

# A Naming Service for Overlay Networks

Greg Mattes  
gmattes@cs.virginia.edu  
Department of Computer Science  
University of Virginia

July 2005

## Abstract

The HyperCast system [1] provides the abstraction of a peer-to-peer overlay network socket that has a logical overlay network address, or simply logical address. An overlay socket's logical address reflects its position in an overlay network topology. The format of a logical address depends on the particular overlay protocol, or protocols, in which a socket participates.

Logical addresses are used for peer identification and message routing. An overlay network message is addressed to a peer socket using the logical address of the intended recipient. Peer sockets on the path from sender to receiver make routing decisions based on the destination logical address of the message.

Although logical addresses are needed to identify peers and to route messages in an overlay network, they are not intended for identifying applications located at particular sockets. This means that it can be cumbersome to work exclusively with logical addresses when writing overlay network applications, e. g. a programmer wishes to establish contact with an application using a convenient identifier like "application foo," not with a socket using logical address 1234. The ability to contact an application using an identifier other than a logical address is particularly useful if an application moves its position in an overlay network, if more than one instance of an application exists in an overlay, or if the logical address format used in an overlay changes due to a change in the selection of overlay protocol. The identifier used as an alternative to a logical address is called a *name*. Names decouple applications from any particular logical addressing scheme.

This document presents a naming service for overlay networks implemented with HyperCast overlay sockets. This service allows names to be used instead of logical addresses. The naming service is incorporated directly into each overlay socket to form an integrated, distributed, peer-to-peer naming service. This system has many desirable properties. First, the format and interpretation of names is independent of any particular overlay topology. Second, the naming service creates name bindings. Each name binding links a name to a logical address and changes in response to logical address changes. Third, logical addresses are not interpreted by the naming service which means that the naming service can be used with any overlay protocol. Finally, the naming service provides mechanisms that ensure the integrity and authenticity of name bindings.

This final property raises the need for a facility that allows the trust of a name binding to be established. The HyperCast naming service solves this problem by defining a protocol for exchanging and verifying trust information.

# Contents

<b>1</b>	<b>Design</b>	<b>4</b>
1.1	Introduction . . . . .	4
1.2	HyperCast Network Services . . . . .	6
1.3	Name Binding Exchange vs. Trust Information Exchange . . . . .	7
1.4	Naming Service API vs. Naming Service FSM . . . . .	7
<b>2</b>	<b>Data Structures</b>	<b>9</b>
2.1	Name Binding . . . . .	9
2.2	Pending Logical Address Query Table . . . . .	9
2.3	Pending Name Query Table . . . . .	10
2.4	Local Name Binding Table . . . . .	11
2.5	Peer Name Binding Cache . . . . .	12
2.6	Pending Certificate Request Table . . . . .	12
2.7	Pending Untrusted Query Table . . . . .	12
2.8	Certificate Cache . . . . .	13
<b>3</b>	<b>Protocol Messages</b>	<b>14</b>
3.1	Pull Name Bindings Message . . . . .	14
3.2	Push Name Bindings Message . . . . .	15
3.3	Invalidate Name Bindings Message . . . . .	16
3.4	Logical Address Query Message . . . . .	17
3.5	Name Query Message . . . . .	18
3.6	Certificate Request Message . . . . .	19
3.7	Certificate Response Message . . . . .	19
3.8	Local Name Query Message . . . . .	20
3.9	Install Trusted Certificate Message . . . . .	20
3.10	Set Binding Message . . . . .	20
3.11	Unset Binding Message . . . . .	21
<b>4</b>	<b>Timers</b>	<b>22</b>
4.1	Periodic Push Timer . . . . .	22
4.2	Peer Bindings Timer . . . . .	22
4.3	Pending Query Timer . . . . .	22
4.4	Certificate Request Timer . . . . .	22
<b>5</b>	<b>Protocol Operations</b>	<b>23</b>
5.1	Logical Address Query Operation . . . . .	23
5.2	Name Query Operation . . . . .	24
5.3	Push Name Bindings Operation . . . . .	25
5.4	Set Name Operation . . . . .	26
5.5	Invalidate Name Bindings Operation . . . . .	26
5.6	Establish Trust Operation . . . . .	26

<b>6</b>	<b>Evaluation</b>	<b>28</b>
6.1	Linear Network . . . . .	31
6.1.1	Linear Network: No Trust . . . . .	31
6.1.2	Linear Network: Trust . . . . .	33
6.2	Grid Network . . . . .	35
6.2.1	Grid Network: Static . . . . .	35
6.2.2	Grid Network: Dynamic . . . . .	47
<b>7</b>	<b>Conclusions</b>	<b>50</b>
<b>A</b>	<b>Naming Service API</b>	<b>52</b>
A.1	setName . . . . .	52
A.2	unsetName . . . . .	52
A.3	getNames . . . . .	53
A.4	getNamesNonBlocking . . . . .	53
A.5	getLogicalAddressByName . . . . .	53
A.6	getLogicalAddressByNameNonBlocking . . . . .	54
A.7	installTrustedNamingCertificate . . . . .	54
<b>B</b>	<b>Configuration</b>	<b>55</b>
B.1	Attributes . . . . .	55
B.2	Statistics . . . . .	57
<b>C</b>	<b>Naming Shell Application</b>	<b>58</b>

# 1 Design

## 1.1 Introduction

Application-layer peer-to-peer overlay networks have become popular for deploying a variety of services. Overlay networks are built on top of existing substrate networks, also called underlay networks, such as the Internet. Since overlays are application-layer networks, they are easily deployed, requiring no changes to the Internet routing and switching infrastructure.

An overlay network is composed of *overlay sockets*<sup>1</sup> that act as end-points for communication. Each overlay socket has a logical address and a physical address. The logical address is taken from the address space of the overlay network. Logical addresses are used in overlay networks to identify individual sockets as well as to route messages among sockets. Each overlay socket also has a physical address. A physical address is an address in the underlay network that an overlay socket uses for message transport.

The space from which logical addresses are chosen for a particular overlay network depends on the overlay topology being used. For example, an overlay network might have a single dimensional logical address space that uses the natural numbers as logical addresses, as seen in spanning tree overlays [3]. Circular or ring structured logical address spaces have been popularized by distributed hash table (DHT) overlay protocols such as Chord [4] and Pastry [5]. Some overlays define multi-dimensional coordinate spaces from which logical addresses are chosen as seen in the CAN [6] protocol or HyperCast's Delaunay triangulation protocol [2].

Physical location sometimes plays a role in the assignment or use of logical addresses. The goal of using physical locality when handling logical addresses is typically to minimize the use of the underlay network resources. For example, a protocol that uses a two dimensional logical address space could use latitude and longitude coordinates as logical addresses. Alternatively, a protocol that uses a ring topology can choose to build its routing tables with the logical addresses of peers that are close to the local socket in the underlay network so that routes are physically short.

Logical addresses are necessary in overlay networks to identify sockets and to route messages between sockets, but they are cumbersome to use in applications built using overlay networks. Just as Internet Protocol (IP) addresses carry no semantic information regarding the applications present at any particular Internet host, logical addresses give no clues as to the applications present at any particular overlay socket.

A mnemonic naming mechanism is needed in overlay networks because logical addresses lack application identification information and because the ability to write programs in terms of application names provides a level of abstraction more powerful than socket logical addresses. Such a naming mechanism should be flexible enough to convey semantic information about applications found on an overlay network. It should also be independent of the various overlay protocols that exist so that it can be used with any of them.

By assigning a name to an application in the network, a convenient identifier is created. Even though applications have names, messages sent to named applications must be sent using logical addresses because all message routing in an overlay network uses logical addresses. This means that

---

<sup>1</sup>In this document some discussions are written with respect to a particular overlay socket where an API is executed or a message is sent or received. The term *local socket* distinguishes such an overlay socket from other overlay sockets. Similarly, the term *peer socket* is used to refer to any overlay socket that is not the local socket.

Also, the terms *socket*, *peer*, and *node* are used as abbreviations for *overlay socket*. *Peer* implies not local, while *socket* and *node* can mean either local or non-local.

Finally, the terms *application* and *application program* mean a program that uses an overlay socket for communication in an overlay network.

some mechanism must exist that maps names to logical addresses. This problem is solved with an overlay network *naming service*.

In the Internet, a typical underlay network on which overlay networks are built, the Domain Name System (DNS) is a distributed naming service that allows application programs to use a name such as `www.example.com` rather than an IP address like `192.0.34.166`. The DNS system is not a peer-to-peer naming service; it is an example of a client/server architecture. Each DNS server answers requests of clients but also acts as a client itself when it communicates with its “parent” DNS server. DNS supports caching and replication to implement a distributed naming database. The DNS hierarchy of servers terminates in a set of “root” DNS servers that have no parent server.

In telephony networks, some mechanisms such as “speed dialing” and “directory assistance” work with names. These services allow callers to make network connections with the name of a person or a business rather than a telephone number. These telephony services are sometimes not completely integrated or automated in the network.

The HyperCast naming service, described in this document, is a distributed, peer-to-peer, topologically independent, and secure naming service for use in overlay networks. It is incorporated directly into each overlay socket. This means that any socket can participate in the naming service. Moreover, all sockets that participate in the naming service will participate in the *same* naming service. This is in stark contrast to the DNS model where all Internet hosts do not participate in the naming service, but there is a separate hierarchy of name servers and to the telephony model where various naming services exist.

In the HyperCast naming service a *name* is defined to be an arbitrary string. There is no structure imposed on a name; its meaning is determined solely by the applications that use it. A name need not be unique. When a name appears many times in an overlay network it is considered to be a name of a group. For example, an “emergency services” overlay could have multiple instances of the name “police officer” to represent all of the officers in the overlay.

The naming service provides a distributed database of *name bindings* stored in name binding tables at each socket. Each name binding links a name to a logical address, denoted as  $\langle name \leftrightarrow logical\ address \rangle$ . Note that sockets do not have names; instead sockets manage name bindings. The naming service provides operations to create, exchange, and destroy name bindings. The naming service also provides a distributed database of *trust information* that is used to verify the trust (integrity and authenticity) of name bindings. An operation is provided that exchanges trust information.

A set of operations defines the *naming service protocol*. Each operation is composed of a sequence of overlay messages:

- A *set name operation* occurs when an application creates a name binding. This operation causes a socket to add a name binding to its name binding tables for the name specified by the application.
- In a *push name bindings operation*, an overlay socket disseminates name bindings to peer sockets without being requested to do so by any peer. The push name bindings operation sends name bindings to network peers with the assumption that peers will cache the name bindings in their name binding tables and generate queries for them in the future.
- There are two types of *query operations* in which applications query name bindings. A *logical address query operation* (also called a *forward query*) which maps a name to one or more logical addresses and with a *name query operation* (also called a *reverse query*) which maps a logical address to one or more names. The forward query is the more difficult of the two since the reverse query specifies a logical address that immediately yields the destination of the reverse query.

A query operation occurs when an application requests name bindings for a name or logical address. When a socket receives a query, it searches for name bindings that satisfy the parameters of the query in its name binding tables. Any name bindings that satisfy the parameters of the query are sent to the peer that initiated the query. The peer that initiates the query is said to *pull* name bindings from the peer that responds to the query.

- An *invalidate name bindings operation* occurs when name bindings are destroyed. This operation causes sockets to remove name bindings from their name binding tables. Invalidate name bindings operations remove name bindings in the name binding tables of the local socket as well as name binding tables of peers.
- An *establish trust operation* occurs when information that verifies name binding trust is needed.

The naming service uses both broadcast and unicast message delivery to send its messages. Broadcast is in the form of multicast as opposed to flooding. When initiating the push name bindings operation, the invalidate name bindings operation, and the logical address query operation, the naming service broadcasts name bindings in the case of push and invalidate, and query parameters in the case of logical address query. This is because there is no structure, hierarchical or other, imposed on names or on sockets with respect to the naming service, i. e. there is no naming address, or naming topology. Push and invalidate name bindings operations do not have a “natural” destination for the name bindings they carry; rather they forward name bindings to any peer that they can reach. Similarly, a logical address query operation cannot be routed so that it is likely to “get closer” to a peer that can respond to the query because names are not structured with respect to overlay topology. Because of these reasons, the push, invalidate, and logical address query operations resort to broadcast delivery to initiate their operations. In contrast, the name query operation is initiated using unicast delivery because the logical address of the peer being queried is known. Similarly, the messages that pull name bindings as part of query operations use unicast delivery because the address of the peer that initiated the query is known. The set name operation is a local operation that does not send messages to peer sockets.

The trust exchange operation uses both unicast delivery and broadcast delivery. Unicast delivery is used when the location of trust information is likely to be known; broadcast delivery is used otherwise. For example, when verifying the trust of the name binding  $\langle \text{baz} \leftrightarrow \{4, 42, 30\} \rangle$ , the logical address  $\{4, 42, 30\}$  would be used as the likely location of relevant trust information for the name “baz.”

In our evaluation of the naming system, we seek to understand the trade-offs between pushing name bindings to peers proactively and pulling name bindings from peers on demand. We also seek to learn the impact of caching and socket mobility (changes of logical address) within an overlay network with respect to the naming service.

## 1.2 HyperCast Network Services

HyperCast overlay sockets have the ability to run a variety of overlay protocols. In addition to overlay protocols, HyperCast sockets can run a variety of network services. The naming service is one such network service.

Network services in HyperCast overlay sockets are implemented with a finite state machine (FSM) mechanism. Each service defines an FSM object that is activated when service messages are received. An FSM object determines the actions taken by a service in response to message receipt and timer expiration.

There are two categories of services used in HyperCast overlay sockets called *message-based* services and *stream-based* services. Message-based services create a new FSM object to manage each

message that is sent or received. Each of these FSMs will manage all activities related to the transmission of their message. This type of FSM could be used to handle retransmissions and acknowledgments for individual messages. Stream-based services create an FSM for a stream of messages rather than for a single message. This means that all messages associated with a particular stream have a single FSM managing their processing. Such stream-based services could be used to provide an in-order delivery of a stream of multimedia content.

The naming service is a stream-based service where the “stream” of information is the content of the distributed database of name bindings. This stream is not a stream of application data, but a stream of overlay network meta-data. A stream-based service was selected for the design of the naming service because the operations of the naming service are not independent. Any name binding contained in a push or pull message can satisfy many queries. The naming service provides a stream of information that is shared among queries.

### **1.3 Name Binding Exchange vs. Trust Information Exchange**

There are two major functions of the naming service: a name binding exchange service that implements the naming service proper and a trust information exchange service that exchanges information that verifies the integrity and authenticity of name bindings.

The name binding exchange service dynamically creates name bindings. Name bindings are created dynamically to allow for changes to the logical address of a socket and to update timestamps contained in name bindings (name binding contents and structure are discussed in detail in section 2.1). Name bindings are created for names that have been used as parameters to the `setName` application programmers interface (API) (Appendix A.1) at the local socket only - the naming service does not create name bindings for the logical address of another socket; there is no delegation of name binding creation. Name bindings are exchanged between network peers and possibly cached for later use. Cached name bindings can be used to respond to queries that originate in the local socket or queries received from peers.

The second function provided by the naming service is realized by an exchange of trust information. This information verifies the integrity and authenticity of name bindings not created at the local socket. The current implementation of the trust information exchange service uses X.509 v3 certificates. The trust exchange service is logically separate from the name binding exchange service so other models of trust can be implemented such as a distributed peer-to-peer model [7].

### **1.4 Naming Service API vs. Naming Service FSM**

The naming service is separated into two modules, the *naming service API* and the *naming service FSM*. The API module provides a means by which application programs can interact with the naming service. The FSM module comprises the bulk of the naming service. The name binding exchange and trust information exchange functions are both located in the FSM module.

The API module defines a set of operations that allow applications to interact with the naming service through an overlay socket. The naming service API module allows an application to:

1. Create and destroy name bindings that use the logical address of the local socket. Name bindings can be created and destroyed using certificates and private keys allowing digital signatures to be computed for name bindings. Digital signatures allow the integrity and authenticity of name bindings to be verified.
2. Query for name bindings using names and logical addresses of network peers.

3. Install certificates that act as *trust anchors* in the construction of *trust chains*. A trust chain is a series of certificates. Each certificate in the series is signed by the certificate that follows it in the series. This chain terminates at a trust anchor, which is a certificate that an application has decided to trust.

The details of the API are discussed in appendix A.

The bulk of the naming service resides in the FSM module, which is located in the message store component of the HyperCast overlay socket architecture. The message store is a component that is designed to contain FSMs that define network services. As described in section 1.2, the FSMs encode the actions that are taken in response to the receipt of messages or the expiration of timers for particular network services. The message store instantiates FSMs as needed to handle the service messages received from network peers. The message store is also responsible for saving any state information required by network services. The naming service saves state information in the form of a distributed database of name bindings and a distributed database of trust information.

The naming service FSM module is responsible for:

1. Creating name bindings with the logical address of the local socket.
2. Storing name bindings that use the logical address of the local socket.
3. Exchanging name bindings with peer sockets.
4. Caching name bindings received from peer sockets.
5. Exchanging trust information with peer sockets.
6. Storing trust information that is received from peer sockets.
7. Building trust chains.

The two modules of the naming service, the API and the FSM, need to communicate with each other. The naming service API module sends information to the naming service FSM module as APIs are invoked. Similarly, the naming service FSM module sends information to the naming service API module as API requests are fulfilled by the FSM.

The communication between the API and FSM modules cannot be done directly because of the position of the FSM within the HyperCast architecture. All communication between these modules must pass through the interfaces of the message store because the naming service FSM is located in the message store. The naming service interacts with the message store without changing the interface of the message store. The message store accepts input by receiving overlay messages. When a naming service API is invoked by an application program an overlay message is created and sent to the message store with the parameters of the API call. This message is called a *dummy message* because it is created and sent by one component of an overlay socket and received by another component of the same overlay socket; it is not passed to peer sockets in the overlay network.

Information is passed from the naming service FSM module to the naming service API module using HyperCast's *asynchronous notification* mechanism. The asynchronous notification mechanism creates and raises events caught by event handlers elsewhere in the system. The naming service API module contains event handlers for *naming service events*. Naming service events carry name bindings requested by the naming service API. Naming service events are caught and processed by the naming service API module after they are created and raised by the naming service FSM module. Dummy messages and asynchronous notification allow the two modules of the naming service to communicate with each other.

