied routing requests based on its policy. Since an application was able to learn the target peer's routing policy before contacting it, the application could avoid the wasted effort of contacting a peer, which would ultimately deny it access, based on the routing policy.

3.3 Simulation Setup

To assess the impact the proposed routing policy definition would have on a peer-to-peer network; several different scenarios were simulated within PlanetSim. The common elements in all these scenarios were the attributes, capabilities, and policies configured on each application, and the work performed by each application. At the start of the simulation, PlanetSim read a set of data files, allowing it to configure each application individually with both its properties and a set of tasks to be performed over the course of the simulation. During the simulation run, PlanetSim recorded statistics such as the number of messages passed over the network, the size of the messages, and the time needed to perform the various tasks. This allowed modification of the data storage configuration, and measurement of the effect of different configurations on applications and the network.

As the simulator created the nodes that made up the network, it configured each end-user application with a list of attributes read from a file. These attributes were selected to resemble the I2P and Tor networks; for example, the distribution of all nodes' "country" attributes was taken from the country distribution observed in the actual Tor network. Similarly, the "version" attribute was distributed like that of the I2P network. Each node

was given a unique nodename. These attributes had no effect on the behavior of the application, but were used in policy decisions made by other applications during the simulation. All attribute data was pre-generated, allowing the same experiment to be run multiple times with the same results. Examples of the attribute data used are provided in Appendix A.

Each application offered certain capabilities to the network. The offered capabilities were distributed randomly among the various applications and grouped into capability sets, each of which was given a random name. Like attribute data, capability set data was pre-generated and the same input was used for each experiment. Appendix A contains examples of the capability data used for applications.

Finally, policy data was generated granting access to each of a node's capability sets. Each capability set offered by a node had at least one policy granting access to it. The attributes required to match a given capability set varied from universal matches, such as "nodename=*," to somewhat common matches, like "country=us," to rare matches like "country=in, version=1.1." This ensured that not all capability queries would result in a list of nodes willing to grant access to the capability desired, and forced the application to consider the policy. Examples of the attribute data used are provided in Appendix A.

After each node's end-user application read its input parameters, it published its attributes, capabilities, and policies as described in section 2.2. For experiments using the distributed hash table, keys and values were written to the distributed hash table; for

experiments using directory servers, data was published onto the directory server with which the application had registered. At this time, directory servers replicated this data to their full peers. Once the data was published, each application proceeded to run various capability queries and act on the results. For example, an application may have queried for "content=discussion." It may have received one or more responses to this query. Upon receiving a response, it attempted to locate the policy of one of the nodes providing the capability. It was then able to evaluate itself against the policy, and if it met the policy's criteria, it attempted to access the desired capability. If it did not meet the criteria, and if multiple nodes provided the capability, it continued requesting policies and evaluating itself against those policies, until it succeeded in finding a match, or ran out of candidates. Pseudo code showing an application's processing for a query is shown in Figure 12.

```
send capability query
receive list of candidates

do while (capability not granted) and (more candidates):
  request next candidate's policy
  evaluate candidate policy against this my attributes
  am I granted access, based on the policy?
    capability = granted
end do

is capability granted?
  contact candidate to access capability
```

Figure 12: Application Query Procedure

Each application ran twenty queries over the course of the simulation. The queries themselves were predefined in a data file, allowing the simulation to be run repeatedly with the same results. An example of the data contained in this file is shown in Appendix C.

When an application attempted to access content defined on another peer, the other peer's application evaluated the requestor's attributes against the policies it had defined, and decided whether the requestor should be granted access. Although applications offering content always enforced access control based on their policy, client applications were able to determine whether they possessed the attributes required to access a capability based on the policy, and avoid contacting nodes which would deny access.

Two separate groups of simulation experiments were performed. The first tested the directory server-based approach, and the second tested a distributed hash table-based approach. In addition to these experiments, a third experiment was performed with no routing policies at all, to determine the overhead the addition of routing policies imposed.

The attributes, capabilities, and policies read by each node, as well as the actions performed by each node, were identical across all experiments, regardless of the underlying data storage method. Applications run under these conditions would expect to see very little difference in functionality; an application would expect to receive the same answers from queries it generated. Applications would expect to notice a

difference in the time taken to receive a response to a query, and in the amount of network traffic processed.

3.3.1 Simulation Parameters

Many different parameters could be set before starting the simulation. These parameters could greatly influence the outcome. Therefore, it was important to select realistic values before running any experiments.

The first parameter to consider was the size of the simulated network. The majority of Chord testing was conducted on a network of 10,000 nodes [Stoica01]. The size of the Tor network varies, but as of January 15, 2011, had approximately 6,300 nodes. I2P also varies in size, but had 2,311 nodes as of February 10, 2011. Therefore, 10,000 was selected as a reasonable and realistic network size to simulate. In order to determine if the network experienced problems with scaling, another series of experiments was conducted using 5,000 nodes and another at 1,000 nodes.

For the directory server-based experiments, only a subset of nodes actually functioned as directory servers. Determining the exact number of directory servers was important. Using a small number of servers would result in less replication traffic among the directories and potentially faster response, but could place an undue burden on the directory servers, since they would be required to support more clients. Again, the Tor and I2P networks were observed to determine a realistic proportion. A considerable difference was observed in these networks: approximately 53% of Tor nodes function as

directory servers, and 2.8% of I2P nodes function as floodfill servers. Using the example in [Yang03], with a cluster size of 10, we chose to have a directory server for every 10 nodes in the network. In this arrangement, one of every 10 nodes, in addition to being a regular peer, also functioned as a directory server. Because of their special function within the network, directory servers would be considered "super-peers" in Yang and Garcia-Molina's terminology [Yang03].

