Scalability in an Intentional Naming System

by

Jeremy Lilley

S.B., Computer Science and Engineering
Massachusetts Institute of Technology (1999)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

Author . . . .
Department of Electrical Engineering and Computer Science
May 15, 2000

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

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

Scalability in an Intentional Naming System

by

Jeremy Lilley

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

With mobile computing devices and services becoming more prevalent, the need is growing for mechanisms in a network that allow higher-level service interaction. Network-layer connectivity is not sufficient for devices to discover and understand relevant services in their vicinity; for example, simply being connected to the network does not allow a device to find the least-loaded printer or locate the nearest web camera. Several projects and systems related to the problem of service discovery are underway, including Sun's Jini, Berkeley's Service Discovery System, and the Intentional Naming System (INS) being developed at MIT LCS.

In the Intentional Naming System, scalability limitations arise when trying to scale beyond the scope of a single local organization. This comes mainly from the assumption, also made by its counterparts, that a server is able to know about all services in the world, and it is furthermore necessitated by the lack of scalability-related structure in its names. To alleviate this problem, we present changes to the Intentional Naming System, specifically the notion of partitioning the namespace to group services into autonomous *virtual space* communities. These virtual spaces facilitate the use of INS both in the context of larger organizations and across multiple administrative domains by reducing name advertisement costs and eliminating inter-domain cooperation requirements. They also preserve the characteristics that make INS work well in the local area. We look at the design issues beneath the changes and evaluate the suitability of the additions to the Intentional Naming System.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor

Acknowledgments

It is fitting to start this thesis thanking a number of people for their help, guidance, and encouragement along the way.

I would especially like to thank my advisor, Prof. Hari Balakrishnan. It was both exciting and enjoyable to work for someone with so much enthusiasm for the field of networking, and I am very appreciative for the flexibility he gave me to pursue what seemed most interesting.

The Intentional Naming System, which is the project related to this thesis, would not have existed without many people. It was great working with William Adjie-Winoto, Elliot Schwartz, and Anit Chakraborty on the system code—they were knowledgeable for discussing architectural ideas and were forgiving when I introduced bugs into the system. I am also appreciative of the people who used, critiqued, and helped refine INS, including Jorge Rafael Nogueras, Allen Miu, Bodhi Priyantha, Andrew Lau, and David Zych.

I am grateful for how my parents were supportive, and particularly my mom, who, when she called from week to week, tried to make sure that things were indeed all right. Also, I am happy that my sisters, Karissa and Miranda, attempted, albeit nearly fruitlessly, to improve my fashion sense over the past year as I worked on my thesis.

I thank Jason Yang, who made my stay at LCS more interesting and provided diversions as we both tried to figure out our futures. The same goes for my officemates, William and Anit.

Finally, I thank Jennifer Maurer and Victor Luchangco, who were there when I was not during thesis season. Jennifer's rule against sleeping in lab was good, but more so, I am grateful for her patience when I spent more time in lab than I anticipated. Victor was a great roommate and friend to learn from this past year, and his personal and thesis advice were very helpful.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

As the cost of computing power and bandwidth drop and portable computers and hand-held devices become more pervasive in everyday life, one of the key challenges is connecting them to interact intelligently. It is rapidly becoming possible to embed computers in every sort of appliance and device. But when that happens, will the resulting pieces be able to communicate with each other? As networks become more dynamic and contain increasingly many mobile or wireless components, the hope is that these devices and services will seamlessly interact with each other, which will ultimately benefit the end-users.

Simply placing a wire or radio between two components, or even implementing a traditional network stack is not enough to enable this functionality. Although network protocols such as TCP/IP [19, 26] and DNS [16] allow different entities to transfer data streams and to look up low-level network addresses, they do not provide higher-level functionality to allow users and applications to understand the capabilities of networked devices and services and interact with them.

The traditional solution to this problem is to require manual configuration to place the additional necessary knowledge in the system. A user needs to install a printer driver on his personal computer; a "universal" remote control needs to be told what type of television the person owns. As long as nothing changes and there are only a few devices to configure, this manual approach, although inconvenient, accomplishes the task. Adding many more devices to the system or making frequent changes, however, increases the administrative burden of the system to the point that it rapidly becomes unmanageable.

This is particularly an issue in mobile computing systems, where the available resources change frequently. Someone with a Personal Digital Assistant (PDA) may want to use the printer and scanner at the office and the stereo and television at home. As the "networked toaster" becomes economically feasible, the person may wish to control that from the PDA as well. Other devices may be added to the environment, while some may be unusable or already used by somebody else. Clearly, simply being able to exchange network packets will not provide this level of connectivity. How can the service providers and consumers be matched, particularly when both may frequently change? Ideally, there should be a higher-level abstraction that helps discover and use these resources effectively.

Many have recently approached this general problem of service discovery in a network. The realization is that adding a level of indirection for service discovery can greatly enhance the ability of different services to find and interact with each other. Various systems have been built to this end. By providing mechanisms for devices to describe themselves and their communication streams, tasks such as "printing to the least-loaded color printer" or "finding all the web cameras on the fifth floor" can enabled and made much more tractable.

While this layer of functionality can be powerful, every system represents various fundamental tradeoffs. For example, flexibility of a system is often at odds with speed or scalability. Faster device updates and reduction in manual configuration consumes more bandwidth. The key in any system is to match these tradeoff choices with the requirements of the underlying project.

1.2 The Intentional Naming System

One particular service discovery solution is the Intentional Naming System [1], which allows services to describe and refer to each other using name which are *intentional*. These names describe a set of properties that the services should have rather than specify a low-level network location. The idea is to allow applications to refer to *what* service they want rather than *where* in the network topology the service resides [24].

As an example, one may want to send a message to a web camera in room 504 of the LCS building. Without a service discovery layer, the application would have to know the DNS address of the camera, which might reflect the location somehow, but very well might not. Here are some examples of DNS names:

```
camera504.lcs.mit.edu
504.cameras.lcs.mit.edu
camera.504.floor5.lcs.mit.edu
randomname.lcs.mit.edu
18.31.0.51
```

These DNS names are difficult to guess at best, and this hard-coding makes it difficult to add and remove names from the system. DNS update times are not intended to be rapid [16], which makes this difficult for frequently changing environments. Furthermore, multiple cameras could be added to the room, and it might be desirable use all of them, or perhaps route a message to all the cameras of the given floor. DNS names carry little semantic information; they only denote a set of IP addresses.

Intentional names, and specifically *name-specifiers*, the designation for names in the INS language, provide a more flexible, more general approach to do this. Name-specifiers use hierarchical attribute-value pairs to describe a service. For example, a message intended for all the color cameras on the fifth floor could be addressed to the following name-specifier:

```
[service=camera][location=floor5[room=*]][type=color]
```

Any attribute in this name may be left unspecified or set as a wild-card, if, for example, the user does not care about the camera's resolution or exact room number. The name schema is completely controlled by the applications rather than being rigidly defined in advance, which allows an application with any set of attributes to join the system. Furthermore, application-specific metrics may be supplied to convey additional information, such as which printer is the least loaded or which camera's view is "best."

INS uses the principle of soft-state[5, 20], whereby periodic messages with expiration times and event-driven triggered updates are used to keep the information up-to-date. Individual services include themselves in the system by periodically advertising their descriptions. Applications using the system are able to learn about and use the cameras that happen to be operating at any time. This soft-state model, which is radically different from the transactional guarantees of a traditional database, works well in a distributed service discovery system that may have many unreliable components. Soft-state relaxes the requirement to do rigid locking or guaranteed reliable replication between different portions of the network, since the information will shortly either be refreshed or expire. We take this soft-state service model, which is normative in Internet routing, and adapt it for service discovery. The main advantages are that it treats failure as a normal case and allows robust healing from losses or outages that might be encountered in an unreliable network.

The Intentional Naming System itself consists of three main components: *clients* using the system, such as services and applications; *Intentional Name Resolvers* (INRs), which resolve all the requests for the clients; and *Domain Space Resolvers* (DSRs), which are a boot-strapping mechanism for finding Intention Name Resolvers.

The core of INS is a network of Intentional Name Resolvers, which interpret the name-specifiers. Figure 1-1 illustrates this architecture. The INRs collect the advertisements from services and create a database from this to provide the necessary functionality to applications. They are deployed as Java-implemented middleware over regular networks and designed to be easily usable by a Client API. They organize themselves into a spanning tree, which is used to disseminate name advertisements

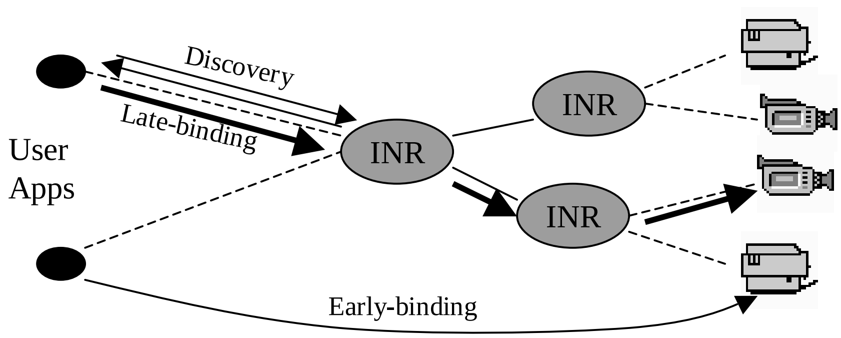

Figure 1-1: INS makes it possible for applications to discover and use services through its INR network. There are three main functions provided by the INRs: (a) **discovery**, where the application asks an INR for names of matching services (b) **late binding**, where messages may be passed through the INRs to the best or all the matching services (c) **early binding**, where the application requests the address of a described service and then communicates directly with it.

and forward multicast announcements efficiently [2]. To enable applications, the relevant functions include:

- *Discovery*, to allow applications to view and browse the available services, filtered by an Intentional Name query.

- *Late binding*, which allows messages to be forwarded to a service or set of services using an name-specifier as the address. Late-binding messages may be sent via *anycast*, to the "best" matching service, or via *multicast*, to every applicable service. This feature integrates name resolution and routing, which are traditionally separate network functions, to provide a more rapidly updated message-passing function.

- *Early binding*, which is much like a traditional lookup service. It allows applications to find the address for a service described by an Intentional Name and to contact the service directly.

The late-binding feature is particularly powerful, since it allows a message to be forwarded to a service or set of services, including arbitrary multicast groups, on the basis of a high-level description. In dynamic systems where the set of services may change over the course of minutes, it prevents the stale information problem—no address information needs to be cached and the correct set of recipients always gets the message.

The Intentional Naming System provides a number of operations that allow services to describe themselves and to find services which they need. The network of Intentional Name Resolvers provide the described discovery, late-binding, and early-binding functionality to services and end applications of the system. The discovery operation provides the ability to search and find the names of services based on a filter. The late-binding feature, which passes messages to destinations as described by an intentional name, consists of two modes, *anycast* and *multicast*. Anycast mode causes the message to be sent to the service matching the intentional name that is ranked as "best." Multicast mode causes the message to be sent to every service matching an intentional name query. Early binding allows applications to connect directly with a specific instantiation of a service once it is found. Overall, INS provides an integrated mechanism that saves application writers from needing to reinvent approaches to the distributed service discovery problem.