## 2 Data Structures

### 2.1 Name Binding

A *name binding* is an object that binds a name to a logical address and meta-data. In particular, a name is not said to be bound to an overlay socket; a name is said to be bound to a logical address. In addition to a name and a logical address, a name binding also contains meta-data. A name binding has a *timestamp* that holds the time at which it was created (measured in seconds since the UNIX epoch). A *logical address change counter* is included that represents the number of times that the logical address of the socket that created the name binding has changed. There is also an optional *digital signature* that allows verification of the name binding's integrity and authenticity. The *signer name* field identifies the subject common name of the signer certificate needed to verify the digital signature. The digital signature is created using the private key corresponding to the public key of the signer certificate. Finally, an *authoritative flag* indicates whether or not it was received from the peer that created it.

Name bindings are created on demand when they are sent to a peer or to an application. Each naming service FSM has a set of names that it uses to create name bindings. This set of name bindings is constructed by invocations of the `setName` API (Appendix A.1).

All data and meta-data in a name binding is immutable with the exception of the authoritative flag. The value of the authoritative flag varies depending on which socket is sending the name binding. A socket that creates a name binding sets the authoritative flag when it transmits the name binding to peers. Peers may cache the name binding if the naming service is configured for caching (see appendix B for caching configuration attributes). A cached name binding is received as authoritative when it is sent by the socket that created it. If this cached name binding is later used in response to a name binding query (Sections 5.1 & 5.2), it cannot be transmitted as authoritative; it is sent with the authoritative flag unset. Note that all data and meta-data except the the authoritative flag is used to create the digital signature. This is because the digital signature is fixed at name binding creation time; the authoritative flag is not fixed at name binding creation time. The data over which the signature is computed cannot be changed.

Name bindings have the following format when serialized for transmission in a pull or push name bindings message (Sections 3.1 & 3.2):

Auth Flag	Name Size	Name	Logical Address Size	Logical Address	Signer Name Size	Signer Name
1 byte	2 bytes	> 0 bytes	1 byte	> 0 bytes	2 bytes	> 0 bytes

Timestamp	Logical Address Change Count	Digital Signature Size	Digital Signature
8 bytes	4 bytes	2 bytes	>= 0 bytes

Only the low order bit of the authoritative flag byte is significant.

The naming service associates two state variables, *verified* and *trusted*, with each name binding that it stores. These variables record the security status of a name binding. A name binding is said to be verified if the naming service has a certificate that has been used to verify the signature of the name binding. A name binding is said to be trusted when the certificate that verified its signature is determined to be trusted.

### 2.2 Pending Logical Address Query Table

The *pending logical address query table* stores information about logical address queries that have been initiated but are not yet completed (the logical address query operation is discussed in detail in

section 5.1). An entry is added to this table each time a naming service API is invoked to perform a logical address query. The APIs that do this are `getLogicalAddressByName` (Appendix A.5) and `getLogicalAddressByNameNonBlocking` (Appendix A.6). Each socket is responsible for tracking its own queries so the pending logical address query table stores information about logical address queries that originate at the local socket only. Entries in the logical address query table are indexed by the name that is being queried. Each entry in this table has a *query serial number* (S/N) that is unique with respect to the local socket.

Entries are added to the pending logical address query table during the processing of logical address query messages (Section 3.4). Entries are removed from the pending logical address query table during the processing of pull name bindings messages (Section 3.2), push name bindings messages (Section 3.1), and certificate response messages (Section 3.7). Entries are also removed from the pending logical address query table when a pending query timer expires (Section 4.3).

Each entry of the pending logical address query table records the query parameters that a name binding must match to satisfy a pending logical address query as specified by a logical address query message. Name bindings that match the parameters of a pending logical address query are called *satisfying name bindings* for the query.

The parameters of a logical address query are:

- The name being queried - satisfying name binding must match.
- Query serial number - satisfying name binding must match if `RequireBindingMessage-SerialNumberMatch` (Appendix B) is set.
- Authoritative response required - satisfying name binding must match.
- Trusted response is required - satisfying name binding must match.
- Maximum age of name binding - satisfying name binding timestamp must indicate that name binding is not older than this number of milliseconds.
- Maximum responses - indicates the number of responses needed (e.g. maybe you need more than one, but at most five, police officers).
- Timeout - the time at which this query expires if it has not been totally satisfied.

An additional field of *start time* is kept for each query, but it is for bookkeeping only; it is not a query parameter used in selecting satisfying name bindings. The format of an entry of the pending logical address query table is shown below:

Name	Query Serial Number	Authoritative Response Required	Trusted Response Required	Max Age (ms)	Max Responses	Start (ms)	Timeout (ms)
------	---------------------	---------------------------------	---------------------------	--------------	---------------	------------	--------------

### 2.3 Pending Name Query Table

The *pending name query table* is similar to the pending logical address query table except that entries are indexed by logical addresses rather than names. The pending name query table stores information about name queries that have been initiated but are not yet completed. An entry is added to this table each time a naming service API is invoked to perform a name query. The APIs that do this are `getNames` (Appendix A.3) and `getNamesNonBlocking` (Appendix A.4). Each socket is responsible

for tracking its own queries so the pending name query table stores information about name queries that originate at the local socket only.

Entries are added to the pending name query table during the processing of name query messages (Section 3.5). Entries are removed from the pending name query table during the processing of pull name bindings messages (Section 3.2), push name bindings messages (Section 3.1), and certificate response messages (Section 3.7). Entries are also removed from the pending name query table when a pending query timer expires (Section 4.3).

Each entry in the pending name query table has a *query serial number* (S/N) that is unique with respect to the local socket. Each entry of the pending name query table records the query parameters that a name binding must match to satisfy a pending name query as specified by a name query message. Name bindings that match the parameters of a pending name query are called *satisfying name bindings* for the query.

The parameters of a name query are:

- The logical address being queried - satisfying name binding must match.
- The serial number of the query - satisfying name binding must match if `RequireBinding-MessageSerialNumberMatch` (Appendix B) is set.
- Authoritative response required - satisfying name binding must match.
- Trusted response is required - satisfying name binding must match.
- Maximum age of name binding - satisfying name binding timestamp must indicate that name binding is not older than this number of milliseconds.
- Maximum responses - indicates the number of responses needed (e.g. maybe you need more than one, but at most five, police officers)
- Timeout - the time at which this query expires if it has not been totally satisfied.

The format of an entry of the pending name query table is shown below:

Logical Address	Query Serial Number	Authoritative Response Required	Trusted Response Required	Max Age (ms)	Max Responses	Start (ms)	Timeout (ms)
-----------------	---------------------	---------------------------------	---------------------------	--------------	---------------	------------	--------------

## 2.4 Local Name Binding Table

The *local name binding table* is a name binding table that contains name bindings created at the local socket. Each table entry contains a name binding indexed by name. The values of a name binding in the local name binding table are updated dynamically when a name binding is accessed. When a name binding is updated its values for timestamp, logical address (if a particular logical address value was not specified in the call of the `setName` API that created the binding (Appendix A.1)), and logical address change counter are changed to reflect the current values of the socket. The name binding is also signed if a signature is required.

Name bindings in the local name binding table are accessed during the processing of logical address query messages (Section 3.4) and name query messages (Section 3.5). Entries are also accessed during push name bindings operations (Section 5.3). Entries are added to the local name binding

table during the processing of set binding messages (Section 3.10) generated by invocations of the `setName` API (Appendix A.1). Entries do not timeout, but they are removed during the processing of unset binding messages (Section 3.11) generated by the `unsetName` API (Appendix A.2).

The format of an entry of the local name binding table is shown here:

Name	Name Binding
------	--------------

## 2.5 Peer Name Binding Cache

The *peer name binding cache* is a name binding table that contains name bindings received from peer sockets. The size of the cache is configured with the configuration attributes `PeerBindingCachePullMaximumSize` and `PeerBindingCachePushMaximumSize` (Appendix B). These quantities can be queried at runtime using the statistics `BindingCachePullSize` and `BindingCachePushSize` (Appendix B). The `BindingCacheSize` statistic reflects the total size of the peer name binding cache.

Entries are added to the peer name binding cache upon the receipt of pull name bindings messages (Section 3.2) and push name bindings messages (Section 3.1). Entries are removed from the cache when invalidate name bindings messages (Section 3.3) are received, the peer bindings timer expires (Section 4.2), or the capacity of the cache is reached. Cached entries are selected for eviction using a least recently used cache replacement strategy.

The peer name binding cache stores name bindings read from pull name bindings messages not destined for the local socket. This behavior is called *name binding snooping* and it is controlled by the configuration attribute `SnoopBindings` (Appendix B).

The entries of the peer name binding cache are indexed by both name and logical address for forward and reverse queries, respectively. The format of an entry of the peer name binding cache is shown here:

Name/ Logical Address	Name Binding
--------------------------	-----------------

## 2.6 Pending Certificate Request Table

The *pending certificate request table* records the state of trust chains being built. Each entry of the table contains the certificate at the start of the trust chain, the current certificate that is being requested to extend the trust chain, and whether or not the current certificate request is unicast or broadcast.

As described in section 5.6, when a unicast certificate request times out, a broadcast certificate request is sent for the same certificate. If that request times out, the building of the trust chain is aborted. Entries are added and removed from the pending certificate table by an establish trust operation (Section 5.6).

The format of an entry of the pending certificate request table is shown here:

Trust Chain Start Certificate Name	Name of Currently Requested Certificate	Unicast/Broadcast Delivery of Request
------------------------------------	---	---------------------------------------

## 2.7 Pending Untrusted Query Table

The *pending untrusted query table* provides a link between the name binding exchange and the trust exchange mechanisms. The name binding exchange and the trust exchange mechanisms are distinct so that they can be modified independently. Even though these mechanisms are separated, there must be some way to use the trust exchange to establish the trust of name bindings acquired via the name binding exchange.

Entries are added to this table when untrusted bindings are received in a pull name bindings message (Section 3.1) or a push name bindings message (Section 3.2) that arrives as part of a logical address query operation (Section 5.1) or a name query operation (Section 5.2). Entries are removed from this table when an establish trust operation (Section 5.6) terminates. The fields of the pending untrusted query table are the name of the certificate that begins the trust chain needed to establish trust, the *trust serial number* (S/N) of the current outstanding certificate request message (Section 3.6), the name or logical address being queried, the query serial number of the query, and a set of untrusted logical addresses or names that could satisfy the query once the corresponding trust chain is built.

The format of an entry of the pending untrusted query table is shown below:

Certificate Name	Trust S/N	Name/ Logical Address	Query S/N	Logical Address/ Name
------------------	-----------	-----------------------	-----------	-----------------------

## 2.8 Certificate Cache

The *certificate cache* is a repository for certificates and private keys. Certificates and private keys are added to the certificate cache using the `setName` API (Appendix A). Trusted certificates are added to the certificate cache using the `installTrustAnchor` API (Appendix A). Certificates received from peers in certificate response messages (Section 3.7) as part of the establish trust operation (Section 5.6) are also added to the certificate cache. The setting of the configuration attribute `CertificateCaching` (Appendix B) control whether or not trust chains are built using cached certificates.

The cache is indexed by certificate name, which is the common name (CN) contained in the certificate. Certificates have private keys associated with them when a certificate and private key pair are used in a call of the `setName` API, otherwise the private key field is null. The trust anchor flag is set if a certificate is used as the parameter of the `installTrustedNamingCertificate` API (Appendix A.7).

Certificates are written to backing store (typically a disk file system) if the configuration attribute `WriteCertificatesToBackingStore` is set. Certificates are stored in a directory called `.certcache` when written to backing store.

The certificate cache records whether or not each certificate is *verified* and *trusted*. The meanings verified and trusted are similar to the meanings of these terms as used with respect to name bindings (Section 2.1). Verified means that the certificate cache has determined that the signature of a certificate was signed by the issuer named in the certificate. Trusted means that a certificate is trusted because a trust chain has been completed from the certificate to a trust anchor.

The certificate cache stores certificates read from certificate request messages (Section 3.6) not destined for the local socket. This behavior is called *certificate snooping* and it is controlled by the configuration attribute `SnoopCertificates` (Appendix B).

Each entry of the certificate cache has the following format:

Subject Common Name	Certificate	Private Key	Trust Anchor Flag
---------------------	-------------	-------------	-------------------

### 3 Protocol Messages

This section describes each type of naming service message and also provides a diagram showing the format of each message type. All messages share a common format prefix that is described before particular messages are discussed.

Naming service messages are encapsulated in overlay messages. Like all network services, the naming service uses an overlay message extension type called the *finite state machine extension* to carry naming service data. A finite state machine extension has a header and a payload. The format of the finite state machine extension used by the naming service is shown here:

FSM Extension Header			
FSM ID	Message Type	Stream ID	Payload
2 bytes	1 byte	4 bytes	variable size

The naming service uses particular values for the fields of the finite state machine extension header that identify an extension as a naming service message. Finite state machine extensions used by the naming service have a `FSM ID` value of `0x7`. The value of the `Message Type` field varies depending on the type of naming service message. The values of the `Message Type` field are listed in the following table:

Set Name	0x01
Unset Name	0x02
Install Certificate	0x03
Logical Address Query	0x04
Name Query	0x05
Local Name Query	0x06
Push Name Bindings	0x07
Pull Name Bindings	0x08
Invalidate Name Bindings	0x09
Certificate Request	0x0B
Certificate Response	0x0C

The naming service does not have multiple streams, so the four bytes of the `Stream ID` field have fixed values of `0x4E`, `0x41`, `0x4D`, `0x45` (ASCII for “NAME”). The `Payload` field contains the information for each naming service message. The following subsections discuss the contents of the `Payload` field in detail for each particular naming service message type.

#### 3.1 Pull Name Bindings Message

A *pull name bindings message* contains a set of name bindings. It is sent in response to a logical address query message (Section 3.4) during a logical address query operation (Section 5.1) or in response to a name query message (Section 3.5) during a name query operation (Section 5.2). A pull name bindings message is sent via unicast delivery to the socket that initiated the query. The format of a pull name bindings message is a query serial number (Section 3.4) followed by a series of name bindings. The query serial number found in a pull name bindings message matches the query serial number of the logical address query message (Section 3.4) or name query message (Section 3.5) that caused the pull name bindings message to be created and sent. See section 2.1 for the format of a name binding.

Query S/N	Binding <sub>0</sub>	Binding <sub>1</sub>	...
4 bytes	variable size	variable size	...

Each socket through which a pull name bindings message passes can read the name bindings in the message and add them to its peer name binding cache (Section 2.5). This reading of pull

bindings messages by intermediate sockets is called *name binding snooping*. Name binding snooping is controlled by the value of the configuration attribute `SNOOPBINDINGS` (Appendix B).

A pull name bindings message is processed by forwarding it to its next hop if it has not reached its destination. After the forwarding step, the name bindings contained in the pull name bindings message are added to the peer name binding cache if name binding snooping is configured or if the pull name bindings message has reached its destination. Next, the name bindings are used to inspect the pending logical address query table (Section 2.2) and pending name query table (Section 2.3). Inspecting the pending query tables causes two sets of queries to be generated. One set contains queries that have been satisfied by the name bindings in the pull name bindings message and the other set contains queries not satisfied by the name bindings that were received because the name bindings are not trusted. Each element of the set of unsatisfied queries causes an establish trust operation (Section 5.6) to begin. The establish trust operation acquires a chain of certificates starting from an untrusted certificate and ending with a trust anchor. Each member of the set of satisfied queries is passed to the application via the asynchronous notification mechanism (Section 1.4). The processing of a pull name bindings message is illustrated in the following pseudo-code:

```

if msg.destination != socket.logicalAddress
    forwardToNextHop (msg)

if snoopingBindings == True or msg.destination == socket.logicalAddress
    peerBindingsCache ← msg.bindings
    satisfiedQueries ← getSatisfiedQueries (pendingLAQueryTable, pendingNameQueryTable, msg.bindings)
    unsatisfiedQueries ← getUnsatisfiedQueriesWithUntrustedBindings (pendingLAQueryTable, pendingNameQueryTable, msg.bindings)

    foreach query in unsatisfiedQueries
        pendingUntrustedQueryTable ← query
        buildTrustChain (query.certificate) // Section 5.6

    foreach query in satisfiedQueries
        notifyApplication (query.bindings) // Section 1.4

```

### 3.2 Push Name Bindings Message

A *push name bindings message*, like a pull name bindings message, contains a set of name bindings. The set of name bindings contained in a push name bindings message is determined by the contents of the local name binding table (Section 2.4) of the socket that sends the push name bindings message or by a parameter of the `setName` API (Appendix A.1). A push name bindings message disseminates name bindings proactively to peer sockets in an overlay network. A push name bindings message contains only authoritative name bindings; name bindings received from peer sockets are not pushed.

Broadcast delivery is used to send push name bindings messages because they have no particular destination; the intent of push name bindings messages is to distribute name bindings throughout the overlay network. The number of network hops traversed by a push name bindings message is limited by the value of the configuration attribute `PUSHRADIUS` (Appendix B). The format of a push name bindings message is a series of name bindings. See section 2.1 for the format of a name binding.

Binding <sub>0</sub>	Binding <sub>1</sub>	...
variable size	variable size	...

Each socket through which a push name bindings message passes will unconditionally read the name bindings in the message and add them to its peer name binding cache (Section 2.5).

A push name bindings message containing all of the name bindings in the local name binding table is sent periodically when the periodic push timer expires (Section 4.1). A push name bindings message is also sent when the `setName` API is called if the `PUSHONSETNAME` configuration attribute

(Appendix B) is set. In the latter case, only the name binding that uses the name specified in the parameter of the `setName` API is contained in the push name bindings message.