Since a directory server could be peered with any number of other directory servers, and multiple peering arrangements are possible (full peering versus proxy peering), the peering configuration was important. Many full-peering arrangements could result in faster lookups by clients, but would increase the amount of replication traffic. Too many proxy-peering arrangements would result in a lengthy lookup path through the network. Since the directory servers in this simulation functioned differently than Tor directory servers or I2P floodfill servers, no direct correlation could be made. Instead, the Goodell thesis was used as a guideline, giving four neighbors per directory server [Goodell06]. Goodell experimented with various peering arrangements, but never mixed them in a single experiment; all were peered as full or proxy neighbors together. In an effort to provide a balance of low-latency query response times and storage efficiency, we used an arrangement of four peering relationships per directory, with any given directory having two full peers and two proxy peers. As in Goodell's examples, the set of directory servers was fully connected. Figure 13 illustrates this for fifteen directory servers. Full peering relationships are represented by solid lines and proxy peering relationships by dashed lines. Note peering relationships themselves are not symmetric; in the

illustration, the double-ended arrows actually represent two separate peering relationships.

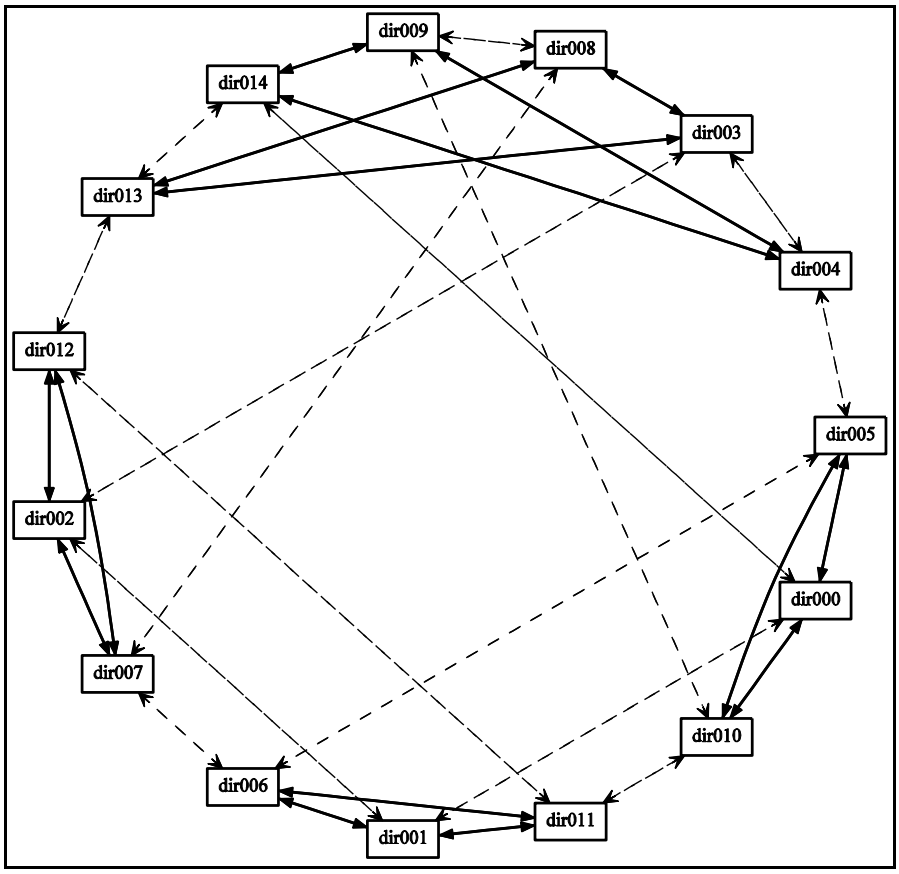

Figure 13: Directory Server Peering Example

Directory servers with proxy peers could forward lookup requests to their proxies; these proxies, in turn, could continue to forward the requests if they were unable to answer. For queries with few or no results, a large number of proxy queries could be generated from a single incoming request. To prevent this, a forwarding limit of two was established, using the guidelines in [Yang03]. Popular capabilities had a good chance of being discovered within two proxy hops, but for more obscure capabilities, the

reachability was limited. In the simulations conducted, the directory server experiments were able to satisfy 83% of all queries satisfied by the DHT. Increasing the proxy-forwarding limit was expected to increase the reach, but also increase the amount of directory traffic.

3.3.2 Routing Policies with Directory-Based Storage and Lookup

The directory-based storage simulations used a Blossom-like directory application. The peering configuration of the directory servers, described above, was pre-determined and read from a data file, allowing consistent results to be produced for each simulation run. Directory server peering took place prior to any application queries being performed. An example of directory server peering data is given in Appendix B.

It was expected that some application queries would perform extremely quickly. If a directory server were able to answer a request directly, then from the end-user's perspective, the lookup would take one round-trip-time (RTT), or two simulation steps. If recursion was required, the lookup could take longer, but the maximum time required for a response would be determined by the distance to the furthest directory server reachable via a proxy peering configuration. It was possible that some application queries would go unanswered; if a directory was not able to answer a query directly, it would forward the request to its proxy peers, but if none of them had the answer, the application would never receive a response. In this case, the application would need to implement a timeout counter, to ensure the user was notified of the failed lookup. On the other hand, if multiple proxy peers returned a response, the application would receive

several separate responses, originating from various proxy peers. Applications assigned a unique query ID to each query submitted, ensuring they could track responses accurately.

