Design and Implementation of Intentional Names

by

Elliot Schwartz

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

Author ....
Department of Electrical Engineering and Computer Science
May 20, 1999

Certified by ....
Hari Balakrishnan
Assistant Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Design and Implementation of Intentional Names

by

Elliot Schwartz

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 1999

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Most network naming schemes force applications to specify the network location of resources they wish to use. However, applications typically want a particular service and do not know where in the network the service is located. We argue that applications should be able to simply describe their needs in an *intentional name*, and that the network infrastructure should be able to resolve this name to its network locations. To this end, we have designed and implemented the Intentional Name System (INS). In this thesis we present the design of a key component of INS: the naming scheme. We describe the expressive syntax of its intentional names and the data structures and algorithms it uses to perform the name-to-location resolution. We analyze these algorithms, describe a prototype implementation, and present performance results that show the feasibility of our ideas. Finally, we describe three applications that demonstrate the utility of intentional names.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor

Acknowledgments

This thesis has my name on it, but there were many people who contributed to its completion. I would especially like to acknowledge the following people:

My thesis supervisor, Professor Hari Balakrishnan, was everything a student could hope for in an advisor. I am indebted to him for the guidance, advice, motivation, and kindness he provided during the year I worked with him.

My research at MIT was supported by a grant from the Nippon Telegraph and Telephone Corporation (NTT). I am grateful to Dr. Ichizo Kogiku for his interest in this project and for hosting me during my visit to the NTT Laboratories in Musashino.

My cohorts in developing the Intentional Naming System were instrumental in my thesis work. Thank you to William Adjie-Winoto, my original partner in designing INS, and to Jeremy Lilley and Anit Chakraborty, who helped the system progress from there.

My officemates at LCS, Suchitra Raman and Hariharan Rahul, were an endless source of answers, suggestions, and entertaining conversation. I'm glad that I had the opportunity to share my days (and nights) with them.

My friend Joanna Kulik motivated me to get my thesis done as quickly as possible, by making me realize there are much more fun things to do. Thank you for introducing me to the real world.

My housemates, Mike Whitson, Nathan Williams, Chris Shabsin ("Shabby"), Megan McCallion, and Robert Ringrose, helped make our house a home that I looked forward to coming back to each night. Thank you for understanding when I was too busy to put the dishes away.

My eternal friend, Wendy Ann Deslauriers, gave me refuge in Tokyo to write my thesis proposal, and was a pleasure to "talk" to during all those evenings when I was still working on my thesis and she was just getting to work. Thank you, again.

My family bore the burden of keeping in contact with me while I was buried in my thesis work, making sure I was alright when they hadn't heard from me in weeks. Their support was and always will be extremely important to me. I love you, Margo, Adinne and Nick.

My thesis relied heavily on a number of free software packages, and their authors have my hearty appreciation. The text of the thesis and the software implementation were written using *GNU Emacs* from the Free Software Foundation. The software was written in Java and run using Sun Microsystems' Java Development Kit (JDK), which was ported to Linux by the Blackdown Java-Linux Porting Team. The software uses many publically available standards of the Internet Engineering Task Force (IETF). The text of the thesis was written in and typeset using Leslie Lamport's LaTeX. All of this work was done on a machine running the Red Hat distribution of Linux, an operating system composed of Linus Torvalds' kernel and many GNU utilities. Figures in the thesis were prepared using *tgif*, which was written by William Chia-Wei Cheng. Graphs in the the thesis were prepared using *Grace*, whose volunteer development team is coordinated by Evgeny Stambulchik.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation for Intentional Naming

Computer systems use naming as a layer of abstraction. For example, in computer architecture, virtual addresses are names that can be resolved to physical addresses. This abstraction allows code to use static addresses, and still be located anywhere in physical memory. It also permits the safe sharing of memory, through a secure resolution process.

In computer networks naming abstractions have also been used to add functionality. Domain Name System (DNS) [25] names are mapped onto IP addresses when a session is initiated, and are used to provide a human-friendly identifier for a host. IP addresses are naming abstractions as well: each router locally maps the IP address of the packet it is forwarding into the next-hop router that the packet should be delivered to. By incrementally performing this resolution the network provides a more robust packet routing service than if it were to resolve the address to a path of routers in advance.

However, all of these naming abstractions maintain the idea that a name should specify *where* the object it names is located: IP addresses need to be assigned according to the routing topology of the network[1]; DNS names reflect the location of

[1] Though one could assign IP addresses randomly, it would have poor scaling properties: every router would have to keep state for every host in its domain, and the core routers would have to keep state for every host on the internet.

the resource within an administrative hierarchy; URLs include a DNS name, but also are extended with a site-local name that is typically a location in a filesystem.

This abstraction from one type of location to another is not always the most useful one. When an application or a user wants to access a service on a computer network, they usually start by knowing *what* they want, rather than *where* it is located. Naming that is based on location forces the user to manually remember and perform the mapping from what they want to where it is. For example, a user may know that she wants to retrieve a closing quote for stock she owns. She needs to remember the name of a specific URL that provides a stock quote service, and then navigate its web pages to find her quote. Other complications may occur in this process: the server she chooses may be far away on the network, heavily overloaded, or even have failed! A more powerful naming system would map what the application wanted to where on the network it is located. Such a system would encourage users to specify what they want, and move the burden of resolving "what is desired" to "where it is" from the user to the network infrastructure.

Our goals, therefore, are to (a) provide a system that allows users to express their intent by defining an *intentional name* for a service, and (b) have the network infrastructure perform the mapping from this intentional name to the actual network location of the service. The following section gives an overview of the network infrastructure, which we call the *Intentional Naming System* or *INS*. The rest of this thesis describes our approach to intentional names.

1.2 The Intentional Naming System (INS)

The Intentional Naming System (INS) is an architecture that allows applications to refer to network resources using intentional names. The architecture contains two distinct components:

- *Applications*, that use a library to access INS services.
- *Intentional Name Resolvers* or *INRs*, that are part of the network infrastructure.

The applications use the library to create and interpret intentional names, and to access services by communicating with the INRs. The INRs provide these services to the applications by sharing information with other INRs. The applications can communicate with the INRs in the following ways (illustrated in Figure 1-1):

- An application can ask an INR to resolve an intentional name to its network location. The application can then send data directly to the network location for an entire session. We call this *early binding*, since the intentional name is bound to the network location at session initiation time.

- An application can ask an INR to send some data to an intentional name. The INR then forwards it through a network of INRs until it gets to the network location. We call this *late binding*, since the intentional name is bound to the network location at data delivery time.

- An application can tell an INR that it is the network location for an intentional name. The INR propagates this information to its neighbors, and begins forwarding data for that intentional name to the application. We call this *advertisement*.

- An application can ask to be notified when an INR learns about new intentional names. We call this *discovery*.

The applications may connect to any INR, but picking the nearest one is beneficial in reducing delay and bandwidth consumption. The INRs form a communication network among themselves. While this network can have an arbitrary topology, it is desirable that neighbor relationships match the underlying network topology. Through this network INRs forward application data and send updated information about the intentional names they know about. The updates contain metrics indicating how desirable that particular advertiser of the intentional name is from the INR's location in the network. INRs run a variant of the distributed Bellman-Ford routing algorithm [4] on these updates to find which advertisers are best for them. This information is then used to resolve requests and forward data.

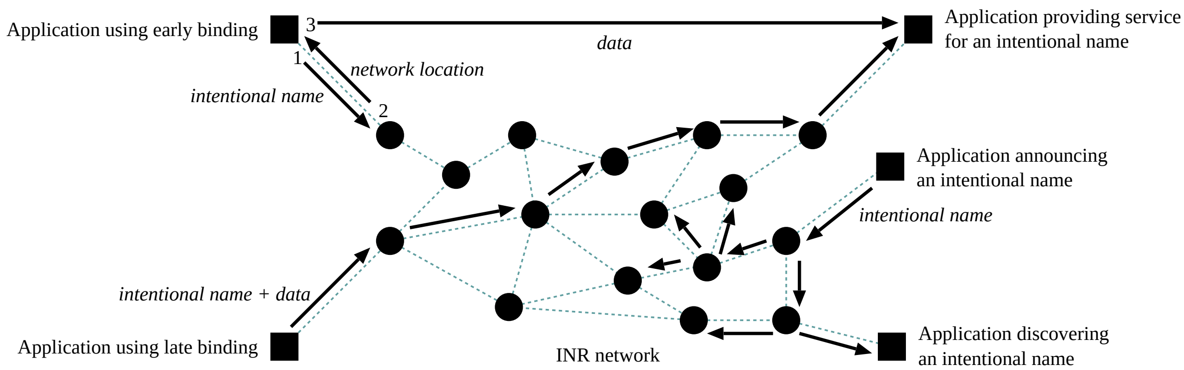

Figure 1-1: The architecture of the Intentional Naming System. The upper-left corner shows an application using early binding: the application sends an intentional name to an INR to be resolved (1), receives the network location (2), and sends the data directly to the destination application (3). The lower-left corner shows an application using late-binding: the application sends an intentional name and the data to an INR, which forwards it through the INR network to the destination application. The lower-right corner shows an application announcing an intentional name to an INR. The intentional name is beginning to be propagated throughout the INR network, and has reached an application performing discovery.

1.3 Related Work

Because intentional names border on many different areas, there exists a wide variety of related work. The areas that contain work that relates to INS are:

- **Naming systems.** Several other network naming systems have been developed. The most commonly used naming system is the Domain Name System (DNS), whose development is detailed in [25]; the most common implementation of DNS software, called BIND, is documented in [38]. Other naming systems include IEN116 [32], XEROX Grapevine [6], and Active Names [36].

- **Intentional names.** There has been some prior work on making the case for intentional names [26, 19, 14]. For example, the Semantic File System [17] uses intentional names for file retrieval.

- **Descriptive names.** Systems have been developed that define object description languages, and allow communication using these descriptions. Variants of these include Jini [20], T Spaces [24], XML [7], and The Information Bus

(R) [27].

- **Generic application-level services.** Other systems exist for providing application-level infrastructure that is not specific to a particular application. These include Application Layer Anycasting [5] and Active Services [1].

- **Research areas where intentional names may be useful.** There are also many efforts related to some of the areas intentional names may have benefits. These include resource discovery [37, 31], caching [2, 8, 10, 15, 23, 34, 39, 18], load balancing [9], group communication via multicast [11, 12, 3, 13, 22] and anycast [28], and mobility [30].

1.4 Outline

This thesis focuses on the design and implementation of intentional names, a key component of the Intentional Naming System; applications use intentional names to describe their services and INRs store a mapping from intentional names to network locations, as well as communicate information on intentional names to other INRs.

First, we discuss our design criteria for intentional names in the context of INS. Then we present a conceptual design of an intentional name for INS, which we call a *name-specifier*. We solidify this design by explaining how INRs represent a name-specifier internally and externally (Chapter 2). Building on this work, we present the *name-tree*, which is the data structure that INRs use to store information about name-specifiers. We also detail and analyze the algorithms that are used to perform operations on the name-tree (Chapter 3). Next we turn to our implementation of name-trees and explain our decisions and the structure of the implementation. To evaluate the performance of our implementation we discuss some experiments and their results (Chapter 4). In order to show how applications use INS, we define an application program interface for creating and manipulating name-specifiers, and describe a few demonstration applications that benefit from using INS (Chapter 5). We then conclude with a summary of our work (Chapter 6).

Chapter 2

Design of Name-Specifiers

An intentional name refers to any name that allows a user to express their intent. We use call the particular kind of intentional names that INS uses *name-specifiers*, and use this term when we are specifically referring to them rather than to the general use of intentional names. In Section 2.1 we present our design criteria for name-specifiers and explain their rationale. Section 2.2 shows our conceptual design of the name-specifier. Section 2.3 gives the concrete representations of the name-specifier: the in-memory data structure that the INS library uses, and the on-the-wire representation that is transmitted among INRs and applications.

2.1 Design Criteria

Our design criteria for name-specifiers fits into four categories: expressiveness, efficiency, generality, and deployability. Expressiveness is critical, since it is what gives users of INS an advantage over other naming approaches. Efficiency is important for any system that expects to provide good performance. Generality makes sure that the system is applicable to a wide range of applications. Deployability is necessary if the system is to be used in a larger community. Our design criteria for name-specifiers is summarized in Figure 2-1.

1. Expressiveness
   (a) Name-specifiers should be able to describe a rich variety of network resources.
   (b) Name-specifiers should be expressive enough that applications can select services without having to learn additional information through other means (for example, by contacting the service).
   (c) Name-specifiers should be expressive enough that applications can use them to name their data without having to place additional information in the packet.
   (d) Name-specifiers should allow applications to describe services without including every detail.
2. Efficiency
   (a) Name-specifiers should support fast resolution by INRs.
   (b) Name-specifiers should support efficient storage by INRs.
   (c) Name-specifiers should be computationally simple to handle, so that applications can run on platforms with small amounts of memory or slow processors (such as mobile nodes or networked devices).
   (d) Name-specifiers should allow for aggregation of information by permitting the suppression of details.
3. Generality
   (a) Name-specifiers should be resolvable by INRs without any additional application-specific information (such as database schema).
   (b) Name-specifiers should not be required to correspond to any physical, network, or administrative boundaries.
   (c) Name-specifiers should not impose implicit semantics regarding whether they refer to a single destination or many destinations.
   (d) Name-specifiers should support forms that are global in scope.
4. Deployability
   (a) Name-specifiers should support distributed application name space design.
   (b) Name-specifiers should be conceptually simple so that application designers can easily adopt them.
   (c) Name-specifiers should allow applications to evolve by providing reasonable approaches to backwards and forwards compatibility in the face of changing name spaces.

Figure 2-1: Summary of design criteria for name-specifiers.

Expressiveness:

1a. Name-specifiers should be able to describe a rich variety of network resources. This is necessary to ensure that INS is applicable to the wide range of present and future network services. These services have a large number of parameters and options, and name-specifiers should be able to express them.

1b. Name-specifiers should be expressive enough that applications can select services without having to learn additional information through other means. If name-specifiers are not expressive enough to do this, they lose many of their benefits (specifically, resource discovery and end point selection) since applications need to consult some other directory service or communicate directly with the service in an application-specific way.

1c. Name-specifiers should be expressive enough that applications can use them to name their data without having to place additional information in the packet. This expressiveness is desirable since such additional information is inaccessible by INRs, and therefore cannot be used to affect resolution decisions. Although there are cases in which it seems that certain information should never need to affect resolution, this is rarely clear at the time the application is created. For example, consider "layer 4 switches" [29]. These are network-layer devices that take application-layer information into account in making their forwarding decisions. To accomplish this, the switches need to be programmed with specific knowledge for each application protocol they handle, which is undesirable.

1d. Name-specifiers should allow applications to describe services without including every detail. Services may offer many options, and an application should not need to specify (or even be aware of) all of them.

Efficiency:

2a. Name-specifiers should support fast resolution by INRs. As the use of the system grows, so will the number of queries the INRs will face. Although additional

INRs can be deployed, designing the name-specifiers such that fast resolution is possible helps to abate this problem.

2b. Name-specifiers should support efficient storage by INRs. Greater use of the system also implies that INRs will need to store more name-specifiers. Therefore, the name-specifiers should be structured in such a way that they can be efficiently stored when there are many of them.

2c. Name-specifiers should be computationally simple to handle, so that applications can run on platforms with small amounts of memory or slow processors (such as mobile nodes or networked devices). These platforms are those most in need of intentional names (since there are many of them, and they have complex parameters), so it is important that applications require only minimal resources to use name-specifiers.

2d. Name-specifiers should allow for aggregation of information by permitting the suppression of details. This helps the system to scale by allowing INRs that are farther away to know less about the services. For example, a cluster of machines providing similar services may appear as one name-specifier to distant INRs, but local INRs may have additional information to dispatch requests within that cluster.

Generality:

3a. Name-specifiers should be resolvable by INRs without any additional application-specific information (such as database schema). The principle behind this is that name-specifiers should be self-contained. Requiring additional detail introduces hard state into the system, which raises the likelihood of failure and the complexity of INR behavior.

3b. Name-specifiers should not be required to correspond to any physical, network, or administrative boundaries. The principle behind this is that name-specifiers should be purely used to describe what an application wants, not where it is. If name-specifiers need to include a physical, network, or administrative location then INS will suffer from some of the same problems as existing naming systems.

3c. Name-specifiers should not impose implicit semantics regarding whether they refer to a single destination or many destinations. Some naming systems implicitly impose delivery semantics on their names. For example, certain IP addresses are set aside for multicast delivery, and the rest are used for unicast delivery. Name-specifiers should avoid this, and allow applications to explicitly select the type of delivery they desire.

3d. Name-specifiers should support forms that are global in scope. It is easy to produce intentional names that are limited in scope, for example, that only make sense to those in a single building or organization. However, for complete generality it should be possible to have name-specifiers that are globally meaningful.