A push name bindings message is processed by forwarding it to peer sockets to continue the broadcast. After the forwarding step, the name bindings contained in the push name bindings message are added to the peer name binding cache. Next, the name bindings are used to inspect the pending logical address query table (Section 2.2) and the pending name query (Section 2.3) table. Inspecting the pending query tables causes two sets of queries to be generated. One set contains queries that have been satisfied by the name bindings in the push name bindings message. The other set contains queries not satisfied by the name bindings that were received because the name bindings are not trusted. Each element of the set of unsatisfied queries causes a establish trust operation (Section 5.6) to begin. The establish trust operation acquires a chain of certificates starting from an untrusted certificate and ending with a trust anchor. Each member of the set of satisfied queries is passed to the application via the asynchronous notification mechanism (Section 1.4). The processing of a push name bindings message is illustrated in the following pseudo-code:

```

forwardBroadcast (msg)
peerBindingsCache ← msg.bindings
satisfiedQueries ← getSatisfiedQueries (pendingLAQueryTable, pendingNameQueryTable, msg.bindings)
unsatisfiedQueries ← getUnsatisfiedQueriesWithUntrustedBindings (pendingLAQueryTable, pendingNameQueryTable, msg.bindings)

foreach query in unsatisfiedQueries
    foreach binding in query.bindings
        buildTrustChain (binding.name)

foreach query in satisfiedQueries
    notifyApplication (query.bindings)

```

### 3.3 Invalidate Name Bindings Message

An *invalidate name bindings message*, like a pull name bindings message and a push name bindings message, contains a set of name bindings. An invalidate name bindings message is sent to remove name bindings in the peer name binding caches (Section 2.5) of peer sockets. Broadcast delivery is used to send invalidate name bindings messages because these messages have no particular destination; the intent of invalidate name bindings messages is to remove name bindings no longer valid throughout the overlay network. The number of network hops traversed by an invalidate name bindings message is limited by the value of the configuration attribute `InvalidateRadius` (Appendix B). The format of an invalidate name bindings message is a series of name bindings. See section 2.1 for the format of a name binding.

Binding <sub>0</sub>	Binding <sub>1</sub>	...
variable size	variable size	...

Each socket through which an invalidate name bindings message passes will unconditionally read the name bindings in the message and remove them from its peer name binding cache.

An invalidate name bindings message is sent when a socket leaves an overlay network and the configuration attribute `InvalidateOnLeaveOverlay` (Appendix B) is true. The invalidate name bindings message that is sent in this case contains all of the name bindings in the local name binding table (Section 2.4).

An invalidate name bindings message is processed by forwarding it to peer sockets to continue the broadcast. After the forwarding step, the name bindings contained in the invalidate name bindings message are removed from the peer name binding cache. The processing of an invalidate name bindings message is illustrated in the following pseudo-code:

forwardBroadcast (msg)  
peerBindingsCache.remove (msg.bindings)

### 3.4 Logical Address Query Message

A *logical address query message* contains a name that is used to search for name bindings in an overlay network. In addition to a name, a logical address query message contains other parameters that describe the logical address query carried in the message: a *query serial number* that is a unique integer relative to the socket that sends the logical address query message, an authoritative flag that determines if an authoritative response is required, the maximum allowable age of a name binding permitted in a response, the maximum number of name bindings permitted in a response, and the minimum number of network hops that the query should traverse. The format of a logical address query message is shown in the following diagram:

Query S/N	Auth Flag	Max Age	Max Responses	Min Hop Count	Name Size	Name
4 bytes	1 byte	4 bytes	4 bytes	4 bytes	2 bytes	variable size

Only the low order bit of the authoritative flag byte is significant.

Broadcast delivery is used to send logical address query messages because there is no particular logical address to use as a destination. The queried name in a logical address query message reveals no information about the location of name bindings that use it.

Forwarding of a logical address query message continues at least until the minimum hop count has been reached. Forwarding ceases when the minimum hop count has been reached and a name binding is found that satisfies the query. Forwarding also ceases if the maximum hop count has been reached as defined by the configuration attribute `QueryRadius` (Appendix B). If an authoritative response is required, a logical address query message must continue to be forwarded until it reaches a socket that has a name binding using the queried name in its local name binding table (Section 2.4) or the hop limit of the logical address query message is reached. A cached name binding cannot be used to satisfy an authoritative query.

The processing of a logical address query message is illustrated in the following pseudo-code:

```
localBindings ← localBindingTable.get (msg.name)
if (msg.fromLocalSocket or not msg.requiresAuth)
    peerBindings ← peerBindingCache.get (msg.name)
if (not msg.fromLocalSocket) // logical address msg.message received from peer socket
    resultBindings ← localBindings + peerBindings
    msg.minHopCount = msg.minHopCount - 1
    if (resultBindings.size < msg.maxResponses or msg.minHopCount > 0)
        forwardBroadcast (msg)
    if (resultBindings.size > 0)
        responseMsg ← PullMessage (resultBindings)
        send responseMsg
else // local message from API
    pendingLAQueryTable ← query
    satisfyingBindings ← pendingLAQueryTable.getSatisfyingBindings (msg.bindings)
    untrustedBindings ← pendingLAQueryTable.getUntrustedBindings (msg.bindings)
    if (satisfyingBindings.size < msg.maxResponses or msg.minHopCount > 0)
        forwardBroadcast (msg)
    if (satisfyingBindings.size > 0)
        notifyApplication (satisfyingBindings)
    if (satisfyingBindings.size < msg.maxResponses)
        foreach binding in untrustedBindings
            buildTrustChain (binding.name)
```

### 3.5 Name Query Message

A *name query message* contains a logical address that is used to search for name bindings in an overlay network. This type of message is used to implement a reverse lookup operation (Section 5.2) in which a logical address is mapped to a name. In addition to a logical address, a name query message contains other parameters that describe the name query represented by the message: a query serial number that is unique relative to the socket that sends the name query message, an authoritative flag that determines if an authoritative response is required, the maximum allowable age of a name binding permitted in a response, the maximum number of name bindings permitted in a response, and the minimum number of network hops that the query should traverse. The format of a name query message is shown in the following diagram:

Serial Number	Auth Flag	Max Age	Max Responses	Min Hop Count	Logical Address Size	Logical Address
4 bytes	1 byte	4 bytes	4 bytes	4 bytes	2 bytes	variable size

Only the low order bit of the authoritative flag byte is significant.

Unicast delivery is used to send name query messages because there is a particular logical address specified in the query. The queried logical address in a name query message specifies precisely where a name query message should be sent to find name bindings that use the queried logical address.

Forwarding of a name message continues at least until the minimum hop count has been reached. Forwarding ceases when the minimum hop count has been reached and a name binding is found that satisfies the query represented by the message. Forwarding also ceases if the maximum hop count has been reached as defined by the configuration attribute `QueryRadius` (Appendix B). If an authoritative response is required, a name query message must continue to be forwarded until it reaches a socket that has a name binding using the queried logical address in its local name binding table (Section 2.4) or the hop limit of the name query message is reached. A cached name binding cannot be used to satisfy an authoritative query.

The processing of a name query message is illustrated in the following pseudo-code:

```
localBindings ← localBindingTable.get (msg.name)
if (msg.fromLocalSocket or not msg.requiresAuth)
    peerBindings ← peerBindingCache.get (msg.name)

if (not msg.fromLocalSocket) // logical address msg.message received from peer socket
    resultBindings ← localBindings + peerBindings
    msg.minHopCount = msg.minHopCount - 1
    if (resultBindings.size < msg.maxResponses or msg.minHopCount > 0)
        forwardBroadcast (msg)
    if (resultBindings.size > 0)
        responseMsg ← PullMessage (resultBindings)
        send responseMsg
else // local message from API
    pendingNameQueryTable ← query
    satisfyingBindings ← pendingNameQueryTable.getSatisfyingBindings (msg.bindings)
    untrustedBindings ← pendingNameQueryTable.getUntrustedBindings (msg.bindings)
    if (satisfyingBindings.size < msg.maxResponses or msg.minHopCount > 0)
        forwardBroadcast (msg)
    if (satisfyingBindings.size > 0)
        notifyApplication (satisfyingBindings)
    if (satisfyingBindings.size < msg.maxResponses)
        foreach binding in untrustedBindings
            buildTrustChain (binding.name)
```

### 3.6 Certificate Request Message

A *certificate request message* contains a name that is used to search for a certificate among peer sockets. Certificate request messages are sent using either unicast or broadcast delivery depending on the context in which they are sent. Unicast delivery is used if the naming service FSM has received a name binding that indicates that a particular socket is likely to have a certificate. If the naming service FSM does not have any information about the possible location of a certificate it will use broadcast delivery when sending a certificate request message. Broadcast delivery is also used after a timeout (Section 4.4) for a certificate request message that was sent with unicast delivery mode. The details of selecting unicast or broadcast delivery for a certificate request message are discussed in the explanation of the establish trust operation (Section 5.6).

The format of a certificate request message is shown below:

Certificate Name Size	Certificate Name
2 bytes	variable size

Any socket can respond to a certificate request message if it has the named certificate in its certificate cache (Section 2.8).

A certificate request message continues to be forwarded until a certificate having a name that matches the name in the message is found or the maximum hop limit of the message is reached. The value of the configuration attribute `QueryRadius` (Appendix B) is used to set the maximum hop limit of a certificate request message.

The following pseudo-code illustrates the processing of a certificate request message:

```
certificate ← certificateCache.get (msg.name)
if (certificate != null)
    responseMsg ← new CertificateResponseMessage (certificate) // Section 3.7
    send (responseMsg)
else
    forward (msg)
```

### 3.7 Certificate Response Message

A *certificate response message* contains a serialized X.509 certificate. This type of message is sent via unicast to the socket that requested the certificate. All sockets on the path between the responding socket and the requesting socket can add the certificate to their certificate caches (Section 2.8) depending on the value of the configuration attribute `SnoopCertificates` (Appendix B). The format of a certificate request message is shown below:

Certificate Size	Certificate
2 bytes	variable size

The following pseudo-code illustrates the processing of a certificate response message:

```
if msg.destination != socket.logicalAddress
    forwardToNextHop (msg)
if ((msg.destination == socket.logicalAddress or snoopingCertificates == True)
    and
    msg.certificate.valid == True
    and
    msg.certificate ∉ certificateCache) // See section 2.8
    certificateCache ← msg.certificate
    (nextCertificate, completedTrustChains) ← pendingCertificateRequestTable.update (msg.certificate) // Section 2.6
    continueEstablishTrustOperation (nextCertificate);
    foreach trustChain in completedTrustChains
```

```

noLongerPendingQueries ← pendingUntrustedQueryTable.update (trustChain)
foreach query in noLongerPendingQueries.logicalAddressQueries
    satisfi edQueries ← pendingLogicalAddressQueryTable.attemptToSatisfy (query);
foreach query in noLongerPendingQueries.nameQueries
    satisfi edQueries ← pendingNameQueryTable.attemptToSatisfy (query);
notifyApplication (satisfi edQueries.bindings) // Section 1.4

```

### 3.8 Local Name Query Message

A *local name query message* is used to query the name bindings in the local name binding table (Section 2.4) of the local socket. It is sent from the naming server API to the naming server FSM; the naming service FSM does not generate messages of this type. Other than the message type itself there is no information that needs to be associated with this message.

The following pseudo-code illustrates the processing of a local name query message:

```

localBindings ← localBindingTable.bindings
notifyApplication (localBindings)

```

### 3.9 Install Trusted Certificate Message

An *install trusted certificate message* contains a serialized X.509 certificate that an application has decided to trust. The certificate contained in the message is added to the certificate cache (Section 2.8) and marked as a trust anchor which means that it can terminate a trust chain of certificates.

Certificate Size	Certificate
2 bytes	variable size

The following pseudo-code illustrates the processing of an install trusted certificate message:

```

msg.certificate.trusted ← True
certificateCache ← msg.certificate

```

### 3.10 Set Binding Message

A *set binding message* is used to create a name binding in the local name binding table (Section 2.4). The contents of a set binding message are a name in string format and, optionally, an X.509 certificate and private key. If a certificate is present, the subject common name (CN) of the certificate must match the name specified in the contents of the message, or the name field of the message must be empty.

Name Size	Name	Certificate Size	Certificate	Private Key Size	Private Key
2 bytes	variable length (can be zero)	2 bytes	variable length (can be zero)	2 bytes	variable length (can be zero)

The following pseudo-code illustrates the processing of a set binding message:

```

validCertificate ← certificate != null and certificate.isValid (privateKey)
if (certificate == null or validCertificate)
    localBindingTable ← name
    if (validCertificate)
        certificateCache ← (certificate, privateKey)
    if (PushOnSetName == true)
        pushOperation (name) // Section 5.3

```

### 3.11 Unset Binding Message

An *unset binding message* is used to remove a name binding from the local name binding table (Section 2.4). The contents of an unset binding message are a name in string format and, optionally, an X.509 certificate and private key. If a certificate is present, the subject common name of the certificate must match the name specified in the contents of the message, or the name field of the message must be empty.

Name Size	Name	Certificate Size	Certificate	Private Key Size	Private Key
2 bytes	variable length (can be zero)	2 bytes	variable length (can be zero)	2 bytes	variable length (can be zero)

The following pseudo-code illustrates the processing of an unset binding message:

```
validCertificate ← certificate != null and certificate.isValid (privateKey)
if (certificate == null or validCertificate)
    localBindingTable.remove (name)
if (InvalidateOnUnsetName == true)
    invalidateOperation (name) // Section 5.5
```

## 4 Timers

### 4.1 Periodic Push Timer

The *periodic push timer* determines when periodic push name bindings operations occur. The period of this timer is set with the configuration attribute `PushPeriod` (Appendix B). If this attribute is set to 0 then periodic push name bindings operations will not occur. Periodic push name bindings operations transmit all bindings in the local binding table (Section 2.4) to network peers for caching.

### 4.2 Peer Bindings Timer

The *peer bindings timer* signals when expired entries of the peer name binding cache (Section 2.5) are removed. The period of this timer is set with the configuration attribute `BindingTimeout` (Appendix B).

### 4.3 Pending Query Timer

A *pending query timer* signals when an expired entry of either the pending logical address query table (Section 2.2) or the pending name query table (Section 2.3) is removed. Each query that is pending has its own pending query timer task. The length of this timer is set with the configuration attribute `BlockingAPITimeout` (Appendix B).

### 4.4 Certificate Request Timer

A *certificate request timer* is set for each certificate request message (Section 3.6) that is sent. When this timer expires the action taken by the naming service FSM depends on the delivery mode used to send the corresponding certificate request message. If unicast delivery was used, then another certificate request message is sent using broadcast delivery and another certificate request timer is set. If broadcast delivery was used, no further certificate requests are sent and no certificate request timer is set.

## 5 Protocol Operations

Operations of the naming service protocol can be categorized into two groups: name binding exchange operations and trust information exchange operations. The former are concerned with the lookup and dissemination of name bindings; the latter are concerned with the exchange of information that is used to establish trust of name bindings.

### 5.1 Logical Address Query Operation

A *logical address query operation* (a *forward query* query) is a name binding exchange operation that is initiated by a socket that wishes to learn the logical addresses associated with a given name. Such a socket is called the *querier* of the logical address query operation. The logical address query operation begins by inspecting name binding tables (Sections 2.4 and 2.5) at the querier for name bindings that satisfy the logical address query. Messages are exchanged with peers of the querier if the logical address query cannot be satisfied by name bindings contained in the querier's name binding tables.

Message exchange with peers begins by sending a logical address query message (Section 3.4) with broadcast delivery. The logical address query message contains the parameters of the query. These parameters are the name that is being queried, the query's serial number (unique with respect to the querier), the maximum allowable age of a name binding containing the specified name, the maximum number of satisfying name bindings in which the querier is interested, whether or not satisfying name bindings must be sent from an authoritative source, whether or not satisfying name bindings must be trusted, and the minimum number of network hops (minimum query radius) that the logical address query message must traverse.

The parameters of the logical address query message can be controlled by the parameters of the `getLogicalAddressByName` API (Appendix A.5) or by the settings of the configuration attributes `MaximumResponseBindingAge`, `MaximumResponsesPerQuery`, `AuthoritativeResponsesOnly`, `TrustedResponsesOnly`, and `MinimumQueryRadius` (Appendix B).

If a logical address query operation has not completed within the number of milliseconds specified by the `BlockingAPITimeout` configuration attribute (which can be overridden with a parameter of the `getLogicalAddressByName` API) the logical address query operation is said to have timed out and will terminate. The timeout is controlled by the pending query timer (Section 4.3).

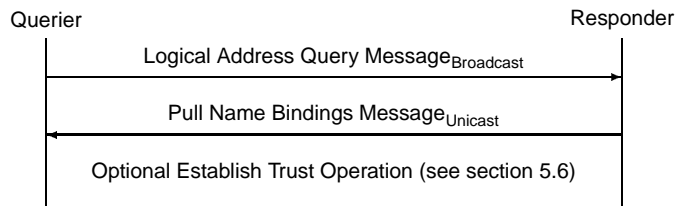
When a logical address query message is received by a socket that has one or more name bindings that satisfy the parameters of the query, the receiving socket responds to the logical address query with a pull name bindings message (Section 3.1) containing the name bindings that satisfy the parameters of the logical address query. A socket that sends a pull name bindings message is called a *responder*. There can be more than one responder for a single logical address query operation because logical address query messages are broadcast. Multiple responders can also occur due to the minimum query radius being greater than 1. The responder sends a pull name bindings message with unicast delivery to the querier.

As a pull name bindings message traverses an overlay network, all intermediate peers on the path from the responder to the querier read the contents of the message. These intermediate peers will store the name bindings contained in the pull name bindings message in their local peer name binding caches (Section 2.5) if name binding snooping (Section 3.1) is configured with the configuration attribute `SnoopBindings` (Appendix B).

The name bindings received in a pull name bindings message are inspected by the pending logical address query table (Section 2.2) to decide if the name bindings satisfy any pending logical address

queries.<sup>2</sup> Satisfying name bindings are delivered to the application by the asynchronous notification system using a naming service event (Section 1.4). Name bindings received from peers in pull name binding messages are added to the peer name binding cache based on the setting of the configuration attribute `CachePullBindings` (Appendix B).

Some logical address query operations specify that only trusted name bindings can satisfy the query. In these cases, an establish trust operation could be initiated if a peer does not have sufficient trust information to establish trust for a name binding that satisfies one of its pending queries. The establish trust operation is discussed in section 5.6.