3.3.3 Routing Policies with DHT-Based Storage and Lookup

The Chord distributed hash table supplied by PlanetSim was used to store attributes, capabilities, and routing policies. Using a distributed hash table instead of directory servers eliminated replication traffic among the directory servers, as well as any recursive lookups and unnecessary proxy requests from directory servers. Applications executed the same queries, but these were serviced by the distributed hash table. The maximum time required for a response depended on the performance of the hash table; in Chord's case, this is O(log n) [Stoica01]. Where a query for a nonexistent attribute would result in no answer from a directory server, an application would receive a negative response from the distributed hash table, because the DHT was able to map the query to a single node responsible for the data. This eliminated the need for applications to implement a timeout counter, as required in the directory server scenario. In addition, since a single node was responsible for a query, an application would never receive multiple responses to a single query, as might happen for directory server queries.

3.4 Metrics and Evaluation

In order to determine whether this work added any value to the field, it needed to be evaluated using criteria accepted by existing research, and against the shortcomings presented in the introduction. The following quantitative metrics were directly measured by experimentation, and should serve to justify the methods outlined in the previous

sections. These statistics were collected for both the initial insertion of data into the network (directory server or distributed hash table) and the subsequent execution of queries against the network.

3.4.1 Wait Time

The end-user experience is important in any application. An application that performs poorly is unlikely to be embraced by users, even if it offers additional functions. Some PlanetSim experimenters have used elapsed time as a measure of algorithm performance [Braunisch06], but this makes many assumptions about the state of the hardware and software system used to run the experiment. A consistent form of measurement must be used within the simulator, so various simulation runs can be compared independently.

Bischofs et al. [Bischofs06] use simulation steps as a measure of node wait time. Because simulation steps provide consistency among experiments, they were measured in all experiments in this work. PlanetSim keeps a simulation counter, which allows applications to check the simulated step count before and after performing an action. This metric is comparable across runs, even if simulations are run on different systems.

3.4.2 Network Traffic

All messages sent between PlanetSim nodes are Java objects. The classes that define these objects were extended to calculate their size in bytes. The node classes in PlanetSim were likewise extended to count the number of messages sent. A single query sent by an application resulted in a number of messages; each query needed to be sent to

one or more directory servers, perhaps recursively, or move around the Chord ring, and each hop resulted in an additional message handled by the network.

In their work, Hu and Xia [Hu09] consider the impact on the network, based on the number of messages per second and the total size per second. Similarly, Yang and Garcia-Molina [Yang03] measure network load, based on bits per second. In the experiments we conducted, using a simulated network with simulation steps rather than seconds, it was more appropriate to compare the total number of messages sent, as well as the average and total size of these messages. Since each experiment ran for the same number of simulation steps, these results were comparable across runs.

3.4.3 Application Requirements

It is possible to measure the amount of storage space used within the client applications. For example, the size of attributes and routing policies stored in directory servers can be counted in bytes. As a comparison, the storage space used by each node participating in the distributed hash table can be counted. Application storage is a metric used in other experiments; for example, Hu and Xia use this metric in determining the node resource requirements in their P2P streaming analysis [Hu09].

To support the collection of application storage data, methods were added to the Java classes that make up the DHT and directory server applications, giving them the ability to calculate the total amount of storage required during each experiment.

3.5 Results

3.5.1 Wait Times

Table 6 and Figure 14 show the average query response time experienced by applications running in a directory server configuration, versus applications running in a distributed hash table configuration. In most cases, the directory server was able to locate the requested information either directly or with a single proxy query. As the number of nodes increased, applications using directory servers continued to observe the same response time. As expected, applications querying the distributed hash table saw slower performance, as the number of nodes increased and increasingly more hops were required to route the query to its destination. It should be noted that the increase in wait time was in line with Chord's O(log n) performance expectations [Stoica01].

	1000 nodes	**5000 nodes**	**10000 nodes**
Directory Server	2.30	2.34	2.32
DHT	6.92	7.92	8.43

Table 6: Average Query Wait Time (steps)