Deployability:

4a. Name-specifiers should support distributed application name space design. If the name space needs to be designed centrally or adopted universally it will be very difficult to define. Different organizations should be able to define parts of the name space without affecting other organizations.

4b. Name-specifiers should be conceptually simple so that application designers can easily adopt them. If name-specifiers are difficult to understand, application designers will shun INS in favor of existing naming systems. The complexity must be commensurate with the benefits it provides.

4c. Name-specifiers should allow applications to evolve by providing reasonable approaches to backwards and forwards compatibility in the face of changing name spaces. Old applications should have some reasonable behavior when faced with name-specifiers that use new name space components they don't understand. New applications should be able to use old name-specifiers which may lack some of the new name space components.

2.2 Conceptual Design

Name-specifiers fill two distinct roles in INS: they are used by applications as (a) query expressions for services they are interested in, and (b) descriptions of the services they provide. Although overloading the name-specifier in this way may reduce the flexibility of the system, it does make the system easier to understand and streamlines the implementation of applications.

Keeping in mind the design criteria, we have developed the following conceptual design for name-specifiers. Most naming schemes in computer systems have either a flat structure (for example, memory addresses) or a simple hierarchical structure (for example, DNS names). In contrast, name-specifiers use a tree structure to allow for even more flexibility in describing intent.

The two basic building blocks of name-specifiers are textual fields called the *attribute* and the *value*. An attribute is a category in which an object can be classified. For example, the "color" of an object is such a category, as is the "floor" it is on. A value is an object's classification within that category. For example, the color of the object could be "red" or "blue" while its floor could be "4" or "5." INS (deliberately) does not understand the semantics of individual attributes and values, and only performs opaque comparisons on them. There is no predefined set of attributes or values that applications must use: they are free to define a set of attributes and values that best suits their application[1].

When a specific value is used to classify an object with respect to a specific attribute, this relationship is called an *attribute-value-pair* or *av-pair*. For example, an av-pair might convey that an object's "color is blue" or that its "floor is 5." In the simplest view an av-pair is just an association between an attribute and a value. However, an av-pair by itself may not carry all the information about the object: in the above examples we don't know what shade of blue the object is, or what room or wing of the 5th floor the object is in. To solve this problem, an av-pair has a set

[1] However, it may be desirable to standardize certain attributes and values. For example, if there were standard ways of describing colors or physical locations, different application designers could use these standards to reduce design time and provide interoperability.

of *child av-pairs*, which further describe the object. For example, the child av-pair of "color is blue" may be "shade is light" while "floor is 5" may have child av-pairs that include "room is 503" and "wing is north." Thus, child av-pairs provide added detail, within the context of their parent av-pair.

A name-specifier, then, is just an arrangement of av-pairs in a tree structure. The root of the tree is an implicit null value, which has as its children a set of *top-level attributes*. Figure 2-2 shows a graphical representation of an example name specifier.

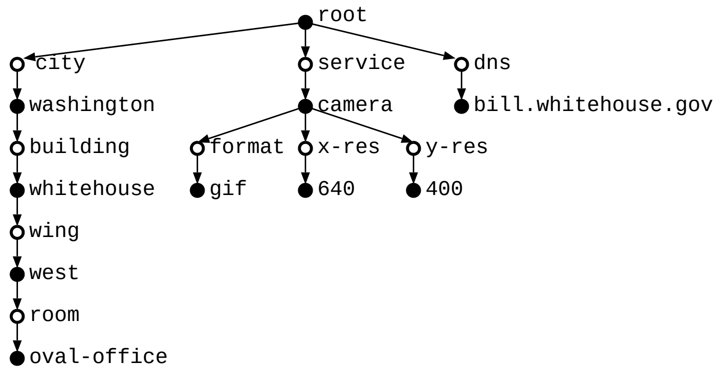

Figure 2-2: A graphical view of an example name-specifier. The hollow circles are used to identify attributes; the filled circles identify values. This name-specifier describes an object in the oval-office that provides a camera service (with 640-by-400 GIF images) and has a DNS name of bill.whitehouse.gov.

2.3 Concrete Representations

The previous section described the conceptual design of the name-specifier; this section describes the concrete representations of the name-specifier. The INS library provides a name-specifier data structure and functions to manipulate it; these are used by applications and within the INRs. The INS library also provides the ability to convert this data structure to and from a wire representation, which is in communication among different INRs and applications.

The name-specifier data structure, which follows the conceptual design closely, is defined in Figure 2-3. The name-specifier itself simply contains an av-pair which is the *root* of the name-specifier. Each av-pair contains an attribute *attribute* and its associated value *value*, as well as a list of av-pairs that are the *children* of this av-pair. There are a set of functions that applications and INRs use to manipulate name-specifiers and their components, without having to understand the details of the representation; these are fully described in the Applications chapter (Chapter 5) and Appendix A.

name-specifier:	
av-pair	*root*
av-pair:	
attribute	*attribute*
value	*value*
list of av-pair	*children*

Figure 2-3: The name-specifier data structure. Each definition contains a list of member variables for which the type and name are given. *list of* represents an abstract compound data structure.

In order to transmit name-specifiers across the network, it is necessary to have a flat representation. We use the following wire representation. Attributes and values are arbitrary length strings. To create an av-pair, they are separated by an equals sign (`=`). To create a tree of av-pairs, children are individually bracketed (with `[` and `]`) and placed after their parents. White space (spaces, tabs, etc.) can be used anywhere within the name-specifier, except in the middle of attributes and values; this allows name-specifiers to be formatted in a way that makes them easy for humans to read, which assists with debugging. An informal[2] BNF syntax of the wire representation is shown in Figure 2-4. Figure 2-5 shows a wire representation of the name specifier from Figure 2-2.

[2]The syntax is "informal" because we don't bother (a) specifying exactly what characters can be part of "a string" and instead assume they are limited to characters that don't cause ambiguities in the syntax, and (b) formally indicating that white space can be used anywhere within the name-specifier except in the middle of attributes and values. The limitations necessary to make the assumptions from (a) hold are:

1. An equals sign cannot appear in an attribute (since it is used between attributes and values).
2. A right bracket cannot appear in a value (since it is used to terminate an av-pair).

For consistency and simplicity, we suggest disallowing equals signs and brackets within either attributes or values.

attribute	= a string
value	= a string
av-pair	= '`[`', *attribute*, '`=`', *value*, { *av-pair* }, '`]`'
name-specifier	= { *av-pair* }

Figure 2-4: Informal BNF syntax of the wire representation of a name-specifier.

```
[city = washington
     [building = whitehouse
          [wing = west
               [room = oval-office]]]]
[service = camera
     [format = gif]
     [x-res = 640]
     [y-res = 400]]
[dns = bill.whitehouse.gov]
```

Figure 2-5: A wire representation of the name-specifier shown in Figure 2-2.

Alternative wire representations. The wire representation we use for name-specifiers was chosen to be readable by humans, in the spirit of other application-level protocols such as SMTP [33], HTTP [16], NNTP [21], etc. Having a human readable format makes testing and debugging quite easy, but is not the most compact or computationally efficient representation. If bandwidth or processing power is scarce, a different format may be desirable. For example, attributes and values need not be human readable strings, since it is expected that applications will be aware of the semantics of the attributes and values they use. More concretely, the use of name-specifiers require that applications have agreed on a set of attributes and values to communicate with a priori; there is no reason that the attribute or value can't be represented as a binary number rather than a human readable string in order to save bandwidth. Similarly, the structure of the name-specifier could be represented using a binary encoding rather than textual delimiters.

Chapter 3

Design of Name-Trees

The data structures that are used by INRs to store information about name-specifiers are called *name-trees*. Many of our design criteria for name-specifiers manifest themselves in name-tree design decisions. The design of name-specifiers and the design of name-trees also have a significant degree of interplay: choices made in designing name-specifiers constrain and guide options for name-trees, while the needs of name-tree operations influence the structure of name-specifiers.

There are two parts of an INR that interact with the name-tree. One of these is the code that handles the resolution of name-specifiers, and the other is the code that is responsible for sharing information about name-specifiers with other INRs. Thus, the function of a name-tree is similar to that of an IP routing table [35] or name server cache [38].

For resolving name-specifiers, the name-tree supports a LOOKUP-NAME operation, which returns the information associated with a name-specifier in the name-tree. Since the rate of lookups an INR must perform will grow as the system becomes more heavily used, lookups must be extremely efficient and scale well.

For sharing information about name-specifiers with other INRs, we use a protocol that is based on the Bellman-Ford routing protocol [4]: each INR sends the name of individual name-specifiers it knows about, as well as their metric and other associated information to its neighbors. To support this, the name-tree provides a GET-NEXT-NAME operation to iterate through the name-specifiers in the name-tree, and an

ADD-NAME operation to add a new name-specifier to the name-tree. Performance is less of a concern for these operations, because they are not used as frequently. The number of neighbors an INR has can be kept small, since the INR network is a controlled topology.

In Section 3.1 we define the name-tree data structure. The next three sections present the name-tree operations: Section 3.2 describes the LOOKUP-NAME operation, Section 3.3 describes the GET-NEXT-NAME operation, and Section 3.4 describes the ADD-NAME operation. An analysis of the LOOKUP-NAME operation is provided in Section 3.5.

3.1 Data Structure

The name-tree data structure is used to store information about the name-specifiers an INR knows. It is similar in structure to a name-specifier, resembling a superposition of all of the name-specifiers.

The fundamental components of name-trees are the *attribute-node* and *value-node*, which are analogous to the attribute and value of a name-specifier. However, in a name-specifier there is a one-to-one relationship between attributes and values, which are associated to form an av-pair. In name-trees, there is no analog to the av-pair. Like values, value-nodes have as their children attribute-nodes that are more specific categorizations of their value. However, unlike attributes, attribute-nodes also have multiple value nodes as children, which represent the different values the name-tree knows.

The name-tree also needs some way to keep distinguish among its different name-specifiers and retrieve information for a particular one. We call this per-name-specifier information a *name-record*. Each value-node in the name-tree contains a list of the name-records for the name-specifiers it is a part of. Thus, if a name-specifier is in the name-tree, the following condition holds: for each of the value-nodes in the name-tree that correspond to a leaf av-pair of the name-specifier, the name-record for the name-specifier will be in the value-node's list of name-records.

A name-tree, then, is an arrangement of alternating levels of value-nodes and attribute-nodes in a tree structure. The root of the tree is a value-node, whose value *value* and attribute-node *parent* are implicitly null. Figure 3-1 shows a graphical representation of part of an example name-tree. The name-tree also contains additional information for bookkeeping and efficiency that is detailed below, but not illustrated in the figure.

The precise definition of a name-tree is given in Figure 3-2. The name-tree itself contains only two variables: a value-node which is the *root* of the tree, and a list of name-record *records*. The list of name-record contains all of the name-records in the name-tree, and is used by the GET-NEXT-NAME algorithm to iterate through the name-specifiers in the tree.

Each value-node has several variables. The value *value* stores the value that this value-node represents. The list of attribute-node *children* contains all of the attribute-nodes that are children of this value-node; these are created when the name-specifiers in the name-tree have more specific av-pairs. The attribute-node *parent* contains the parent attribute-node of this value-node; it is used by the GET-NEXT-NAME algorithm when it traces up the tree. The list of name-record *records* contains all of the name-records for name-specifiers that end at this part of the name-tree. Finally, the av-pair PTR is usually null, but is used to store temporary values during the execution of the GET-NEXT-NAME algorithm.

An attribute-node has fewer variables. The attribute *attribute* stores the attribute that this attribute-node represents. The list of value-node *children* contains all of the value-nodes that are children of this attribute-node; these contain the possible values for the attribute that are in this name-tree. The value-node *parent* contains the parent value-node of this attribute-node; it is used by the GET-NEXT-NAME algorithm when it traces up the tree.

A name-record contains a list of value-nodes that are its *parents* in the name-tree. These are used by the GET-NEXT-NAME algorithm when it traces up the tree. It also contains other information about the name-specifier which the INR uses. These include next-hop INRs, network layer locations, metrics for these locations, and an expiration time for the name-record.

In order to simplify the presentation of the operations in the following sections, the recursive functions are written to be called on the *root* value-node of the name-tree. In an actual implementation, a short wrapper function would be called on the name-tree, which in turn would start the recursive function on the *root* value-node.

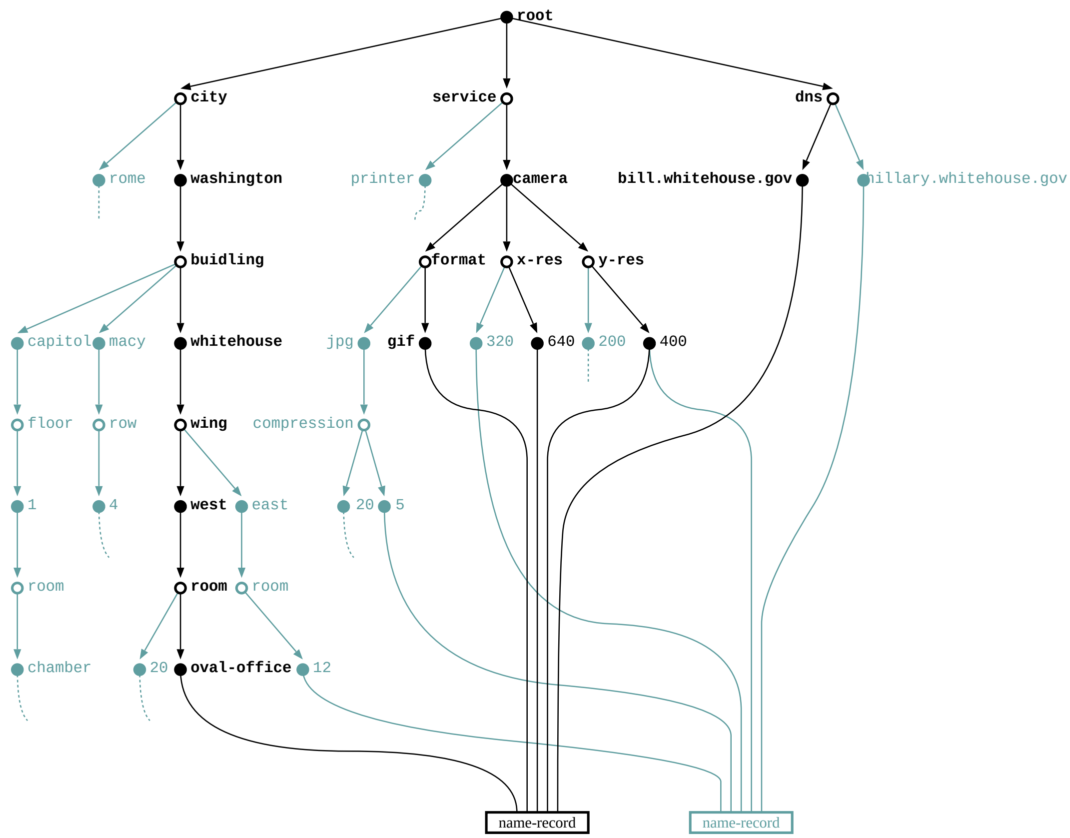

Figure 3-1: A graphical view of part of an example name-tree. The name-tree consists of alternating levels of attribute-nodes and value-nodes. Value-nodes contain a list of name-records for all the name-specifiers with corresponding leaf av-pairs. The part of the name-tree corresponding to the example name-specifier shown in Figure 2-2 is in bold.

name-tree:	
value-node	*root*
list of name-record	*records*
value-node:	
value	*value*
list of attribute-node	*children*
attribute-node	*parent*
list of name-record	*records*
av-pair	*PTR*
attribute-node:	
attribute	*attribute*
list of value-node	*children*
value-node	*parent*
name-record:	
list of value-node	*parents*
...other name information...	

Figure 3-2: The name-tree data structure. Each definition contains a list of member variables for which the type and name are given. *list of* represents an abstract compound data structure. The *other name information* is information used by the INR in resolving requests; these include the next-hop INR, the destination IP address, and a unique announcer ID. This information is not used by the name-tree algorithms.

3.2 The Lookup-Name Operation

The algorithm for the Lookup-Name operation, shown in Figure 3-3, is used to retrieve the name-records for a particular name-specifier n from the name-tree T. The main idea behind the algorithm is that a series of recursive calls reduce the set of candidate name-records S by intersecting it with the *records* at the value nodes of T that correspond to the leaf values of n. When the algorithm terminates, S contains only the relevant name-records.