Furthermore, evidence of the vitality of this problem domain can be found in looking at the diversity of systems trying to solve similar problems—Sun's Jini [14], the IETF Service Location Protocol [18, 12], Microsoft's Universal Plug and Play [27], and Berkeley's Service Discovery System [7] are a few. Clearly, there are a number of significant issues and tradeoffs to encounter in engineering any such system. The existence of tradeoffs and active problems brings us to reevaluate some aspects of the Intentional Naming System more closely.

1.3 Contributions

From the experience of developing and using the Intentional Naming System, we encountered various limits to the original version. The major area for study was scalability to enable INS to work better with greater numbers of devices. The contributions of this thesis can be divided primarily into local-area scalability and scalability in the wide area.

1.3.1 Scalability

Scalability in INS becomes an issue as large numbers of services enter the system. INS assumes that the Intentional Name Resolvers know all the services they should ever need to route to. In other words, some type of "global knowledge" is assumed. While this is not a bad assumption in a system designed for a single local organization and other service discovery projects also make this assumption, it cannot be made in a system that will span many organizations.

This global knowledge assumption is necessitated by other areas of INS that make the system difficult to distribute. Namely, applications can define their own schemas, and INS allows partial matches based on any arbitrary subset of attributes. The idea of global knowledge is fairly widespread in other service discovery systems [14, 18, 27, 7] as well.

Certain factors make the global knowledge assumption expensive. The most significant problem is advertising costs; services need to advertise themselves periodically to an INR server, and all the INRs need to share their knowledge of existing services periodically. Propagating names among INRs uses a significant amount of bandwidth and processing power, and is especially wasteful for services that are seldom needed in a given area. As an example, if people in the Laboratory for Computer Science and Artificial Intelligence Lab need to use each other's services only occasionally, it is a waste of bandwidth to synchronize all the service information between both sites continuously.

In addition, the resolver's internal data structure, the name-tree, which is used to store and retrieve the name-specifiers, works well with thousands of names, but its ability to operate efficiently starts to diminish with tens of thousands of names. The former scale is more than enough for a typical organization, but it would be inefficient to store names for every service on the Internet.

We address this scalability concern is to partition the namespace of the system into many *virtual spaces*, or *vspaces*. A virtual space is an autonomous community of services that have characteristics in common and may frequently interact with each other. The vspaces are intended to correspond to "natural" divisions of the world that might not otherwise be captured by a system that indexes everything in the world by location or service type. For example, there may be a virtual space for a given lab or floor of a building, or for all the printers in an area. Intentional Name Resolvers still need to know all the services in a virtual space which they host, but requests for another virtual space can be forwarded to another INR, thus relaxing the "global knowledge" assumption and in effect creating application-defined partitioning. The usability cost of this mechanism is requiring a *vspace* entry in the names for communication between virtual spaces, but there are discovery mechanisms to locate the proper virtual space.

This thesis discusses the design and implementation of the virtual space mechanism, and it evaluates the result. This approach is also compared with other approaches, taking into account scalability, functionality, and usability.

1.3.2 Wide-area operation

Another problem is allowing the system to work across administrative domains well. The Intentional Naming System allows sets of INRs to form in various domains, producing many self-contained INS enclaves. However, the original implementation does not allow these communities to be loosely linked, that is, to connect without keeping all their data synchronized. This is difficult to require when the domains are managed by entirely different parties with potentially conflicting goals.

This thesis presents a way to use virtual spaces for operating in the wide area and spanning administrative domains. The idea is to create a component called a Domain Space Resolver (DSR), which is responsible for tracking which virtual spaces exist in its domain and which INRs contain hosting information about these vspaces. This information is kept updated by listening to periodic messages from each of the local INRs. The idea is to use the DSR component to help enable communication between virtual spaces in different administrative domains in a manner nearly as simple as within the same domain.

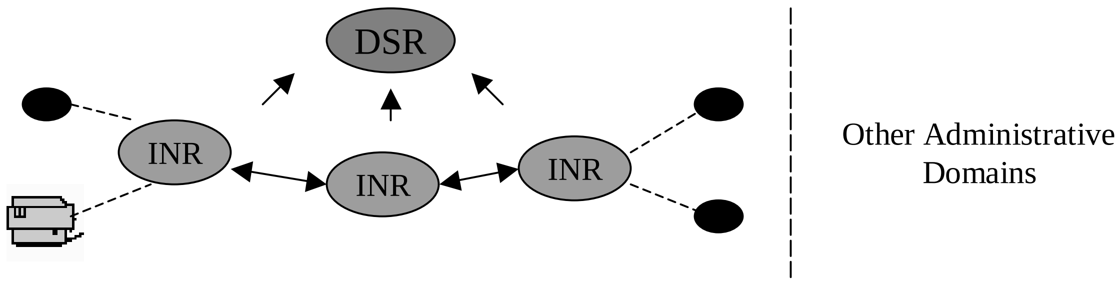

Figure 1-2: The Intentional Naming System consists of INRs to respond to services and applications as well as Domain Space Resolvers (DSR), which track all the INRs and the virtual spaces in a domain. DSRs are key for inter-domain operation.

The DSR, as illustrated with the INS architecture in Figure 1-2, helps with both inter-domain connectivity local-area bootstrapping. For working between domains, it allows wide-area applications to query for an INR which supports vspaces in another domain. In the local area, it provides the ability for applications to find an appropriate INR without resorting to a broadcast-style approach.

Furthermore, in a mobile environment which might contain services from different domains, we present a means which allows clients to see a coherent picture of what is available. One of the issues in this area is trying to solve too many or too few problems—this thesis presents a workable solution that is evaluated and may also be augmented in the future.

1.4 Outline

The focus of this thesis is on scaling the Intentional Naming System. This includes both improving the scalability inside an administrative domain and being able to operate between administrative domains. While we have developed and implemented thesis ideas in the Intentional Naming System, they are generally applicable to other systems as well.

In the next chapter, we discuss the design of the virtual space mechanism and of the architecture for the supporting system. This includes a look at the goals and tradeoffs, as well as how it solves many of the problems.

In Chapter 3, we discuss operation in the wide area in more detail. This includes looking at the different criteria for inter-domain operation and prescribing a manner for dealing with it in INS.

In Chapter 4, we examine the implementation of the changes to the system. In Chapter 5, we give a quantitative and qualitative evaluation of the virtual space mechanism's effectiveness in the Intentional Naming System. Finally, in Chapter 6, we proceed to a conclusion of the work.

Chapter 2

Virtual Spaces

Observing the limits of the Intentional Naming System as the number of services, users, and administrative domains increase, we present the design goals and architecture for a scalability mechanism. The result is a partitioning mechanism, known as the *virtual space*, or *vspace*. This chapter presents the basic design for virtual spaces within a single administrative domain; Chapter 3 presents additional goals and design for using virtual spaces in the wide area.

Virtual spaces are introduced to solve a number of problems discussed in Chapter 1. We strive to reduce the volume of periodic advertisements needed to maintain state in the system, since this becomes excessive as the system grows to encompass greater areas. Similarly, we want to reduce the amount of state that needs to be kept in any part of the system by better distributing it, which reduces memory requirements and increase performance. In addition, it would be helpful to be able to isolate disjoint parts of the namespace, partly to avoid namespace pollution but also to provide a unit of granularity for load balancing and distributing sets of names among servers, preferably in an application-defined manner.

2.1 Design Criteria

There are a number of relevant goals for creating a scalability mechanism for a service discovery system like the Intentional Naming System.

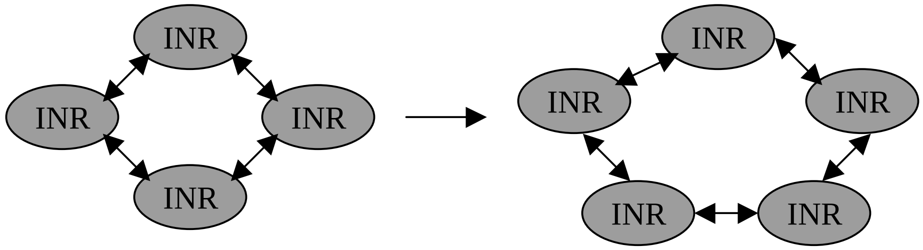

Figure 2-1: INRs in the system need to keep the sum of their state synchronized and replicated by periodically advertisements. The amount of state in the system and the number of places where all this state is stored both grow as as increasingly many services and INRs enter the system. This $O(n^2)$ total stored-state growth needs to be addressed in scaling INS.

1. **Scalability**

   (a) **Minimize advertisement traffic**: A big factor in the cost of INS as more services are added to the system is the overhead from periodic advertisements. Each service must advertise itself to an INR server, and each INR must advertise the services it knows about to its neighboring INRs. These messages are mandated by the soft-state [5, 20] service model which allows the system to recover from losses and outages robustly. Any way to minimize these advertisement costs without compromising the soft-state service model is a large gain. This growth in advertised state is illustrated in Figure 2-1.

   (b) **Scale the data structures**: The name-tree data structure used in the INR servers is quite suitable in intra-domain deployment when even thousands of services exist. But, like any other data structure, as the requirements for the number of names to store goes beyond the intended scale, its performance diminishes. There are two possible directions for moving the structure to inter-domain and large intra-domain work: making the existing name-tree more scalable or using more, but smaller, name-trees.

2. **Functionality**

(a) **Usability**: INS should not become much more difficult to use, particularly with the small-sized deployments that are currently common, because of the addition of the scalability mechanism. The overall *quality of service* should not be compromised.

(b) **Enable load-balancing**: The system should be robust as more servers are added to handle increased load. It should use this scalability mechanism to help distribute requests and advertisements. In addition, resources should be intelligently managed.

3. **Simplicity**

(a) **Minimal changes to system concepts, applications**: In scaling the Intentional Naming System, care needs to be taken to avoid compromising concepts and assumptions which are fundamental to its operation. INS is an existing project that meets its local-area design goals well; it would be a mistake to compromise those original goals in a newer version of the system.

(b) **Low overhead**: The implemented mechanism should not consume excessive resources, whether that is bandwidth, processing power, or memory. There should not be a significant "scalability penalty" or other obstacle to deter people from using a version of INS with the scalability mechanism.

4. **Robustness**

(a) **Enhance fault-tolerance**: One of the features of the Intentional Naming System and its soft-state advertisement model is that portions of the system can fail or become disabled without the whole being drastically affected. Routinely, INRs and services can be stopped and restarted without worry for the stability of the system. This fault-tolerance should be maintained as the scalability is improved.

(b) **Minimize configuration**: From the start, INS has held the goal of near-zero configuration; for example, devices should be able to enter and exit

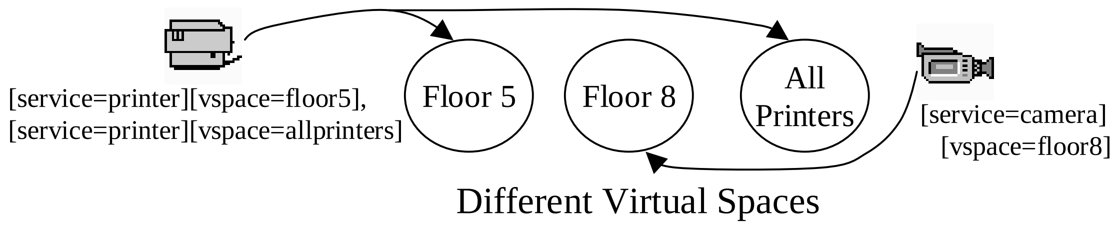

Figure 2-2: Services in virtual spaces. Rather than all services existing in a single "global" realm, they are part of smaller, more autonomous realms, or virtual spaces. Services may be in any number of these vspaces.

a room and use the local services without having to enter configuration information. This configuration should be minimized whenever possible, particularly in end-applications.