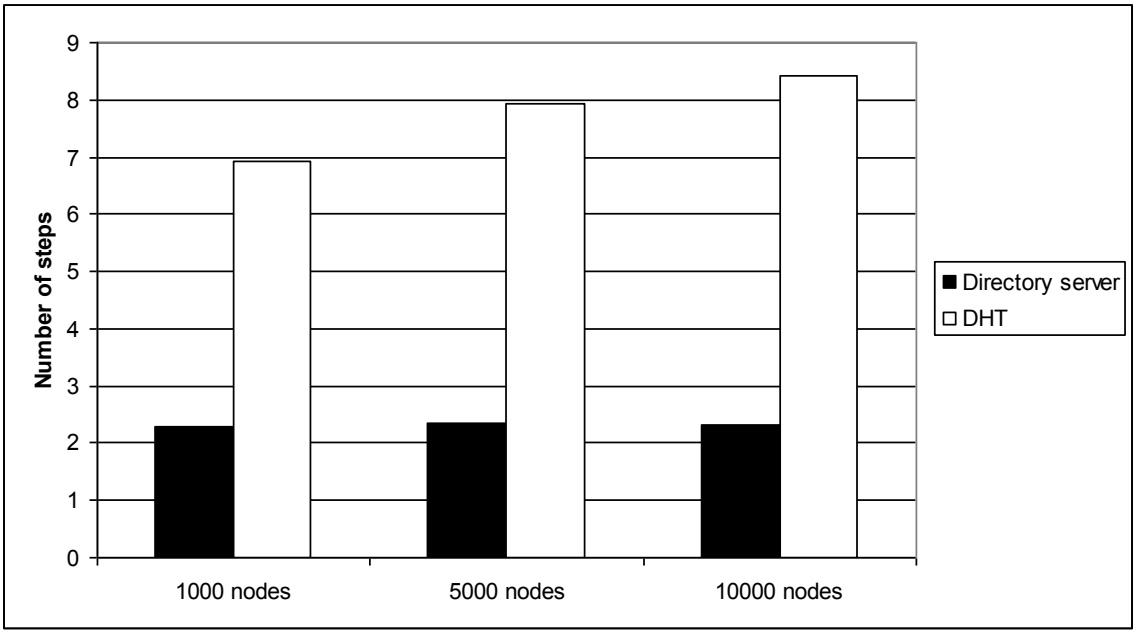

Figure 14: Average Query Wait Time

3.5.2 Network Traffic

Both the number of messages sent and the total size of all messages increased linearly with the number of nodes in both the directory server and DHT experiments. In the directory server experiments, a ten-fold increase in the number of nodes resulted in the number of messages and total byte count increasing just under ten times. With the same increase in the number of nodes, the DHT experiments resulted in a larger, but still linear, increase. These results are presented in Figures 15 and 16, and in Table 7.

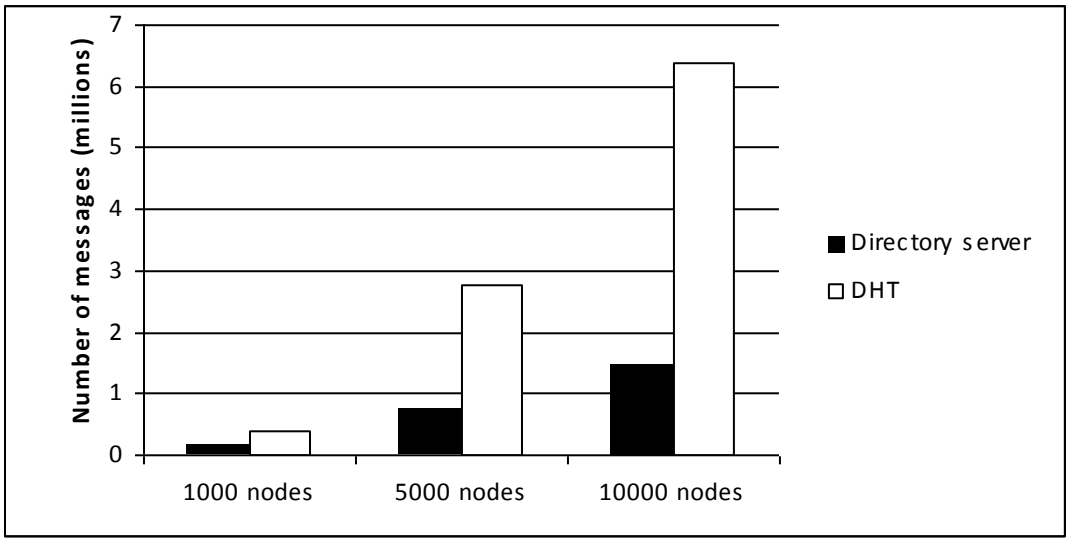

Figure 15: Total Network Messages Sent

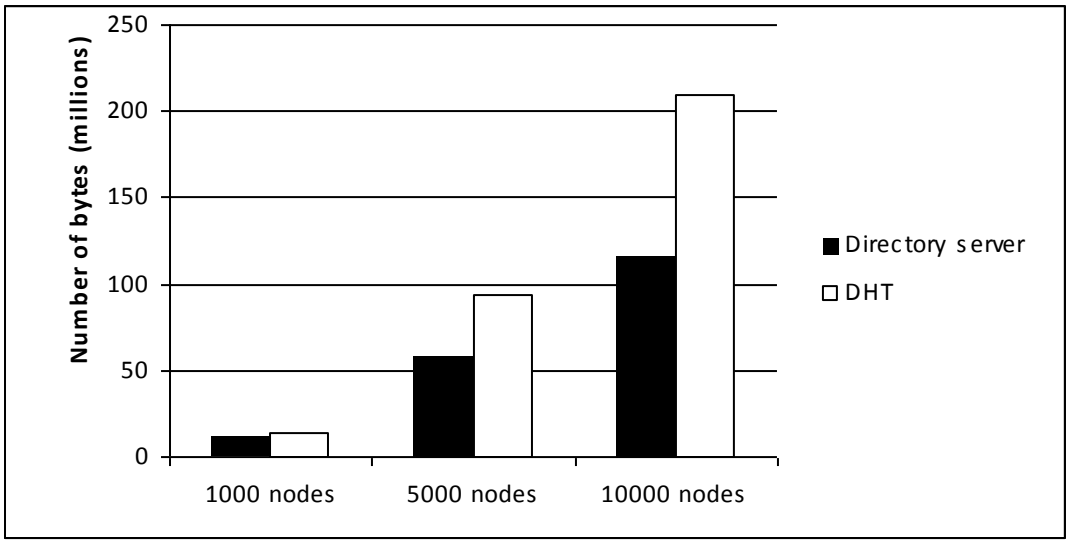

Figure 16: Total Network Traffic (bytes)

Comparing the two approaches to each other, the directory server experiment resulted in far fewer messages than the DHT experiment and considerably fewer bytes as well. It should be noted here that the average message size in directory server experiments was

nearly double the average DHT message size, because of the increased size of the directory synchronization messages. Table 7 provides the results of these experiments.