Description. The algorithm starts by initializing S to the set of all possible name-records[1] (line 1). Then, for each av-pair p that is a child of n it finds T_a, the child attribute-node of T that corresponds to p's attribute (2-4). If there is no corresponding attribute-node, it just moves on to the next one (5-6). This means that applications can make queries that specify attributes which were not announced, and still have the lookup match some routes. This allows announcers the option of not advertising particular attributes and could be used, for example, if the announcer supports all the values of that attribute. Next the algorithm looks at the value of p. If the value is a wild card, then it computes S' as the union of all name-records in the subtree rooted at the corresponding attribute-node, and intersects S with S' (8-14). This means that specifying an attribute with a wild card value is actually more restrictive than not specifying the attribute at all, since only the name-specifiers that provide additional (more-specific) av-pairs are retained. If the value is not a wild card, it finds the corresponding value-node in the name-tree (15-18). If it has reached the leaf of either the name-specifier or the name-tree, the algorithm intersects S with the name-records pointed to by the corresponding value-node (19-20), thus narrowing down the set S to those name-records that meet the criteria of this branch. If not, it makes a recursive call to compute the relevant set from the subtree rooted at the corresponding value, and intersects that with S (21-22). Finally, the algorithm returns the union of the name-records it has found below it that match the criteria

[1] Note that an implementation does not actually need to store all possible name-records explicitly, since after the first intersection operation S will be reduced to the smaller list of records.

(S), and the name-records at this location in the name-tree (24); this too assists in allowing announcements to be made for only higher level attributes.

```
LOOKUP-NAME(T, n)
1    S ← the set of all possible name-records
2    for p ← each av-pair in n.children
3        T_a ← the attribute-node of T.children such that
4             T_a.attribute = p.attribute
5        if T_a = null
6            continue
7
8        if p.value = *
9            ▷ Wild card matching.
10           S′ ← ∅
11           for T_v ← each value-node in T_a.children
12               S′ ← S′ ∪ all of the records of value-nodes
13                           in the subtree rooted at T_v
14           S ← S ∩ S′
15       else
16           ▷ Normal matching.
17           T_v ← the value-node of T_a.children such that
18                 T_v.value = p.value
19           if T_v is a leaf node or p is a leaf node
20               S ← S ∩ T_v.records
21           else
22               S ← S ∩ LOOKUP-NAME(p, T_v)
23
24   return S ∪ T.records
```

Figure 3-3: The LOOKUP-NAME algorithm. This algorithm looks up the name-specifier n in the name-tree T and returns all appropriate name-records.

3.3 The Get-Next-Name Operation

The Get-Next-Name algorithm, shown in Figure 3-4, is used to iterate through the name-specifiers stored in the name-tree. The INR calls Get-Next-Name to retrieve the next name-specifier to be sent in a periodic update to one of its neighbors. The algorithm works by iterating through a list of name-records that is stored in the name-tree, and returning the name-specifier associated with each one. To find the name-specifier from a name-record, it uses the Trace algorithm (shown in Figure 3-5) to trace up through the name-tree from the name-record, reconstructing the name-specifier as it goes along.

Description. The parameters for the algorithm are T, the name-tree, and j, a counter that keeps the state of the iteration. The algorithm starts starts by pulling a name-record out from $records$, based on j, and storing that in the variable $record$ (lines 1-2). Next it creates a new, empty list *touched*, which will be used to store all of the value-nodes whose PTR has been set (4-5). It also creates a new, empty name-specifier n that will eventually contain the return value of the algorithm (7-8). It ties the name-specifier n to the name-tree T by setting the PTR of the root value-node of the name-tree to point to the top of the name-specifier (10-12). Now the algorithm is ready to begin reconstructing the rest of the name-specifier from the name-tree. To do this, it traces back from all of the value-nodes that contain $record$ (14-16). It passes to the Trace algorithm the place to start tracing from T_v, the fragment of the name-specifier reconstructed so far (*null* at this point), and the list of *touched* value-nodes. Now that the name-specifier n has been reconstructed, it restores the name-tree to its original condition by resetting all of the PTRs of the value-nodes in *touched* back to *null* (18-20). Finally it returns the name-specifier n (22).

The Trace algorithm traces up the name-tree from value-node T_v. As it traces it creates a name-specifier fragment (of which n is the bottom part), and adds any value-nodes whose PTR it sets to the list *touched*. The main idea behind the algorithm is that as it traces up the name-tree it looks at the PTR variable of the value-nodes. If the PTR is filled in, it has found part of the name-specifier that has been reconstructed already, and it adds the fragment it has reconstructed on to it; if it isn't filled in, it reconstructs another av-pair of the name-specifier fragment and proceeds upwards.

The details of the TRACE algorithm as as follows. First, the algorithm checks the PTR (T_v.PTR) to see if it has been filled in. If so, then it contains an av-pair, which means the algorithm has found the main name-specifier (lines 1-2). If the algorithm has reconstructed a fragment of the name-specifier (i.e. $n \neq null$), then it attaches the fragment (n) to the av-pair of the main name-specifier and terminates (3-5). If the PTR hasn't been filled in (i.e. T_v.PTR $= null$), it reconstructs more of the name-specifier fragment (6-7). It does this by creating a new av-pair, whose value is taken from T_v's value and whose attribute is taken from T_v's parent's attribute; this av-pair is assigned to T_v.PTR (8-9). Since T_v.PTR has been modified, T_v is added to the list of *touched* value-nodes (10). If part of the name-specifier has been reconstructed already (i.e. $n \neq null$), then it attaches n to the new av-pair stored in T_v.PTR (12-14). Finally, it continues up the name-tree by recursively calling TRACE to trace up from the grandparent of T_v (i.e. the value-node above T_v in the name-tree), adding onto the top of the name-specifier fragment T_v.PTR (which will be the new n), and passing along the list of *touched* value-nodes.

Example. An illustration of an in-progress execution of the GET-NEXT-NAME algorithm is shown in Figure 3-6. In this execution, almost all of a name-specifier has been recreated. The left branch of the name-specifier associated with the name-record *record* has been traced all the way up to the *root*, as can be seen by the presence of PTR assignments. The bottom value-node of the right branch has also been traced, and the algorithm is currently grafting the name-specifier fragment of the right branch onto the main name-specifier. It does this by noticing it has reached a value-node T_v whose PTR (T_v.PTR) already has an av-pair, and then adding the fragment rooted at n as a child av-pair of T_v.PTR. Since all branches of *record* have been traced, the name-specifier is fully created and the algorithm terminates.

```
GET-NEXT-NAME(T, j)
    ▷ Get the next name-record to extract a name for.
    record ← the jth element of records

    ▷ Create a list of value-nodes whose PTR has been touched.
    touched ← a new, empty list of value-node

    ▷ Create the name-specifier to be returned.
    n ← a new, empty name-specifier

    ▷ Attach the name-specifier to the root, and mark it as touched.
    T.PTR ← n
    append T to touched

    ▷ Construct the name-specifier by tracing backwards.
    for T_v ← each value-node in record.parents
        TRACE(T_v, null, touched)

    ▷ Clear the PTRs that have been touched.
    for T_v ← each value node in touched
        T_v.PTR ← null

    return n
```

Figure 3-4: The GET-NEXT-NAME algorithm. This algorithm extracts and returns the name-specifier for the name-record r in the name-tree T.

```
TRACE(T_v, n, touched)
  if T_v.PTR ≠ null
      ▷ We've found the main name-specifier!
      if n ≠ null
          ▷ Graft the fragment we've recreated on to it, and we're done.
          append n to T_v.PTR.children
  else
      ▷ We haven't found the main one yet; recreate more of the fragment.
      T_v.PTR ← a new av-pair consisting of
                    T_v.value and T_v.parent.attribute
      append T_v to touched

      if n ≠ null
          ▷ Graft the fragment we've recreated onto the new part.
          append n to T_v.PTR.children

      ▷ Continue up the name-tree.
      TRACE(T_v.parent.parent, T_v.PTR, touched)
```

Figure 3-5: The TRACE algorithm. This algorithm recursively traces up the name-tree from a value-node T_v, building part of a name-specifier as at goes up, and grafting it on to the name-specifier to be returned when it reaches it.

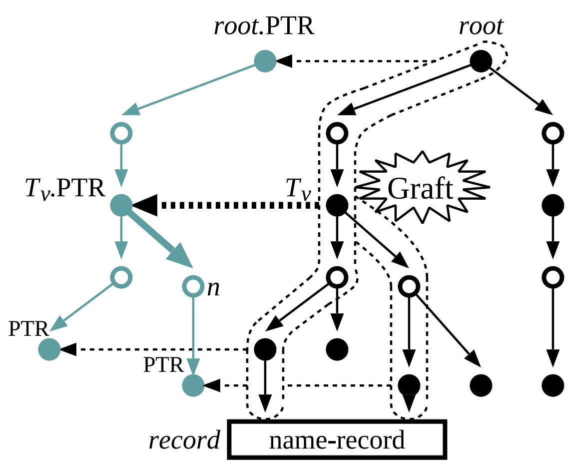

Figure 3-6: An illustration of an in-progress execution of the GET-NEXT-NAME algorithm. The name-specifier being created is shown in gray on the left, while the name-tree it is being created from is shown black on the right. The parts of the name-tree that are circled with dotted lines are the paths through the name-tree that have been traced. The dotted arrows are used to illustrate the assignments of the PTR variables. The thick arrows indicate the parts of the data structures that are currently being manipulated. In this example, the name-specifier fragment rooted at n is being grafted onto T_v.PTR, which is part of the main name-specifier.

3.4 The Add-Name Operation

The Add-Name algorithm, shown in Figure 3-7, is used to add a name-record r for a name-specifier n into the name-tree T. The INR calls Add-Name to add new name-specifiers that it has learned about to the name-tree. The algorithm is similar in structure to the Lookup-Name algorithm: a set of recursive calls are used to descend through the name-tree, creating additional nodes as needed, and adding the name-record to the records at the value-nodes that correspond to the values at the bottom of the name-specifier.

Description. The algorithm starts by adding the name-record r to the list of $records$ (lines 1-2); this list is used by Get-Next-Name algorithm to iterate through the name-records in the name-tree. Then, for each av-pair p that is a child of n it does the following (4). First it finds T_a, the child attribute-node of T that corresponds to p's attribute (5-7). If there is no corresponding attribute-node (i.e. $T_a = null$), it creates a new attribute-node, assigns it to T_a, and then adds it as a child of T (8-10). Next the algorithm finds T_v, the child value-node of T_a that corresponds to p's value (12-14). If there is no corresponding value-node (i.e. $T_v = null$), it creates a new value-node, assigns it to T_v, and then adds it as a child of T_a (15-17). Finally the algorithm checks if it is at the bottom of the name-specifier (i.e. if p is a leaf node) (19-20). If so, it adds the name-record r to the records at this node in the name-tree (21); if not, it makes a recursive call to add the name-record r to the part of the subtree rooted at T_v that corresponds to the part of the name-specifier rooted at p.

```
Add-Name(T, n, r)
    ▷ Add the name-record to the list of records.
    append r to records

    for p ← each av-pair in n.children
        ▷ Find or create the attribute-node.
        T_a ← the attribute-node of of T.children such that
              T_a.attribute = p.attribute
        if T_a = null
            T_a ← a new attribute-node with p.attribute
            append T_a to T.children

        ▷ Find or create the value-node.
        T_v ← the value-node of T_a.children such that
              T_v.value = p.attribute
        if T_v = null
            T_v ← a new value-node with p.value
            append T_v to T_a.children

        ▷ Add the name-record here, or recurse further down.
        if p is a leaf node
            append r to T_v.records
        else
            Add-Name(T_v, p, r)
```

Figure 3-7: The Add-Name algorithm. This algorithm adds the name-record r for the name-specifier n in the name-tree T.

3.5 Analysis

Since the scalability of INRs under heavy load is a major concern, it is important to analyze the performance of the lookup algorithm. While many of the tasks involved in resolving an name-specifier take a constant amount of time (e.g., copying the data, transmitting it over the network), the time to perform a lookup depends on the structure and quantity of name-specifiers in the name-tree. It is therefore important to determine the worst-case run-time of the algorithm.

To simplify the analysis of our lookup algorithm, we assume that name-specifiers are uniform trees with the following dimensions: d, one-half the depth of name-specifiers; r_a, the range of possible attributes in name-specifiers; r_v, the range of possible values in name-specifiers; and, n_am the actual number of attributes in name-specifiers. These are summarized in Figure 3-8 and illustrated in Figure 3-9.

d	One-half the depth of name-specifiers
r_a	Range of possible attributes in name-specifiers
r_v	Range of possible values in name-specifiers
n_a	Actual number of attributes in name-specifiers

Figure 3-8: The dimensions of a uniform name-specifier.

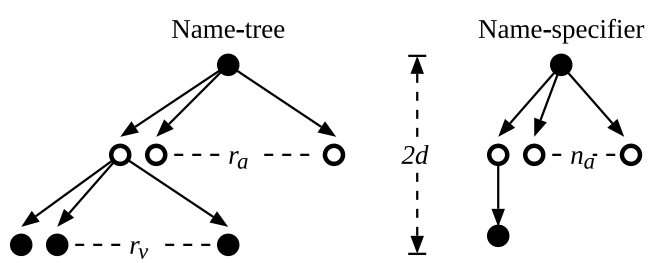

Figure 3-9: An illustration of the dimensions for a uniform name-specifier. Note that $d = \text{depth}/2 = 1$ for this tree.

Analysis In each invocation, the algorithm iterates through the attributes in the name-specifier, finding the corresponding attribute-node and value-node in the name-tree, and making a recursive call. Thus, the run-time is given by the recurrence,

$$T(d) = n_a \cdot (t_a + t_v + T(d-1)),$$

where t_a and t_v represent the time to find the corresponding attribute-node and value-node, respectively. Assume that it takes time b for the base case such that:

$$T(0) = b$$

Setting $t = t_a + t_v$ and performing the algebra yields:

$$\begin{aligned}
T(d) &= n_a \cdot (t + T(d-1)) \\
&= n_a \cdot t + n_a \cdot T(d-1) \\
&= n_a \cdot t + n_a^2 \cdot (t) + n_a^2 \cdot T(d-2) \\
&= n_a \cdot t + \ldots + n_a^{d-1} \cdot t + n_a^{d-1} \cdot b \\
&= \frac{n_a^d - 1}{n_a - 1} \cdot t + n_a^{d-1} \cdot b \\
&= \Theta(n_a^d \cdot (t + b))
\end{aligned}$$

If linear search is used to find attribute-nodes and value-nodes, the running time would be:

$$T(d) = \Theta(n_a^d \cdot (r_a + r_v + b)),$$

because $t_a \propto r_a$ and $t_v \propto r_v$ in this case.

However, using a straightforward hash table to find these reduces the running time to:

$$T(d) = \Theta(n_a^d \cdot (1 + b))$$

Implications. From the above analysis, it seems that the n_a^d factor may suffer from scaling problems if d grows large. However, both n_a and d, will scale up with *the complexity of a single application* associated with the name-specifier. There are only as many attributes or levels to a name-specifier as the application designer needs to describe the objects that are used by their application. Consequently, we expect that that n_a and d will be near constant and relatively small; indeed, the sample applications we describe in Applications chapter (Chapter 5) have this property.

The cost of the base case, b, is the cost of an intersection operation between the set of route entries at the leaf of the name-tree and the current target route set. Taking the intersection of the two sets of size s_1 and s_2 takes $\Theta(max(s_1, s_2))$ time, assuming the two sets are sorted (as in our implementation). In the *worst* case the value of b is on the order of the size of the universal set of route entries ($\Theta(|U|)$), but is usually significantly smaller. Unfortunately, an average case analysis of b is difficult to calculate analytically since it depends on the number and distribution of names. However, in the following chapter we discuss experimental results that were produced by an actual implementation, and indicate that the cost of the base case is relatively low.

Chapter 4

Implementation

In order to demonstrate the advantages of intentional names, we developed a prototype implementation of the INS components. This chapter discusses our implementation of the INS library, which allows applications and INRs to manipulate name-specifiers, and of the name-tree, which is used by the INRs to store information on name-specifiers.

In Section 4.1 we describe the components of our implementation. Section 4.2 explains our choice of Java as an implementation language, and discusses the language features we use. Section 4.3 evaluates the performance of our implementation.

4.1 Implementation Components

Our implementation of the INS library and name-tree is made up of several components. In our Java prototype, these components are classes that are named closely after their data structures. Figure 4-1 contains a list of the classes in the implementation; the actual source code can be found in Appendix B.

The classes that make up the INS library are *NameSpecifier*, *AVPair*, *Attribute*, and *Value*. *NameSpecifier*, *Attribute*, and *Value* fairly simple, since they only contain methods to access their data; *Value* contains more methods than *Attribute* since it handles the special case of wild card values. In addition to the access methods, *AVPair* contains a method that parses the wire representation of a name-specifier,

Java class	**Data structure or use**	**Lines**
NameSpecifier	name-specifier	96
AVPair	av-pair	314
Attribute	attribute	62
Value	value	103
NameTree	name-tree	140
AttributeNode	attribute-node	123
ValueNode	value-node	299
NameRecordSet	name-record-set	274
NameRecord	name-record	71
TestNameSpecifier	name-specifier tests	73
TestNameTree	name-tree tests	436
TestNameRecordSet	name-record-set tests	109
TestNameTreePerformance	name-tree performance tests	320
Total		2420

Figure 4-1: Classes in the prototype Java implementation and their associated data structure or use. *Lines* gives the number of source lines of code, including comments and whitespace.

and an equality testing method.