## 5.2 Name Query Operation

A *name query operation* (a *reverse query*) is a name binding exchange operation that is initiated by a socket that wishes to learn the names associated with a given logical address. Such a socket is called the *querier* of the name query operation. The name query operation begins by inspecting name binding tables (Sections 2.4 and 2.5) at the querier for name bindings that satisfy the name query. Messages are exchanged with peers of the querier if the name query cannot be satisfied by name bindings contained in the querier's name binding tables.

Message exchange with peers begins by sending a name query message (Section 3.5) with unicast delivery. The name query message contains the parameters of the query. These parameters are the logical address that is being queried, the query's serial number (unique with respect to the querier), the maximum allowable age of a name binding containing the specified logical address, the maximum number of satisfying name bindings in which the querier is interested, whether or not satisfying name bindings must be sent from an authoritative source, whether or not satisfying name bindings must be trusted, and the minimum number of network hops (minimum query radius) that the name query message must traverse.

The parameters of the name query message can be controlled by the parameters of the `getNames` API (Appendix A.3) or by the settings of the configuration attributes `MaximumResponseBindingAge`, `MaximumResponsesPerQuery`, `AuthoritativeResponsesOnly`, `TrustedResponsesOnly`, and `MinimumQueryRadius` (Appendix B).

If a name query operation has not completed within the number of milliseconds specified by the `BlockingAPITimeout` configuration attribute (which can be overridden with a parameter of the `getLogicalAddressByName` API) the name query operation is said to have timed out and will terminate. The timeout is controlled by the pending query timer (Section 4.3).

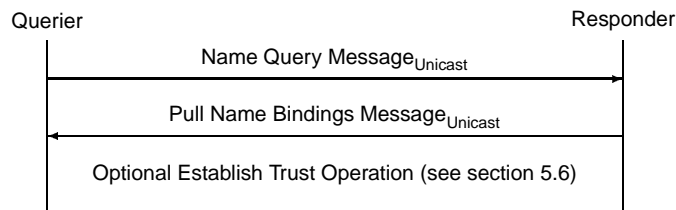
When a name query message is received by a socket that has one or more name bindings that satisfy the parameters of the query, the receiving socket responds to the name query with a pull name bindings message (Section 3.1) containing the name bindings that satisfy the parameters of the name query. A socket that sends a pull name bindings message is called a *responder*. There can be more than one responder for a single name query operation if the minimum query radius is greater than 1. The responder sends a pull name bindings message with unicast delivery to the querier.

<sup>2</sup>Pending naming queries are considered as well. Also, name bindings received by push name bindings messages may cause pending queries to be satisfied.

As a pull name bindings message traverses an overlay network all intermediate peers on the path from the responder to the querier read the contents of the message. These intermediate peers will store the name bindings contained in the pull name bindings message in their local peer name binding caches (Section 2.5) if name binding snooping (Section 3.1) is configured with the configuration attribute `SnoopBindings` (Appendix B).

The name bindings received in a pull name bindings message are inspected by the pending name query table (Section 2.3) to decide if the name bindings satisfy any pending name queries<sup>3</sup>. Satisfying name bindings are delivered to the application by the asynchronous notification system using a naming service event (Section 1.4). Name bindings received from peers in pull name binding messages are added to the peer name binding cache based on the setting of the configuration attributes `CachePullBindings` (Appendix B).

Some name query operations specify that only trusted name bindings can satisfy the query. In these cases an establish trust operation could be initiated if a peer does not have sufficient trust information to establish trust for a name binding that satisfies one of its pending queries. The establish trust operation is discussed in section 5.6.

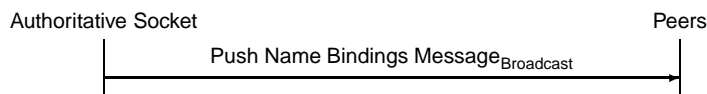


### 5.3 Push Name Bindings Operation

A *push name bindings operation* is a name binding exchange operation that causes authoritative name bindings to be disseminated to peers via broadcast. A push name bindings operation is implemented by a push name bindings message (Section 3.2) that contains name bindings of the local name binding table (Section 2.4) of the socket that initiates the push name bindings operation. Each peer that receives a push name bindings message stores the name bindings contained in the message in its peer name binding cache (Section 2.5).

A push name bindings operation occurs when the periodic push timer (Section 4.1) expires. A push name bindings operation also occurs when a socket changes its logical address or joins an overlay network, depending on the settings of the configuration attributes `PushOnChangeOfLogicalAddress` and `PushOnJoinOverlay`, respectively (Appendix B). In all of these cases, all of the name bindings in the local name binding table will be pushed.

Calling the `setName` API (Appendix A.1) when the configuration attribute `PushOnSetName` is set (Appendix B) also causes a push name bindings operation to occur. Only the name that was used in the API call will be pushed in this case.



<sup>3</sup>Pending logical address queries are considered as well. Also, name bindings received by push name bindings messages may cause pending queries to be satisfied

## 5.4 Set Name Operation

The *set name operation* is a name binding exchange operation that creates an entry in the local name binding table (Section 2.4). This operation occurs when the `setName` API (Appendix A.1) is called.

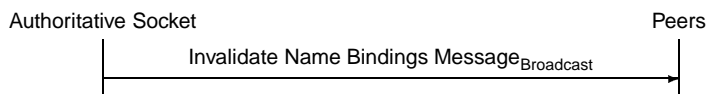
This operation can occur with a name only or with a certificate and private key pair. If a certificate and private key are used, the subject common name contained in the certificate is the name added to the local name binding table. Passing a certificate and a private key allows the naming service FSM to create digital signatures for name bindings that use the name contained in the certificate.

The set name operation is a local operation that does not exchange messages among peers. The set name operation could cause a push name bindings operation (Section 5.3) to occur if the configuration attribute `PushOnSetName` is set (Appendix B).

## 5.5 Invalidate Name Bindings Operation

An *invalidate name bindings operation* is a name binding exchange operation that causes name bindings to be invalidated both at the socket that initiates the invalidate name bindings operation as well as at peer sockets. A socket can only invalidate name bindings for which it is authoritative, i. e. only name bindings contained in its local name binding table (Section 2.4). An invalidate name bindings operation is implemented by an invalidate name bindings message (Section 3.3) that contains name bindings no longer considered to be valid. Broadcast delivery is used to send invalidate name bindings messages. Each peer that receives an invalidate message removes all name bindings contained in the message from its peer name binding cache (Section 2.5).

Invalidate name bindings operations are executed when the `unsetName` API (Appendix A.2) is invoked or when a socket leaves an overlay network if the configuration attribute `InvalidateOnLeaveOverlay` (Appendix B) is set.



## 5.6 Establish Trust Operation

An *establish trust operation* is a trust information exchange operation that constructs trust chains. Trust chains are built to verify the integrity and authenticity of name bindings. A trust chain is a series of certificates. Each certificate in the series is signed by the certificate that follows it in the series. This chain terminates at a trust anchor, which is a certificate that an application has indicated it trusts by calling the `installTrustedNamingCertificate` API (Appendix A.7). An establish trust operation begins after a logical address query operation (Section 5.1) or a name query operation (Section 5.2) receives an untrusted name binding.

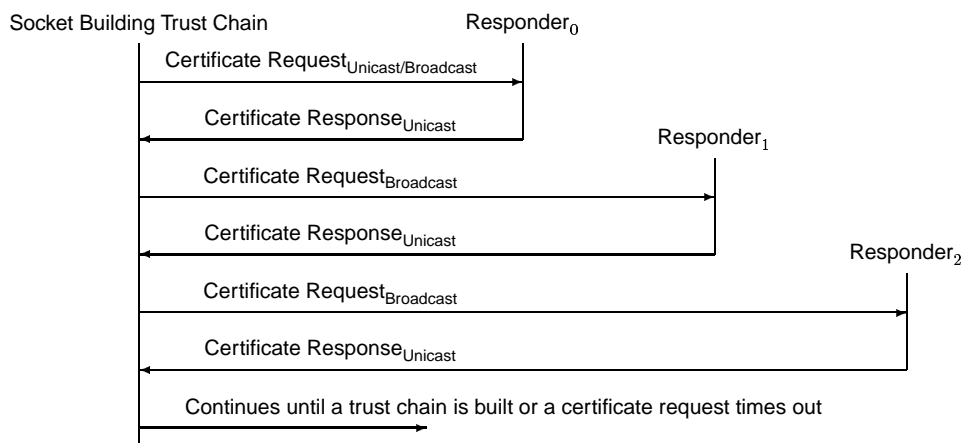
The establish trust operation consists of a series of certificate request messages (Section 3.6) and certificate response messages (Section 3.7). The initial certificate request message is sent with unicast delivery if the name of the certificate being requested is the name of the name binding that caused the establish trust operation to occur. In all other situations broadcast delivery is used for the initial certificate request message. Broadcast delivery is used for all subsequent certificate request messages.

When a certificate request message is sent, a certificate request timer is set (Section 4.4). This timer determines how long to wait for a corresponding certificate response message. If this timer expires for a certificate request message that was sent with unicast delivery, another certificate request message, for the same certificate, will be sent using broadcast delivery and the certificate request timer

is reset. If this timer expires for a certificate request message that was sent with broadcast delivery the establish trust operation terminates.

Certificate response messages are sent in response to certificate request messages when a peer that receives a certificate request message has the certificate named in the certificate request message. All certificate response messages are sent via unicast. Each peer on the path traveled by a certificate response message saves the certificate in its certificate cache.

When building a trust chain, certificate requests continue to be sent for successive certificates until the trust chain is completed or until a certificate request message times out and a matching certificate response message is not yet received. The progress of trust chain construction is tracked in the pending certificate request table (Section 2.6).



## 6 Evaluation

Performance of the naming service is evaluated with a series of experiments running on a cluster of Sun Microsystems Sunfire computers each having dual 2.8 GHz Xeon processors, 512 MB of physical memory, and a 1 Gbps ethernet interface. All machines in the cluster are connected together by a single 1 Gbps ethernet switch. UDP datagrams are used for message transport in all experiments. The performance evaluation of the naming service has many goals:

- Measurement of query response latency and query success ratio in the absence of name binding caching. Latency is defined to be the amount of time that elapses between the sending of a query and the receipt of a query response. Success ratio is the measure of the percentage of queries that receive a response. All caching is disabled when collecting these measurements to force all messages to travel the entire distance between a querying socket and a responding socket. Latency and success ratio are measured as a function of the rate at which queries are sent as well as the number of network hops between a socket that sends a query and the socket that responds to the query. These measurements show the rate at which the naming service can successfully process queries and the length of time needed to resolve a query.
- Measurement of adding a trust operation to the basic query operation. This measurement shows the cost of trust: exchanging trust information and performing cryptographic trust operations after a query operation. Again, latency and success ratio are measured. Trust information is not cached so each query causes trust information to flow between the querying socket and the responding socket.
- Given an understanding of latency and success ratio from earlier experiments, how do different settings of push and pull radii effect the performance of query operations in the naming service? Peer name binding cache (Section 2.5) settings are inherently tied to the analysis of push and pull radii because name bindings transmitted in push and pull messages are stored in peer name bindings caches. Without caching, name bindings would not be distributed throughout an overlay network. Measurements are taken for query response latency, query success ratio, amount of data sent, and cache size in experiments that vary pull radius, push radius, and maximum cache size.
- Measurement of the effect of socket mobility on the naming service. Experiments are conducted in which a mobility setting is specified that controls how many sockets per second change their logical address. This parameter is varied and measurements are taken for query response latency, query success ratio, amount of data sent, and cache size.

The complete list the experiment parameters is:

- Network size: measured in hops for linear experiments (Section 6.1) and rows and columns of peers for grid experiments (Section 6.2).
- Push radius: the maximum number of hops that push name bindings message travels, measured in network hops.
- Pull (query) radius: the maximum number of hops that query message travels, measured in network hops.
- Number of queries sent. In linear experiments (Section 6.1), a single peer sends all queries; in grid experiments (Section 6.2), all peers send queries.

- Query rate: the rate at which queries are sent: queries per second (qps). This rate always means the total number of queries sent over the entire network, not per socket.
- Maximum cache size: This parameter is divided into the maximum number of name bindings that are cached as a result of receiving pull name bindings messages and the maximum number of name bindings that are cached as a result of receiving push name bindings messages.
- Mobility: the number of sockets per second that change their logical addresses in an overlay network. Changing logical addresses of sockets emulates mobility in realistic situations where physical position is changed, e.g. an overlay that uses latitude and longitude tuples as logical addresses in which each peer is a taxi moving in city streets. When logical addresses are changed the overlay protocol automatically restabilizes the overlay network. This causes added network traffic that happens simultaneously with experimental measurements.
- Trust: whether or not name bindings received from peers must be trusted to satisfy a query.

Performance metrics are presented as averages computed over all sockets in the network under test. The complete list of performance metrics is:

- Success ratio: the measure of the percentage of queries that receive a response. The success ratio measurement gives insight into the maximum load that the naming service can sustain. Queries can fail due to forwarding errors in an unstable network, UDP packet loss, exceeding maximum hop limit, and queue overflow due to backlog at individual peers.
- Latency: the amount of time that elapses between the sending of a query and the receipt of a query response for that query.
- Cache size: a measurement of the number of name bindings held in name bindings tables of overlay sockets.
- Data sent: a measurement of the amount of information sent in the overlay network.

The overlay protocol for all experiments is the Delaunay triangulation. In a Delaunay triangulation, each overlay socket has a 2-element coordinate tuple as its logical address. Each socket is logically placed at some point on the Cartesian plane such that the  $x$  and  $y$  coordinate of each logical address is an integer in the range of  $[0, 9999]$ .

Two types of network topologies are built for the evaluation: a single-dimensional linear network and a two-dimensional grid network. Linear networks are used to acquire baseline latency and success ratio data. The cost of trust operations is also considered using linear networks. With grid networks, the settings of push and pull (query) radii and maximum cache size are varied to explore how each impacts the performance of the naming service. Grid experiments are conducted under static (fixed logical addresses) and dynamic (changing logical addresses) network conditions.

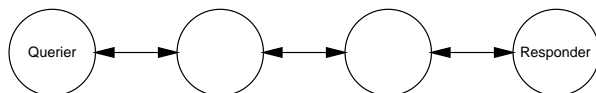
Performance measurements are presented in a series of graphs. Each graph has an  $x$ -axis and a  $y$ -axis. The  $x$ -axis shows the values of some experimental parameter; the  $y$ -axis shows the values of some observed performance metric. Each graph contains a set of curves, each curve corresponding to a setting of some experimental parameter. The points on each curve are observed values of the performance metric for a single experiment. With the  $x$ -axis and a set of curves, two experimental parameters can be varied in each graph. For example, one graph may have an  $x$ -axis in units of network hops, a  $y$ -axis that measures query success rates, and a set of curves with each curve representing a different value of query rate. When a graph reports that 25,000 queries are performed for each

experiment it means that each point on each curve represents an average value computed over 25,000 queries.

Three-dimensional graphs are also used to display the results depicted in the two-dimensional graphs. These graphs are provided as visual aids that allow the viewer to consider the experimental results in a different perspective.

## 6.1 Linear Network

The first topology is a single-dimensional linear network of overlay sockets. In linear networks, sockets are placed in a line. For example, a four socket (three hop) linear network might have sockets with logical addresses: (10, 10), (20, 10), (30, 10), and (40, 10). In linear networks, the socket having the least  $x$ -coordinate value (the “leftmost” socket) is the *querier*. The querier socket sends logical address query messages at a fixed rate in search of the logical address of a particular name. The socket having this name, the *responder*, is the socket with the greatest  $x$ -coordinate value (the “rightmost” socket). The responder sends pull name bindings messages (Section 3.1) to the querier. Thus, logical address query messages travel from left to right across the entire length of the linear network and pull name bindings messages travel across the entire network from right to left.



Each logical address query message must traverse the entire length of the network as there is no caching of name bindings. For simplicity, all logical address queries request the same name. Logical address queries are distinguished from one another by a unique query serial number carried by each logical address query message. Pull name bindings messages echo query serial numbers so that queries can be matched with responses.

Linear experiments are conducted to understand the latency and success ratio of query operations for various network hop distances and various query rates, measured in queries per second. Up to 18 cluster nodes are used for linear experiments. Cluster nodes are given an ordering, and sockets are allocated on the nodes in a round-robin fashion beginning with the querier and ending with the responder. This is done so that sockets that are adjacent in the overlay network do not reside on the same system. Many overlay sockets can be on a single cluster node as long as they are not adjacent in the logical overlay. This forces all messages to be sent physically in the underlay substrate network (IP) using ethernet interfaces.

### 6.1.1 Linear Network: No Trust

The goal of the first set of linear network experiments is to measure the query latency and success ratio of logical address query operations (Section 5.1). The experimental parameters that are varied are hop count between querier and responder and query rate. Hop count is varied from 10 to 100 network hops, in increments of 10 hops. Query rate is varied from 50 qps to 100 qps in increments of 50 qps. Each data point represents an average value computed over 6,000 logical address queries.

Figure 1 shows a graph in which latency is measured as a function of hop count. Each curve on the graph represents a different setting of queries per second (qps) from 50 qps to 1000 qps in increments of 50 qps. Figure 2 shows the same data as Figure 1, but latency is shown as a function of queries per second instead of as a function of hop count. Each curve represents a different hop count from 10 hops to 100 hops in increments of 10 hops. Figure 3 shows the success ratio of queries as a function of queries per second with each curve representing a different hop count, also from 10 hops to 100 hops in increments of 10 hops. Query rate is varied from 50 qps to 1000 qps in increments of 50 qps.

These graphs show that latency is linear in the number of network hops. Moreover, the data show the naming service is capable of handling on the order of 1000 queries per second with response times of less than 0.1 second in networks of length 100 hops or less. Queries are nearly 100% successful for 500 queries per second or less over all network lengths. At higher traffic loads, the success ratio drops slightly due to UDP packet loss.