	1000 nodes	**5000 nodes**	**10000 nodes**
Directory server total messages (millions)	0.150	0.747	1.480
DHT total messages (millions)	0.381	2.77	6.40
Directory server total bytes (millions)	11.8	58.3	116
DHT total bytes (millions)	14.1	93.5	21.0
Directory server average message size (bytes)	78.7	78.1	78.3
DHT average message size (bytes)	33.5	33.8	32.8

Table 7: Network Statistics

3.5.3 Application Requirements

All applications stored their own attributes, capability sets, and policies, by design. The different storage requirements came into play at the point data was being made available to the network, using directory servers or a distributed hash table. Since all nodes served as Chord nodes in the DHT experiment, but only one in ten nodes served as directory nodes, it was expected that directory servers would have heavier storage requirements. Directory servers required more than ten times the space of DHT nodes, because each directory server stored both the data of its direct clients and of its full peers' clients. In

both DHT and directory server experiments, the amount of storage space remained constant as the size of the network increased, because the increased amount of data was distributed among an increased number of Chord nodes or directory servers. The application storage requirements are presented in Table 8.

	1000 nodes	**5000 nodes**	**10000 nodes**
Average directory server application storage (bytes)	10,040	10,035	10,025
Average DHT server application storage (bytes)	455	455	453

Table 8: Application Storage Requirements

3.5.4 Routing Policy Overhead

A series of experiments were conducted using no routing policies at all. In these experiments, nodes still registered attributes and capabilities, but no policy information. Registration of a capability was considered an implicit grant of that capability to all nodes. In this way, capabilities became a sort of global policy, much like in the Tor network. The same set of experiments were run without routing policies, to determine the overhead the introduction of routing policies added to the network. Since in these experiments, the existence of a capability implied access to the capability, applications did not need to submit an additional query to determine a candidate node's policy. Thus, the application query procedure noted in Figure 12 was modified to eliminate the iteration over candidate nodes.

Not surprisingly, the addition of routing policies carried a penalty both in terms of the storage required by applications and the amount of data traveling over the network. Table 9 shows the decreased application storage requirements. While routing policies do require more space, the absolute difference is not excessive and does not change with the size of the network.

	1000 nodes	**5000 nodes**	**10000 nodes**
Average directory server application storage (bytes)	6,288	6,292	6,296
Average DHT server application storage (bytes)	315	315	313

Table 9: Application Storage Requirements (no policies)

The differences in network traffic are shown in Figure 17 and Figure 18. Adding policies to the network did affect the network, in terms of both message count and the total number of bytes. This increase was less pronounced in the directory server experiments.