The classes that implement the name-tree are *NameTree*, *AttributeNode*, *ValueNode*, *NameRecord*, and *NameRecordSet*. *NameTree* contains only wrapper methods, which initialize the algorithms that the name-tree provides; *ValueNode* provides most of the implementation of these algorithms. *AttributeNode* is simple and provides only access methods; a future implementation may find it more elegant to move some of the algorithm code into this class, if possible. *NameRecordSet* implements an ordered list, to allow fast set union and intersection operations. *NameRecord* contains many access methods for the information that the INR uses; these are not used by the name-tree and are omitted from the line count in Figure 4-1. A better way to implement NameRecord would be to move that information into a separate class that can be treated opaquely by the name-tree. There are also two other classes, *CantParseString* and *ElementNotFound*, which are used only as identifiers for Java exceptions.

There are also classes that non-operational code and are used for testing. *TestNameSpecifier*, *TestNameTree*, and *TestNameRecordSet* are used to test that the implementations of their associated classes are working correctly. *TestNameTreePerformance* is used to produce the experimental results in Section 4.3.

4.2 Implementation Language

We made the decision to use Java as the language for our prototype implementation of INS based on its cross platform portability, and the rapid development that a high-level, object-oriented language with well-documented and relatively complete libraries allows. However, since the INS design is deliberately language-independent, we also strove to make our implementation techniques as language-independent as possible.

The following features of Java are used by our implementation. The implementation makes modest use of inheritance: *NameSpecifier* is a subclass of *AVPair* and *NameTree* is a subclass of *ValueNode*. These could easily be changed to have *NameSpecifier* store a copy of *AVPair*, for example, if inheritance isn't available. The implementation also only makes minimal use of exceptions, and they could easily be replaced by special return values. However, the implementation does make heavy use of a few Java library classes. Vectors, which provide an automatically resizable array, are used to store sets that hold the children of attribute-nodes and value-nodes, the name-records of a value-node, the list of all name-records, the parents of name-records, and the list of value-nodes whose PTR has been touched by the TRACE algorithm. Enumerations, which provide an iteration abstraction, are used to iterate through these sets. StringTokenizers, which flexibly break a string into tokens, are used to parse the wire representation of name-specifiers.

4.3 Evaluation

In this section we evaluate the performance of our implementation. In particular, we look at the rate of lookups that the name-tree is capable of sustaining, and how this depends on the number of name-specifiers the INR knows about. Then we examine if the memory usage of the name-tree is a limiting factor in the number of name-specifiers an INR can maintain.

Methodology. We create a number of randomly constructed name-trees, and time how long it takes to perform 1000 random lookup operations on the tree. Since this experiment is designed to extend the analytical work, the name-tree and name-specifiers are chosen to be uniform with the same parameters as the analysis in Section 3.5. We create an empty name-tree, and add to it a set of n random name-specifiers, each with a unique name-record. These random name-specifiers are also recorded separately so they can be looked up in the name-tree.

The random name-specifiers are created as follows. For each of the d levels of the name-specifier, n_a different attributes are chosen randomly from a total of r_a possible attributes. Then for each attribute, a value is chosen randomly from a total of r_v possible values. The actual attributes and values used are just the string representation of the integers from 0 to $r_a - 1$ and $r_v - 1$, respectively; these are smaller than the actual attributes and values used in our sample applications, but are representative of what attributes and values would be like if our design and implementation supported integer types.

The experiment then precomputes a set of 1000 random numbers that are indices into the separate list of name-specifiers. It then records the start time, does a lookup in the name-tree for each one of those name-specifiers, and subtracts the start time from the finish time to get the total time taken. From this we calculate the number of lookups per second.

We also attempt to determine the total amount of memory used by the name-tree as follows. After we have run our lookup test there are three major consumers of memory: the name-tree, the list of name-specifiers, and other objects. Of these, only the name-tree and the list of name-specifiers depend on the number of name-specifiers in the name-tree; the space consumed by the other objects is constant across different trials. There are two ways to measure the memory consumed by the name-tree. The obvious way is to call the Java garbage collector to remove any unreferenced objects, record the amount of free memory, remove any references to the name-tree, call the garbage collector again, and then subtract the original amount of free memory from the current amount of free memory. However, in doing this we discovered that the garbage collector does not always free the name-tree; this produces results in which the name-tree appears to have consumed no space. The method actually used is a more rough estimation, but does not seem to produce this anomaly. Instead of trying to look at the difference in free space, we just remove our reference to the list of name-specifiers and calculate the name-tree size as the difference between the total memory and the free memory. The assumption made here is that it is the list of name-specifiers consumes most of the memory, not the name-tree. This approach produces results that are consistent with the more obvious approach, but are off by the amount of space consumed by other objects associated with the experiment (approximately 200 kb).

Parameters and environment. We chose the parameters to be $r_a = 3$, $r_v = 3$, $n_a = 2$, and $d = 3$, as illustrated in Figure 4-2. These parameters were chosen based on the name spaces of our demonstration applications, which we present in Chapter 5. We varied n, the number of name-specifiers in the name-tree between 100 and 14300, in increments of 100. The range of experiments possible was limited by the extra memory required to store the list of name-specifiers to be looked up in the name-tree, and not by the much smaller name-tree.

We performed our experiment on a standard PC with an Intel Pentium II processor running at 450 MHz and containing 512 kb cache and 128 Mb RAM. The machine was running Red Hat Linux 5.2 with the default kernel (version 2.0.36). The code was compiled and run with the Blackdown Java-Linux port of Sun's JDK (Java Development Kit) 1.1.7. We limited the maximum heap size of the Java interpreter

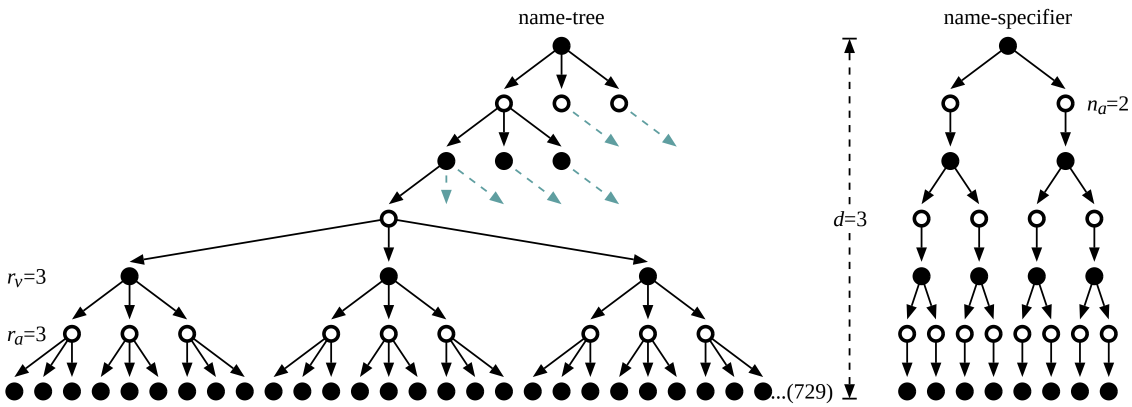

Figure 4-2: Experiment parameter. This diagram shows the structure of a uniform name-tree and name-specifier with parameters $r_a = 3$, $r_v = 3$, $n_a = 2$, and $d = 3$.

to 64 Mb and set the initial allocation to that amount to avoid artifacts from other memory allocation on the machine.

Results. The name-tree lookup performance results are shown in Figure 4-3. For this name-tree and name-specifier structure, our performance went from a maximum of about 900 lookups per second to a minimum of about 700 lookups per second. Extrapolating from the trend in the graph, it appears that the name-tree should be capable of sustaining over 100 lookups per second for name-trees of up to 75000 names. This experiment also give us a practical idea of the cost of set intersection an union operations (the base case b from Section 3.5) affects the performance of the lookup algorithm.

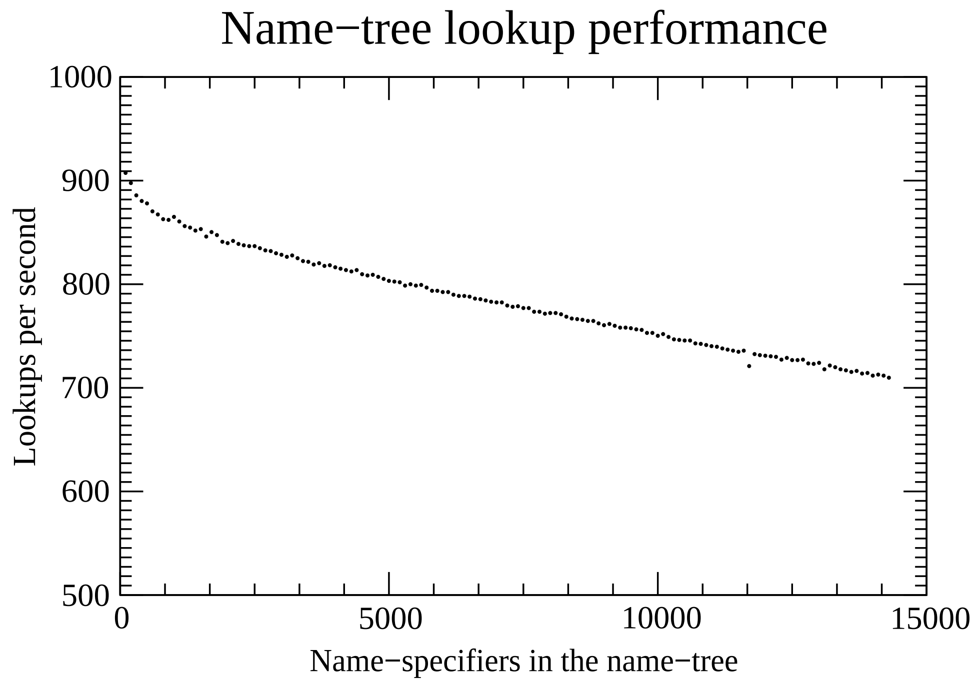

Figure 4-3: Experimental name-tree lookup performance. This graph shows how the name-tree lookup performance of an INR varies according to the number of names in its name-tree.

The name-tree size results are shown in Figure 4-4. The amount of memory allocated to the name-tree went from approximately 0.5 megabytes to 4 megabytes as the number of names was increased. Based on these results, a name-tree with 75000 names would require approximately 16 megabytes of space.

We believe that this order-of-magnitude of lookup performance is adequate for intra-domain deployment, because of the load balancing provided by having multiple INRs, and the parallelism inherent in independent name lookups. In addition, the memory consumption is low enough that it doesn't present a barrier to deployment.

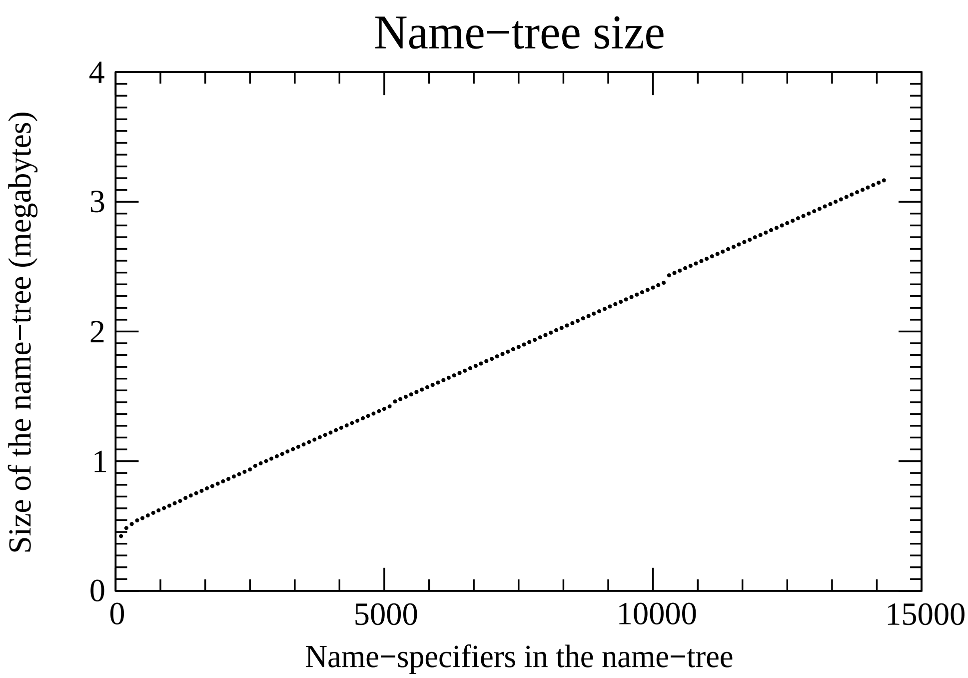

Figure 4-4: Experimental name-tree size. This graph shows how the name-tree size varies according to the number of names in its name-tree.

Chapter 5

Applications

In this chapter we discuss how applications use INS. In Section 5.1 we present the Application Program Interface (API) for the INS library, and offer some advice on its use. Next, we present some of the demonstration applications that we have implemented to validate our design and get some experience in application design using INS. Section 5.2 presents *Floorplan*, a graphical service discovery tool; Section 5.3 presents *Camera*, a mobile camera service; and Section 5.4 presents *Printer*, a printer service.

5.1 Application Program Interface

In order to allow applications to easily handle intentional names we provide a library that has abstractions for creating, understanding, and manipulating name-specifiers and their components. It is expected that applications will insulate the user from the structure of name-specifiers. In other words, a name-specifier should never explicitly be presented as output or required as input from a user. Instead, the application should present an interface that matches the intent of the name-specifier. For example, it would be poor practice to have a user enter a name-specifier for a location. Instead, the user should be able to use a map or hierarchical list to choose the location they want to express. Similarly, the choice of resolution for a camera application should be presented to the user as a list of options, rather than as an arbitrary text or numeric input.

The functions of the Application Program Interface (API) for the library are shown in Figure 5-1; the complete API can be found in Appendix A. The design of the library is object-oriented, and objects are the four main components of name-specifiers: the name-specifier itself, the av-pair, the attribute, and the value.

All objects provide a constructor that creates an object from its wire representation, a copy constructor that creates an equal but different instance of the object, an equality operator for testing whether two objects are the same, and an operator that can produce a human readable representation of the object.

The name-specifier object provides methods for creating an empty name-specifier, creating a name-specifier from its wire representation, and getting the av-pair that is the root of the name-specifier. The equality and other common operators are omitted from the name-specifier, since they can be performed by first getting the av-pair that is the root of the name-specifier, and then performing the desired operation on the av-pair.

The av-pair object provides the largest number of methods. It provides constructor methods for creating an av-pair from an attribute and value, access methods for retrieving its attribute and value, and methods for handling its child av-pairs which allow adding a child, removing a child or all children, testing for the presence of children, iterating through the children, and retrieving children with a particular attribute.

The attribute object provides only the common operators, but the value object also includes methods for creating wild card values and testing if a value is a wild card.

name-specifier:	
name-specifier	createNamespecifierEmpty()
name-specifier	createNamespecifierFromString(string s)
name-specifier	createNamespecifierFromNamespecifier(name-specifier n)
av-pair	namespecifierGetRoot(name-specifier n)
av-pair:	
av-pair	createAvpair(attribute a, value v)
av-pair	createAvpairFromString(string s)
av-pair	createAvpairFromAvpair(av-pair p)
attribute	avpairGetAttribute(av-pair p)
value	avpairGetValue(av-pair p)
void	avpairAddChild(av-pair p, av-pair *child*)
void	avpairRemoveChild(av-pair p, av-pair *child*)
void	avpairRemoveAllChildren(av-pair p)
boolean	avpairIsLeaf(av-pair p)
av-pair	avpairGetChild(av-pair p, attribute a)
av-pair	avpairGetNextChild(av-pair p, iteration-state j)
boolean	avpairsEqual(av-pair $p1$, av-pair $p2$)
string	avpairToString(av-pair p)
attribute:	
attribute	createAttributeFromString(string s)
attribute	createAttributeFromAttribute(attribute a)
boolean	attributesEqual(attribute $a1$, attribute $a2$)
string	attributeToString(attribute a)
value:	
value	createValueFromString(string s)
value	createValueWildcard()
value	createValueFromValue(value v)
boolean	valueIsWildcard(value v)
boolean	valuesEqual(value $v1$, value $v2$)
string	valueToString(value v)

Figure 5-1: The functions of the name-specifier API.