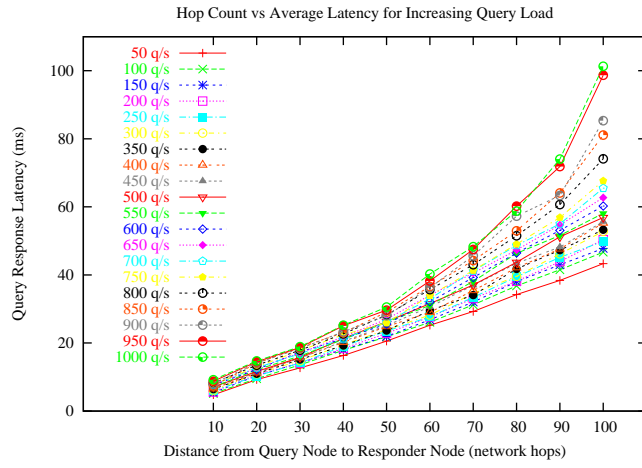


Figure 1: Linear Network, No Trust, Hop Count vs. Latency

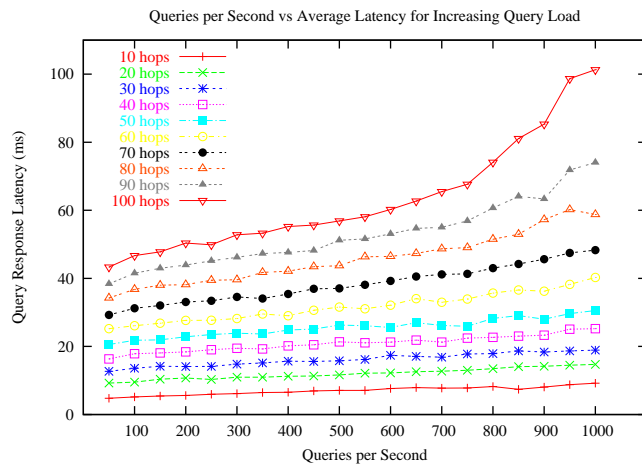


Figure 2: Linear Network, No Trust, Queries Per Second vs. Latency

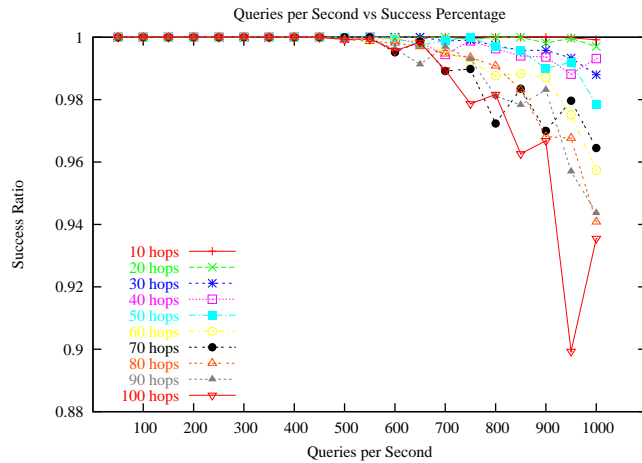


Figure 3: Linear Network, No Trust, Success Ratio

### 6.1.2 Linear Network: Trust

Single-dimensional linear network topology experiments are used to measure the performance of a logical address query operation (Section 5.1) followed by an establish trust operation (Section 5.6). An establish trust operation following a logical address query operation verifies the trust (integrity and authenticity) of the name binding received in response to the query. The goal of this set of experiments is to understand the added cost of verifying trust in terms of latency and success ratio when exchanging a certificate and performing cryptographic functions.

In addition to exchanging a logical address query message (Section 3.4) and a pull name bindings message (Section 3.1), each logical address query operation (Section 5.1) causes an establish trust operation (Section 5.6) to occur in which a certificate request message and a certificate response message are exchanged. In these experiments, the querier socket sends logical address query messages (Section 3.4) at a fixed rate to the responder. When the responder receives a logical address query message, it updates the fields of the name binding being queried, in particular it recomputes the name binding's digital signature. The responder sends the updated name binding in a pull name bindings message to the querier socket in reply to the query. When the querier receives the pull name bindings message it attempts to verify the trust of the name binding but fails because it does not have the required certificate. The querier sends a certificate request message to the responder for the certificate that was used to sign the name binding. The responder receives the certificate request message and replies to the querier with a certificate response message containing the requested certificate. The querier receives the certificate request message and verifies the name binding with the certificate in the message.

As is the case with the linear network experiments of the previous section (Section 6.1.2), linear network experiments with trust operations do not cache query responses at intermediate sockets between the querier and the responder. However, in experiments with trust operations, the querier does store name bindings that it receives so that it can later verify name binding signatures upon certificate receipt. The name bindings stored by the querier are not considered when logical address query messages are sent - all logical address query messages proceed to the responder. For simplicity, all queries request the same name. Queries are distinguished from each other by a unique query serial number carried by each logical address query message. Pull name bindings messages echo the query serial numbers so that queries can be matched with responses. All certificate request messages ask for the same certificate since the same name is used for all logical address query messages. Certificate request messages also carry a unique serial number that is echoed by certificate response messages. Echoing the serial number from a request allows requests and responses to be matched.

The graphs in Figures 4, 5, and 6 show the results of the experiments that use a linear network with trust operations. Each data point represents an average value computed over 6,000 logical address queries. Figure 4 shows query latency as a function of the number of network hops between the querier and the responder. The number of hops is varied from 10 to 100, in increments of 10 hops. Each curve in the graph represents a particular query rate. Query rate is varied from 50 qps to 100 qps in increments of 10 qps. Figure 5 shows the same data as Figure 4, but latency is shown as a function of queries per second instead of as a function of hop count. Each curve represents a different hop count from 10 hops to 100 hops in increments of 10 hops. Figure 6 shows the success ratio of queries as a function of queries per second with each curve representing a different hop count, also from 10 hops to 100 hops in increments of 10 hops. Query rate is varied from 50 qps to 100 qps in increments of 10 qps.

These data show that the name service can support on the order of 80 trusted queries per second before serious performance degradation occurs. This sharp decrease in performance corresponds to CPU saturation, not a network bottleneck.

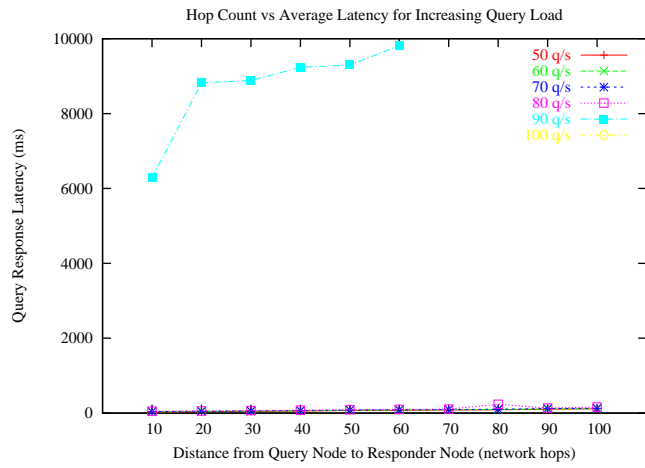


Figure 4: Linear Network, Trust, Hop Count vs. Latency

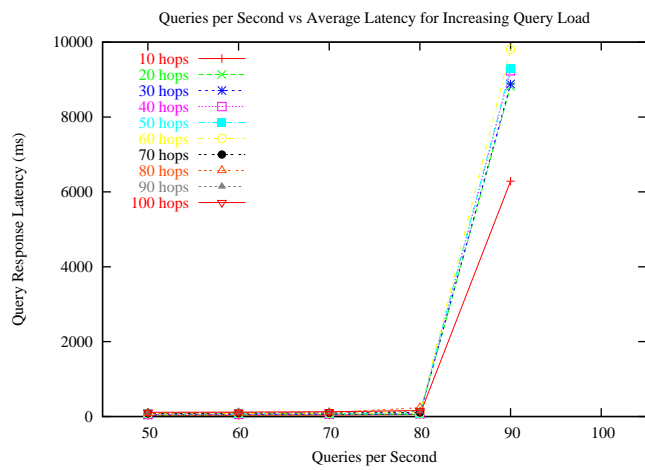


Figure 5: Linear Network, Trust, Queries Per Second vs. Latency

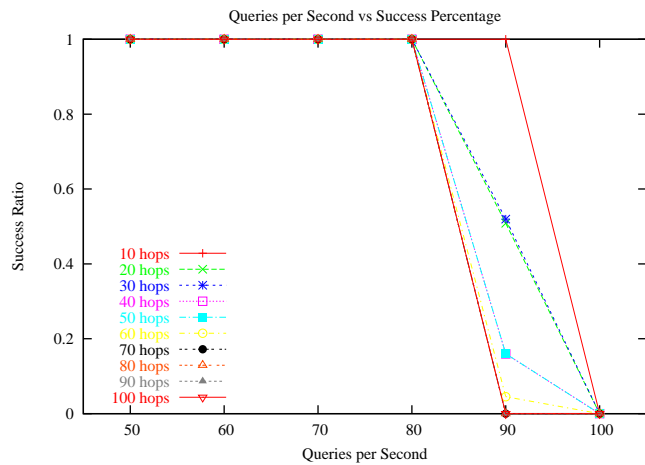


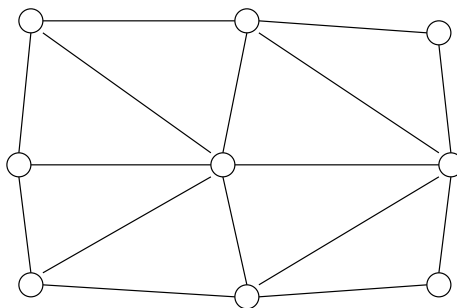
Figure 6: Linear Network, Trust, Success Ratio

## 6.2 Grid Network

The second type of network topology used to measure the performance of the naming service is a two-dimensional grid. In grid network experiments, sockets are arranged in the logical address space in a regular grid pattern instead of sockets arranged in a line. For example, a nine socket grid might have coordinates: (10, 10), (10, 20), (10, 30), (20, 10), (20, 20), (20, 30), (30, 10), (30, 20), and (30, 30). Two-dimensional grids are used in experiments that vary push and pull radii. The grid allows push name bindings messages and logical address query messages to radiate in all directions from pushing and querying sockets.

The Delaunay triangulation protocol does not permit four or more sockets to be on the same circle because the triangulation is not unique in that case. To avoid having four or more sockets on the same circle exact grid coordinates are not used. Instead, the socket coordinates are randomly shifted by a small amount to reduce the probability of four or more sockets being placed on a circle. If four sockets are on a circle after this random movement, the Delaunay triangulation protocol will detect this situation automatically and force sockets to change their logical addresses until this is no longer the case.

In the grid pattern, each socket has neighbors to its left, right, top, and bottom (unless the socket is on the edge of the grid). In addition to these neighbors, the diagonals of the grid connect neighbors because a triangulation is formed. The orientation of diagonal links is determined by the positions of the sockets that are linked by the diagonal.



In a grid there is not a single querier as in the linear experiments, instead all sockets participate in the querying. Sockets are chosen at random to initiate logical address query operations so that the specified aggregate query rate is achieved. Each logical address query operation randomly chooses a name to query. There are as many names in the network as there are sockets in the network. Each socket has a single entry in its local name binding table (Section 2.4). No two sockets have name bindings for the same name.

The grid experiments are separated into static networks and dynamic networks. In static grid experiments, each socket keeps the same logical address for the duration of the experiment. In dynamic grid experiments, sockets change their logical addresses at a configurable rate.

Each grid experiment uses 16 cluster nodes. The size of the grid overlay network in each experiment is 40 rows  $\times$  40 columns of overlay sockets for a total of 1600 overlay sockets.

Grid experiments are conducted to understand how the performance of the naming service is affected by various push and pull (query) radii settings. Various settings of maximum cache size are considered as well. Finally, socket mobility is considered in dynamic grid experiments.

### 6.2.1 Grid Network: Static

Static grid network experiments measure the effects of various push and pull radii as well as various settings of the maximum peer name bindings cache size. Name bindings received in push name

bindings messages are always cached, but the amount of caching of name bindings received in pull name bindings messages is varied. The performance metrics that are measured in static grid network experiments are query response latency, query success ratio, amount of data sent, and amount of name bindings cached at each socket.

Each static grid network experiment has two phases. In the first phase, name bindings are pushed to network peers. The pushed name bindings are stored in peer name binding caches for the duration of each experiment. All logical address query operations take place in the second phase of each experiment. Pull name bindings messages are sent in response to logical address query messages. Pull name bindings messages are sent from the socket that responds to a logical address query to the socket that initiated the logical address query. All intermediate sockets that forward pull name bindings message between responder and querier store the name bindings contained in the pull name bindings messages (see the description of binding snooping in section 2.5).

The measurements for each performance metric are presented in a set of three graphs. Each graph has a different value for the maximum number of name bindings cached as a result of pull name bindings messages. The three cache settings are: (1) no name bindings received in pull name bindings messages are cached, (2) all name bindings received in pull name bindings messages are cached, and (3) a maximum of 400 name bindings (one-quarter of all name bindings in the network) received in pull name bindings messages are cached. In each graph, the pull (query) radius is varied between 2 hops and 30 hops, in increments of 2 hops. The push radius is varied between 2 hops and 30 hops, in increments of 4 hops. The  $x$ -axis is used to show pull (query) radius and each curve in each graph represents a particular push radius setting. An aggregate query rate of 100 logical address queries per second over all sockets in the grid is used for each experiment. Each experiment produces 25,000 queries.

The graphs in Figures 7, 8, and 9 show success ratio for various cache settings. The graphs in Figures 10, 11, and 12 show latency for various cache settings. The graphs in Figures 13, 14, and 15 show data sent for various cache settings. The graphs in Figures 16, 17, and 18 show cache size for various cache settings.

The graphs in Figures 19, 20, and 21 are three-dimensional views of the data shown in Figures 7, 8, and 9. The graphs in Figures 22, 23, and 24 are three-dimensional views of the data shown in Figures 10, 11, and 12. The graphs in Figures 25, 26, and 27 are three-dimensional views of the data shown in Figures 13, 14, and 15. The graphs in Figures 28, 29, and 30 are three-dimensional views of the data shown in Figures 16, 17, and 18.

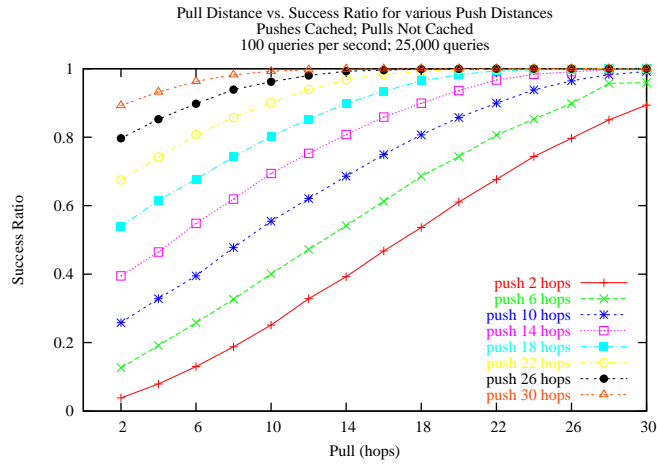


Figure 7: Grid Network, Success Ratio, Only Pushes Cached

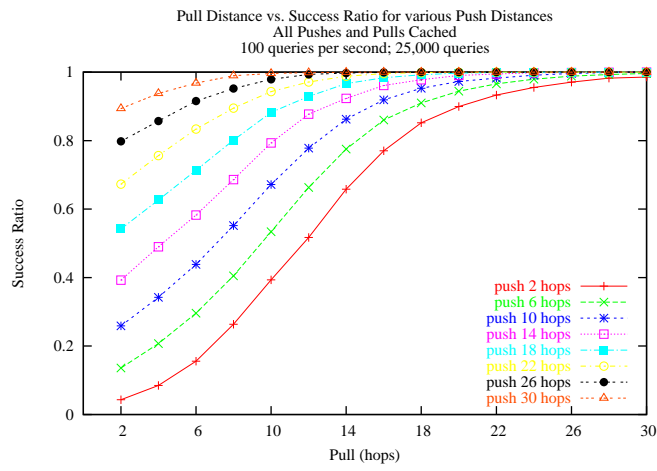


Figure 8: Grid Network, Success Ratio, Pulls and Pushes Cached

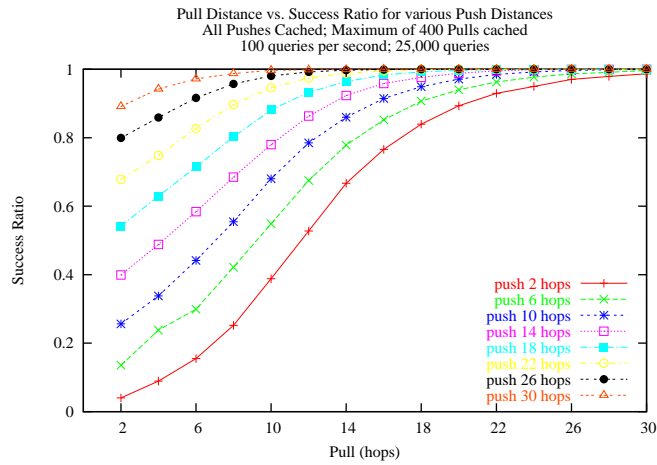


Figure 9: Grid Network, Success Ratio, All Pushes Cached; Maximum of 400 Pulls Cached