2.2 Design of Virtual Spaces

The idea behind the design is to partition the names of the system into autonomous virtual spaces. In their representation, the new name-specifiers have have an additional `vspace` attribute and matching value, designating which virtual space the name is in. Since the INS name-specifier language already consists of matching attributes and values, this fits cleanly into the existing syntax. When there is no `vspace` tag in a query, there are rules in the client API for sending to the default for the local area.

Thus, a sample name-specifier query for a printer on the floor that is written with or without the accompanying `vspace` tag:

```
[service=printer][location=floor5[room=*]]
[service=printer][location=floor5[room=*]][vspace=lcs]
```

Inside the scope of the *lcs* virtual space, both name-specifiers are the same. Outside the locality, such as to a device in the *AILab* vspace, the service may still be available, but the name needs the proper `vspace` tag.

Furthermore, there is no restriction to how many virtual spaces a single service may advertise itself to, and thus join. In the simplest case, each service is only a

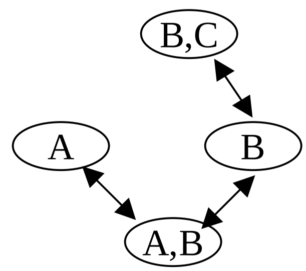

Figure 2-3: Virtual space INS architecture. Each INR "hosts" a set of vspaces and periodically keeps its contents updated with other INRs that host the same vspaces. The vspaces are A, B, and C. The arrows represent the spanning tree network that each virtual space maintains.

member of one vspace. But, with overlapping vspaces, there is no artificial restriction against that, as shown in Figure 2-2. An example of this could be the printer service advertising itself both to a $floor5$ vspace and an *allprinters* virtual space.

Inside the system, each Intentional Name Resolver may "host" any number of these virtual spaces, and a given vspace may be hosted by any number of INRs. By "hosting a virtual space," an INR joins the vspace's advertisement network and keeps updated information about all the services in the virtual space (Figure 2-3). An Intentional Name Resolver assumes it has authoritative information about all the services in the vspaces which it hosts. This is the equivalent to taking a large database and splitting it into a number of smaller databases that are replicated on different machines. The key here is that the partitions are application-defined and make sense to the applications. Any requests for a service in a locally-hosted virtual space can then be processed by the INR and returned.

If an INR receives a request for a service in a virtual space it does not host, the request is forwarded to an INR in the proper vspace. This forwarding information is provided by a unit known as the `Domain Space Resolver`, which manages all the virtual spaces in an administrative domain, and this information is cached. Since every INR does not need to know about every virtual space, advertising bandwidth between INR nodes can be made much lower.

The robustness of the soft-state service advertisement model is still present but

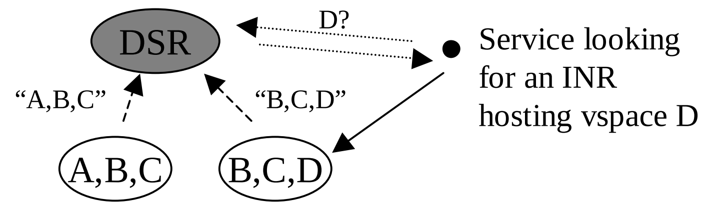

Figure 2-4: DSR interaction: INRs periodically announce the virtual spaces they host, with an expiration time. Services and INRs may look up and cache that information.

contained within the virtual space mechanism. Since the INR can host an arbitrary number of virtual spaces, the amount of authoritative knowledge that is assumed, and thus the local name advertisement costs and lookup load, can be suitably adjusted to help scale the system.

All the virtual space information in an area, with a list of known virtual spaces and INRs which host them, is maintained by the Domain Space Resolver. In the local area, the DSR primarily solves the bootstrapping problem of finding which INRs host a specific virtual space. While one could use a broadcast mechanism to get this information, that approach assumes the existence of a broadcast medium and is not particularly bandwidth efficient. The DSR is a server that employs a soft-state mechanism to maintain vspace-to-INR mappings; it gets this information from INRs that periodically announce to the DSR which virtual spaces they host. Clients may cache the data they request from the DSR, for the duration of their expiration time. The cost of this mechanism is low, and it allows INRs and services to find out which INRs host a given vspace.

This revives another connectivity issue—since no periodic name advertisement messages are sent between different virtual spaces, there are no major periodic costs for allowing their connectivity. This bridges connectivity where it may not have been practical due to advertising bandwidth with the original INS system. If, for example, two neighboring labs have services which they do not mind sharing, but they do not want to waste bandwidth keeping all their information synchronized, the virtual space mechanism allows them to do this.

2.2.1 Unions of virtual spaces

Emphasis so far has been mostly on reducing name advertisement costs by partitioning the namespace to enhance performance. From the standpoint of usability, however, it can be convenient to create larger virtual spaces that act as unions of other virtual spaces. This is similar, in the situation of email lists, to having mailing lists for each floor of an building and then also having an encompassing mailing list consists of all the floors of the building.

There are two ways to approach this type of problem in INS, where there might be different "floor" vspaces and an entire "building" vspace:

1. Each service advertises itself to both the floor and the building virtual spaces.

2. Services advertise themselves to the floor virtual space, and the floors advertise themselves as being part of the master building virtual space.

Approach 1 effectively doubles the advertisement overhead required for the service and might require extra administrative effort to get the services to advertise to both. This is an inelegant means of aggregation, but it is more efficient if there are many accesses to the entire building virtual space. It is similar to creating an index in a database—there is additional overhead to maintain it, but it may be beneficial if it is frequently used. It is straightforward to do this in INS with the virtual space design thus far.

It then becomes interesting to add a provision in the virtual space mechanism design that follows Approach 2, which might better match the situations of some applications. There may be some larger virtual spaces that are not heavily used but are convenient to have. Or, a virtual space may grow so large that it would be helpful to divide it into sub-virtual spaces to reduce advertisement messages but to also make it viewable as a single unit. To accomplish this, we extend the virtual space mechanism to include the notion of *aggregate virtual spaces*, which are unions of a number of child vspaces.

The design goals for these aggregate vspaces should be similar to the model for standard vspaces:

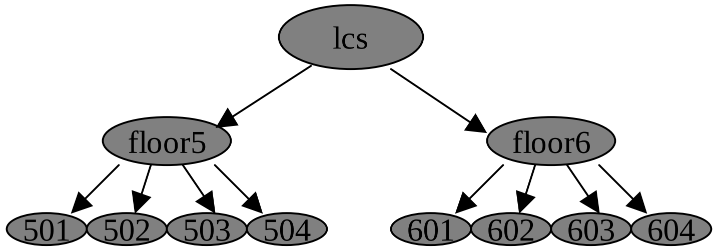

Figure 2-5: An illustration of aggregate virtual spaces. The top-level lcs vspace is an aggregate vspace, defined as the union of all the names in child vspaces $floor5$ and $floor6$. Likewise, $floor5$ and $floor6$ are both aggregate vspaces with a set of child vspaces. A service that advertises itself in the 504 vspace is then automatically recognized as also being in $floor5$ and lcs without any additional advertisements, but at a cost of extra query time.

- Their names should not be any different from those of standard virtual spaces, since INS ordinarily imposes no rigid restrictions in naming vspaces. Thus, aggregate vspaces should be able to look exactly like regular vspaces to the end user.

- In every area of application functionality, aggregate virtual spaces should work just as if all the contained services had been manually added into the larger vspace. Queries should be done on the basis of the services in the whole aggregate set, particularly with operations like anycast that single out one element—they should use the single best match of all the child vspaces, rather than the best match in every child vspace.

- The mechanism in which INRs periodically advertise their virtual spaces with the DSR should be largely unaltered with the advent of aggregate vspaces.

We now examine a design for implementing the concept of aggregate virtual spaces. The idea is illustrated in Figure 2-5, whereby top-level vspaces can be defined as the union of all the names in several child vspaces. To describe this design, we first look at the additional state that must be kept in an INR to support aggregate vspaces. Then, we examine how this state must be interpreted to support the different operations in an INR properly for aggregate vspaces. Also, we make arrangements in the design for the system to work with remotely-hosted vspaces across the network.

Fortunately, the design can be made simpler by leveraging on existing features of INS to do some of the book-keeping. The two extra pieces of information that an INR needs is whether a given virtual space is an aggregate vspace, and, if it is, what the names of the child vspaces are. This child vspace data must be kept consistent across the network, since many INRs can host the same virtual space.

To keep an aggregate vspace's children consistent across the network, we take advantage of the fact that INS's namespace is designed to stay roughly consistent across a network. Thus, we can store an aggregate vspace's child vspace names as "pseudo-services" in its name-tree.

To illustrate how the names of the aggregate vspace's children are stored, we present an example. Suppose there is an *lcs* virtual space which is in fact an aggregate vspace made up of $floor5$, $floor6$, $floor7$, and so on. Under this design, all the virtual spaces listed, including *lcs* and the individual floors, would first be constructed as normal, but applications only join their respective floor virtual spaces. That is, the fifth floor services join $floor5$ (or perhaps the room vspaces, such as 504), the sixth floor services join $floor6$, and so on. The aggregate *lcs* vspace is then built to represent the services on all these floors. As far as the data structures go, this means tagging the vspace as type *aggregate* and then adding the child vspace names as as specially-interpreted "services" in *lcs*'s name-tree:

```
[child=floor1][service=vspace][vspace=lcs]
[child=floor2][service=vspace][vspace=lcs]
...
[child=floor9][service=vspace][vspace=lcs]
```

The rest of the design requires changes in the code to properly interpret the aggregate virtual spaces. The main functions that need to be modified are late binding, including anycast and multicast forwarding; early binding; and discovery.

Multicast forwarding is straightforward—a request sent to the aggregate vspace just needs to be forwarded to each of the children. Anycast forwarding requires doing the match among all the child vspaces and then looking at which, over the entire aggregate set, has the best application metric. Discovery and early binding both take an approach where a query is sent to each child virtual space's name-tree, and then the results are accumulated and returned.

Aggregate virtual spaces also work when an INR does not locally host all the child virtual spaces. In this case, the INR hosting the aggregate vspace must generate a request to send to the INRs hosting the non-local child vspaces. Once the children's results come back, the total may be returned to the original caller. Unfortunately, aggregating non-local virtual spaces incurs a performance penalty, which is quantified in Chapter 5. However, it may be a reasonable choice when a larger vspace is not heavily used, at least compared to the advertisement bandwidth that would be required to maintain it.

2.2.2 Usability Features

Besides the aggregate virtual space mechanism which helps view sets of smaller vspaces as a single coherent unit, other virtual space usability features are needed in the design. This includes the ideas of default virtual spaces, a discovery mechanism, and location beacons.

For the most part, messages are intended to be generally sent inside a single virtual space. For example, within the Laboratory of Computer Science, it is likely that most service requests will be for services in the building. This is particularly true in small and ad-hoc environments, where there may not be a large number of relevant vspaces. To address this situation, we develop the the notion of a *default virtual space*. This helps follow the reasonable assumption that applications only dealing with a single local region should not need to deal with more complicated machinery.

A means for discovering the available virtual spaces is present in the design for an application to discover the names of local vspaces. The Domain Space Resolver, which "owns" all an area's virtual spaces, may be queried for which vspaces it knows about. This allows applications to view the virtual spaces which are available and even to browse them, particularly with the aggregate vspace hierarchy.