5.2 *Floorplan*: A Graphical Service Discovery Tool

In the process of building the mobile camera and printer services, we realized that we needed an easy way to start up applications that access these services. The simplest way would have been to have an application which just takes the name-specifier of the camera or printer to contact on the command line, but this is problematic because the camera or printer you wish to contact may not exist. A slightly more complex way would be to use the discovery services of INS to present a list of the existing name-specifiers to user, and have her pick one. However, this has the undesirable quality that it exposes raw name-specifiers to the user, which may be difficult to understand. Instead, we have constructed a graphical service discovery tool called *Floorplan* which presents the intent conveyed by the name-specifiers to the users in a way that makes intuitive sense.

Floorplan uses the location information contained in the name-specifiers to show the user where the services it knows about are located, and uses the service information in the name-specifiers to decide what icon to use for the service. Figure 5-2 shows a screenshot from *Floorplan*.

Location information is conveyed through the use of a location name space within the name-specifier. Figure 5-3 shows the layout of the location name space. The top-level location attribute that we have chosen to use is “organization.” All applications need to be aware of this top-level location attribute in order to find the location information. The value of the organization attribute indicates the organization that the location information is for. After this, all the attributes and values can be arbitrarily chosen by each organization. For MIT, whose value is “mit,” the next attribute is “building” and is followed by “floor” and “room.” Although MIT’s part of the location branch doesn’t take advantage of the fact that attributes can vary based on the value (all MIT locations contain building, floor and room attributes), the top part of the location branch uses this to allow per-organization attributes. For example, an organization that has only one building may want to start directly with a “floor” attribute, or use “wing” or some other subdivision instead of “building.”

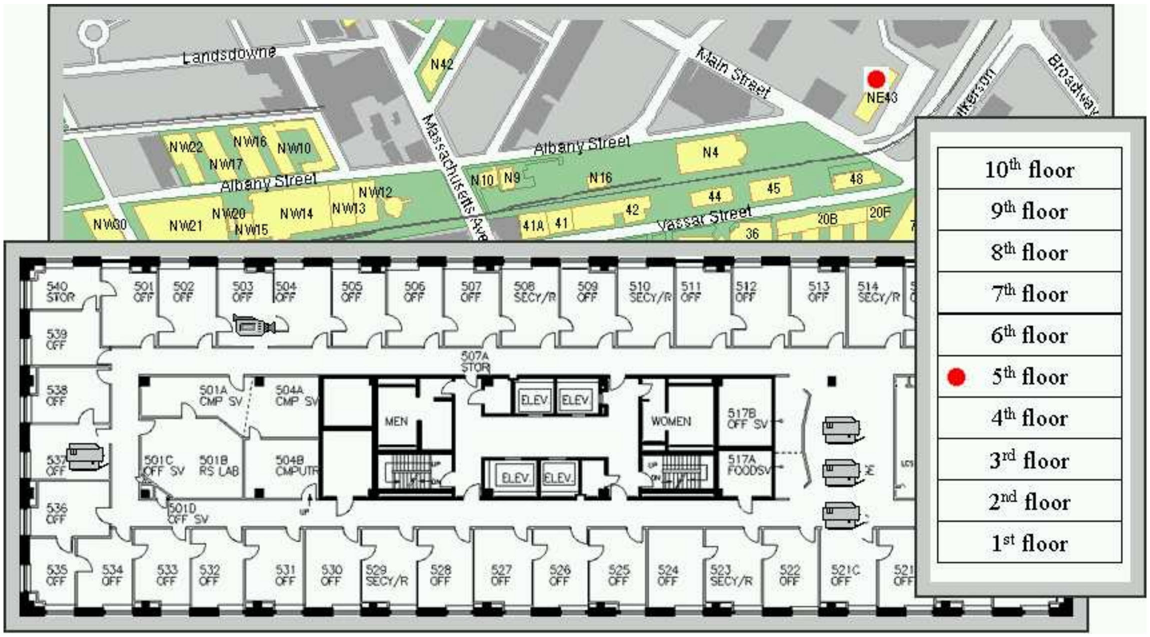

Figure 5-2: A screenshot from *Floorplan*. The circle by NE43 on the MIT map indicates a service has been discovered at that location. Clicking on the circle brings up a map of NE43, which has another circle on the 5th floor. Clicking on that brings up a floorplan of the 5th floor. The floorplan shows the discovered services: three printers in the lounge, one by room 537, and a camera in 503.

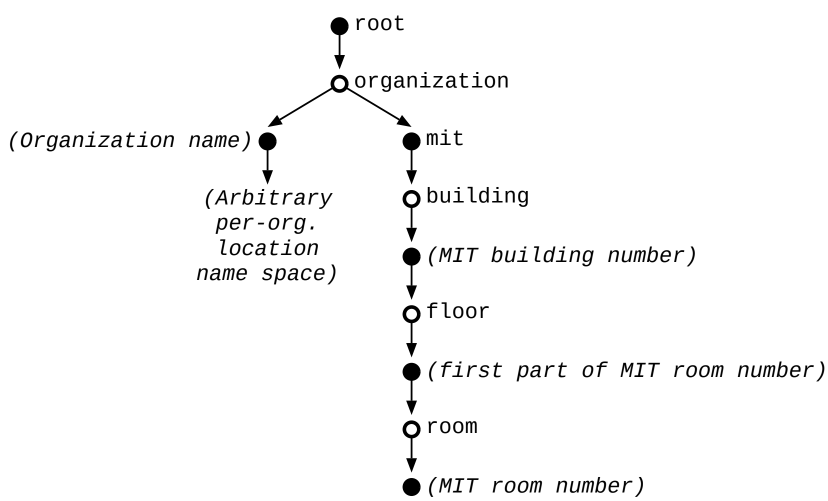

Figure 5-3: The location name space. The top-level attribute is "organization." The value of organization is the name of the organization, and the children of this value form an arbitrary per-organization location branch. The right hand branch is an example of such a branch for MIT.

5.3 *Camera*: A Mobile Camera Service

Camera is an application that provides a mobile camera service. When a user of *Floorplan* notices a camera icon in a particular location she can click on it to start the *Camera* client, shown in Figure 5-4. *Camera* supports two methods of access. The first way via a request-response protocol, the second is via a subscription style protocol. In the request-response protocol the client just sends a request to the server, which sends back the picture. In the subscription style protocol, the client sends a request informs the INRs on the path towards the server that it is interested in receiving pictures. The pictures which the server has been multicasting will then be sent to the client as well. If the *Camera* server changes location, it just updates the location information to indicate its new location. *Camera* clients can choose to track either the camera (following it to different locations) or the location (regardless of what camera happens to be there at the moment) by specifying one or the other as a destination of requests or subscriptions.

Figure 5-4: A screenshot from *Camera*. The display shows the latest image that has been received. "Update" will request an update of the picture from that location, which will trigger a response from the camera. "Subscribe" and "Unsubscribe" are used to start and stop sending subscription requests towards the camera; these requests inform the INRs on the path that the application wishes to receive pictures, and the multicast images the camera is sending out will be forwarded to it.

The *Camera* name space is shown in Figure 5-5. To convey location information, *Camera* uses the location name space that was described in the previous section. For

Camera-specific information, it uses the name space underneath the top-level av-pair "service is camera." (The printer application will use the name space underneath "service is printer.") There are two pieces of *Camera*-specific information. The "entity" attribute is used to identify the different participants in the *Camera* protocol. When the *Camera* server advertises its service, it uses the value "transmitter." The client can then send data towards this entity. When the client uses the request-response protocol, the "receiver" value is used to identify the client; the response will then be sent back to that particular receiver. When the client uses the subscription style protocol, the "subscriber" value is advertised by the client to the INRs on the path to the server. The multicast pictures that the server sends out this value will go to the client, as well as any other multicast subscribes. The "id" attribute is a unique ID associated with the particular entity. It is used by the *Camera* server to make sure that responses go back to only a single receiver. If the *Camera* client wants to track a camera as its location changes, it identifies the camera by its id but omits the location name space; if the client just wants a camera at that location, it uses the location name space but omits the camera id.

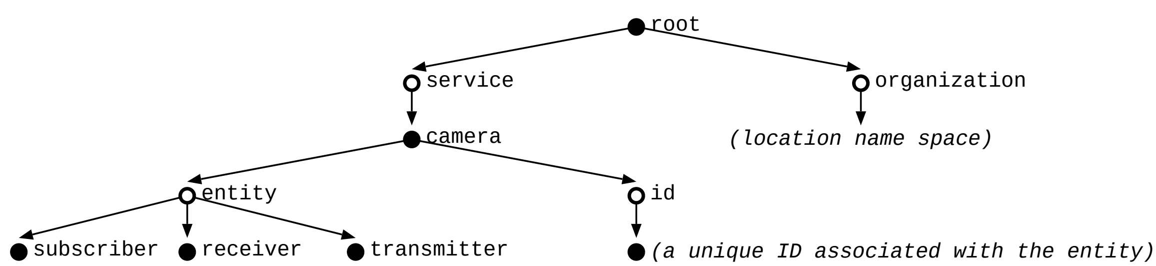

Figure 5-5: The *Camera* name space. The area under the top-level av-pair "service is camera" is where the *Camera*-specific information is. The "entity" attribute is used to identify the different participants in the *Camera* protocol: the subscriber, the receiver, and the transmitter. The "id" attribute is used to give entity a unique identifier, even if it changes location. The location name space is used to express the camera's location.

5.4 *Printer*: A Printer Service

Printer is an application that provides a printer service. When a user of *Floorplan* notices a printer icon in a particular location she can click on it to start the *Printer* client, shown in Figure 5-6. The *Printer* client has several features. It can retrieve a list of the jobs that are in the queue for the printer. It can remove a selected job from the queue, if the user has permission to remove that job. It also allows the user to submit files to the printer in two ways. The "submit job to *name*" will submit a new job to the printer that the user has selected. The "submit job to *location*" will submit a new job to any printer at the same location as the printer that the user has selected. The *Printer* servers change the metric they advertise to the INRs based on their queue length and error status, so the data will be sent to which ever printer is currently the least loaded.

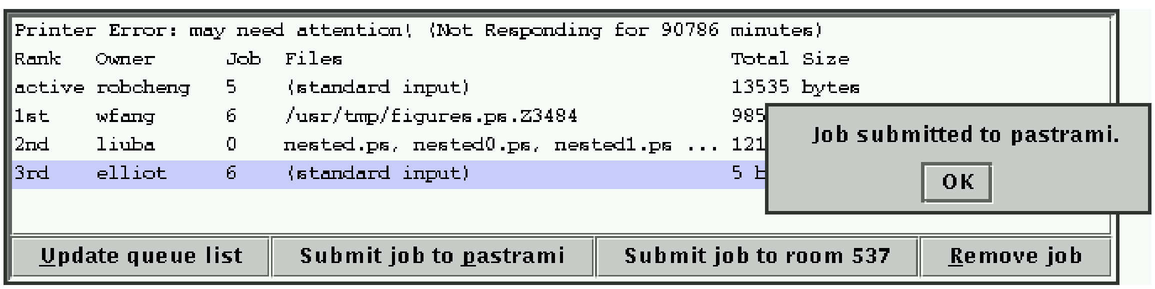

Figure 5-6: A screenshot from the printer application. The queue list displays the jobs currently in the printer queue for the named printer. "Submit job to pastrami" will submit a new job to the printer named pastrami, while "Submit job to room 537 will submit a new job to any printer whose location is room 537. The INRs will send the job to the printer that is advertising the best metric at the time.

The *Printer* name space is shown in Figure 5-7. To convey location information, *Printer* uses the location name space in the same way *Camera* does. For *Printer*-specific information, it uses the name space underneath the top-level av-pair "service is printer." As with *Camera*, there is an "entity" attribute which is used to identify the different participants. However, since *Printer* only support a request-response protocol, the only values for entity are "client" and "spooler." Rather than having an id attribute, *Printer* has a "name" attribute, which is the name of the printer that is used by the traditional print spooling system. When the client wishes to submit a job to a particular printer, it sends the data to a name-specifier that includes the entity and name attributes. When the client wishes to submit a job to any printer at that location, the name-specifier includes the entity attribute and the location name space, but not the name attribute; this allows INS to forward the data to any printer at that location. Since the printers choose their metric based on their operational status, the job will be directed to the best one.

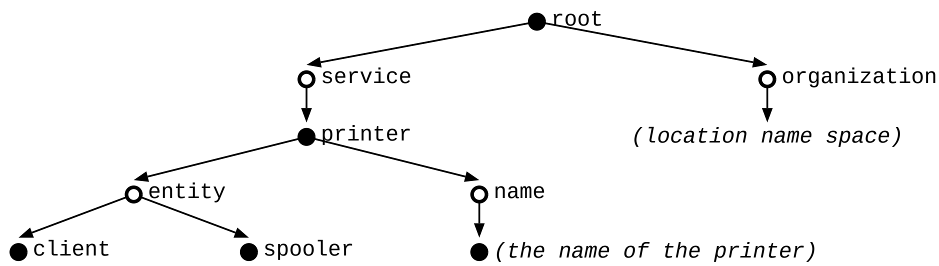

Figure 5-7: The *Printer* name space. The area under the top-level av-pair "service is printer is where the *Printer*-specific information is. The "entity" attribute is used to identify the different participants in the *Printer* protocol: the spooler and the client. The "name" attribute is the name of the printer.

5.5 Benefits of Intentional Names

Our experience with designing applications for INS has shown us that there are several benefits of intentional names which were not obvious beforehand.

Floorplan demonstrated how intentional names enable resource discovery, by allowing users to discover new services. Since the services have a very descriptive name, the name can be used to determine and select the service best suits the a user's needs.

When designing *Camera*, we noticed that it would be useful to cache pictures at various places in the network, so that requests don't have to go back to the original server every time. To this end, we have designed an application-inspecific extension to INS that allows INRs to cache data packets; this means that once a particular piece of data has passed through an INR nearby applications can have their requests served locally. Intentional names made this design fairly simple. With traditional naming schemes each application provides its own opaque names for its data units. In contrast, intentional names give applications a rich vocabulary with which to name their data, yet keep the structure of these names understandable without any application-specific knowledge. Thus, the intentional names can be used as a handle for a cached object. Of course, it is still necessary to provide additional information to describe if or how the object should be cached; INS provides additional fields in its packet header that convey this information to the INRs.

Camera also demonstrated how intentional names can be used to support mobility in a network. Both mobility of individual machines, as well as of users and services, can be supported with intentional names. If a user wants to track a particular machine as it moves across multiple subnets, she can use an intentional name that specifies a unique and persistent identifier for the machine. More commonly, users are just interested in particular services: as long as the user specifies her intent, the naming system can map the intent to whichever machine happens to be providing the service at the moment. Similarly, if a user moves from machine to machine, another user can use intentional names to maintain a persistent relationship with her (for example, a "talk" session) by referring to the her name rather than the name of the machine she was originally using.

Intentional names made providing application-level group communication services easy, since the names can be used to refer to either a single entity or many of them. In *Camera*, the servers are periodically sending out new pictures to a name-specifier that expresses the server's intent to send the picture to all "clients that wish to receive a picture from this location." The INRs deliver the picture to all clients who have informed the INR (by sending a subscription towards the server) that they wish to receive the picture. In *Printer*, the clients have the option to submit a job to any "printer at this location." The INRs submit the job to whichever printer has advertised the best metric at the time—this ability to change metric of a name enables load balancing among multiple destinations that provide the same service. INS provides a field in its header which allows the applications to select whether data is forwarded to any destination that matches a name-specifier (anycast semantics), or to all destinations that match the name-specifier (multicast semantics).

Chapter 6

Conclusions

As the Internet grows to include more services, it is increasingly necessary to be able to identify these services in a more flexible way than giving their network location. To this end, we have proposed the use of *intentional names*, which allow applications to describe "what they want" rather than "where it is." We have designed, implemented, and evaluated the Intentional Naming System (INS), a system that provides network infrastructure to resolve these intentional names to their network locations.

In this thesis have presented the design of a key component of INS: the naming scheme. We described the expressive syntax of its intentional names, and the data structures and algorithms it uses to perform the name-to-location resolution. We have analyzed these algorithms, described a prototype implementation, and presented performance results that show the feasibility of our ideas.

Based on our experience in developing demonstration applications for use with INS, we believe that the use of intentional names has significant benefits for the development of distributed applications. By using INS, our applications have access to network infrastructure that simplifies resource discovery, caching, mobility, and load balancing. Furthermore, the use of intentional names simplifies the development of the applications, making them easier and faster to develop than they would be with a traditional network programming model. We believe that as the Intentional Naming System evolves to include even greater flexibility and scalability, it will provide even more significant advantages for the developers of Internet services.

Appendix A

Name-Specifier API

name-specifier functions

createNamespecifierEmpty

name-specifier createNameSpecifierEmpty()

Creates an new, empty name-specifier. This is the default constructor.

Parameters: None.

Returns: The new, empty *name-specifier*.

createNamespecifierFromString

name-specifier createNamespecifierFromString(string s)

Creates a name-specifier from its wire representation.