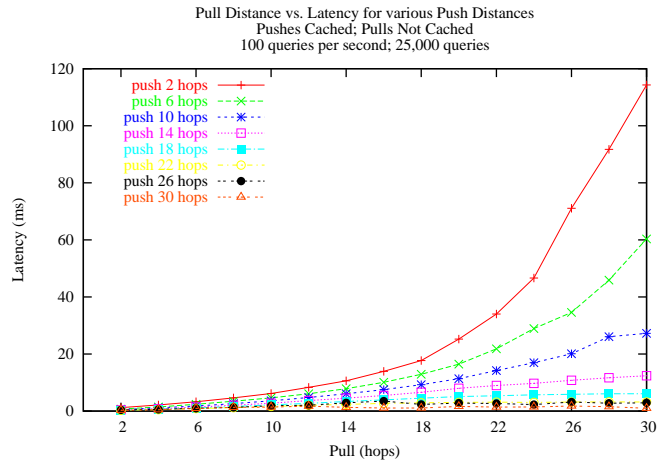


Figure 10: Grid Network, Latency, Only Pushes Cached

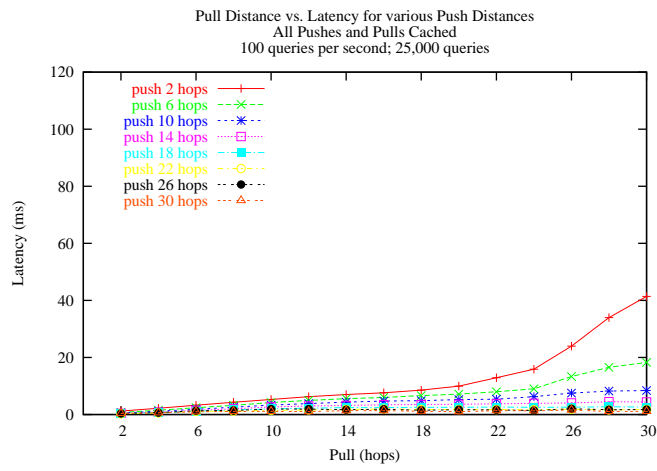


Figure 11: Grid Network, Latency, Pulls and Pushes Cached

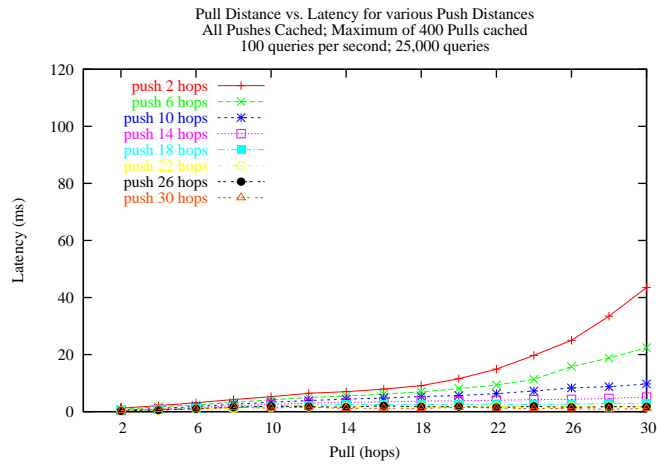


Figure 12: Grid Network, Latency, All Pushes Cached; Maximum of 400 Pulls Cached

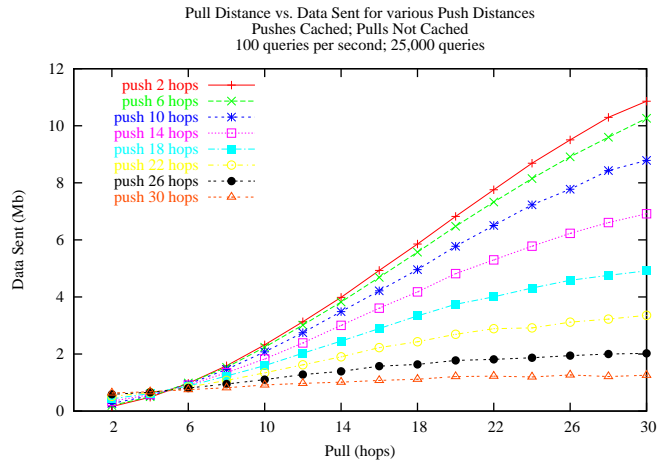


Figure 13: Grid Network, Data Sent, Only Pushes Cached

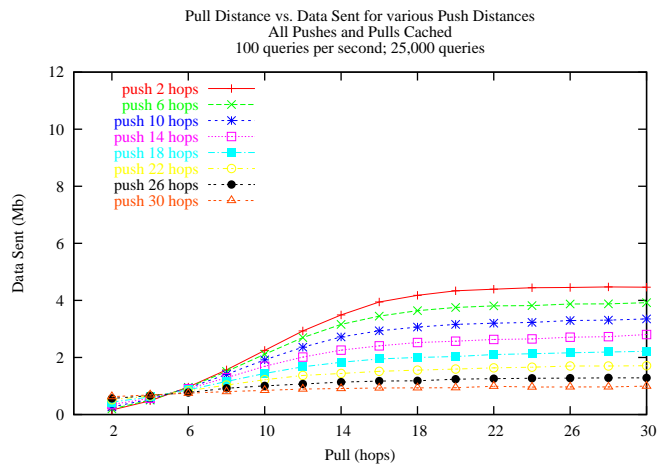


Figure 14: Grid Network, Data Sent, Pulls and Pushes Cached

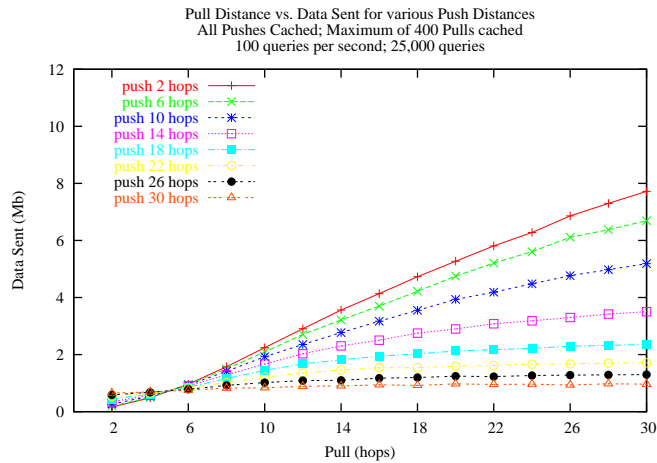


Figure 15: Grid Network, Data Sent, All Pushes Cached; Maximum of 400 Pulls Cached

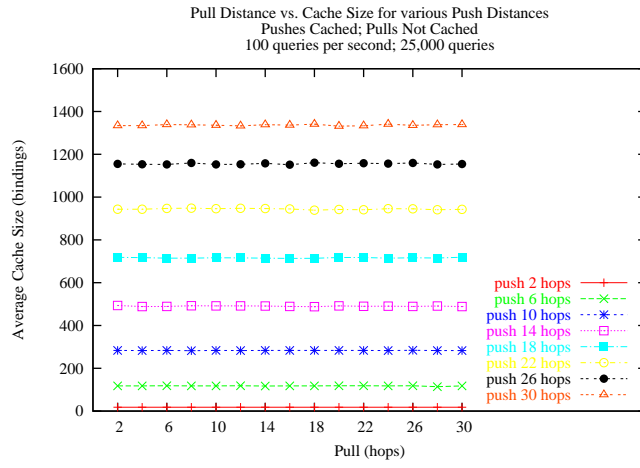


Figure 16: Grid Network, Cache Size, Only Pushes Cached

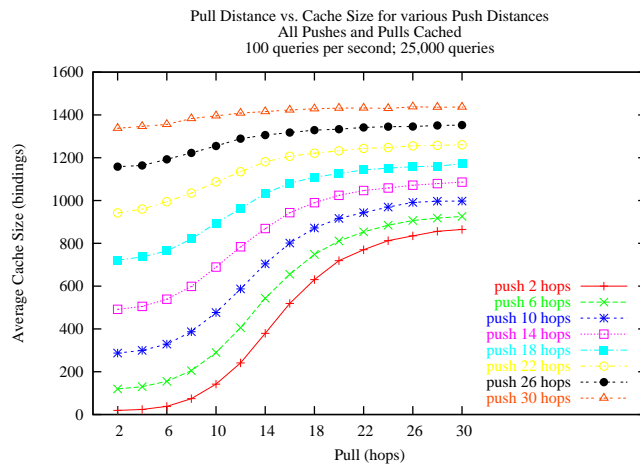


Figure 17: Grid Network, Cache Size, Pulls and Pushes Cached

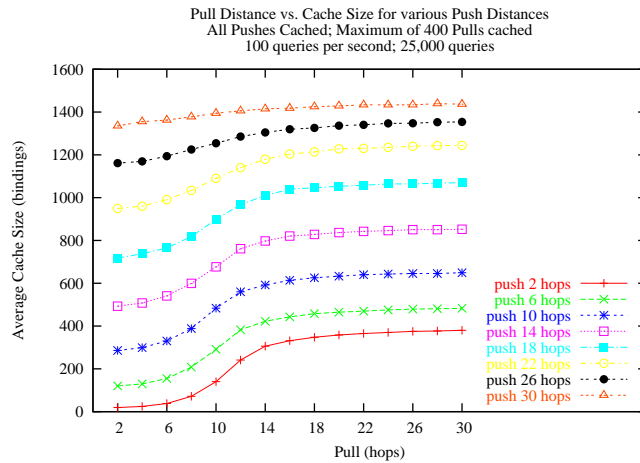


Figure 18: Grid Network, Cache Size, All Pushes Cached; Maximum of 400 Pulls Cached

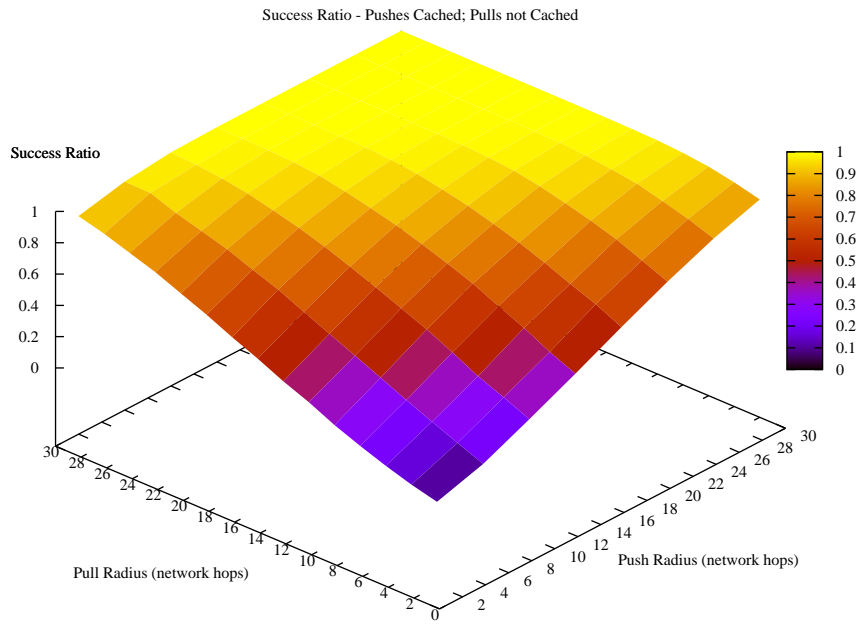


Figure 19: Grid Network, Success Ratio, Only Pushes Cached

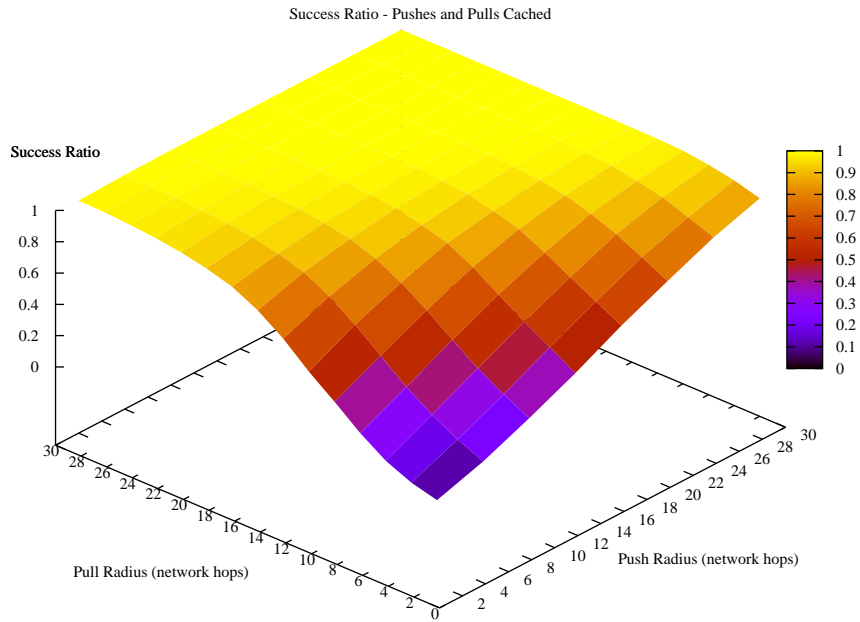


Figure 20: Grid Network, Success Ratio, Pulls and Pushes Cached

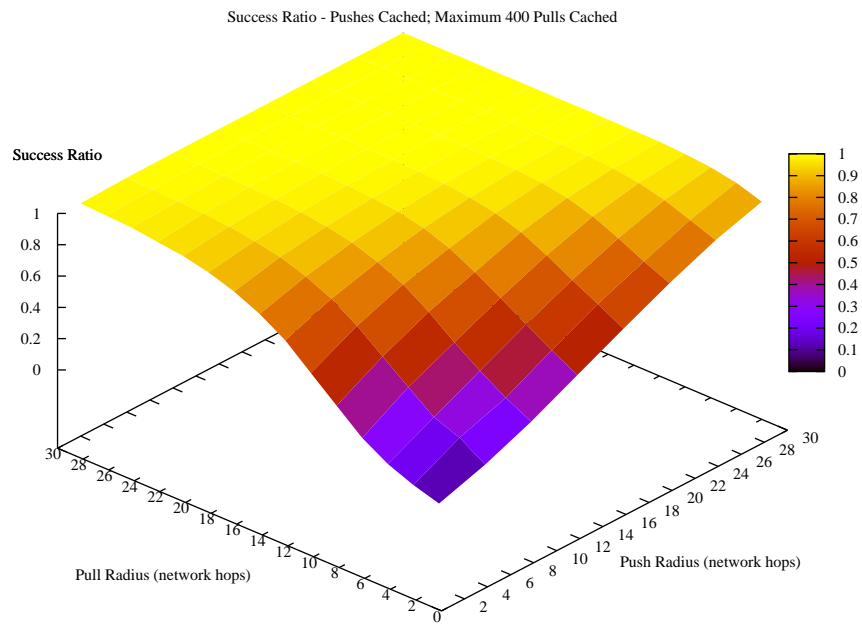


Figure 21: Grid Network, Success Ratio, All Pushes Cached; Maximum of 400 Pulls Cached