Additionally, a location beacon system [4] is being developed for use in INS which will be able to broadcast the name of a virtual space containing local services. This is type of approach is useful for mobile devices—the vspace information is in effect beamed to them and they may then join the virtual space and interact with services that are available.

2.3 Suitability of the Design

For this design, we proceed to investigate the use of virtual spaces in certain situations and to show how the mechanism meets the required criteria. The focus is on how vspaces scale the existing INS system better inside an administrative domain, since Chapter 3 looks in detail about specifically wide-area operation. In addition, we consider load balancing, and we finally look at why the virtual space mechanism is better than its possible alternatives. Actual experimental results of an implementation of this design are in Chapter 5.

2.3.1 Scalability in an administrative domain

The virtual space scheme delivers on the goal of minimizing advertisement traffic (Goal 1a) by partitioning the namespace into manageable subsets. Given a monolithic namespace of size N in a system of M INRs that could be divided into k roughly evenly divided namespaces of approximately size $\frac{N}{k}$, many possibilities arise for reducing the required update bandwidth.

As a first-order indicator, we look at the amount of *maintained knowledge* in the system. If all M INRs continue to host all k vspaces, there can be no reduction in advertisement bandwidth, since there is the same amount of knowledge to be maintained by the system. If, however, the INRs are perfectly divided between the virtual spaces such that $\frac{M}{k}$ INRs each host one of k vspaces, the knowledge maintained by the system decreases by a factor of k^2 from $M \cdot N$ to $\frac{M \cdot N}{k \cdot k}$. This maintained knowledge can be compared with the actual non-redundant knowledge in the system, which is N, but the multiplier drops as more vspaces are introduced.

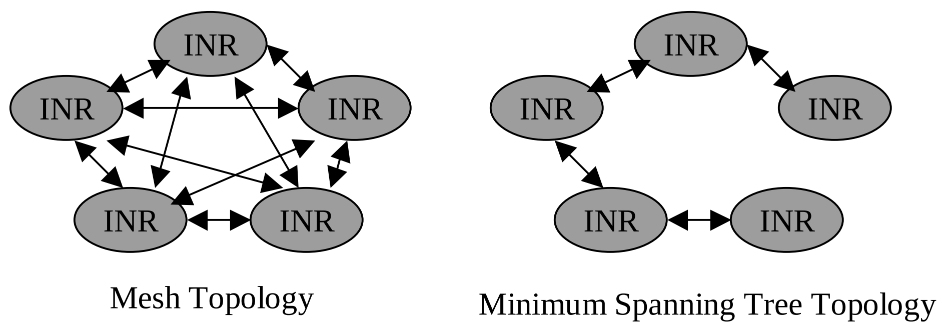

Figure 2-6: The maximally and minimally possible topologies for a connected advertisement network among a set of INRs. When not all of the INRs need to host all vspaces, the necessary advertisement bandwidth drops.

The effect of reducing this maintained knowledge depends on how the topology is structured. Each INR advertises all its names to all its neighbors at the beginning of each refresh period, so a less efficient network topology with a greater number of neighbors for its relative size will benefit more from the knowledge reduction. To illustrate this, we look at a full mesh topology, where every node in the network is considered a neighbor, and a minimum spanning tree topology [6], where a network of M nodes has $M-1$ connections. These topologies, shown in Figure 2-6, represent the maximal and minimal ways to connect the INRs, and between them is some tradeoff between bandwidth consumption and fault-tolerance.

A mesh topology with M nodes each having $M-1$ neighbors has $\frac{(M-1)M}{2}$ two-way links, which translates into $(M-1)M$ one-way communications of the set of names. If there are N names in the system, this becomes $N \cdot (M-1)M$ names sent per refresh cycle. Perfectly dividing these names among k vspaces and the INRs into k separate networks of $\frac{M}{k}$ nodes yields a total of $(\frac{M}{k}-1)\frac{M}{k}$ connections each transmitting $\frac{N}{k}$ names. This is a total of $k \cdot (\frac{N}{k} \cdot (\frac{M}{k}-1)\frac{M}{k})$ names per refresh cycle. When reduced to $N\frac{(M-k)M}{k^2}$, this indeed is an $O(k^2)$ improvement on average.

With a more efficient spanning tree topology, which newer portions of the INS code approximate [2], the bandwidth use still improves. Such a system has only $2(M-1)$ unidirectional links. In the undivided monolithic case, there are $2N(M-1)$ names transmitted per refresh cycle, which is an order $\frac{M}{2}$ improvement over the mesh architecture, leaving less inefficiency to optimize away with the vspaces. But, if the namespace is perfectly divided into k vspaces and the INRs are similarly divided into k networks of $\frac{M}{k}$ nodes, each of the k subnetworks will send $2\frac{N}{K}(\frac{M}{K}-1)$ names per refresh cycle, or $2N(\frac{M}{K}-1)$ total names per refresh cycle. The namespace partitioning thus produces an $O(k)$ factor improvement in name advertisement traffic.

Partitioning also helps scale the name-tree structures, Goal 1b. Since each virtual space hosted on an INR has its own name-tree, they can be kept smaller, on the order of $\frac{N}{K}$ rather than N names with k vspaces, regardless of how many name-trees are hosted on an INR.

The key to doing this partitioning while maintaining the usability and functionality of the system (Goal 2a) is that the partitions are designed to be along "natural" high-level divisions which exist in larger organizations, such as departments or floors or lab groups. In smaller scale and with ad-hoc use of the system, the virtual spaces can be ignored altogether in favor of a single default virtual space. As the system grows larger, this mechanism provides a way to shape the namespace along the natural boundaries of the organization, while improving the scalability.

This virtual space mechanism, along the same lines, makes minimal changes to the existing system (Goal 3a). The new name-specifiers look the same and simply leverage one optional `vspace` attribute to go outside of the default scope. The routing infrastructure and the main internal data structures, such as the name-tree, remain very similar with some changes to support many lightweight virtual spaces on the same INR.

The cost of maintaining these virtual spaces is low as well, satisfying Goal 3b. Periodic announcements to the DSR of the list of virtual spaces that an INR supports are lightweight and less frequent than the name announcements. The DSR announcements are particularly compact realizing that they in some sense represent all the names in their partition. Also related to virtual space overhead is the cost of maintaining an separate self-organizing overlay network for each virtual space, but the specifics of this are within the scope of another thesis [2]. As implementation optimizations inside the INR, there are no additional threads per virtual space, and the memory footprint of adding many virtual spaces is small.

We also observe that the system with virtual spaces satisfies the Goal 4, robustness. Rather than being one large network of Intentional Name Resolvers and one large namespace, it has been replaced with a greater number of smaller, more agile, more decoupled components. This can improve fault-tolerance (Goal 4a), since this modularity shields vspaces from each other and makes it harder for one wildly misbehaving component to have catastrophic effects on the entire system. It also requires less responsibility for each component of the system, reducing the chance for error. But, one concern about partitioning is that there may be fewer INRs servicing a given set of names. This may be alleviated by not under-replicating the system and ensuring that enough servers cover each vspace to maintain good fault tolerance.

In addition, the configuration complexity in the virtual space mechanism are kept small to satisfy Goal 4b. The use of a default local virtual space is used to prevent needing any other configuration information in a small environment. In addition, virtual spaces in a specific administrative domain can be completely discovered and in effect browsed, minimizing the amount of "hard state" configuration information in the system. The servers need to be configured with the virtual spaces which they host, but this information can be dynamically adjusted to move the system toward the optimum.

2.3.2 Load balancing

Since services are organized into similar, self-interacting collections, this makes an ideal unit for load balancing. This level of granularity is large enough to be managed individually by the INS system yet not excessively large that the addition of one will throw the unit off. We present this section as a design rather than a completed implementation, since core areas such as local scalability and wide-area support took precedence over other factors like vspace-level load-balancing and security.

Typically, the system may be out of tune for two reasons:

- **Excessive lookups**: The lookup load on the server is too high. Adding servers with the replicated information fixes this situation.

- **Excessive advertisements**: The servers are overly replicated, and the soft-state advertisement bandwidth required to maintain this replication is too great. Removing a server would aid this problem.

Both the advertisement bandwidth and the number of lookups to the server are easy metrics to obtain. When an exponentially weighted moving average (EWMA) of the lookup traffic passes a determined threshold, the INR can know that it is being over-utilized. Likewise, when the advertisement bandwidth is consistently low in relation to the lookup traffic, it has a good idea that its presence may be not needed.

When this information is known at the INRs, messages can be sent to the DSR to request activity. The DSR policy manager can then analyze the request. If the INR sends a *dropout_req* message stating that it is under-utilized for its advertising bandwidth, the DSR responds stating whether it should drop out or not. In the same manner, when an INR sends a *replicate_req* message stating that it is over-utilized, the DSR can have a better idea whether or not the additional INR is necessary. With this information, the virtual space may be hosted on another INR with its permission, or another INR may be spawned.

The advantage of the virtual space design is that it allows virtual spaces to be moved around without losing data in the process. If an INR used to support a virtual space but does so no longer, the inter-vspace mechanism simply routes it to an INR, returned from the DSR, which really does host the virtual space. Likewise, the presence of new INRs supporting a virtual space in time is propagated to through the system, more evenly balancing the load.

2.3.3 Alternatives Approaches

The benefits of this basic vspace approach are clear in that it provides the ability to reduce name advertisement bandwidth by partitioning overly large namespaces and the ability to bridge previously unconnected sets of services. There is a question, though, about whether the choice in mechanism is appropriate and how it compares to other alternatives.

The main drawback of our approach is that the namespace is in fact partitioned. Some of the original elegance is gone when there is another unit in the system to discover and specify. In some respects, it would be ideal to have a globally scalable version of the original INS system where matches are done using any number of properties against all the accessible INS services in the world.

Compression

Some have proposed systems that do try to keep a global namespace using compression. In general, compression will give some amount improvement, but only to a point which is generally not far enough. In addition, it has the potential to make the system more fragile or more brittle.

One design that uses compression is the Service Discovery Service (SDS) from U.C. Berkeley [7]. Their optimizing technique is to use Bloom filters [3] to produce a lossy aggregation of data. In this lossy aggregation, there may be a false positive, but not a false negative. First, they run a message digest or hash function on the name. To enable partial matches, they take many snapshots of the same name, and they employ some tricks to greatly reduce the combinatoric explosion involved when partial-matching among many attributes. Then, they use a Bloom filter to take the hash values of all the names in the system and collapse them into a large fixed-length field, by using the hash values to set specific combinations of bits. One can later tell from that field if a given name could have been in the system by checking if the appropriate combination of bits is set. Here, there is a fundamental tradeoff between this fixed-field length and the portion of false positives. Names in the form of these Bloom fixed-length fields are propagated and aggregated along a global tree structure, which reduces the amount of advertisement bandwidth required to maintain the global service namespace.

While this is an interesting approach and its design calls for a globally searchable namespace, it has many important limitations. First, though the Bloom filters do reduce the magnitude of bandwidth and storage required for the system, they only allow the system to scale to some point. This may in fact be sufficient for the local-area scaling and optimization problem, but it becomes inadequate for wide area mechanisms, as discussed in the next chapter. Once the gain of 10-100 is used up, such that ten or a hundred organizations can all share services, the process of scaling the system again stalls.

Similarly, the false hit rate, though it can be decreased by allocating more representation space, in such a system is generally on the order of 1-2% [9],which is not insignificant. Just a few services with name collisions, particularly when they are opposite branches of the global tree, are enough to require a significant amount of unnecessary traversal every time a service lookup is needed.