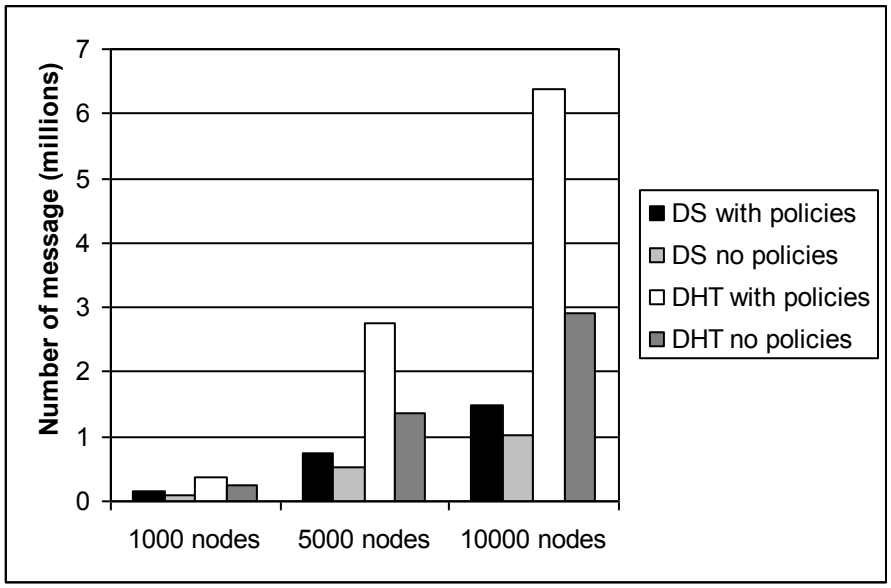

Figure 17: Comparison of Network Messages, Policies vs. No Policies

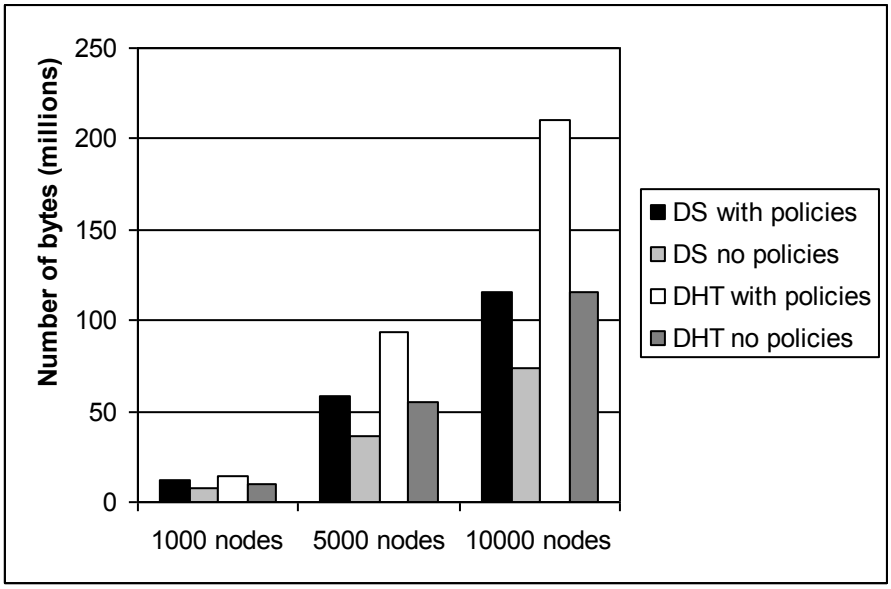

Figure 18: Comparison of Network Traffic, Policies vs. No Policies

3.5.5 Discussion

The experiments showed using directory servers provided the best overall performance. In addition to serving ten other nodes on the network, each directory server contained a replica of its two full peers' data, giving its clients access to the attribute, capability, and policy data of 30 different peers within a single RTT. In addition, through the two proxy peers, access to the data of 60 additional peers was available via a single proxy request. This allowed the directory servers to answer queries more quickly than the distributed hash table. In addition, applications received consistent directory server response times, regardless of the size of the network.

Using directory servers for data storage put the least load on the network. Due to the large messages sent during full-peer replication, the average message size was greater when using directory servers. However, fewer messages were needed overall, since directory servers were generally able to answer queries either directly or via a proxy request. The overall amount of data sent over the network was significantly less using directory servers for storage.

The full set of capabilities was not reachable given the experimental parameters defined in section 3.3.1. When limiting the number of proxy hops to two, applications were able to get answers to approximately 83% of their capability queries. Increasing this limit would result in higher reachability, at the expense of more network traffic. Varying the peering arrangements could also result in higher reachability; this is an interesting area for future research.

Directory servers, by their nature, have additional demands placed on them, because they are required to store data and process queries for multiple nodes on the network. Directory servers must store attribute, capability, and policy information for each node they serve, as well as for each node served by full directory peers. An average directory server required 20 times the amount of storage space as an average node in the distributed hash table.

Chapter 4

CONCLUSION AND FUTURE WORK

4.1 Contributions

Peer-to-peer networks can benefit from the addition of flexible routing policies and from a way to express and publish these policies, so other peers on the network can find and use them. It is important to measure the impact these policies have on clients, in terms of their expected response time and additional storage requirements. It is also important to determine the impact on the network, so the most efficient means possible is used to store and query for policy information.

In this work, a language was introduced which allows node owners to define capabilities they wish to offer to the network and to group these into capability sets. Node owners reference these capability sets when defining policies, allowing them to grant access to capability sets based on criteria they define. These criteria are based on the attributes of requesting peers, which allows nodes to offer different capabilities selectively to different peers, rather than offering only a global, one-size-fits-all policy. A method was shown allowing attributes, capabilities, and policies to be stored in both directory servers and distributed hash tables, and to allow nodes to query the stored policy data in either arrangement. The advantages and disadvantages to each approach were discussed, and while both methods are functionally suited to simple queries, directory servers were shown to be better able to process complex queries.

Since data can be stored and queried in either a directory server or DHT arrangement, this work showed experimentally that directory servers provide advantages in both query response time for individual peers and in lighter aggregate load on the network. The directory servers themselves, however, were required to take on the additional workload of processing queries and replicating traffic to their peers and were shown to have greater storage requirements than nodes in a distributed hash table.

4.2 Future Work

Directory servers present many interesting possibilities not considered in this work. For purposes of accurate comparison with distributed hash tables, the capabilities of directory servers could not be fully exploited. For example, because a directory server stores a node's capability sets and policies together, it can instantly evaluate a set of attributes against any node's policy. This information allows directory servers to answer queries differently based on the requestor's attributes. A directory server could filter its answer to a capability query to include only nodes that would allow the requestor to access the capability, based on the policy, both reducing the amount of data sent to a requestor and eliminating the need for the requestor to issue a separate policy query. Directory servers also have the ability to answer more complex queries, such as requests for combinations of capabilities or ranges of capability values.

Goodell mentioned a third type of directory server peering where directory servers summarize their data and send the summaries, rather than a full replica, to their peers

[Goodell06]. These summaries could be used to reduce the number of proxy requests sent, since directory servers could avoid sending proxy requests to peers that had not indicated knowledge of a capability.

The results of this experimentation would appear to rule out distributed hash tables for policy data storage. However, there may be certain conditions in which a distributed hash table would perform better than directory servers. If most capabilities being advertised were rare in the network, directory servers would suffer from the coverage problem described in section 3.3.1. Distributed hash tables may actually show a performance advantage here. In addition, directory servers and distributed hash tables are not the only ways of storing data. For example, routing indexes as described by [Crespo] and [Hose09] could be used to guide queries through the network.

REFERENCES

Print publications

[Andersen01A]
Andersen, D., H. Balakrishnan, F. Kaashoek, and R. Morris, "Resilient Overlay Networks," Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, ACM, New York, 2001, pp. 131–145.

[Andersen01B]
Andersen, D. G., "Resilient Overlay Networks," Master's thesis, Massachusetts Institute of Technology (May 2001).