Parameters: s - the wire representation.

Returns: The *name-specifier* that s represents.

createNamespecifierFromNamespecifier

name-specifier createNamespecifierFromNamespecifier(name-specifier n)

Creates a copy of a name-specifier. This is the copy constructor.

Parameters: n - the name-specifier to be copied.

Returns: A *name-specifier* that is a copy of n.

namespecifierGetRoot

av-pair namespecifierGetRoot(name-specifier n)

Returns the root av-pair of a name-specifier.

Parameters: n - the name-specifier to get the root of.

Returns: The *av-pair* that is the root of n.

av-pair functions

createAvpair

av-pair createAvpair(attribute a, value v)

Creates an av-pair from an attribute and a value.

Parameters: a - the attribute of this av-pair.
v - the value of this av-pair.

Returns: A new *av-pair* with attribute a and value v.

createAvpairFromString

av-pair createAvpairFromString(string s)

Creates an av-pair from its wire representation.

Parameters: s - the wire representation.

Returns: The *av-pair* that s represents.

avpairGetAttribute

attribute avpairGetAttribute(av-pair p)

Gets the attribute of an av-pair.

Parameters: p - the av-pair to get the attribute of.

Returns: The *attribute* of p.

avpairGetValue

value avpairGetValue(av-pair p)

Gets the value of an av-pair.

Parameters: p - the av-pair to get the value of.

Returns: The *value* of p.

createAvpairFromAvpair

av-pair createAvpairFromAvpair(av-pair p)

Creates a copy of an av-pair. This is the copy constructor.

Parameters: p - the av-pair to be copied.

Returns: An *av-pair* that is a copy of p.

avpairAddChild

void avpairAddChild(av-pair p, av-pair *child*)

Adds an av-pair as a child of another av-pair.

Parameters: p - the parent av-pair.
child - the av-pair to be added as a child.

Returns: Nothing.

avpairRemoveChild

void avpairRemoveChild(av-pair p, av-pair *child*)

Removes a child of an av-pair.

Parameters: p - the parent av-pair.
child - the child av-pair to removed.

Returns: Nothing.

avpairRemoveAllChildren

void avpairRemoveAllChildren(av-pair p)

Removes all children of an av-pair.

Parameters: p - the av-pair to remove all children of.

Returns: Nothing.

avpairIsLeaf

boolean avpairIsLeaf(av-pair p)

Tests if an av-pair is a leaf (i.e. has no children).

Parameters: p - the av-pair to test.

Returns: *true* if p is a leaf, $false$ otherwise.

avpairGetChild

av-pair avpairGetChild(av-pair p, attribute a)

Gets the av-pair child of an av-pair with a particular attribute.

Parameters: p - the av-pair to get a child av-pair of.
a - the attribute that the child av-pair should have.

Returns: The child *av-pair* with attribute a, or *null* if not found.

avpairGetNextChild

av-pair avpairGetNextChild(av-pair p, iteration-state j)

Gets the "next" av-pair child of an av-pair.
This provides an iteration abstraction.

Parameters: p - the av-pair to get the next child av-pair of.
j - the state of the iteration; determines what "next" is.

Returns: The next (according to j) av-pair child of p.

avpairsEqual

boolean avpairsEqual(av-pair $p1$, av-pair $p2$)

Tests the equality of two av-pairs.
Two av-pairs are equal if their attribute and value are equal,
and they have an identical set of child av-pairs.

Parameters: $p1$ - one of the av-pairs.
$p2$ - the other av-pair.

Returns: *true* if $p1$ and $p2$ are equal, $false$ otherwise.

avpairToString

string avpairToString(av-pair p)

Produces a human readable representation of an av-pair.

Parameters: p - the av-pair to produce a human readable representation of.

Returns: The *string* that is a human readable representation of p.

attribute functions

createAttributeFromString

attribute createAttributeFromString(string s)

Creates an attribute from its wire representation.

Parameters: s - the wire representation.

Returns: A new *attribute* that s represents.

createAttributeFromAttribute

attribute createAttributeFromAttribute(attribute a)

Creates a copy of an attribute. This is the copy constructor.

Parameters: a - the attribute to be copied.

Returns: An *attribute* that is a copy of a.

attributesEqual

boolean attributesEqual(attribute $a1$, attribute $a2$)

Tests the equality of two attributes.

Parameters: $a1$ - one of the attributes.
$a2$ - the other attribute.

Returns: *true* if $a1$ and $a2$ are equal, $false$ otherwise.

attributeToString

string attributeToString(attribute a)

Produces a human readable representation of an attribute.

Parameters: a - the attribute to produce a readable representation of.

Returns: The *string* that is a readable representation of a.

value functions

createValueFromString

value createValueFromString(string s)

Creates a value from its wire representation.

Parameters: s - the wire representation.

Returns: A new *value* that s represents.

createValueWildcard

value createValueWildcard()

Creates a value that is a wildcard.

Parameters: None.

Returns: A new *value* that is a wildcard.

createValueFromValue

value createValueFromValue(value v)

Creates a copy of a value. This is the copy constructor.

Parameters: v - the value to be copied.

Returns: A *value* that is a copy of v.

valueIsWildcard

boolean valueIsWildcard(value v)

Tests if a value is a wildcard.

Parameters: v - the value to test.

Returns: *true* if v is a wildcard, $false$ otherwise.

valuesEqual

boolean valuesEqual(value $v1$, value $v2$)

Tests the equality of two values.

Parameters: $v1$ - one of the values.
$v2$ - the other value.

Returns: *true* if $v1$ and $v2$ are equal, $false$ otherwise.

valueToString

string valueToString(value v)

Produces a human readable representation of a value.

Parameters: v - the value to produce a human readable representation of.

Returns: The *string* that is a human readable representation of v.

Appendix B

Source Code

NameSpecifier

```
import java.util.Vector;
import java.util.Enumeration;
import java.util.StringTokenizer;

/**
 * Implements a NameSpecifier.
 */
public class NameSpecifier extends AVPair
{
    // VARIABLES

    // all inherited from AVPair
    // a and v are not used

    /**
     * Creates a new, empty NameSpecifier. This is the default constructor.
     */
    public NameSpecifier()
    {
        super();
    }

    /**
     * Creates a NameSpecifier that is a copy of ns.
     * @param ns the NameSpecifier to be copied.
     */
    public NameSpecifier(NameSpecifier ns)
    {
        super();

        for (Enumeration e = ns.getAVPairs(); e.hasMoreElements(); ) {
            addAVPair(new AVPair((AVPair)e.nextElement()));
```

```
    }
}

/**
 * Creates a NameSpecifier.
 * @param s the String to create the NameSpecifier from
 * @exception CantParseString the String given was bad
 */
public NameSpecifier(String s)
    throws CantParseString
{
    super();

    StringTokenizer st;
    int level;     // bracket nesting level
    String next;  // the next token
    String child; // the current AVPair string

    st = new StringTokenizer(s, "[]", true);

    while (st.hasMoreTokens()) {

        // The child starts when we reach a "[" (and includes it).
        child = "";
        while (st.hasMoreTokens()) {
            child = st.nextToken();

            if (child.equals("["))
                break;
        }

        // We're inside the child, so we're at level 1.
        level = 1;

        while (st.hasMoreTokens()) {
            // Append the token to the child.
            next = st.nextToken();
            child = child + next;

            // Keep track of brackets to figure out when the child
            // has ended. "[" indicates another level of nesting
            // within the child; "]" indicates a level of nesting has
            // finished. When we go back down to level 0, this child
            // is done.
            if (next.equals("[")) {
                level++;
            } else if (next.equals("]")) {
                level--;
                if (level == 0)
                    break;
            }
        }

        // Create the AVPair from the child string
```

```
            // and add it to our list.
            AVPair c = new AVPair(child);
            addAVPair(c);
        }
    }

}
```

AVPair

```
import java.util.Vector;
import java.util.Enumeration;
import java.util.StringTokenizer;

/**
 * Represents an Attribute-Value tree node in a NameSpecifier.
 */
public class AVPair
{
    // VARIABLES

    private Attribute a;
    private Value v;
    private Vector children; // Vector of AVPair
                             // (children of this AVPair)

    // CONSTRUCTORS

    /**
     * Creates an AVPair. Used by NameSpecifier.
     */
    protected AVPair()
    {
        children = new Vector();
    }

    /**
     * Creates an AVPair from an Attribute and a Value.
     * @param a the Attribute in the AVPair.
     * @param v the Value in the AVPair.
     */
    public AVPair(Attribute a, Value v)
    {
        this.a = a;
        this.v = v;

        children = new Vector();
    }

    /**
     * Creates an AVPair from its wire representation..
     * @param s the String to create the AVPair from.
```

```
 * @exception CantParseString the String given was bad.
 */
public AVPair(String s)
    throws CantParseString
{
    StringTokenizer st;
    int level;     // bracket nesting level
    String next;  // the next token
    String child; // the currnet AVPair string

    children = new Vector();

    st = new StringTokenizer(s, "[=]", true);

    // We discard stuff until we get to the first "["
    while (st.hasMoreTokens()) {
        next = st.nextToken();

        if (next.equals("["))
            break;
    }

    // The next token is the Attribute

    if (! st.hasMoreTokens()) {
        throw new CantParseString("Expected an Attribute, " +
                                  "but ran out of tokens in " + s);
    }

    next = st.nextToken();

    if (next.equals("[") || next.equals("=") || next.equals("]")) {
        throw new CantParseString("Expected an Attribute, not a " +
                                  next + " in " + s);
    }

    a = new Attribute(next);

    // The next token is the =

    if (! st.hasMoreTokens()) {
        throw new CantParseString("Expected an =, " +
                                  "but ran out of tokens in " + s);
    }

    next = st.nextToken();

    if (! next.equals("=")) {
        throw new CantParseString("Expected an =, not a " +
                                  next + " in " + s);
    }

    // The next token is the Value
```

```
    if (! st.hasMoreTokens()) {
        throw new CantParseString("Expected a Value, " +
                                  "but ran out of tokens in " + s);
    }

    next = st.nextToken();

    if (next.equals("[") || next.equals("=") || next.equals("]")) {
        throw new CantParseString("Expected a Value, not a " +
                                  next + " in " + s);
    }

    v = new Value(next);

    // Now we deal with the children of this AVPair.
    // We start off at bracket level 1, add one every time we see a "[",
    // and subtract one every time we see a "]".
    // If we get back to 1, we've finished a child.
    // If we get back to 0, we've finished this AVPair.

    level = 1;
    child = "";
    while (st.hasMoreTokens()) {
        // Append the token to the child.
        next = st.nextToken();
        child = child + next;

        if (next.equals("[")) {
            level++;
        } else if (next.equals("]")) {
            level--;
            if (level == 1) {
                // Done the child. Create it, add it, and clear the string.

                AVPair c = new AVPair(child);
                children.addElement((Object)c);
                child = "";
            } else if (level == 0) {
                // Done the whole AVPair.

                break;
            }
        }
    }
}

/**
 * Creates an AVPair that is a copy of ave.
 * @param ave the AVPair to copy.
 */
public AVPair(AVPair ave)
{
    a = new Attribute(ave.a);
    v = new Value(ave.v);
```

```
    children = new Vector();

    for (Enumeration e = ave.children.elements(); e.hasMoreElements(); ) {
        children.addElement(new AVPair((AVPair)e.nextElement()));
    }

}

// METHODS

/**
 * Add AVPair c as a child of this AVPair.
 * @param c the AVPair to be added as a child
 */
public void addAVPair(AVPair c)
{
    children.addElement(c);
}

/**
 * Remove AVPair c as a child of this AVPair.
 * @param c the AVPair to be removed as a child
 */
public void removeAVPair(AVPair c)
{
    children.removeElement(c);
}

/**
 * Remove all AVPairs (children) of this AVPair.
 */
public void removeAllAVPairs()
{
    children.removeAllElements();
}

/**
 * Checks if an AVPair is a leaf (i.e. has no children).
 * @return true if this AVPair is a leaf node, false otherwise.
 */
public boolean isLeaf()
{
    return(children.isEmpty());
}

/**
 * Gets the Attribute of this AVPair.
 * @return the Attribute of this AVPair.
 */
public Attribute getAttribute()
{
    return(a);
}
```

```
/**
 * Gets the Value of this AVPair.
 * @return the Value of this AVPair.
 */
public Value getValue()
{
    return(v);
}

/**
 * Returns an Enumeration of the AVPair children of this AVPair.
 */
public Enumeration getAVPairs()
{
    return(children.elements());
}

/**
 * Gets the AVPair child of this AVPair with a particular Attribute.
 * @param a the Attribute of the child AVPair to be returned.
 * @return the child AVPair with Attribute a, or null if not found.
 */
public AVPair getAVPair(Attribute a)
{

    for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
        AVPair c = (AVPair)e.nextElement();

        if (c.a.equals(a)) {
            return(c);
        }
    }

    return(null);
}

/**
 * Tests two AVPairs for equality.
 * @param obj the object to test equality with.
 * @return true if they are equal, false otherwise.
 */
public boolean equals(Object obj)
{
    if (!(obj instanceof AVPair)) return false;
    AVPair ave = (AVPair)obj;

    // First part of test is because the could both be null,
    // as they are in a NameSpecifier

    if (! (a == null && ave.a == null)) {
        if (a == null || ave.a == null) {
            return(false);
        }
```

```
            if (! a.equals(ave.a)) {
                return(false);
            }
        }

        if (! (v == null && ave.v == null)) {
            if (v == null || ave.v == null) {
                return(false);
            }

            if (! v.equals(ave.v)) {
                return(false);
            }
        }

        for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
            AVPair c1 = (AVPair)e.nextElement();

            // Not particularly efficient - may want to optimize for the
            // common case that both AVPairs have their children in
            // the same order.
            AVPair c2 = ave.getAVPair(c1.a);

            if (c2 == null || ! c1.equals(c2)) {
                return(false);
            }
        }

        return(true);
    }

    /**
     * Returns a String representation of the AVPair.
     * @return a String representation of the AVPair.
     */
    public String toString()
    {
        String output = "";

        if (a != null && v != null)
          output = output + "[" + a + "=" + v;

        for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
            output = output + (AVPair)e.nextElement();
        }

        if (a != null && v != null)
          output = output + "]";

        return(output);
    }

}
```

Attribute

```java
/**
 * Represents an Attribute.
 */
public class Attribute
{
    // VARIABLES

    private String attribute;

    // CONSTRUCTORS

    /**
     * Creates an Attribute from string s.
     * @param s the wire representation of the Attribute.
     */
    public Attribute(String s)
    {
        attribute = s.trim();
    }

    /**
     * Creates an Attribute that's a copy of a.
     * @param a the Attribute to be copied.
     */
    public Attribute(Attribute a)
    {
        attribute = a.attribute;
    }

    // METHODS

    /**
     * Returns a String representation of the Attribute.
     * @return a human readable String that represents this Attribute.
     */
    public String toString()
    {
        return(attribute);
    }

    /**
     * Tests if an Attribute is equal to this Attribute.
     * @param a the Attribute to be tested.
     * @return true if a equals this Attribute, false otherwise.
     */
    public boolean equals(Attribute a)
    {
        return(attribute.equalsIgnoreCase(a.attribute));
    }
```

```
    /**
     * Tests if two Attributes are equal.
     * @param a1 one of the Attributes.
     * @param a2 the other Attribute.
     * @return true if a1 and a2 are equal, false otherwise.
     */
    public static boolean equal(Attribute a1, Attribute a2)
    {
        return(a1.equals(a2));
    }

}
```

Value

```
/**
 * Represents a Value.
 */
public class Value
{
  // VARIABLES

  private String value;
  private boolean wildcard;

  // Representation Invariant:
  // wildcard == true -> String == null
  // wildcard != true -> String != null

  // CONSTRUCTORS

  /**
   * Creates a Valuer from string s.
   * @param s the wire representation of the Value.
   */
  public Value(String s)
  {
    // Remove any extra spaces around the string,
    // since they're not part of the value.
    value = s.trim();

    if (value.equals("*")) {
      // wildcard
      value = null;
      wildcard = true;
    } else {
      wildcard = false;
    }
  }

  /**
```

```
 * Creates a Value that's a wildcard.
 */
public Value()
{
  value = null;
  wildcard = true;
}

  /**
   * Creates a Value that's a copy of v.
   * @param v the Value to be copied.
   */
  public Value(Value v)
  {
      value = v.value;
      wildcard = v.wildcard;
  }

// METHODS

/**
 * Returns a String representation of the Value.
 * @return a human readable String that represents this Value.
 */
public String toString()
{
  if (wildcard)
    return("*");
  else
    return(value);
}

/**
 * Tests if a Value is equal to this Value.
 * @param v the Value to be tested.
 * @return true if v equals this Value, false otherwise.
 */
public boolean equals(Value v)
{
  // Values are case-insensitive.
  if (wildcard || v.wildcard)
    return(true);

  return(value.equalsIgnoreCase(v.value));
}

/**
 * Tests if two Values are equal.
 * @param v1 one of the Values.
 * @param v2 the other Value.
 * @return true if v1 and v2 are equal, false otherwise.
 */
public static boolean equal(Value v1, Value v2)
{
```

```
    return(v1.equals(v2));
  }

  /**
   * Tests if a Value is a wildcard.
   * @return true if the Value is a wildcard, false otherwise.
   */
  public boolean isWildcard()
  {
    return(wildcard);
  }

}
```