Furthermore, the service model under such a system is not as robust as that in INS. While the system state gets periodically refreshed, names cannot be prematurely unregistered from the system before the next refresh time. The application metric, an important component in INS, is normally propagated among INRs in addition to the names; it is not possible to get this without possibly traversing the entire global server tree in this architecture.

In general, there are many problems in approaches that do not use any part of the name for scalability. The part that makes this difficult is the need to make partial-matches—if one wanted a system for which the problem was reduced to matching a full string, one could take a hash function of the string, have an arbitrary number N servers, and send the request to the $(hash \bmod N)^{th}$ computer. But with the requirement to match on any partial number of attributes, there can be little to deduce from the query to help route it, at least without additional constraints.

Otherwise, when the problem grows to an increasingly larger area, one would have to store data and generate some potentially very long results for queries such as `[country=us[state=ma]]` (all the resources in Massachusetts), `[service=printer]` (all the printers in the world), or simply `[service=*]` (every service in the world). It is absurd that a general discovery system should be required to generate a response for those queries, and perhaps more so that the user would have wanted such a broad response. For the most part, when `[service=printer]` is requested, one only cares about looking in the local scope. Generally, when something of wider scope is desired, the user will have some extra higher-level knowledge to help locate it.

Adding structure

To proceed, then, some structure needs to be added to the name in order for a naming system like INS to scale. The idea of taking INS's relatively free-form model of completely application-defined names and expecting to use them for a global namespace without any tweaking is clearly untenable.

Observing some scalable naming systems, such as the Domain Name System [16] and X.500 [21], it becomes apparent that their namespaces are carefully structured. DNS names are completely hierarchical, where each part of the name is a step closer to finding the response in what is a large distributed database. The X.500 directory system provides a namespace for attribute lookup, but it is scalable by making the schema more rigid. Portions of the namespace are then delegated along the lines of this rigid schema. For example, *country* and *organization* are attributes used to get through most of the namespace.

It might then be easy to require that every intentional name have the organization, location, or perhaps a DNS name. However, there are many potential boundary cases which may not necessarily fit a rigid, predefined model of the world. The Intentional Naming System team wished to avoid unnecessarily forcing a global hierarchy on the namespace in advance.

2.4 Summary

Instead of partitioning on the basis of any predetermined attributes, our design calls for application-defined partitioning of the namespace. An attribute in name-specifiers, now known as the `vspace`, is used to specify that partition. It can be keyed off any convenient unit, whether that is in fact a location or organization, or if it might be a building floor or set of lab groups who frequently interact. It is more workable than any “global INS” solution could be, yet the changes are minimal and flexible enough that the system can be both scalable and usable.

Looking at the original problems, we find that the virtual space approach provides a feasible solution. Virtual spaces provide a means for reducing periodic advertisement bandwidth by allowing the namespace to be partitioned into manageable pieces. By the same token, the amount of state needed to be kept in the system can be reduced, both by putting the names into different virtual spaces and by optimizing the number of INRs that need to replicate each of these sets of names. Furthermore, this mechanism provides isolation for portions of the namespace when desired, to avoid namespace pollution and to aid in server load balancing.

Chapter 3

Virtual Spaces for the Wide Area

By *wide-area operation*, we refer to working in situations that involve multiple administrative domains, or sets of devices under common administrative control. Since we operate in the context of the Internet, we treat administrative domains as nearly synonymous with domains in DNS; for example, `mit.edu`, `cnn.com`, and `cs.berkeley.edu` are domains.

As the problem of service discovery grows to encompass multiple administrative domains, more complexities arise. There can be no central location that knows about or controls every vspace, since there may be large numbers of vspaces that may be owned by different entities with potentially conflicting goals. The delegation must be completed: the basic virtual space scheme in Chapter 2 transforms the problem of "global knowledge of services" in the INRs to "global knowledge of virtual spaces" in the DSRs; the global knowledge requirements need to be removed altogether and the virtual space system needs to be fully distributed. We endeavor to ensure that the virtual space scheme can work in the wide area, to ensure that services and applications from many independent administrative domains are able to interact.

3.1 Criteria for Wide-Area Operation

In setting goals for wide-area operation in INS, we first look at the problems. The main problem is being able to describe and find every object in the world in a way that is concise, meaningful, and scalable. It may be possible to give an exhaustive list of all the properties of an entity, yet such an approach is not concise and it may be intractable to build a scalable system that uses this approach. This scalability is particularly difficult if partial matches are allowed and there is no attribute that is a priori designated as a "primary key" (which is impossible to choose perfectly in a global system). Every entity in the world could be tagged with a large number, which may be concise and perhaps scalable. But, these descriptions are not meaningful semantically, as it would likely be difficult to "find the printer *next to* entity 1234567" in such a system.

Thus, it is important to note that we cannot attempt to solve every wide-area problem. The focus of our design and goals is to allow INS communities to be *bridged*, rather than to provide a one-stop solution by which every service in the world may be discovered by a partial set of attributes alone. The original goal in INS of having completely free-form, application-defined schemas and allowing partial matches on any set of attributes, simply cannot scale from the local to the wide area without extra assumptions.

So, just as the previous chapter scales the system with the assumption that the `vspace` attribute has a special meaning, it should be acceptable to further interpret the virtual space name. In areas where the design does not solve every problem, it should be acceptable that some extra information may be embedded in the name, or perhaps come from an external search engine, for expediency.

Thus, the goals for working among various administrative domains are as follows:

1. **Connectivity anywhere**: A major goal of our work is to enable increasingly larger groups to communicate. The overarching goal of wide-area operation should be to enable connectivity and some type of discovery with any publicly available service in the world.

2. **Minimize need for explicit cooperation**: Members of administrative domains should not need to agree in advance to be able to exchange packets. A requirement like this is the equivalent of needing to reinvent the IP routing

layer, where ISPs need to make individual peering agreements with each other, for service discovery. Explicit cooperation requirements between administrative domains should be avoided whenever possible.

3. **Minimal changes to local-area operation**: Services should not need to be fundamentally altered in order to be used in another administrative domain. In many ways, operation in the wide area should be transparent to services, though the underlying discovery process will be different.

4. **Implementable solution**: Categorizing *everything* in the whole world is a difficult problem that seems intractable. Something lightweight and implementable that works across administrative domains, particularly if it can be incrementally improved, is much preferable to a grand solution that is too slow to be usable.

5. **Integrate with location-dependent mobile devices**: Location-dependent applications are among the most important ones being enabled by INS. It is important to ensure that mobile device using geographically local services is able to operate effectively in an environment consisting of services from many administrative domains.

3.2 Wide-Area Virtual Spaces

The assumption thus far is that virtual space names are arbitrary tokens; the names themselves consist of no explicit hierarchy. Furthermore, the Domain Space Resolver unit for an area is supposed to know about every vspace that might be encountered. Our model for wide-area operation relaxes both these properties.

The idea behind the wide-area extensions is to allow virtual space names to consist of an optional *domain qualifier*. To illustrate this, there may be a *cameras* virtual space inside the *lcs.mit.edu* administrative domain. While the vspace can be referred to as *cameras* inside its home domain, it becomes possible with the extensions to refer to it outside the scope of LCS as *cameras:lcs.mit.edu*.

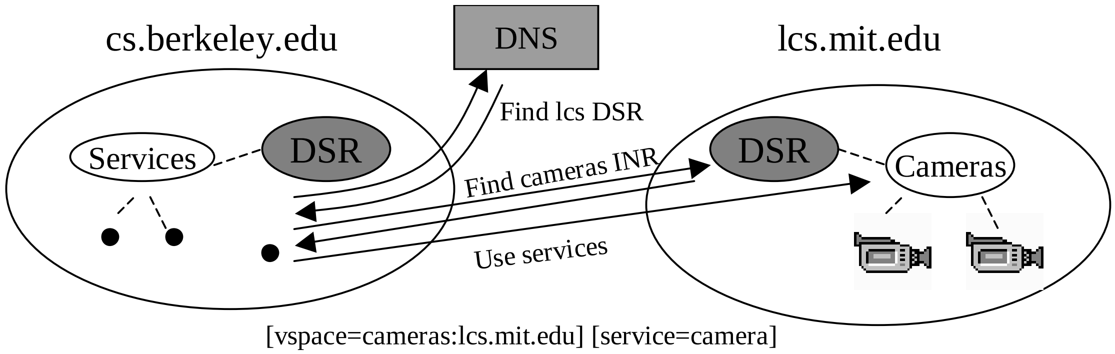

Figure 3-1: Inter-domain operation: services in vspaces hosted in another administrative domain can be used by using the wide-area version of the virtual space name. The request for an INR knowing about the *cameras* vspace is sent to the *lcs.mit.edu* domain's DSR, as located by the Domain Name System, rather than the local DSR. The response, the address of the INR, enables services in the remote vspace to be used as normal.

In this system, the DSRs become explicitly in charge of all the virtual spaces in their respective administrative domains. Each virtual space is owned by exactly one DSR, but nothing precludes services and INRs from one administrative domain from joining another domain's virtual space. Given that the proper DSR can be found to process the vspace request, most of the wide-area connectivity problems can then be reduced to the standard INS model. We solve this problem of finding the DSR by leveraging the Domain Name System.

To find the appropriate DSR for a vspace, the domain qualifier needs to be examined. If there is no domain qualifier or if it is local, the message is a standard intra-domain request that can be routed to the local DSR as described in Section 2.2. If it is a remote request, such as the *cameras:lcs.mit.edu* vspace being referred to from *berkeley.edu*, *lcs.mit.edu* must be used to find the appropriate DSR. The current convention involves using DNS to look for a DSR host named *domainname*, and then *dsr.domainname*. Thus, the request for *cameras:lcs.mit.edu* is routed to *dsr.lcs.mit.edu*.

There are other possible approaches to find the correct DSR for a virtual space. The alternate solution that we most carefully considered was using the SRV Resource

Record in DNS [11] rather than relying on the *dsr* prefix. The SRV record, which is a recent[1] addition to DNS, is intended for applications such as INS that wish to embed specialized information in DNS without requiring a new Resource Record type. But, due to the newness of the SRV Resource Record, the lack of support for the type in legacy DNS installations, and the fact that "dsr" prefix is a workable solution, we decided use the prefix in the current implementation. That, however, does not preclude the SRV entry from being used as well in the future.

This DNS approach also allows a single DSR to serve multiple DNS domains, so that administrative and DNS domains can be distinct if desired. Since DNS entries normally just return IP addresses, multiple DNS domains can point to the same DSR. For example, at MIT there are various labs which have their own DNS domains. Should they be considered part of the same administrative domain or not? By pointing DNS entries to the proper DSRs, it becomes possible for LCS and the AI Lab to share a single set of vspaces while the Media Lab has its own independent set. Furthermore, the EECS department can point the DNS entry in its *eecs.mit.edu* domain to MIT's DSR if it wishes to consider itself inside MIT's main administrative domain. Thus the problem of deciding "when asking for all the printers at MIT, do I normally count the ones at Sloan?" does not need to be solved in general but can be customized for any given environment by proper DNS delegation.