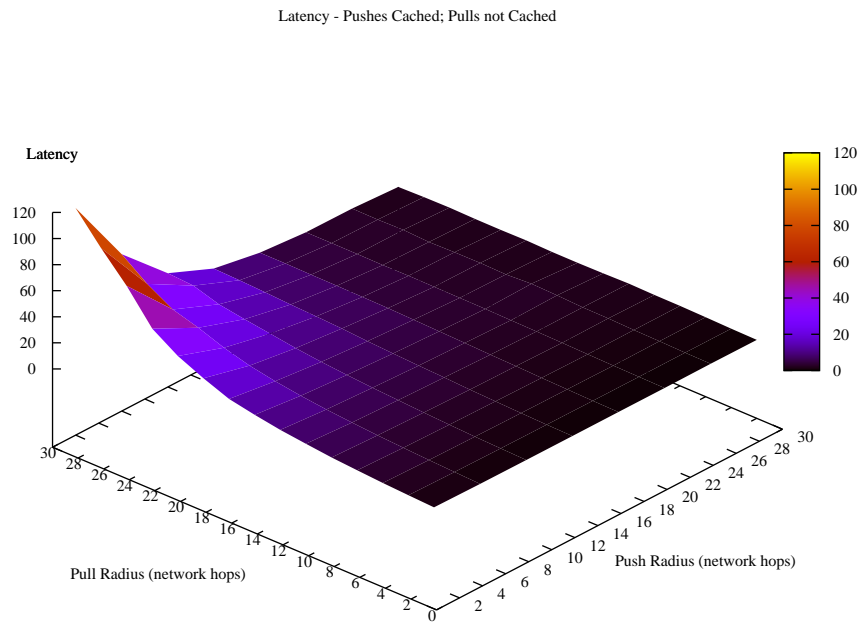


Figure 22: Grid Network, Latency, Only Pushes Cached

Latency - Pushes and Pulls Cached

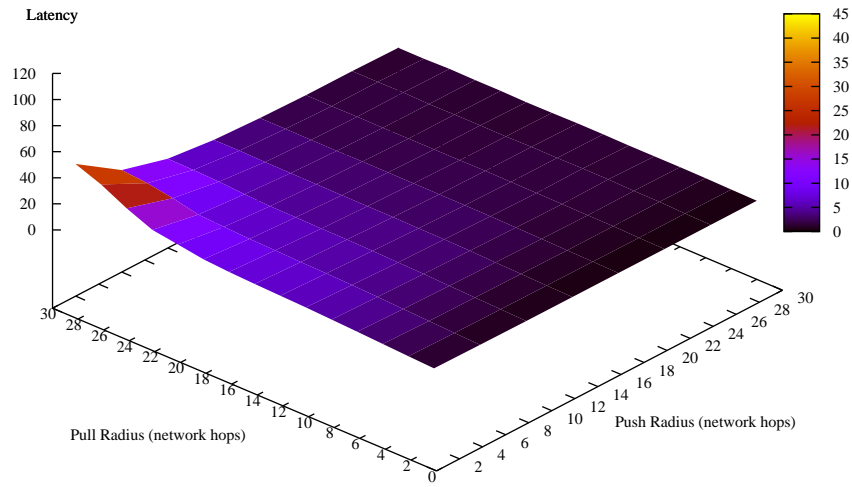


Figure 23: Grid Network, Latency, Pulls and Pushes Cached

Latency - Pushes Cached; Maximum 400 Pulls Cached

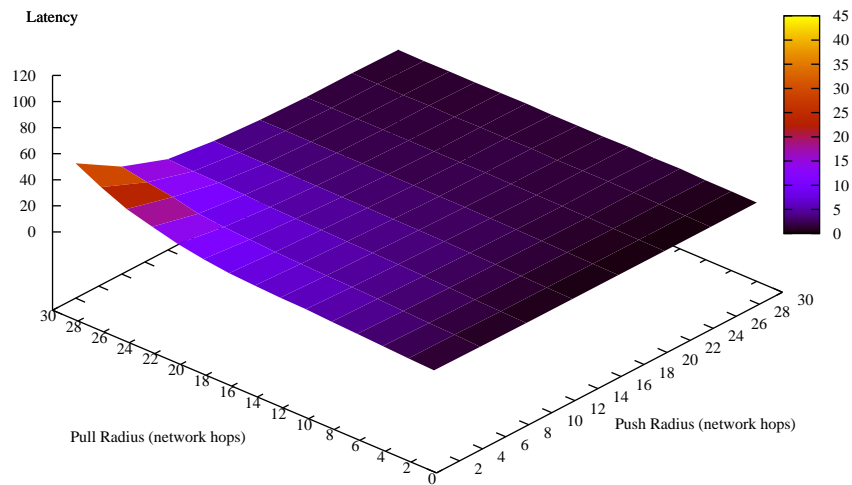


Figure 24: Grid Network, Latency, All Pushes Cached; Maximum of 400 Pulls Cached

Data Sent - Pushes Cached; Pulls not Cached

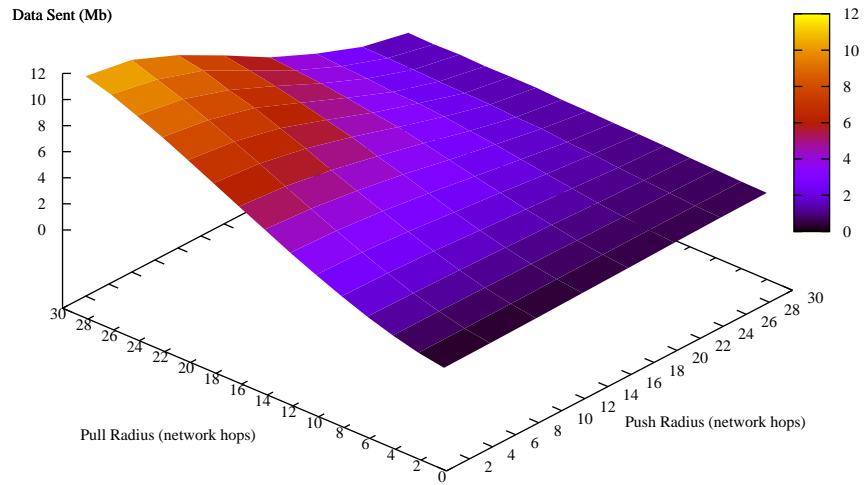


Figure 25: Grid Network, Data Sent, Only Pushes Cached

Data Sent - Pushes and Pulls Cached

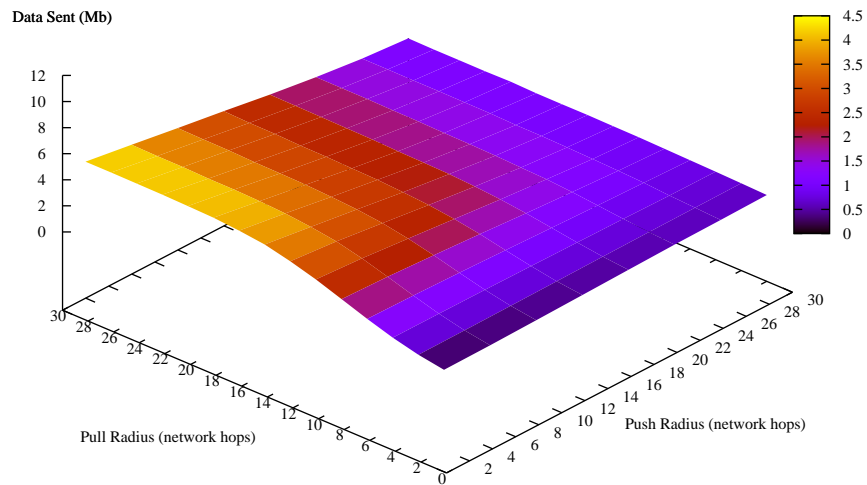


Figure 26: Grid Network, Data Sent, Pulls and Pushes Cached

Data Sent - Pushes Cached; Maximum 400 Pulls Cached

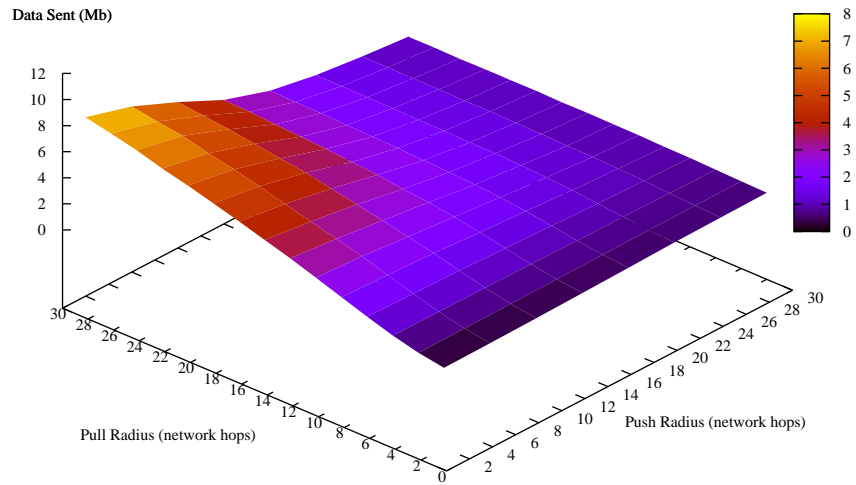


Figure 27: Grid Network, Data Sent, All Pushes Cached; Maximum of 400 Pulls Cached

Average Cache Size - Pushes Cached; Pulls not Cached

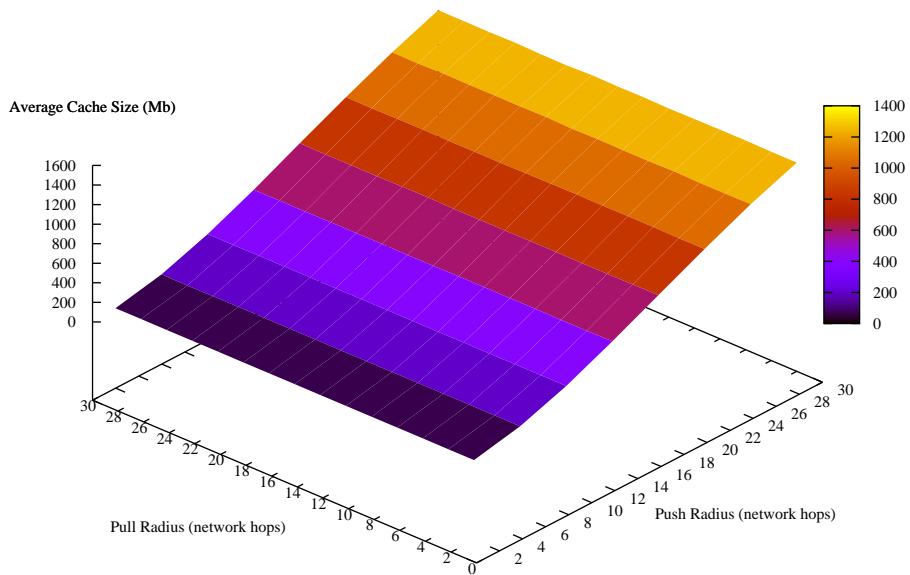


Figure 28: Grid Network, Cache Size, Only Pushes Cached

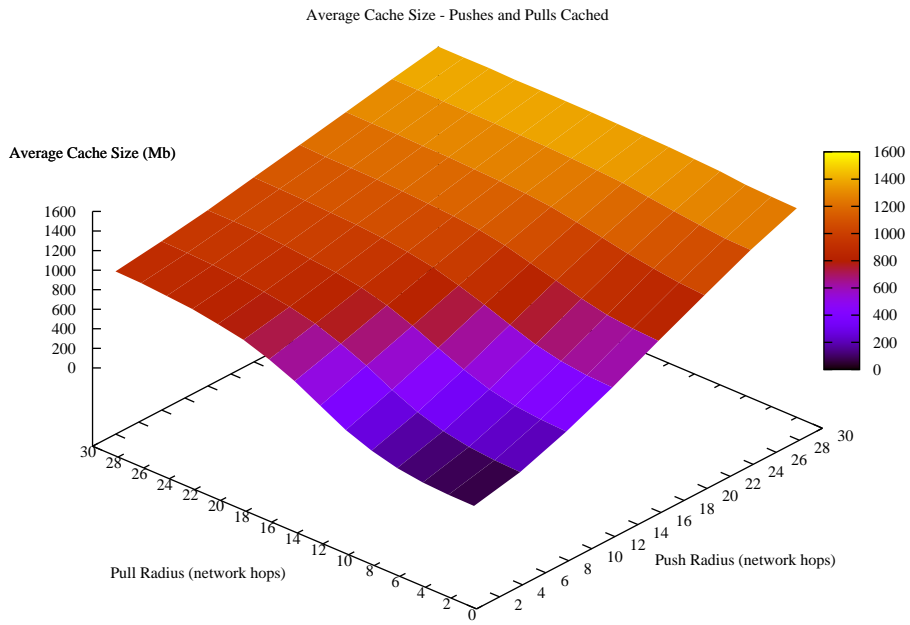


Figure 29: Grid Network, Cache Size, Pulls and Pushes Cached

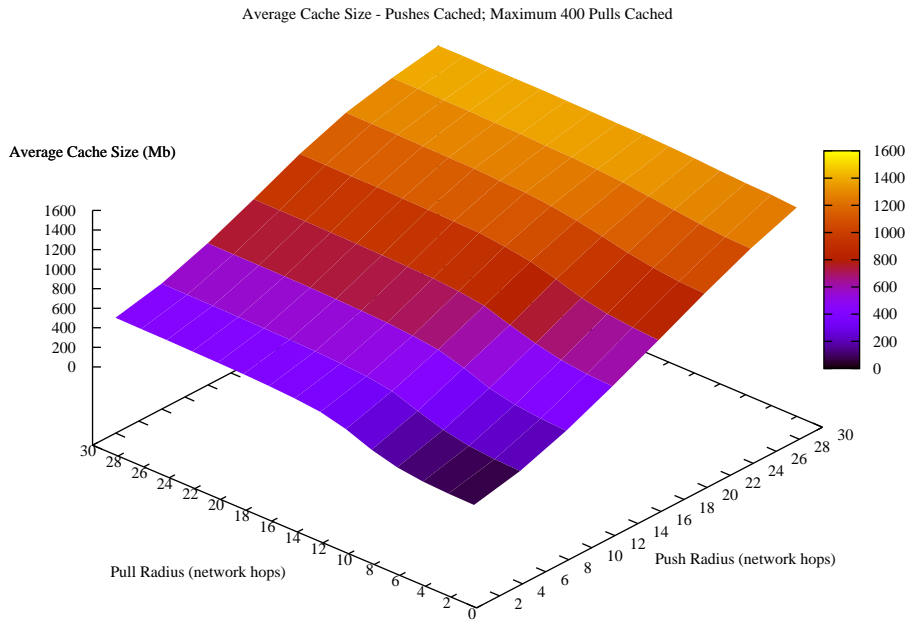


Figure 30: Grid Network, Cache Size, All Pushes Cached; Maximum of 400 Pulls Cached

### 6.2.2 Grid Network: Dynamic

The goal of dynamic grid network experiments is to measure the impact of mobility (changes of logical address) on the naming service. In dynamic grid network experiments, an experimental parameter controls the rate at which sockets change their logical addresses. When a socket changes its logical address it increments either its  $x$ -coordinate value or its  $y$ -coordinate value by one unit. The coordinate that is incremented is chosen at random for each change of logical address. By incrementing a single coordinate by a small amount, the topology of the network is preserved. In addition to mobility, the experimental parameters of push radius and pull radius are also varied. The push and pull radii are varied so that their sum remains constant. Four performance metrics are measured in dynamic grid network experiments: success ratio of queries, latency of successful queries, amount of data sent, and size of peer naming bindings caches.

During dynamic grid network experiments each socket periodically pushes the name binding that it has in its local name binding table (Section 2.4). Periodic pushes are done so that the most recent name binding is regularly distributed to network peers. Logical address query operations occur at a configurable rate during the entire duration of each experiment. Pull name bindings messages are sent in response to logical address query messages. Pull name bindings messages are sent from the socket that responds to a logical address query to the socket that initiated the logical address query. All intermediate sockets that forward pull name bindings message between responder and querier store the name bindings contained in the pull name bindings messages (see the description of binding snooping in section 2.5).

The results of dynamic grid network experiments are shown in a series of four graphs. Each graph shows the measurements of a different performance metric. In each graph, the mobility parameter is varied from 10 logical address changes per second to 100 logical address changes per second, in increments of 10 logical address changes per second. Push and pull radii are varied between 10 hops and 30 hops, in increments steps of 5 hops, such that the sum of the push radius and the pull radius is 40 hops (network diameter). Logical address query operations occur at a fixed rate of 50 queries per second (qps). 25,000 queries are produced in each experiment.

Figure 31 shows success ratio measurements. Figure 32 shows latency measurements. Figure 33 shows the amount of data sent measurements. Figure 34 shows the size of peer bindings caches measurements.

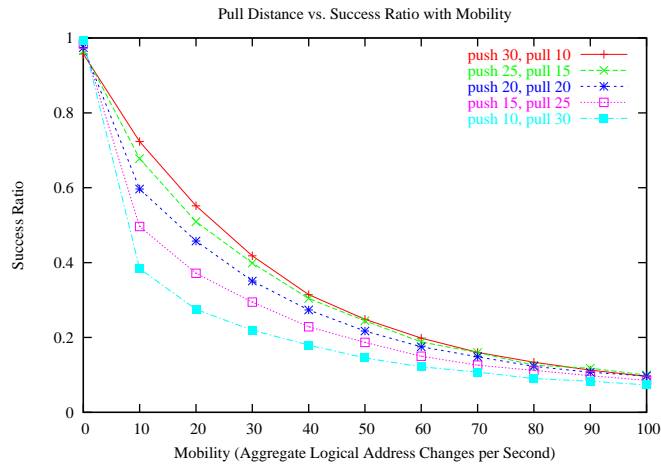


Figure 31: Grid Network, Mobility, Success Ratio

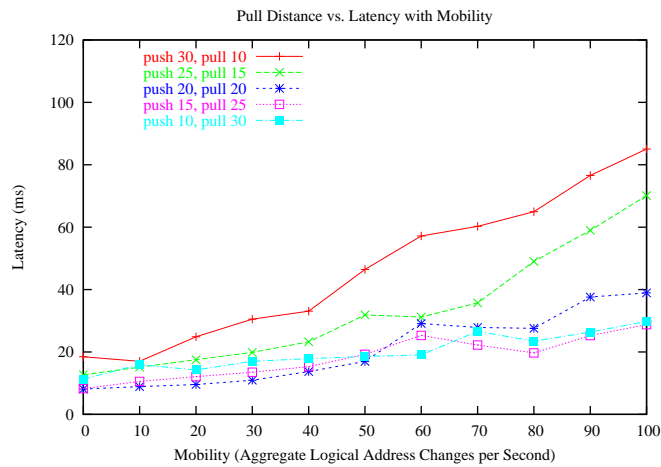


Figure 32: Grid Network, Mobility, Latency

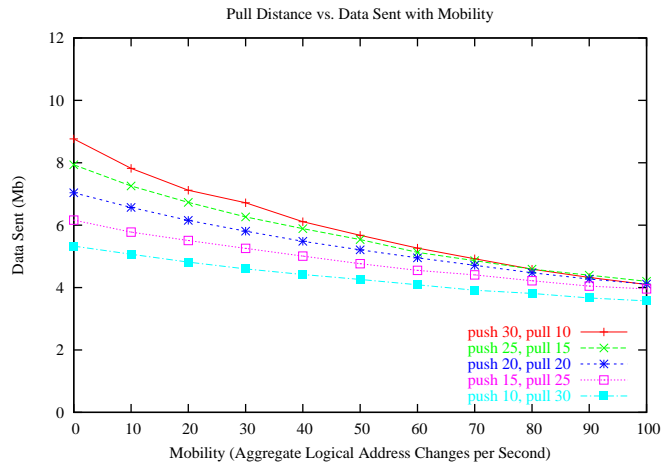


Figure 33: Grid Network, Mobility, Data Sent

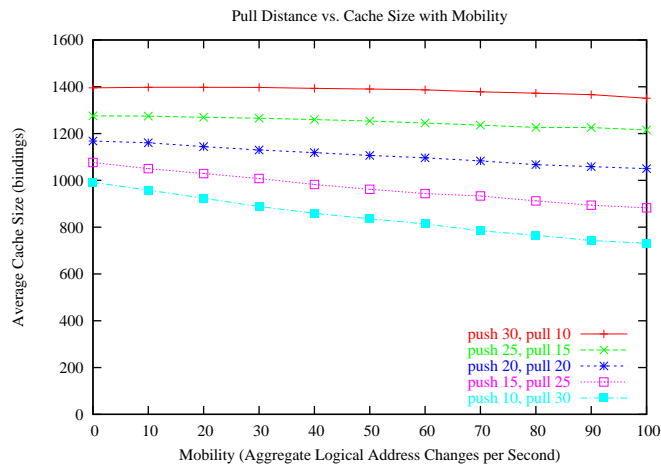


Figure 34: Grid Network, Mobility, Cache Size

## 7 Conclusions

The this project has explored of the design and development of a naming service for dynamically changing application-layer overlay networks. The practicality of such a system was evaluated using various performance metrics and experimental configurations. The study was done without the assumption of any fixed network infrastructure (central servers). The trade-off between disseminating name bindings at creation time (push) verses disseminating name bindings on demand (pull) were considered. The impact of caching and mobility were also explored. A solution was presented for establishing the trust of name bindings in a dynamic environment in which there is no trusted third party.

The scope of the evaluation was limited by available experimental resources. These limitation leave open the question of large scale performance. The abstraction of groups is open to further elaboration, e.g. sub-grouping constructs.