[Androutsellis-Theotokis04]
Androutsellis-Theotokis, S. and D. Spinellis, "A Survey of Peer-to-Peer Content Distribution Technologies," ACM Computing Surveys 36, 4, (2004), pp. 335–371.

[Bischofs06]
Bischofs, L., S. Giesecke, M. Gottschalk, W. Hasselbring, T. Warns, and S. Willer. "Comparative Evaluation of Dependability Characteristics for Peer-to-Peer Architectural Styles by Simulation," Journal of Systems and Software 79, 10 (2006), pp. 1419–1432.

[Braunisch06]
Braunisch, M. H., "Chord and Symphony: An Examination of Distributed Hash Tables and Extension of PlanetSim," Master's thesis, Harvard University (June 2006).

[Crespo]
Crespo, A. and H. Garcia-Molina. "Routing Indices For Peer-to-Peer Systems," Proceedings of the International Conference on Distributed Computing (July 2002).

[Dabek03]
Dabek, F., B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (2003).

[Dingledine04]
Dingledine, R., N, Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," Proceedings of the 13th USENIX Security Symposium (2004), pp. 303–320.

[Feamster04]
Feamster, N., H. Balakrishnan, J. Rexford, A. Shaikh, and J. V. D. Merwe, "The Case for Separating Routing from Routers," ACM SIGCOMM Workshop on Future Directions in Network Architecture, ACM Press, 2004.

[García05]
García, P., C. Pairot, R. Mondéjar, J. Pujol, H. Tejedor, and R. Rallo, "Planetsim: A New Overlay Network Simulation Framework," Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 2005, volume 3437, pp. 123–137.

[Goldschlag96]
Goldschlag, D., M. Reed, and P. Syverson, "Hiding Routing Information," R. Anderson, ed., Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 1996, volume 1174, pp. 137–150.

[Goodell06]
Goodell, G. L., "Perspective Access Networks," Ph.D. thesis, Harvard University (July 2006).

[Govindan98]
Govindan, R., C. Alaettinoğlu, K. Varadhan, and D. Estrin, "Route Servers for Inter-Domain Routing," Computer Networks and ISDN Systems 30 (1998), pp. 1157–1174.

[Hose09]
Hose, K., C. Lemke, and K.-U. Sattler. "Maintenance Strategies for Routing Indexes," Distributed and Parallel Databases 26 (2009), pp. 231–259.

[Huitema95]
Huitema, C., Routing in the Internet, Prentice Hall, Englewood Cliffs, NJ, 1995.

[Jesi06]
Jesi, G., A. Montresor, and O. Babaoglu, "Proximity-Aware Superpeer Overlay Topologies," Proceedings of SelfMan'06 (2006).

[Jovanovic01]
Jovanovic, M., F. Annexstein, and K. Berman, "Scalability Issues in Large Peer-to-Peer Networks - A Case Study of Gnutella," ECECS Department, University of Cincinnati, Cincinnati, OH, 2001.

[Manku03]
Manku, G. S., M. Bawa, and P. Raghavan, "Symphony: Distributed Hashing in a Small World," Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (2003), pp. 127–140.

[Marti05]
Marti, S. and H. Garcia-Molina, "Taxonomy of Trust: Categorizing P2P Reputation Systems," Working Paper 2005-11, Stanford InfoLab, Stanford University, Palo Alto, CA, 2005.

[Maymounkov02]
Maymounkov, P. and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," Proceedings of the International Workshop on Peer-to-Peer Systems (2002).

[Savetz95]
Savetz, K., N. Randall, and Y. Lepage, MBONE: Multicasting Tomorrow's Internet, IDG Books Worldwide, Inc., Foster City, CA, USA, 1995.

[Stoica01]
Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (2001), pp. 149–160.

[Stoica02]
Stoica, I., D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet Indirection Infrastructure," Proceedings of ACM SIGCOMM (2002), pp. 73–86.

[Xu03]
Xu, Z., R. Min, and Y. Hu, "HIERAS: A DHT Based Hierarchical P2P Routing Algorithm," Proceedings of the International Conference on Parallel Processing (2003), p. 187.

[Yang03]
Yang, B. and H. Garcia-Molina, "Designing a Super-Peer Network," Proceedings of the International Conference on Data Engineering (2003), p. 49.

[Zhang05]
Zhang, R. and Y. C. Hu, "Assisted Peer-to-Peer Search with Partial Indexing," Proceedings of IEEE INFOCOM (2005).

Electronic Sources

[Fink04]
Fink, R. and R. Hinden, "6bone (IPv6 Testing Address Allocation) Phaseout (RFC 3701)," http://www.ietf.org/rfc/rfc3701.txt, last revision March 2004, last accessed March 3, 2011.

[Hu09]
Hu, Y. and Y. Xia, "Tracker vs. DHT Performance Comparison for P2P Streaming," http://tools.ietf.org/html/draft-hu-ppsp-tracker-dht-performance-comparison-01, last accessed March 10, 2011.

[I2P11A]
"I2P Anonymous Network," http://www.i2p2.de/, last accessed March 3, 2011.

[I2P11B]
"The Network Database," http://www.i2p2.de/how_networkdatabase, last accessed March 3, 2011.

[I2P11C]
"Network Database Discussion," http://www.i2p2.de/netdb_discussion.html, last accessed March 3, 2011.

[OMNeT11]
OMNeT++ Community, "OMNeT++ Network Simulation Framework," last accessed March 10, 2011.

[OverlayWeaver11]
"Overlay Weaver: An Overlay Construction Toolkit," http://overlayweaver.sourceforge.net, last accessed March 10, 2011.