With this basic framework in place, there are only a few remaining areas in the architecture where modifications need to be made to integrate with the design described in Chapter 2. One common operation in INS, particularly in service provides, is taking the source and destination name-specifiers from a received packet, swapping them, and expecting that address to deliver the packet back to the source. It is plausible, however, that somebody sending a packet to such a system may not use their fully-qualified wide-area source address, which would cause problems when returning a response, since the domain to return to was not specified. To prevent this problem, which relates to the general problem of context in name resolution, packets sent outside the local domain automatically get their source address converted

[1]The Request For Comment for SRV, RFC 2782, is dated February, 2000.

to a fully-qualified wide-area address. Likewise, when packets are received by the final destination, the destination address is truncated to the local version of the virtual space to avoid confusion of the larger name in the application. The discovery mechanism similarly needs translation checks so that the names that are returned are fully-qualified if the discovery spanned administrative domains. Similar ideas are used in Internet email—a sender in the *mit.edu* domain who specifies his address as *johndoe* needs the address to be expanded to *johndoe@mit.edu* so that it may be interpreted in the right context when it leaves *mit.edu*. Along those lines, an email sent to *johndoe@mit.edu* also needs to be recognized when received by the *mit.edu* server as being local so that it might be routed to the proper internal mailbox.

And, while not the focus of the current work on INS, adding security to the design would be possible as well. The virtual space mechanism provides an ideal level of granularity for setting security access permissions. The DSRs could implement a security policy that prohibits various hosts from discovering, finding INRs for, and joining a virtual space. The DSRs messages are compact and infrequent enough that cryptographically signing them would not be prohibitive. INRs could also be set to drop packets from unauthorized hosts. This is not a part of our current implementation due to a focus on basic functionality; nevertheless the architecture allows the enhancements to be made.

3.2.1 Communication With Barriers

In the real world, communication between networks is often hindered by security mechanisms that may be deployed. Specifically, firewalls may filter out the UDP traffic and block incoming TCP connections necessary for operating INS. Since we experienced this inter-domain issue while developing INS, the design should allow some type of operation in these cases.

There are many types of firewalls and security mechanisms; we are not going to attempt to design specific solutions to all of them. However, an illustration of one case should highlight the idea of communicating by proxy in those situations.

For example, with one particular firewall in use, shown in Figure 3-2, it was

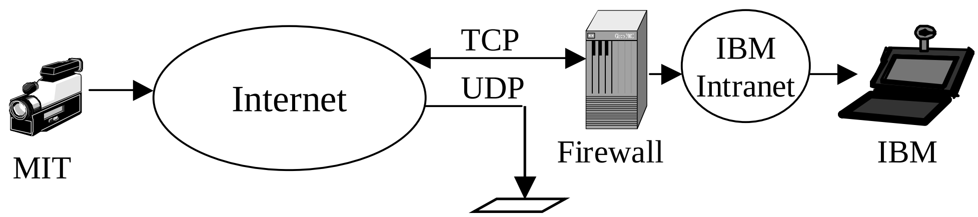

Figure 3-2: When security barriers such as firewalls are in place, the UDP packets that INS uses may not get through. These messages can be tunnelled through the firewall using a TCP/IP connection initiated from inside the protected network.

possible to make TCP connections from inside the firewall, provided specific libraries are loaded to interface with the SOCKS v4 [25] filtering protocol. Outgoing HTTP connections were also permitted, as well as incoming multicast UDP. The problem is enabling regular UDP connection with the nodes behind the firewall.

While there are many possible solutions to this connectivity problem, including the idea of using multicast UDP to receive data packets and HTTP to tunnel UDP packets from inside the firewall, the model which we wish to employ in general is proxying. A proxy node outside the firewall is set up to handle the connections for the INR outside the firewall. Outside applications can connect to this proxy, as directed by the DSR, and use all the services without worrying about the network connection.

The basic part of this solution involves no implementation changes in the regular Java INS code, since it can be enabled by the architecture of Java, which allows its underlying runtime network libraries to be changed. Whenever a network socket is created or packets are sent and received by the INR on the isolated side of the firewall, the action is done on the proxy which can communicate with the rest of the world.

Additionally, the DSR should be set to return the appropriate INR depending on where the request comes from. Combined with an INR and a proxy on the outside of a firewall, this provides a means with even a minimal amount of outside connectivity, for INS to use and communicate with services in the outside world.

3.3 Suitability of the Design

We believe that although our design is simple and efficient, integrating well with the framework of INS, it certainly does not solve every facet of the wide-area service discovery problem. We discuss below the strengths and weaknesses of the system, investigating the functionality provided by the mechanism.

Connectivity between sets of virtual spaces in the wide area is effectively enabled by our design. The optional vspace domain qualifiers address the problem of bridging virtual spaces belonging to different DSRs and domains, and they make accessible the complete functionality of the domain. For instance, the virtual spaces in another domain may be browsed, and the services in them discovered. This does require the domain name to be known; thus it serves intentionally as a bridging mechanism rather than as a general wide-area discovery solution.

Furthermore, little cooperation is needed to bridge domains. As long as a DSR wishes to make a virtual space publicly available outside its domain, there are no per-domain costs to the connectivity because the existing Internet infrastructure is used. This is contrasted with Sun's Jini [8], which needs to be explicitly "federated" with outside lookup services and requires IP multicast support. While the comparison is incomplete, since federation in Jini is only intended to glue a small number of communities together, it does show the burden that per-domain cooperation agreements can be.

In addition, the changes in the INS system to enable the domain qualifiers are minimal. The qualifiers need to be recognized and interpreted, and requests for INRs on remote DSRs need to be sent to the proper domain's DSR. But, besides some provisions for making name-specifiers canonical for the wide area in inter-domain conversations, the infrastructure is almost entirely untouched.

By the same token, the design is implementable in its full specification, as documented in Chapter 4 and evaluated in Chapter 5. An alternate approach could have required an extra level of indirection in the virtual spaces—rather than using DNS, an entirely high-level layer of wide-area abstraction could have been invented.

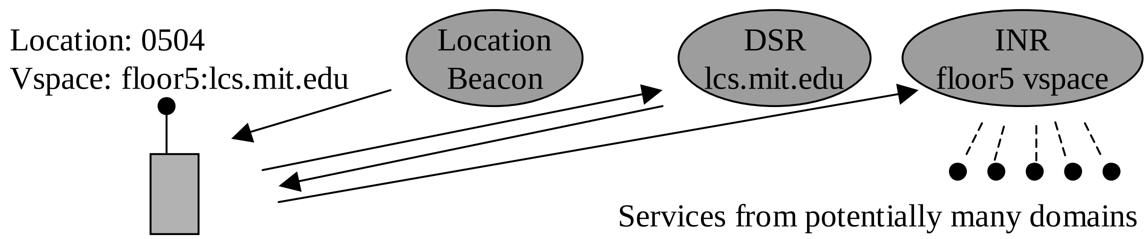

Figure 3-3: Location-dependent computing in an inter-domain setting: upon entering an area, the mobile device learns of a virtual space containing a collection of local services. Given a fully-qualified vspace name, the appropriate services may be discovered. Additionally, using services from another "home territory" or domain is possible due to connectivity provided by inter-domain vspaces.

Rather than specifying *cameras:lcs.mit.edu*, the requirement could have been to resolve *cameras:cameras-of-the-world*. The design does not preclude this extra layer from being added in a future revision of INS; it may be welcome as an incremental improvement once known to be beneficial. Rather, we present a basic, easily implementable design for bridging communities of services.

The last design goal, of explicitly being able to work well with location-dependent mobile devices across administrative domains is best illustrated using some examples. The general picture is of a mobile device brought into a new environment consisting of many services. This device receives location information from a location beaconing system [4], and it maintains some type of wireless network connectivity, whether from a local or wide-area provider. The test is how effectively the local services can be discovered, browsed, and used, particularly when they are owned or controlled by different administrative domains.

In this situation, we conceive of a location as being represented by a virtual space and some extra information. For instance, the room 504 might be identified as location "0504" in the $floor5$ virtual space. The location beaconing system can be then configured to send out this appropriate location information. The vspace should be completely wide-area qualified; if the device is connected by a wide-area network ISP, rather than via a local-area connection such as Bluetooth [15] or 802.11 [13], the domain qualifier is necessary to know the correct domain.

This location virtual space should contain the collection of available local services. That does not mean, however, that they are all necessarily owned or controlled in the same domain. Since a service may be a member of many virtual spaces, the service simply needs to advertise itself to an additional appropriate virtual space to be visible. This works for wide-area addresses; a camera may join both the *cameras:mit.edu* vspace and the *services:lcs.mit.edu* virtual space. In the case this is difficult, a "proxy" for a non-local service may be established, which joins the local vspace and routes the requests to the proper wide-area service.

Other mechanisms aid inter-domain service availability. The aggregate vspaces mechanism (Section 2.2.1) may be used to establish collections of virtual spaces without requiring the services to explicitly advertise for the extra virtual spaces. The virtual space union mechanism may also be used to establish a "default set of services" which are available using aggregation in many virtual spaces.

The key to this situation is that a location beacon is able to broadcast the name of a virtual space that offers a coherent view of an area. The local vspace's job can be reduced to *collecting* services from relevant areas. Some of this requires human intervention in the same way that the development of any other collection requires human effort. But, it is not hindered or complicated by the services being from many domains, since inter-domain vspaces can be treated the same as any other vspaces—services are not a priori prohibited from joining another domain's vspace, INR's are not necessarily prohibited from hosting another domain's vspace, and services are not prevented to using other domain's services. Location-dependent services with global connectivity are thus enabled.

3.3.1 Alternatives and tradeoffs

With this design, different virtual space communities may be bridged and inter-domain connectivity enabled. However, the approach does not solve the completely general wide-area service discovery problem of being able to find any service in the world by doing a partial match on any subset of relevant attributes.

With this said, it is not clear that the type of discovery, even if feasible, is necessary, at least as part of the base architecture. For example, in the regular Internet, TCP/IP and DNS may be used to communicate between hosts and discover host addresses, but search engines are higher-level units with specialized algorithms. The Internet provides some discovery and communication services, but they are supplemented by indexing units beyond the original design. While a service discovery system should surely provide greater discovery functionality with services than the regular Internet with hosts, there is a line between functionality that should be implemented in the base system and functionality that should be left to higher-level units.

This is a classic end-to-end argument for the simplicity of the system [23]. A mechanism and infrastructure for providing the extra indirection can be devised as an addition. But, this may not be necessary or urgent if there arises a popular site with commonly used services, say *services:yahoo.com*.

Some have observed the differences in character between the local-area and wide-area discovery problem. According to the committee on the Wide Area Service Location Protocol [22], location does not typically matter with wide-area queries, so it should not be considered in the queries. A statement like this seems acceptable in a system such as SLP [18, 28], which is designed from the beginning to hard-code special meaning into its attributes. However, WASLP's assumptions about how the wide area is divided makes the system ineffective to use in the local area. Perhaps the local and wide area problems are so distinct that they should be addressed in separate systems; however, our goal is to try to do otherwise. A glance at the many differences between a precise local service discovery system such as INS or Jini and a broad, general Internet search engine such as Google [10] or Northern Light [17] is enough to point out how the wide-area problem is more complex and more difficult to define. Particularly, in such a model, it is virtually impossible to return an answer that is nearly "complete" or accurate—something that is very much a part of the INS local-area service model.

So, the functionality of a search engine or the power of an inter-domain abstraction could have been included in the design. This might involve a master record which indexes the services in many many domains or a global advertisement tree like

Berkeley SDS [7] which attempts to attain some type of global service knowledge. This would avoid the need of explicit DNS addresses in the names and make a more uniform global service discovery system. However, these schemes are complex, are not significantly better for inter-domain location-dependent applications, and they fall short of the regular INS service model specifications.