## References

- [1] Hypercast design documents and materials. <http://www.cs.virginia.edu/hypercast/materials.html>, 2005.
- [2] J. Liebeherr and M. Nahas, "Application-layer Multicast with Delaunay Triangulations," March 2001, Global Internet Symposium, IEEE Globecom 2001, November 2001.
- [3] Hypercast spanning tree design documents?
- [4] I. Stocia et. al., "Chord, A Scalable Peer-to-Peer Lookup Service for Internet Applications," Proc. ACM SIGCOMM, ACM Press, 2001, pp. 149-160.
- [5] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001.
- [6] S. Ratnasamy et. al., "A Scalable Content-Addressable Network," Proc. ACM SIGCOMM, ACM Press, 2001, pp 161-172.
- [7] A. Bahora, T. Collins, S. Davis, S. Goknur, J. Kearns, T. Lieu, T. Nguyen, J. Zeng, S. Patek, B. M. Horowitz, "Integrated Peer-To-Peer Applications for Advanced Emergency Response Systems. Part I and II.," In Proceedings of the 2003 IEEE Systems and Information Engineering Design Symposium (SIEDS'03), Pages 255-268, April 2003.

## A Naming Service API

This is a catalog of all application programmer interfaces (APIs) of the HyperCast naming service. These APIs are implemented in Java as members of the HyperCast `I_OverlaySocket` interface. The behavior of these APIs is influenced by configuration parameters described in Appendix B.

A programmer must configure a HyperCast overlay socket to use the naming service to use naming service API. An `UnsupportedOperationException` is thrown if a naming service API is invoked on a socket that does not have the naming service configured.

### A.1 setName

```
void setName (String name)
void setName (String name, I_LogicalAddress LA)
void setName (Certificate certificate, PrivateKey privateKey)
void setName (Certificate certificate, PrivateKey privateKey, String name, I_LogicalAddress LA)
```

This method is used to create a name binding at the local overlay socket. It may be called arbitrarily many times to create arbitrarily many name bindings. The `setName` API may be called with or without a logical address parameter. If no logical address is specified, the logical address of the local overlay socket is used in the new binding. If the logical address of the local socket changes, the binding's logical address field changes as well. If a logical address is specified, the logical address field of the new binding will use this value and this value will not change when the logical address of the local socket changes.

Certificates are assumed to be X.509 v3 certificates with RSA keys. Private keys are assumed to be RSA keys that match the specified certificate. The subject common name or "CN" field of the certificate's X.500 name is used as the name that will be bound to a logical address if no other name is specified.

Calling `setName` with the same parameter multiple times (without an intervening call to `unsetName` with the same parameter) has no effect. Note that calling `setName` with the same name, once as a `String` and once as a certificate (again without an intervening call to `unsetName`), is an error. That is, it is an error to call `setName(someCertificate)` followed by `setName( "Foo" )` where "Foo" is the subject common name of the certificate. Conversely, it is an error to call `setName( "Foo" )` and then `setName(someCertificate)` with "Foo" as the subject name of `someCertificate`.

Calling `setName` may cause a push name bindings operation (Section 5.3) to occur depending on the setting of the configuration attribute `PushOnSetName` (Appendix B).

### A.2 unsetName

```
void unsetName (String name)
void unsetName (Certificate certificate, PrivateKey privateKey)
```

This method is used to destroy name bindings at the local socket. After this method is invoked, an overlay socket will cease to advertise a name binding with this name during push name binding operations (Section 5.3). It will also cease to reply to queries for this name.

Calling `unsetName` may cause an invalidate name bindings operation (Section 5.5) to occur depending on the configuration attribute `InvalidateOnSetName` (Appendix B).

### A.3 `getNames`

```
String [] getNames()
String [] getNames (LogicalAddress peer)
String [] getNames (LogicalAddress peer,
                    boolean    requireTrusted,
                    long        timeout,
                    boolean    requireAuthoritative,
                    int         maxAge,
                    int         maxResponses,
                    int         minHopCount)
```

The first version of this API, With no parameters, this method returns an array of all names currently use by name bindings in the local name binding table (Section 2.4). The list may be empty in which case a zero length array is returned - not the null reference.

The second and third versions of this API, with parameters, implement a distributed reverse lookup function for the naming service. Given the logical address of a peer network node, this method returns the names bound to it. If no response is received from this query, the null reference is returned. If the peer node exists but has no names bound to it, it will respond and a zero length array will be returned.

As is the case with `getLogicalAddresses(String,int,long)`, the constants `Naming.NO_RESPONSE_LIMIT` and `Naming.WAIT_INDEFINITELY` can be used for `maxResponses` and `timeout`, respectively. If both of these constants are used, `getNames` will raise an error since otherwise the method would never return.

### A.4 `getNamesNonBlocking`

```
void getNamesNonBlocking (LogicalAddress peer)
void getNamesNonBlocking (LogicalAddress peer,
                           boolean    requireTrusted,
                           long        timeout,
                           boolean    requireAuthoritative,
                           int         maxAge,
                           int         maxResponses,
                           int         minHopCount)
```

This method implements the same reverse lookup function as `getNames`, however, it is a synchronous call that relies on the HyperCast notification system to provide reply information to applications. The implementation of this method blocks waiting for the receipt of a naming event from the naming service FSM. When a naming event arrives (or a timeout occurs) this API returns to its caller.

See the example code in the description of `getLogicalAddressNonBlocking`.

### A.5 `getLogicalAddressByName`

```
LogicalAddress [] getLogicalAddressByName (String name)
LogicalAddress [] getLogicalAddressByName (String name,
                                           boolean requireTrusted,
                                           long    timeout,
                                           boolean requireSerialNumberMatch,
                                           boolean requireAuthoritative,
                                           int     maxAge,
                                           int     maxResponses,
                                           int     minHopCount)
```

This method implements a lookup function that maps a name to a set of logical addresses. This method returns the null reference if no response is received.

The constants `Naming.NO_RESPONSE_LIMIT` and `Naming.WAIT_INDEFINITELY` can be used for `maxResponses` and `timeout`, respectively. If both of these constants are used, `getLogicalAddresses` will raise an error since otherwise the method would never return.

## A.6 `getLogicalAddressByNameNonBlocking`

```
void getLogicalAddressByNameNonBlocking (String name)
void getLogicalAddressByNameNonBlocking (String name,
                                         boolean requireTrusted,
                                         long timeout,
                                         boolean requireSerialNumberMatch,
                                         boolean requireAuthoritative,
                                         int maxAge,
                                         int maxResponses,
                                         int minHopCount)
```

This method implements the same name lookup as the `getLogicalAddress` method however, it is an asynchronous call that relies on the HyperCast notification system to provide reply information to an application.

The following example code illustrates a handler that processes responses to an asynchronous name lookup.

```
class MyNamingEventHandler extends NotificationHandler {
    public void handle_NAMING_EVENT (NAMING_EVENT event) {
        NamingBindings [] bindings = event.getBindings();
        for (int i = 0; bindings.length > 0; ++i)
            System.out.println (bindings[i]);
    }
}
```

## A.7 `installTrustedNamingCertificate`

```
void installTrustedNamingCertificate (Certificate certificate)
```

This method adds a certificate to the certificate cache (Section 2.8) and marks it as trusted, making it a trust anchor. Trust chains terminate at trust anchors.

## B Configuration

This appendix summarizes the configuration attributes and statistics available with the naming service.

### B.1 Attributes

`AuthoritativeResponsesOnly` (default `No`) This attribute is used to specify whether or not acceptable bindings must be received from the peer that created them.

`BindingTimeout` (default `60000`) This value, specified in milliseconds, defines the amount of time that a name binding is considered to be usable after it is received. After this time has expired, the name binding will be removed from the peer name binding cache. Note that if the peer name binding cache is not being used, this setting is not used.

`BlockingAPITimeout` (default `10000`) The number of milliseconds that a blocking naming API will wait until it has a timeout.

`BypassPeerBindingsCacheOnQuery` (default `No`) Determines if queries inspect the peer name binding cache of the local socket before being forwarded to peers.

`CachePullBindings` (default `No`) There are two ways that a node can receive bindings from peers: push name bindings messages and pull name bindings messages. When the peer name binding cache is used, all name bindings received via push name bindings messages are eligible for caching. This setting determines whether or not bindings received in pull name bindings messages are subject for caching.

`CertificateCaching` (default `Yes`) If set to `No`, a certificate exchange operation will be forced for every step in building a trust chain.

`CertificateRequestTimeout` (default `5000`) Measured in milliseconds, this setting defines how long the naming service will wait for a response to a certificate request message.

`DigitalSignatureAlgorithm` (default `None`) This attribute defines the algorithm that will be used to create and verify digital signatures of bindings. `SHA1withRSA` is supported.

`DoNamingQueries` (experiments only; default `Yes`) Determines whether or not naming traffic is generated if `No` messages with application data will be sent.

`InvalidateOnLeaveOverlay` (default `No`) Determines if an invalidate name bindings operation will occur when a socket leaves an overlay.

`InvalidateOnUnsetName` (default `No`) Determines if an invalidate name bindings operation will occur when the unset name API is invoked.

`InvalidateRadius` (default `10`) The maximum distance that an invalidate name bindings message will travel. This distance is measured in network in hops. Invalidate messages are sent via broadcast.

`MaximumResponseBindingAge` (default `60000`) Each binding contains a timestamp that indicates when it was created. This setting, in milliseconds, specifies the maximum allowable age of an acceptable binding.

`MaximumResponsesPerQuery` (default 1) This attribute sets the maximum number of responses that will be returned to the application for a query.

`MinimumQueryRadius` (default 0) The minimum number of hops that a query message should traverse regardless of responses found on the way. Normally, (`MinimumQueryRadius = 0`) a query message will not forward further when it encounters a peer that can respond with at least one binding.

`PeerBindingCachePullMaximumSize` (default 0) This attribute sets the maximum number of bindings that can be placed into the peer name binding caches with pull messages.

`PeerBindingCachePushMaximumSize` (default 0) This attribute sets the maximum number of bindings that can be placed into the peer name binding caches with push messages.

`PushOnChangeOfLogicalAddress` (default No) Determines if a push name bindings operation will occur when a socket changes its logical address.

`PushOnJoinOverlay` (default No) Determines if a push name bindings operation will occur when a socket joins an overlay.

`PushOnSetName` (default No) Determines if a push name bindings operation will occur when the set name API is invoked.

`PushPeriod` (default 60000) The number of milliseconds that will elapse between consecutive push name bindings operations.

`PushRadius` (default 4) The maximum distance that a push message will travel. This distance is measured in network in hops. All push messages are send via broadcast.

`QueryCount` (experiments only; default 10000) Determines the number of queries for an experiment.

`QueryFrequency` (experiments only; default 0) Used to configure experimental rate controller.

`QueryRadius` (default 10) The maximum distance that a query message travels. This distance is measured in network in hops. Logical address query messages are sent via unicast; name query messages are sent via broadcast.

`QueryTracing` (default No) If set to Yes, query messages will be traced to the console.

`QueryTracingModulus` (default 1) Determines how often a query trace will occur with respect to query serial numbers. For example if this value is set to 100, then every 100 query messages will be traced to console.

`RequireBindingMessageSerialNumberMatch` (default No) Each query message has a serial number as does each response (pull) message. This attribute determines if the serial number of a response must match the serial number of a query.

`RequireTrustMessageSerialNumberMatch` (default No) Each certificate request message has a serial number as does each certificate response message. This attribute determines if the serial number of a response must match the serial number of a certficate request.

`SnoopBindings` (default `Yes`) This attribute specifies that the name bindings of all query responses (pull name bindings messages) should be read and processed by intermediate nodes as well as the destination node.

`SnoopCertificates` (default `Yes`) This attribute specifies that the certificates of all certificate responses should be read and processed by intermediate nodes as well as the destination node.

`SocketsInNetwork` (experiments only; default 0) Used to configure experimental rate controller.

`TrustedResponsesOnly` (default `No`) Determines if an acceptable binding must have a trusted signature.

`WriteCertificatesToBackingStore` (default `Yes`) If set to `Yes`, certificates will be saved to backing store when they are received. If set to `No`, certificates will be saved in memory only.

## **B.2 Statistics**

`BindingCacheEvictions` This statistic counts the number of bindings removed from the peer name binding cache. A name binding can be removed because the cache has reached its maximum size or because of a timeout.

`BindingCacheHits` This statistic records the number of times that a query message finds a name binding in the peer name binding cache.

`BindingCachePullSize` This statistic reports the current number of bindings in the peer name binding cache as a result of pull name bindings messages.

`BindingCachePushSize` This statistic reports the current number of bindings in the peer name binding cache as a result of push name bindings messages.

`BindingCacheSize` This statistic reports the current number of bindings in the peer name binding cache as a result of pull and push name bindings messages.

`QueryMessagesProcessed` This statistic counts the number of query messages processed.

## C Naming Shell Application

The HyperCast distribution contains a command line utility that uses the naming service API (Appendix A) and has functionality that is analogous to the DNS utilities `nslookup`, `dig`, and `host`. This utility is called the *naming shell*. In addition to offering users a simple way in which to interact with the naming service, the naming shell also acts as an example program that demonstrates how the naming service API is called from application programs.

What follows is a simple interaction between two naming shell instances. The interaction demonstrates how a name is set with a certificate/private key pair and without a certificate and private key. The Delaunay triangulation overlay topology is used.

The first naming shell is started and commands are issued to its socket to set two names, “foo” and “0000”. The name “0000” is set using a certificate and a private key so name bindings that use this name will have digital signatures created. The naming service will be able to verify the integrity and authenticity of these name bindings. The logical address of the socket is changed using the `changela` command to a simple address (100, 100). The first naming shell is also used to illustrate the use of the `help` command which lists all of the commands available in the naming shell:

```
> java -classpath hypercast.jar:bcprov-jdk14-122.jar:xalan.jar NamingShell
Waiting for Naming Shell socket to become stable in the overlay...stability notification received!
Welcome to the Naming Shell
6605,6354> help
Available commands are:
  set <name> ; set name of local socket
  crypto_set <certificate filename> <private key filename>; set name of local socket using crypto
  unset <name> ; unset name of local socket
  crypto_unset <certificate filename> <private key filename> ; unset name of local socket using crypto
  name [-timeout<secs>] [-block] [-trusted] [-auth] [-age<secs>] [-minhops<count>] [-maxhops<count>] [-responses<count>] \
    [-matchsn] <logical address>; name query
  la [-timeout<secs>] [-block] [-trusted] [-auth] [-age<secs>] [-minhops<count>] [-maxhops<count>] [-responses<count>] \
    [-matchsn] <name>; logical address query
  changela <new logical address> ; changes logical address of the local socket
  trusted_cert <certificate filename>; install a trusted certificate
  certs ; displays table of certificates
  bindings ; displays table of bindings
  pending ; displays table of pendings queries and pending certificates
  help
  quit
  exit
6605,6354> changela 100,100
100,100> set foo
100,100> crypto_set cert-0000.der privkey-0000.der
100,100> bindings
BindingTable
  = Local Name Binding Table:

foo -> <name -> foo, LA -> 100,100, local -> true, authoritative -> true, timestamp -> 0 sec old, sig -> false, \
  trusted -> true, verified -> true, LA changes -> 1>
0000 -> <name -> 0000, LA -> 100,100, local -> true, authoritative -> true, timestamp -> 0 sec old, sig -> false, \
  trusted -> true, verified -> true, LA changes -> 1>

Push:
Peer Name Binding Cache:

Pull:
Peer Name Binding Cache:

100,100> certs
CertificateCache
  = Certificate Cache:

Subject (0000) Issuer (HyperCast CA) PrivateKey (Yes) Trusted (No) Verified (No)
100,100>
```

The second naming shell has its logical address changed to (200,200) and then begins to make logical address queries. Both the name that was set without a certificate and a private key as well as the name that was set with a certificate and private key are queried. Trust cannot be established for the name that is not signed with a private key. Trust cannot be established for the name that is signed until a trust anchor is installed. Notice that successful queries have a query serial number displayed.

```

> java -classpath hypercast.jar:bcprov-jdk14-122.jar:xalan.jar NamingShell
Waiting for Naming Shell socket to become stable in the overlay...stability notification received!
Welcome to the Naming Shell
9700,3628> changela 200,200
200,200> bindings
  BindingTable
    = Local Name Binding Table:

Push:
Peer Name Binding Cache:

Pull:
Peer Name Binding Cache:

200,200> certs
  CertificateCache
    = Certificate Cache:

200,200> la foo
Issuing logical address query for name "foo"
NAMING_EVENT handler:
Query S/N: 1
<name -> foo, logical address -> 100,100, LA changes -> 1, timestamp -> 0 sec old, trusted -> false, \
  authoritative -> true>

200,200> la -trusted foo
Issuing logical address query for name "foo"
200,200> la 0000
Issuing logical address query for name "0000"
NAMING_EVENT handler:
Query S/N: 3
<name -> 0000, logical address -> 100,100, LA changes -> 1, timestamp -> 0 sec old, trusted -> false, \
  authoritative -> true>

200,200> la -trusted 0000
Issuing logical address query for name "0000"
200,200> certs
  CertificateCache
    = Certificate Cache:

Subject (0000) Issuer (HyperCast CA) PrivateKey (No) Trusted (No) Verified (No)

200,200> trusted_cert cert-CA-HC.der
200,200> la -trusted 0000
Issuing logical address query for name "0000"
NAMING_EVENT handler:
Query S/N: 5
<name -> 0000, logical address -> 100,100, LA changes -> 1, timestamp -> 34 sec old, trusted -> true, \
  authoritative -> true>

200,200> certs
  CertificateCache
    = Certificate Cache:

Subject (HyperCast CA) Issuer (HyperCast CA) PrivateKey (No) Trusted (Yes) Verified (Yes)
Subject (0000) Issuer (HyperCast CA) PrivateKey (No) Trusted (Yes) Verified (Yes)

200,200> bindings
  BindingTable
    = Local Name Binding Table:

Push:
Peer Name Binding Cache:

Pull:
Peer Name Binding Cache:

<name -> foo, LA -> 100,100, local -> false, authoritative -> true, timestamp -> 63 sec old, sig -> false, \
  trusted -> false, verified -> false, LA changes -> 1>
<name -> 0000, LA -> 100,100, local -> false, authoritative -> true, timestamp -> 39 sec old, sig -> true, \
  trusted -> true, verified -> true, LA changes -> 1>

200,200>

```