[OverSim11]
Karlsruher Institut für Technologie, "The OverSim P2P Simulator," http://www.oversim.org, last accessed March 10, 2011.

[P2Psim11]
"p2psim: A Simulator for Peer-to-Peer (p2p) Protocols," http://pdos.csail.mit.edu/p2psim/, last accessed March 10, 2011.

[PeerSim11]
"PeerSim P2P Simulator," http://peersim.sourceforge.net/, last accessed March 10, 2011.

[RIPE11]
RIPE Community, http://www.ripe.net/ripe/, last accessed March 3, 2011.

[BBC00]
BBC News, "Napster Shut Down," http://news.bbc.co.uk/1/hi/entertainment/852283.stm, last revision July 2000, last accessed March 3, 2011.

[TorDir11]
The Tor Project, "Tor Directory Protocol Version 3," https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=dir-spec.txt, last accessed March 3, 2011.

APPENDIX A

EXAMPLES OF ATTRIBUTE, CAPABILITY, AND POLICY DATA

Description of Data

During the initialization phase of the simulation, each application is assigned certain attributes, capabilities, and policies. The application presents this information to the directory server or distributed hash table when it starts. The attribute, policy, and capability data for three nodes is shown below.

Example Attribute Data

Attribute data is defined using a simple attribute_name = attribute_value format. Lines beginning with "#" mark the start of a new node's data.

```
# Node 0
country=in
netname=Net05
version=2.0
nodename=alice-blue
# Node 1
country=pt
netname=Net03
version=1.2
nodename=antique-white
# Node 2
country=vn
netname=Net01
version=0.2
nodename=aquamarine
```

Example Capability Data

Capability set definitions are written in the language described in section 2.1.2. Applications register between zero and five capability sets, and they are assigned random names. Lines beginning with "#" mark the start of a new node's data.

```
# Node 0
capset qsbbxk0guq {
 content = web, streamingaudio;
 perspective = au;
}
capset 2s84ae0uni {
 content = mailmix;
```

```
 port = 25;
}
capset bsttmjk65k {
 content = web;
 perspective = uk;
}
capset 60we0mzgrr {
 content = politics;
 perspective = nl;
 service = chat;
}
# Node 1
capset nkxd128y1y {
 content = news;
 encryption = aes256;
 perspective = uk;
}
# Node 2
capset 0vrnkj8vc8 {
 content = email, anonymity;
 port = 25, 587;
 perspective = us;
}
capset wb87l9gq9u {
 content = streamingvideo;
 port = 80, 8080;
 perspective = us;
}
```

Example Policy Data

At least one policy item grants access to every capability set offered by an application. Policy definitions are written in the language described in section 2.1.3. Lines beginning with "#" mark the start of a new node's data.

```
# Node 0
for country = nl
  allow qsbbxk0guq;
for country = aq, netname = Net05
  allow 2s84ae0uni, bsttmjk65k;
for country = nl
  allow bsttmjk65k, qsbbxk0guq, 2s84ae0uni;
for not netname = Net01, country = es
  allow 60we0mzgrr, qsbbxk0guq;
# Node 1
for country = mx
  allow nkxd128y1y;
# Node 2
for country = us
  allow 0vrnkj8vc8, wb87l9gq9u;
for country = es
  allow wb87l9gq9u, 0vrnkj8vc8;
```

APPENDIX B

EXAMPLE OF DIRECTORY SERVER PEERING DATA

During the initialization phase of the simulation, each directory server application makes a series of peering connections with other directory servers. The peering configuration is read from a file, so multiple simulation runs result in the same peering arrangements. The following represents a peering configuration for ten directory servers, numbered from 0 to 9.

Using this file, directory 0 would have directories 3 and 6 as full peers, and directories 1 and 8 as proxy peers.

```
0 full 3
0 full 6
0 proxy 1
0 proxy 8
1 full 4
1 full 7
1 proxy 2
1 proxy 9
2 full 5
2 full 8
2 proxy 3
2 proxy 0
3 full 6
3 full 9
3 proxy 4
3 proxy 1
4 full 7
4 full 0
4 proxy 5
4 proxy 2
5 full 8
5 full 1
5 proxy 6
5 proxy 3
6 full 9
6 full 2
6 proxy 7
6 proxy 4
7 full 0
7 full 3
7 proxy 8
7 proxy 5
8 full 1
8 full 4
8 proxy 9
8 proxy 6
9 full 2
9 full 5
9 proxy 0
9 proxy 7
```

APPENDIX C

EXAMPLE OF APPLICATION WORKLOAD DATA

During the course of the simulation, each end-user application makes 20 queries against either its directory server or the distributed hash table, depending on the test configuration. This data is read from a file, so multiple simulation runs result in the same workload executing against the peer-to-peer network. The queries are randomly distributed over the course of 100,000 simulation steps.

Using this file, at simulation step 903, the application on node 2 would execute the query "port = 25".

```
at 903 2 query port = 25
at 982 1 query content = streamingvideo
at 1188 0 query encryption = aes256
at 1210 8 query nonexistent = nonexistent
at 1487 6 query perspective = fr
at 2523 8 query content = streamingvideo
at 3558 8 query nonexistent = nonexistent
at 4174 7 query port = 21
at 4650 1 query content = web
```

VITA

Michael Pickering has a Bachelor of Music degree from Jacksonville University in Music Theory and Composition, 1995, and expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida, April 2011. Dr. Sherif Elfayoumy of the University of North Florida is serving as Michael's thesis advisor.