NameTree

```
import java.util.Vector;
import java.util.Enumeration;

/**
 * This class implements a tree that stores routing information.
 */
public class NameTree extends ValueNode
{
    // VARIABLES

    // v is not used.

    private Vector nameRecords; // Vector of NameRecord
    // (contains all NameRecord objects)
    // used to do an iteration over all of the nameRecords
    // but whoops, it looked like I deleted the storage of backpointer
    // information in the NameRecord because I forgot why I put it in <sigh>

    // CONSTRUCTORS

    /**
     * Creates an empty NameTree.
     */
    public NameTree()
    {
        super(null);
        nameRecords = new Vector();
    }

    // METHODS

    /**
     * Lookup NameSpecifier s in the NameTree, and return its NameRecordSet.
     */
    public synchronized NameRecord[] lookup(NameSpecifier s)
    {
```

```
    NameRecordSet rs = super.lookup((AVPair)s);

    return(rs.toArray());
}

/**
 * Adds NameRecord r for NameSpecifier s to this NameTree.
 * Overrides addNameRecord in ValueNode.
 */
public synchronized void addNameRecord(NameSpecifier s, NameRecord r)
{
    // Add r to the list of all Route Entries.
    nameRecords.addElement(r);

    // Now recursively add the Route Entry to the tree.
    super.addNameRecord((AVPair)s, r);
}

/**
 * Extracts and returns the NameSpecifier associated with the NameRecord r.
 */
public synchronized NameSpecifier getName(NameRecord r)
{
    NameSpecifier n = new NameSpecifier();
    Vector touched = new Vector(); // Vector of ValueNode

    // Precondition: all PTR's in the NameTree are null

    // Set the root to the new NameSpecifier,
    // and add it to the touched list
    PTR = n;
    touched.addElement((ValueNode)this);

    // For each parent of the NameRecord
    for (Enumeration e = r.getParents(); e.hasMoreElements(); ) {
        ValueNode v = (ValueNode)e.nextElement();

        // Trace its way up to an AVE we've filled in already.
        v.trace((AVPair)null, touched);
    }

    // Clear all the pointers we've touched
    for (Enumeration e = touched.elements(); e.hasMoreElements(); ) {
        ValueNode v = (ValueNode)e.nextElement();

        v.PTR = null;
    }

    return(n);
}

/**
 * Returns an Enumeration of the nameRecords in this NameTree.
 */
```

```java
    public synchronized Enumeration getnameRecords()
    {
        return(nameRecords.elements());
    }

    /**
     * Returns an String that is a printed version of the nameRecords.
     */
    public synchronized String nameRecordsToString()
    {
        String output = "";

        for (Enumeration e = nameRecords.elements(); e.hasMoreElements(); ) {
            NameRecord re = (NameRecord)e.nextElement();
            NameSpecifier ns = getName(re);
            output = output +
                "For NameRecord:\n  " + re + "\n" +
                "The NameSpecifier is:\n  " + ns + "\n";
        }

        return(output);
    }

    /**
     * Return a String representation of this NameTree that is pretty.
     * Overrides toPrettyString in ValueNode.
     */
    public synchronized String toPrettyString()
    {
        String output = "";

        output = output + super.toPrettyString(0);

        return(output);
    }

    /**
     * Returns a String representation of this NameTree.
     * Overrides toString in ValueNode.
     */
    public synchronized String toString()
    {
        String output = "";

        output = output + super.toString();

        return(output);
    }

}
```

AttributeNode

```
import java.util.Vector;
import java.util.Enumeration;

/**
 * Represents a AttributeNode in a NameTree.
 */
class AttributeNode
{
    // VARIABLES

    Attribute a;
    private Vector children;      // Vector of ValueNode
                                  // (contains possible values)
    private ValueNode parent;

    // CONSTRUCTORS

    /**
     * Creates a AttributeNode with a as its Attribute.
     */
    AttributeNode(Attribute a)
    {
        this.a = a;
        children = new Vector();
        parent = null;
    }

    // METHODS

    /**
     * Sets the parent of this AttributeNode to be the ValueNode v.
     */
    void setParent(ValueNode v)
    {
      parent = v;
    }

    /**
     * Returns the parent of this AttributeNode.
     */
    ValueNode getParent()
    {
      return(parent);
    }

    /**
     * Add ValueNode v as a child of this AttributeNode.
     */
    void addValueNode(ValueNode v)
    {
        v.setParent(this);
```

```
    children.addElement((Object)v);
}

/**
 * Return a String representation of this sub-NameTree that is pretty,
 * and is indented to indentLevel.
 */
String toPrettyString(int indentLevel)
{
    String output = "";
    ValueNode v;

    for (int i = 0; i < indentLevel; i++)
        output = output + "  ";
    output = output + a + "\n";

    for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
        v = (ValueNode)e.nextElement();
        output = output + v.toPrettyString(indentLevel+1);
    }

    return(output);
}

/**
 * Returns a String representation of the AttributeNode.
 */
public String toString()
{
    String output;
    ValueNode v;

    output = "(" + a;

    for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
        v = (ValueNode)e.nextElement();
        output = output + " " + v.toString();
    }

    output = output + ")";

    return(output);
}

/**
 * Returns the ValueNode with Value = v1, or throws an exception
 * if not found.
 */
ValueNode findValueNode(Value v1)
    throws ElementNotFound
{
    for (Enumeration e = children.elements(); e.hasMoreElements();) {
        ValueNode ve = (ValueNode)e.nextElement();
        Value v2 = ve.v;
```

```
            if (v1.equals(v2)) {
                return(ve);
            }
        }

        throw(new ElementNotFound());
    }

    /**
     * Returns an Enumeration of the ValueNodes.
     */
    Enumeration getValueNodes()
    {
        return(children.elements());
    }

}
```

ValueNode

```
import java.util.Vector;
import java.util.Enumeration;

/**
 * Represents a Value element in a NameTree.
 */
class ValueNode
{
    // VARIABLES

    Value v;
    private Vector children;          // Vector of AttributeNode
                                      // (contains orthogonal values)
    private AttributeNode parent;
    private NameRecordSet routeSet;

    AVPair PTR;                       // Temporary data storage
                                      // used by an invocation of
                                      // NameRecord.trace()
                                      // Access to this field in the
                                      // entire route tree should be
                                      // serialized.
                                      //
                                      // Only not "private" because NameTree
                                      // needs to access it.

    // CONSTRUCTORS

    /**
     * Creates a new ValueNode.
     * @param v the Value of the ValueNode
```

```
 */
ValueNode(Value v)
{
    this.v = v;
    children = new Vector();
    parent = null;
    routeSet = new NameRecordSet();
    PTR = null;
}

// METHODS

/**
 * Sets the parent of this ValueNode.
 * @param a the AttributeNode to set the parent to
 */
void setParent(AttributeNode a)
{
    parent = a;
}

/**
 * Add a child AttributeNode to this ValueNode.
 * @param a the AttributeNode to be added
 */
void addAttributeNode(AttributeNode a)
{
    a.setParent(this);
    children.addElement((Object)a);
}

/**
 * Adds a NameRecord to the NameRecordSet of this ValueNode.
 * @param r the NameRecord to be added
 */
private void addNameRecordHere(NameRecord r)
{
    r.addParent(this);
    routeSet.addNameRecord(r);
}

/**
 * Remove a NameRecord from the NameRecordSet of this ValueNode.
 * @param r the NameRecord to be removed
 */
void removeNameRecordHere(NameRecord r)
{
    routeSet.removeNameRecord(r);
}

/**
 * Returns true if this ValueNode is a leaf node.
 */
boolean isLeaf()
```

```
{
    return(children.isEmpty());
}

/**
 * Looks up (part of) a NameSpecifier.
 * @param n the part of the NameSpecifier (AVPair) to be looked up
 */
NameRecordSet lookup(AVPair n)
{
    // This code is the same as the code in NameTree.java

    NameRecordSet S;  // S is the NameRecordSet we're eventually going to return.

    // Start with S = the set of all possible route entries
    S = new NameRecordSet();
    S.addAllRouteEntries();

    // for each attribute-value pair p = (na,nv) in n
    for (Enumeration e1 = n.getAVPairs(); e1.hasMoreElements(); ) {
        AVPair p   = (AVPair)e1.nextElement();
        Attribute na = p.getAttribute();
        Value     nv = p.getValue();

        // Ta = the child of T (this) such that name(Ta) = name(na)
        AttributeNode Ta;
        try {
            Ta = findAttributeNode(na);
        } catch (ElementNotFound ex) {
            continue;
        }

        ValueNode Tv;
        // if nv == *
        if (nv.isWildcard()) {
            // Wildcard matching.

            NameRecordSet Sprime = new NameRecordSet();

            for (Enumeration e2 = Ta.getValueNodes();
                 e2.hasMoreElements();
                 ) {
                Tv = (ValueNode)e2.nextElement();
                Sprime.unionWith(Tv.routeSet);
            }

            S.intersectWith(Sprime);
        } else {
            // Normal matching.

            try {
                Tv = Ta.findValueNode(nv);
            } catch (ElementNotFound ex) {
                return(new NameRecordSet());
```

```
            }

            if (Tv.isLeaf() || p.isLeaf()) {
                S.intersectWith(Tv.routeSet);
            } else {
                S.intersectWith(Tv.lookup(p));
            }
        }
    }

    S.unionWith(routeSet);

    return(S);
}

/**
 * Recursively add a NameRecord to this substree of the NameTree.
 * @param r the NameRecord to add
 * @param n the part of the NameSpecifier to work on from here (this)
 */
void addNameRecord(AVPair n, NameRecord r)
{

    // Figure out where to put it in the tree.
    for (Enumeration e = n.getAVPairs(); e.hasMoreElements(); ) {
        AVPair p  = (AVPair)e.nextElement();
        Attribute na = p.getAttribute();
        Value     nv = p.getValue();

        AttributeNode Ta;
        try {
            Ta = findAttributeNode(na);
        } catch (ElementNotFound ex) {
            // Didn't find the AttributeNode; create it.

            Ta = new AttributeNode(na);
            addAttributeNode(Ta);
        }

        ValueNode Tv;
        try {
            Tv = Ta.findValueNode(nv);
        } catch (ElementNotFound ex) {
            // Didn't find the ValueNode; create it.

            Tv = new ValueNode(nv);
            Ta.addValueNode(Tv);
        }

        // If this is the bottom of this part of the name specifier,
        // add the route. Otherwise, to a recursive call (sort of).
        if (p.isLeaf()) {
            Tv.addNameRecordHere(r);
        } else {
```

```
            Tv.addNameRecord(p,r);
        }

    }
}

/**
 * Finds an AttributeNode with a particular Attribute.
 * @param a1 the Attribute that is being looked for
 * @return the AttributeNode that matches
 * @exception ElementNotFound
 *  No AttributeNode with that Attribute was found.
 */
private AttributeNode findAttributeNode(Attribute a1)
    throws ElementNotFound
{
    for (Enumeration e = children.elements(); e.hasMoreElements();) {
        AttributeNode ae = (AttributeNode)e.nextElement();
        Attribute a2 = ae.a;

        if (a1.equals(a2)) {
            return(ae);
        }
    }

    throw(new ElementNotFound());
}

/**
 * Fills in the current PTR entry, if necessary, and then recursively
 * calls itself on its parent.
 */
void trace(AVPair s, Vector touched)
{
    if (PTR != null) {
        // This has already been done, just add the subtree s on.

        if (s != null)
            PTR.addAVPair(s); // graft

    } else {
        // This hasn't been done yet.

        // Create new AVPair, and mark the PTR as touched.
        PTR = new AVPair(parent.a, v);
        touched.addElement(this);

        // Add the subtree on.
        if (s != null)
            PTR.addAVPair(s); // graft

        // Now try to create the parent, with the AVPair we
        // just created as the subtree.
        parent.getParent().trace(PTR, touched);
```

```
        }
    }

    /**
     * Return a String representation of this sub-NameTree that is pretty,
     * and is indented to indentLevel.
     */
    public String toPrettyString(int indentLevel)
    {
        String output = "";
        AttributeNode a;

        for (int i = 0; i < indentLevel; i++)
            output = output + "  ";
        output = output + v;

        String rs = routeSet.toString();
        int spaces = 79 - output.length() - rs.length();

        for (int i = 0; i < spaces; i++)
            output = output + " ";
        output = output + rs + "\n";

        for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
            a = (AttributeNode)e.nextElement();
            output = output + a.toPrettyString(indentLevel+1);
        }

        return(output);
    }

    /**
     * Returns a String representation of the ValueNode.
     */
    public String toString()
    {
        String output;
        AttributeNode a;

        output = "(" + v;

        for (Enumeration e = children.elements(); e.hasMoreElements(); ) {
            a = (AttributeNode)e.nextElement();
            output = output + " " + a.toString();
        }

        output = output + ")";

        return(output);
    }

}
```

NameRecordSet

```
import java.util.Vector;
import java.util.Enumeration;
import java.lang.ArrayIndexOutOfBoundsException;

/**
 * This class represents a set of NameRecord objects.
 */
public class NameRecordSet
{
    // VARIABLES

    // if true, NameRecordSet contains all NameRecord objs
    private boolean allNameRecords;

    private Vector nameRecords;     // Vector contains type NameRecord

    // Representation Invariant:
    // if (allNameRecords == true) nameRecords.isEmpty() = true;
    // nameRecords is sorted in ascending order of NameRecord.id

    // CONSTRUCTORS

    /**
     * Creates an empty NameRecordSet.
     */
    NameRecordSet()
    {
        allNameRecords = false;
        nameRecords = new Vector();
    }

    // METHODS

    /**
     * Adds all the NameRecord objects to the NameRecordSet.
     */
    void addallNameRecords()
    {
        allNameRecords = true;
        nameRecords.removeAllElements();
    }

    /**
     * Adds the NameRecord r to the NameRecordSet.
     */
    void addNameRecord(NameRecord r)
    {
        // If we're storing allNameRecords, it's already in here.
        if (allNameRecords) {
            return;
        }
```

```
    // If nameRecords is empty, we can just add it.
    if (nameRecords.isEmpty()) {
        nameRecords.addElement(r);
        return;
    }

    // We've got at least one entry.
    for (int i=0; i < nameRecords.size(); i++) {
        NameRecord r2 = (NameRecord)nameRecords.elementAt(i);

        // It's already in there.
        if (r2.id == r.id) {
            return;
        }

        // This is the spot.
        if (r2.id > r.id) {
            nameRecords.insertElementAt(r, i);
            return;
        }
    }

    // We didn't find a spot for it, so we insert it at the end.
    nameRecords.addElement(r);
}

/**
 * Removes the NameRecord r from the NameRecordSet.
 */
void removeNameRecord(NameRecord r)
{
    // Can't really remove a single route from allNameRecords.
    if (allNameRecords) {
        return;
    }

    nameRecords.removeElement(r);
}

/**
 * Intersects the NameRecordSet with NameRecordSet s.
 * (r = r intersect s)
 */
void intersectWith(NameRecordSet s)
{
    int i, j;         // index in this, s
    NameRecord m, n; // NameRecord at i, j

    // If s contains allNameRecords, intersection with it is same.
    if (s.allNameRecords) {
        return;
    }
```

```
// If this contains allNameRecords, copy routes from s.
if (allNameRecords) {
    allNameRecords = false;
    for (j=0; j < s.nameRecords.size(); j++) {
        nameRecords.addElement(s.nameRecords.elementAt(j));
    }

    return;
}

// Neither contains allNameRecords; see which ones in this are in s.

// Start with the first element of each Vector.
i = 0;
j = 0;
try {
    m = (NameRecord)nameRecords.elementAt(i);
    n = (NameRecord)s.nameRecords.elementAt(j);

    // We'll eventually break out of this infinite loop with an
    // exception, when we get to the end of one of the Vectors,
    // since each option of the if statement makes progress towards
    // the end.
    while (true) {
        if (n.id == m.id) {
            // The id is the same; we keep the element and move on
            // in both Vectors.

            i++;
            m = (NameRecord)nameRecords.elementAt(i);
            j++;
            n = (NameRecord)s.nameRecords.elementAt(j);
        } else if (n.id < m.id) {
            // j is behind; help it catch up by increasing it.

            j++;
            n = (NameRecord)s.nameRecords.elementAt(j);
        } else { // (n.id > m.id)
            // i is behind; help it catch up by removing the
            // current element, which slides everything else back.

            nameRecords.removeElementAt(i);
            m = (NameRecord)nameRecords.elementAt(i);
        }
    }
} catch (ArrayIndexOutOfBoundsException e) {
    // If we've come to the end of i, then we can stop.
    // If we've come to the end of j, we delete everything else in i,
    // and then we can stop.

    if (j == s.nameRecords.size())
        nameRecords.setSize(i);

    // falls through to return
```

```
    }
}

/**
 * Unions the NameRecordSet with NameRecordSet s.
 * (r = r union s)
 */
void unionWith(NameRecordSet s)
{
    int i;                  // index in this
    NameRecord m;           // NameRecord in this at i
    NameRecord n = null; // Route entry in s; null to placate compiler

    // If this contains allNameRecords, it still does.
    if (allNameRecords)
        return;

    // If s contains allNameRecords, now this does too.
    if (s.allNameRecords) {
        allNameRecords = true;
        nameRecords.removeAllElements();
        return;
    }

    // Neither contains allNameRecords; add ones from s to this.

    Enumeration e = s.getnameRecords();
    try {
        // If this ends before s, we'll get an exception and then we can
        // add everything from s onto the end of this.

        // Start with the first element of this
        i = 0;

        // Add the elements of s to the right spot in this
        while (e.hasMoreElements()) {
            n = (NameRecord)e.nextElement();

            // Try to fetch the NameRecord for this
            m = (NameRecord)nameRecords.elementAt(i);

            // Keep on going through nameRecords until at the right spot
            while (m.id <= n.id) {
                i++;
                m = (NameRecord)nameRecords.elementAt(i);
            }

            // Stick it here, and advance i to the element it was at before
            // the insert.
            nameRecords.insertElementAt(n,i);
            i++;
        }
    } catch (ArrayIndexOutOfBoundsException ex) {
        // We've come to the end of this, add everything from s.
```

```
        nameRecords.addElement(n);
        while (e.hasMoreElements()) {
            n = (NameRecord)e.nextElement();
            nameRecords.addElement(n);
        }

    }
}

/**
 * Return true if the NameRecordSet is empty.
 */
public boolean isEmpty()
{
    return((! allNameRecords) && nameRecords.isEmpty());
}

/**
 * Return an Enumeration over the NameRecord objects in the set.
 */
public Enumeration getnameRecords()
{
    return nameRecords.elements();
}

public NameRecord[] toArray()
{
    NameRecord[] a = new NameRecord[nameRecords.size()];

    nameRecords.copyInto(a);

    return(a);
}

/**
 * Returns a String representation of this NameRecordSet.
 */
public String toString()
{
    String output;

    if (allNameRecords) {
        return("{ * }");
    }

    output = "{";

    for (Enumeration e = nameRecords.elements(); e.hasMoreElements(); ) {
        output = output + ((NameRecord)e.nextElement()).id;
        if (e.hasMoreElements()) {
            output = output + ", ";
        }
    }
```

```
        output = output + "}";

        return(output);
    }

}
```