Our wide-area domain qualifier design suffers from none of these complexity drawbacks and it provides an implementable scheme by which INS virtual space communities may be bridged. Furthermore, provisions are included to enable services to be integrated regardless of administrative domain.

Chapter 4

Implementation

To evaluate the effectiveness of our virtual space design, we implemented the mechanism in Chapter 2 and 3 in the MIT Intentional Naming System (INS). We describe the changes in the INS architecture, particularly in the Intentional Name Resolver and Domain Space Resolver components.

The three main code components of INS are the *Intentional Name Resolvers* (INRs), which are the servers that connect INS services and applications; the *Domain Space Resolvers* (DSRs), which serve as a boot-strapping mechanism for finding INRs; and the *Client Library*, which is an API for services and applications to access the system. Architectural and implementation changes were needed in all these components to enable the virtual space scheme. Nevertheless, the bulk of the effort focuses on the INRs and DSRs and is best described in these two contexts.

4.1 INR Implementation

The Intentional Name Resolvers form the core of the INS system by aggregating advertisements from services, providing a discovery and lookup mechanism, and enabling routing based on intentional names. The method by which messages are routed and forwarded in the system, as well as the way the INRs are organized into a network are the basis of other work [1, 2]. Here, we look at partitioning the namespace, which involves making changes to many aspects of the INR, including its name stor-

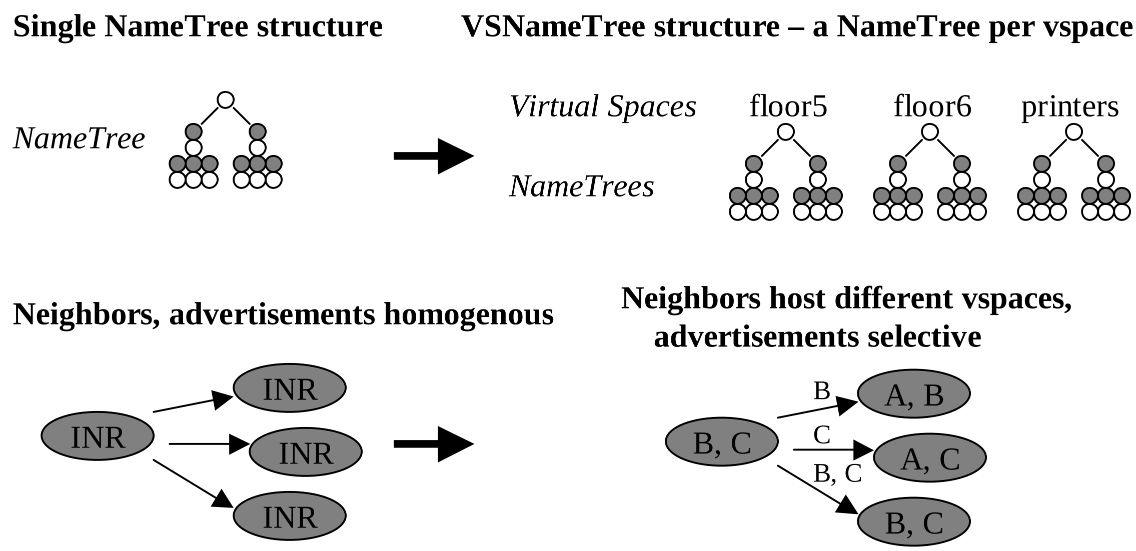

Figure 4-1: Architectural differences in adding virtual spaces. In the virtual-space enabled INR code, the names are stored in separate name-trees by virtual space. Likewise, since INRs each host a certain subset of vspaces, this neighbor vspace information must be known and the advertisements be properly directed.

age structures, advertisement routines, and neighboring mechanisms. This section discusses the provisions that were made to reduce the INR's assumption of knowing everything about the world to knowing only about certain well-defined portions of the world.

4.1.1 Basic Structural Changes

The goals for the virtual space mechanism include reducing the advertisement bandwidth and reducing the strain on the internal data structures. To accomplish this, changes were made in the name-tree structure, the advertisement mechanism, and the way other INR nodes are referenced in the system. An illustration of this is in Figure 4-1.

Names in the INRs are stored in the *name-tree* data structure [24]. This is designed to scale to a moderate number of names that might be found in a local area. Since names either contain a virtual space or otherwise find a default vspace, they can be separated into different name-trees by vspace. Rather than the INR containing a

```
ADVERTISE-NAMES()
    for V ← each hosted vspace in VSNameTree
        updateBuffer ← empty update buffer
        nt ← VSNameTree.GETNAMETREE(V)

        for n ← each name in nt
            U ← name-update created from n
            updateBuffer.ADD(U)

            if updateBuffer.FULL or
                    U.sourcePath different from last
                updateBuffer.FLUSH(V)

        updateBuffer.FLUSH(V)
    return

UPDATEBUFFER.FLUSH(vspace)
    for n ← each neighbor in vspace
        if (n not the source path)
            SEND(buffer.contents, n)
    return
```

Figure 4-2: ADVERTISE-NAMES Pseudo-Code: the data from each hosted virtual space gets shared with its neighbors who also host that virtual space.

single *name-tree* object, it contains a collection of name-trees for each hosted virtual space, called the $VSNameTree$. This maps virtual space names to name-trees using a hash table, and provides appropriate accessor functions. It furthermore serves as the authoritative list in the INR for which virtual spaces it hosts. And, in this structure, each virtual space's corresponding *name-tree* represents the INR's complete knowledge for that vspace. This isolates each virtual space as a separate, contained unit in the INR.

The main benefit comes from going a step beyond separating how names are stored to separating how names are advertised. Name advertisements for a given virtual space should only be sent to other INRs hosting the same vspace, rather than to every local INR. Otherwise, it is just a waste of bandwidth. This then requires knowing which virtual spaces each neighboring INR supports, which requires augmenting neighbor data structures to store this. One this is known, the name advertisements can be sent to only the neighbors hosting the specific vspace, by an algorithm such as in Figure 4-2. In effect, this creates distinct advertisement networks for each virtual space. A key in the implementation is avoiding per-virtual space threads in the code, which would make vspaces too heavyweight to put large numbers of them on a single server.

Beyond the use of multiple name-trees, augmented neighbor information, and a pruned advertisement method, other details are required to enable the basic mechanism. The Client Library typically ensures that requests sent to the system contain the default virtual space, but when that otherwise does not happen, names are tagged with the appropriate Resolver default virtual space. The starting set of virtual spaces that an INR hosts is loaded at initialization, but this may be changed at runtime. System-use names are also appropriately tagged with virtual space names to ensure they are properly routed inside the INRs.

Communication within a single virtual space, or when a vspace is hosted by the INR, is straightforward. The necessary name queries to discover names, to route a packet by late-binding, or to look up an address for early-binding can be obtained from a local name-tree for the appropriate virtual space. After this name-tree is retrieved from the $VSNameTree$ structure, the query can occur and the appropriate actions taken. Variations on this process such as that for inter-vspace communication, unions of virtual spaces, or inter-domain operation require addition steps.

4.1.2 Inter-Vspace Communication

An INR may need to forward a packet to an intentional name address by late binding and not have the virtual space hosted locally. In this case, the packet needs to be forwarded to an Intentional Name Resolver that does host the appropriate virtual space.

In the code, a list of INR servers which host a specific vspace can be obtained by the mechanism in Section 4.2.3. These are cached in the $VSResolvers$ list in the INR. Given a vspace name, the packet can be forwarded to an INR entry picked from the correct list. The implementation does all this inter-domain forwarding by UDP

```
Forward-Packet(packet, params)
    Vspace ← packet.destination.GetVspace()

    if (Vspace in VSNameTree)            // hosted locally
        nt ← VSNameTree.GetNameTree(Vspace)
        RouteSet ← nt.Lookup(packet.destination)
        ForwardUsingRouteSet(packet,params,RouteSet)

    else                                 // hosted remotely
        if (Vspace not in VSResolvers)
            VSResolvers.RetrieveInfoForVspace(Vspace)

        inr ← VSResolvers.GetINRForVspace(Vspace)

        if (inr not null)
            ForwardToAnotherINR(packet,params,inr)
        else
            Drop Packet
    return
```

Figure 4-3: Forward-Packet Pseudo-code: An illustration of how a packet may be processed locally if the necessary virtual space is hosted on the local INR, or it may be forwarded to an appropriate INR otherwise. It is also possible that the virtual space name may be malformed or not exist, in which case the packet is dropped. The *params* field represents characteristics such as anycast vs. multicast.

to avoid setting up short-lived TCP connections. A feature of this scheme is that it does not need to trust the correctness of the local cache—if an INR originally hosted a virtual space v and then stopped hosting v for some reason, it can still forward the data on to an INR which does support v. Figure 4-3 contains pseudo-code for this process.

With the inter-vspace late-binding code in place, the INR's other functions, namely discovery and early-binding, are nearly automatically working between virtual spaces. This is because discovery and early-binding are both implemented between the Client API and the INR by exchanging regular INS late-binding packets. Thus, if a client wishes to make a discovery request but sends the request to an INR which does not host the relevant virtual space, the system will forward the request to a proper INR like any other packet.

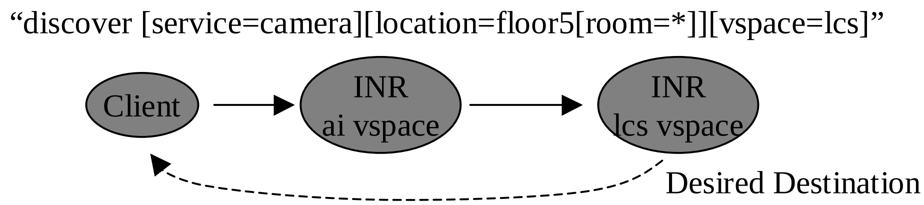

Figure 4-4: Inter-vspace discovery problem. Late binding solves the problem of getting discovery and early-binding messages to the right INR, but it may not get the response back in the inter-vspace multi-hop case. Extra data is added to the request packet to ensure the correct return client is known.

There is one obstacle, however. The original model for returning discovery and early-binding messages was simply to respond to the host which sent the message. This scheme is good because does not need the client's name to be propagated through the system to work and is efficient. In the inter-vspace case, though, the discovery message is sent from the client c to an INR ai and then forwarded to an appropriate INR lcs. Here, the return address from the standpoint of the lcs INR is not the client c but the first INR ai. This is illustrated in Figure 4-4. To alleviate this, the discovery and early-binding request formats have been changed to include the return address so that the response may go back to the proper host.

Along the lines of the discovery and early-binding messages, one further addition to the implementation of the system was adding sequence numbers that are unique for a requesting host. This addition allows the Intentional Naming System to behave more smoothly with potential concurrency.

4.1.3 Unions of Virtual Spaces

Unions of virtual spaces are implemented, as described in the the design in Section 2.2.1, by tagging a virtual space with an *aggregate* flag and using its namespace to track which virtual spaces are its children. This mechanism is ideal to use in the case where the advertisements required to maintain an aggregate vspace are more significant than the volume of requests that will be issued to it. A virtual space can be represented as a union of smaller virtual spaces and also have services of its own.

```
Recursive-Lookup(name, maxSteps, foundNames, remoteVspaces)
    vspace ← name.GetVspace()
    if (vspace not in VSNameTree)                    // hosted remotely
        remoteVspaces.add(vspace)
        return

    if (vspace is aggregate and maxSteps > 0) // unions of vspaces
        children ← GetChildVspaces(vspace)
        for v ← each vspace in children
            subname ← name
            subname.SetVspace(v)
            Recursive-Lookup(subname, maxSteps − 1,
                             foundNames, remoteVspaces)
        return

    RouteSet ← VSNameTree.Lookup(name) // hosted locally
    for r ← each element in RouteSet
        discoveredName = ExtractName(r)
        foundNames.add(discoveredName)

return
```

Figure 4-5: Recursive-Lookup Pseudo-code: the lookup routine for the discover mechanism modified to handle unions of vspaces. Two lists are recursively built, *foundNames* of discovered names and *remoteVspaces* of remotely hosted vspaces which need to be contacted.