NameRecord

```
import java.util.Vector;
import java.util.Enumeration;

/**
 * This represents a NameRecord in the NameTree.
 */
public class NameRecord
{

    // VARIABLES

    // the id to be assigned to the next NameRecord
    private static int next_id = 0;

    final int id;                    // the id of this NameRecord

    private Vector parents;          // Vector of ValueNode
    // (contains parents in the routing tree)
    // used for determining the NameSpecifier
    // of a NameRecord

    //////////////////////////////////////////////////////
    // Non-NameTree related instance variables omitted. //
    //////////////////////////////////////////////////////

    // CONSTRUCTORS

    /**
     * Creates a new NameRecord.
     */
    public NameRecord(...)
    {
        // Access to next_id needs to be serialized
        id = next_id++;

        parents = new Vector();
    }

    // METHODS

    /**
     * Adds ValueNode p as a parent of this NameRecord.
```

```
    */
    void addParent(ValueNode p)
    {
        parents.addElement(p);
    }

    /**
     * Returns an Enumeration over the parents of this NameRecord.
     */
    Enumeration getParents()
    {
        return(parents.elements());
    }

    /**
     * Returns a String representation of the NameRecord.
     */
    public String toString()
    {
        String output = "[" + id + "]";

        return(output);
    }

    ///////////////////////////////////////////
    // Non-NameTree related methods omitted. //
    ///////////////////////////////////////////

}
```

TestNameTreePerformance

```
import java.lang.Math;
import java.lang.Runtime;

/**
 * This program tests the performance of routing.
 * @see NameTree
 * @see NameSpecifier
 * @author Elliot Schwartz <elliot@mit.edu>
 */
public class TestNameTreePerformance
{

    /**
     * Test routing performance.
     */
    public static void main(String[] args)
    {
        Parameter[] p = new Parameter[6];

        //                          low  high   inc   def
```

```
        p[0] = new Parameter( "d", 0,     4,    0,    3);
        p[1] = new Parameter("ra", 2,    10,    0,    3);
        p[2] = new Parameter("rv", 1,    10,    0,    3);
        p[3] = new Parameter("na", 1,     3,    0,    2);
        p[4] = new Parameter( "n", 100, 15000, 100,  50);
        p[5] = new Parameter( "l", 1000, 1000,  0, 1000); // No inc of l.

        runPerfTests(p);
    }

    /**
     * Runs many performance tests.
     */
    private static void runPerfTests(Parameter[] p)
    {
        boolean repeat_param;

        // Repeat the first run of this parameter to avoid caching problems.
        repeat_param = false;

        // For each parameter
        for (int i=0; i<6; i++) {

            // Skip over this parameter if there's no increment.
            if (p[i].i == 0)
                continue;

            // Set all parameters to the default
            System.out.print("# Constants:");
            int[] params = new int[6];
            for (int j=0; j<6; j++) {
                params[j] = p[j].d;
                if (j != i) {
                    System.out.print(" " + p[j].n + "=" + p[j].d);
                }
            }
            System.out.println("\n" + "# Variables: "+p[i].n+" lps");

            // Range through the chosen parameter
            for (int k=p[i].l; k<=p[i].h; k+=p[i].i) {
                params[i] = k;
                int range_repeat = 1;
                for (int rr=0; rr < range_repeat; rr++) {
                    Result result
                        = perfTest(params[0],  // d
                                   params[1],  // ra
                                   params[2],  // rv
                                   params[3],  // na
                                   params[4],  // n
                                   params[5]); // l
                    System.gc();

                    System.out.println(k + " " +
                                   (float)params[5]*1000.0/
```

```
                                (float)result.time + " " +
                                result.memory);
            }
        }

        System.out.print("\nNEXT SET\n\n");

        // Repeat if first run of this parameter.
        if (repeat_param) {
            i--;
            repeat_param = false;
        }
    }
}

/**
 * Runs the performance test.
 * @param d the depth of the NameTree
 * @param ra the number of possible different attributes in the NameTree
 * @param rv the number of possible different values in the NameTree
 * @param na the number of attributes for each Route/NameSpecifier
 * @param n the number of routes in the NameTree
 * @param l the number of lookups to do
 * @return the Result of the test
 */
private static Result perfTest(int d, int ra, int rv, int na, int n, int l)
{
    NameTree rt;        // NameTree to run tests on.
    NameSpecifier[] ns; // The NameSpecifiers that are in rt.
    NameRecord[] re;    // The RouteEntries that are in rt.
    int[] rand;         // Random numbers.
    long time;          // For computing the elapsed time.

    // Preliminary sanity checks.

    if (na > ra) {
        System.out.println("na must be <= ra!");
        return new Result();
    }

    // Create the new NameTree.
    rt = new NameTree();

    // Add RouteEntries with random NameSpecifiers to it. Keep a list of
    // the NameSpecifiers and Route Entries.
    ns = new NameSpecifier[n];
    re = new NameRecord[n];

    // NameTree creation.
    // time = System.currentTimeMillis();

    for (int i=0; i<n; i++) {

        NameSpecifier s = createNameSpecifier(d, ra, rv, na);
```

```
        ns[i] = s;

        NameRecord r = new NameRecord(...);
        re[i] = r;

        rt.addNameRecord(s, r);
    }

    // NameTree lookups.

    // Pregenerate random numbers
    rand = new int[l];
    for (int i=0; i<l; i++) {
        rand[i] = random(n);
    }

    NameRecord[] lres;

    System.gc();
    time = System.currentTimeMillis();
    for (int i=0; i<l; i++) {
        lres = rt.lookup(ns[rand[i]]);
    }
    time = System.currentTimeMillis() - time;
    // System.out.println("Looked up "+l+" Routes in "+time+" ms.");

    Runtime runtime = Runtime.getRuntime();
    long total, free;
    ns = null;
    re = null;
    rand = null;
    System.gc();
    total = runtime.totalMemory();
    free = runtime.freeMemory();

    Result result = new Result();
    result.time = time;
    result.memory = total-free;
    return(result);
}

/**
 * Creates a random NameSpecifier.
 * @param d the depth of the NameSpecifier
 * @param ra the number of possible different attributes
 * @param rv the number of possible different values
 * @param na the actual number of attributes used at each level
 * @return the random NameSpecifier.
 */
private static NameSpecifier createNameSpecifier(int d, int ra, int rv,
                                                 int na)
{
    // Create the new NameSpecifier
    NameSpecifier ns = new NameSpecifier();
```

```
    // Check if the requested depth is zero.
    if (d == 0) {
        return(ns);
    }

    // If not, create the first level of AVPairs.

    // Create an array to make sure we don't reuse attribute numbers.
    boolean[] used = new boolean[ra];

    // Create na AVPairs.
    for (int i=0; i < na; i++) {

        // Pick a random number between 0 and ra-1,
        // and keep picking one if it's already used.
        int a;
        do {
            a = random(ra);
        } while (used[a]);

        // Mark it as being used.
        used[a] = true;

        // Pick a value to use; it's alright to reuse values.
        int v;
        v = random(rv);

        // Create the AVPair by calling createAVPair to
        // recursively fill in the rest of the NameSpecifier.
        AVPair ave = createAVPair(new Attribute(Integer.toString(a)),
                                  new Value(Integer.toString(v)),
                                  d - 1, ra, rv, na);

        // Add the AVPair to the NameSpecifier.
        ns.addAVPair(ave);
    }

    return(ns);
}

/**
 * Creates a random AVPair.
 * @param d the depth of the NameSpecifier from this AVPair
 * @param ra the number of possible different attributes
 * @param rv the number of possible different values
 * @param na the actual number of attributes used at each level
 * @return the random AVPair.
 */
private static AVPair createAVPair(Attribute attribute, Value value,
                                   int d, int ra, int rv, int na)
{
    // Create the new AVPair
    AVPair avElement = new AVPair(attribute, value);
```

```
    // Check if we're at the end of the recursion.
    if (d == 0) {
        return(avElement);
    }

    // If not, create the next level of AVPairs.

    // Create an array to make sure we don't reuse attribute numbers.
    boolean[] used = new boolean[ra];

    // Create na AVPairs.
    for (int i=0; i < na; i++) {

        // Pick a random number between 0 and ra-1,
        // and keep picking one if it's already used.
        int a;
        do {
            a = random(ra);
        } while (used[a]);

        // Mark it as being used.
        used[a] = true;

        // Pick a value to use; it's alright to reuse values.
        int v;
        v = random(rv);

        // Create the child AVPair by calling createAVPair to
        // recursively fill in the rest of the AVPair.
        AVPair ave = createAVPair(new Attribute(Integer.toString(a)),
                                  new Value(Integer.toString(v)),
                                  d - 1, ra, rv, na);

        // Add the child AVPair to our AVPair.
        avElement.addAVPair(ave);
    }

    return(avElement);
}

/**
 * Returns a random number between 0 and x - 1.
 */
final static int random(int x)
{
    double r;

    do {
        r = Math.random();
    } while (r == 1.0);

    return((int)(r * x));
}
```

```
    private static void usage()
    {
        System.err.println("Usage: java namespace.TestNameTreePerformance n");
        System.err.println("         n = the number of routes");
        System.exit(-1);
    }

}

class Result
{
    public long time;
    public long memory;
}

class Parameter
{
    public String n;
    public int l, h, i, d;

    public Parameter(String n, int l, int h, int i, int d)
    {
        this.n = n;
        this.l = l;
        this.h = h;
        this.i = i;
        this.d = d;
    }
}
```

Bibliography

[1] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Media Transcoding. In *Proceedings of ACM SIGCOMM '98*, September 1998.

[2] Adaptive Web Caching. `http://irl.cs.ucla.edu/AWC/`, 1998.

[3] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT) An Architecture for Scalable Inter-Domain Multicast Routing. In *Proceedings of SIGCOMM '93 (San Francisco, CA)*, August 1993.

[4] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics, 16(1):87-90*, 1958.

[5] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and Z. Fei. Application layer anycasting. In *Proc Infocom '97*, April 1997.

[6] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[7] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*, February 1998. W3C Recommendation.

[8] A. Chankhunthod, P. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings 1996 USENIX Symposium*, San Diego, CA, Jan 1996.

[9] Cisco—Web Scaling Products & Technologies: DistributedDirector. `http://www.cisco.com/warp/public/751/distdir/`, 1998.

[10] Cisco Cache Engine. `http://www.cisco.com/warp/public/751/cache/cds_ds.htm`, 1998.

[11] S. Deering. *Host Extensions for IP Multicasting*, Aug 1989. RFC-1112.

[12] S. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, December 1991.

[13] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. An Architecture for Scalable Inter-Domain Multicast Routing. In *Proceedings of SIGCOMM '94 (London, UK)*, September 1994.

[14] Estrin, D., et al. Simple Systems. ISAT study group, 1998.

[15] Fan, L. and Cao, P. and Almeida, J. and Broder, A. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. ACM SIGCOMM*, September 1998.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, Jan 1997. RFC-2068.

[17] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Semantic File Systems. In *13th ACM Symp. on Operating Systems Principles*, October 1991.

[18] A. Heddaya, S. Mirdad, and D. Yates. Diffusion-based Caching along Routing Paths. In *Proc. 2nd International WWW Caching Workshop*, June 1997.

[19] V. Jacobson. How to Kill the Internet. Talk at the SIGCOMM 95 Middleware Workshop, available from `http://www-nrg.ee.lbl.gov/nrg-talks.html`, August 1995.

[20] Jini (TM). `http://java.sun.com/products/jini/`, 1998.

[21] B. Kantor and P. Lapsley. *Network News Transfer Protocol*, February 1986. RFC-977.

[22] S. Kumar, P. Radoslavov, D. Thaler, C. Alaettinoğlo, D. Estrin, and M. Handley. The MASC/BGMP Architecture for Inter-Domain Multicast Routing. In *Proceedings of SIGCOMM '98 (Vancouver, BC)*, September 1998.

[23] U. Legedza and J. Guttag. Using Network Level Support to Improve Cache Routing. In *Proc. 3rd International WWW Caching Workshop*, June 1998.

[24] T.J. Lehman, S. McLaughry, and P. Wyckoff. T Spaces: The Next Wave. `http://www.almaden.ibm.com/cs/TSpaces/`, 1998.

[25] P. V. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of ACM SIGCOMM '88*, August 1988.

[26] J. O' Toole and D. Gifford. Names should mean What, not Where. In *ACM 5th European Workshop on Distributed Systems*, September 1992.

[27] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus (R) – An Architecture for Extensible Distributed Systems. In *SIGOPS '93*, pages 58–78, 1993.

[28] C. Partridge, T. Mendez, and W. Milliken. *Host Anycasting Service*, November 1993. RFC-1546.

[29] E. Passarge. Layer 4 Switching: The Magic Combination. *Network World*, 15 February 1999.

[30] C. Perkins. *IP Mobility Support*, October 1996. RFC-2002.

[31] C. Perkins. Service Location Protocol White Paper. `http://playground.sun.com/srvloc/slp_white_paper.html`, May 1997.

[32] J. B. Postel. *Internet Name Server*, August 1979. IEN 116.

[33] J. B. Postel. *Simple Mail Transport Protocol*, August 1982. RFC-821.

[34] K. Ross. Hash-Routing for Collections for Shared Web Caches. *IEEE Network Magazine*, 11(7), Nov-Dec 1997.

[35] K. Sklower. A Tree-Based Packet Routing Table for Berkeley Unix. Technical report, University of California, Berkeley, 1993.

[36] A. Vahdat, T. Anderson, and M. Dahlin. Active Naming: Programmable Location and Transport of Wide-Area Resources, 1998.

[37] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol*, June 1997. RFC-2165.

[38] P. Vixie. *Name Server Operations Guide for BIND – Release 4.9.5*. Internet Software Consortium, 1996.

[39] D. Wessels and K. Claffy. ICP and the Squid Web Cache. `http://ircache.nlanr.net/~wessels/Papers/icp-squid.ps`, August 1997.