Besides adding the aggregate flag and putting names representing the child vspaces such as `[child=floor1][service=vspace][vspace=lcsbuilding]` in the name-tree, the main thrust of the implementation is modifying the different functions of the INR to interpret these names properly. The four functions to modify are discovery, early-binding, anycast late-binding, and multicast late-binding. The multicast case is the most straightforward, since the packet just needs to be forwarded to all the child vspaces. In the other cases, a query and response of all the child virtual spaces is required.

With *discovery*, a client asks for a list of names which fits a filter. This requires discovering the names in each of the child vspaces and then merging the results. If the children are aggregate vspaces, the routine should be recursive and search its child vspaces. But, there should be a limit to avoid the possibility of looping infinitely. It is important to note that the names are returned with the canonical, service-described vspace. This is helpful as an optimization to avoid unnecessarily going through this aggregate vspace mechanism when connecting back to a returned name.

The requirements become more complex, however, when there may be unions of virtual spaces which are not hosted locally. To deal with this, the lookup routine is modified, as in Figure 4-5, to recursively build two lists. One, $foundNames$, consists of all the names discovered in locally hosted virtual spaces. The other list, $remoteVspaces$, catalogs all the remote child virtual spaces which need to be queried. Once these lists are known, the idea is to send discovery request messages to each vspace in $remoteVspaces$. As the responses come back, the names can be added to the $foundNames$ list and the vspace removed from the $remoteVspaces$ list until the discovery is complete. Once the $remoteVspaces$ list is empty, all the names are discovered and the response can be sent to back to the original host.

If a child virtual space goes down or ceases to exist, the discovery mechanism bypasses it and returns the result from all the other child vspaces. The current implementation uses no time-outs maximums in the INRs, so the discovery response time is the greatest of the existing child response times. The Client Library's API, however, does provide a time-out value in the discover function to avoid indefinitely waiting due to a lost packet or similar situation. And, the library is implemented to facilitate a number of concurrent requests because the messages all contain sequence numbers.

This process of looking at all the child vspaces is nearly the same in the case of *early-binding* requests. In fact, the same Recursive-Lookup routine as in Figure 4-5 is used in the implementation with the addition of a flag to differentiate between discovery and early-binding for the base case.

For *late binding*, multicast and anycast operations need to be treated separately. This is due to the underlying semantics of the operations. A multicast message is intended for all the services who match a given query, whereas an anycast message is only intended for the service which matches the query and has the highest application-advertised metric. The unions of virtual spaces mechanism needs to retain these se-

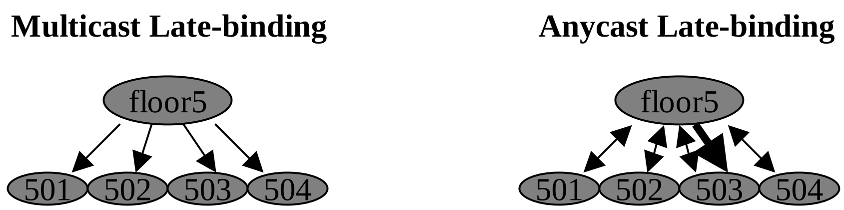

Figure 4-6: Late binding with Unions of Vspaces: for multicast, the packet can be multicast to every child virtual space. In the anycast case, however, the message must be sent to the "best" overall service. This requires asking each child vspace for its "best" entry and sending the packet to the highest ranked of those entries.

mantics, which in the anycast case requires querying all the child virtual spaces for their "best" entry and then finally sending the message to the highest ranked of those entries. This is shown in Figure 4-6. When child virtual spaces are hosted on different INRs, this is accomplished with a message similar to the early-binding request message. A caching mechanism could be invented to reduce the messages across different INR servers, but that would compromise many of the dynamic properties of INS.

4.1.4 Inter-Domain Operation

Inter-domain virtual spaces provide a extra scoping information for which DSR to query about the virtual space. For example, the *mp3players:lcs.mit.edu* vspace has a *domain qualifier* for the *lcs.mit.edu* domain. The INR needs to be able to process this addition information, which from an implementation standpoint can be accomplished mostly inside a single *DSRManager* module.

For the most part, the inter-domain implementation is in the INR, since it is essentially a DSR selection mechanism for the INR. All the DSR needs to do is normalize the requests and operate as before.

The three main operations necessary in the INR to support inter-domain vspaces are as follows:

- **GetDSRForVspace**: Given a domain-qualified vspace, this returns the address of a DSR in the appropriate domain which should know about that vspace.

- **AdjustIncomingVspace**: If the virtual space of a packet entering the INR has a domain-qualifier that represents the local area, this removes the domain-qualifier so that the system may recognize it as a normal local vspace.

- **AdjustOutgoingVspace**: Used on packets being sent outside the local administrative domain, this fully qualifies a virtual space to include the correct local domain name.

When a packet or request needs to be sent to another virtual space, the regular procedure is to check if it is hosted locally. If not, the INR checks its *VSResolvers* list to see the address of an INR which hosts it is cached. Otherwise, DSR is gets queried for an INR that hosts the virtual space. It is this last step that needs to be modified to support domain-qualified vspaces. Rather than sending the request to the local DSR, the implementation uses the domain-qualifier, such as *lcs.mit.edu* to find an alternate DSR to send the request to. Once this is done, much of the basic work is accomplished in the INR.

The GetDSRForVspace routine abstracts away the process of finding a DSR for a domain. The current process, as described in the design in Section 3.2, involves checking two addresses: *dsr.domainname* and *domainname*. The former is an obvious prefix, whereas the latter is implemented so that one could specify a complete machine name in the domain-qualifier. For example, it one wanted to make the machine *fenway.lcs.mit.edu* a DSR in charge of a small, experimental "subdomain" within *lcs*, the latter check makes it possible. The well-known port currently assumed for a DSR is 5678. Since this functionality is abstracted to a single procedure, the lookup process could be straightforwardly extended to include additional means such as the DNS SRV Resource Record [11], which is intended for putting application-specific data such as this in DNS.

After this, it is important for the INR to know which incoming virtual spaces are local and to maintain its internal structures consistently. If an incoming packet is addressed to the *cameras:localdomain* virtual space, the INR should realize that the address is the same as its local *cameras* vspace and process it accordingly. Therefore, all the domain-qualified name-specifiers on incoming packets are truncated to their local versions if they are in fact local. This keeps a single version of all the local vspace names in the data structures. The AdjustIncomingVspace routine performs this function of truncating local vspaces. The test of whether a virtual space is local involves checking whether the domain qualifier can be mapped using GetDSRForVspace to the same address as any of the local DSRs—this is a more accurate comparison than naively checking if the domain strings are equal. Once this truncation occurs, the INR can process incoming packets with domain-qualified vspaces in the same way it did before.

Otherwise, as a usability feature for services and applications, responses to packets sent in the wide area should have their return addresses qualified to the wide area. In other words, if a packet is sent from *services:cs.berkeley.edu* to *cameras:lcs.mit.edu*, the sender at Berkeley may potentially use the local version of the vspace name, *services*. The INR takes care, with the AdjustOutgoingVspace routine, to ensure that packets going to virtual spaces outside the *cs.berkeley.edu* scope have their return addresses qualified with the full *cs.berkeley.edu* domain. Otherwise, the receiver at *cameras:lcs.mit.edu* would not only know that the packet came from the *services* vspace, which would be insufficient to send a response. The implementation of this AdjustOutgoingVspace routine is straightforward—the string representation of the DSR address, with the "dsr." prefix removed if present, is appended as the domain for the vspace name. This is also done on names found with the *discovery* mechanism so that the client may know the fully-qualified versions of names it discovers.

Beyond this, there are a few boundary cases to enable. INRs from one domain should be able to host virtual spaces from another administrative domain. Appropriate implementation is made in the INR to check the names and to ensure the messages for an INR joining a virtual space go to the proper DSR. In addition, services should be able to join virtual spaces from other administrative domains. This requires sending the name announcements for the service to the INR in the proper domain, which is accomplished for the most part by the inter-domain changes to the late-binding mechanism.

4.2 DSR Implementation

The Domain Space Resolver, which serves as a bootstrapping mechanism to find Intentional Name Resolvers for a given virtual space, was adapted to support virtual spaces. With the DSR in this section, we look mostly at its communication mechanism with the INRs and Client Library. The necessary changes include making provisions to store each INR's vspaces, incorporating vspace information in its communication messages, and moving its messaging architecture to soft-state [20] to be more robust.

At the most basic level, some of the functionality of a DSR could be performed by a alternate approaches such as a flooding protocol or a set of DNS entries. The first approach would involve broadcasting a message to the network to request a response from INRs supporting a given vspace. However, it is bandwidth inefficient, cannot scale beyond the scope of a local area network, and does not readily provision for external access. The latter approach could be accomplished by setting up DNS entries, such as *cameras.vspaces.lcs.mit.edu* for the *cameras* vspace in LCS, to point to the proper INRs. This DNS approach is more adaptable to the wide area, but it is somewhat less dynamic; it requires a well-deployed mechanism for modifying DNS entries, perhaps outside the administrative domain; and it is difficult to refine the security model if necessary. The DSR's implementation, as described in the coming text, gets around the limitations of these other approaches to provide an efficient, dynamic, and accessible means to store virtual space information.

4.2.1 DSR Messages

The motivating feature of the DSR is for INRs and clients to be able to interact with it. For this reason, the best way to view the DSR architecture and implementation is to look at the messages which are exchanged. In order to simplify the DSR, INR, and Client Library implementation, the DSR messages are encapsulated with two classes: *DSRRequest*, for requests going from the INR or Client Library to the DSR; and *DSRResponse*, for responses from the DSR. These classes provide constructors and interface methods to the different messages and abstract away the details of

DSRRequest Messages

ADVERTISEVSPACES (*INRaddress*, *vspacelist*, *ttl*, *thisComplete*, *wantVspaceData*, *sequenceNumber*)
GETVSPACERESOLVERS (*vspacename*, *sequenceNumber*)
VSPACESRESOLVERS (*vspacelist*, *sequenceNumber*)
DISCOVERVSPACES (*sequenceNumber*)
PING (*sequenceNumber*)

DSRResponse Messages

ACKADVERTISEVSPACES (*sequenceNumber*)
GETVSPACERESOLVERS (*INRlist*, *sequenceNumber*)
VSPACESRESOLVERS (*vspacelist* with *INRlists*, *sequenceNumber*)
DISCOVERVSPACES (*vspacelist*, *sequenceNumber*)
PING (*sequenceNumber*)

Figure 4-7: DSR Messages, both the requests from the INR or Client Library to the DSR as well as the DSR's responses.

converting to and from a wire-form representation.

The basic set of DSR messages is listed in Figure 4-7. Each *DSRRequest* message returns a counterpart *DSRResponse* message, with the name sequence number to identify the message. The transport protocol for these messages is UDP, but the DSR could support TCP/IP as a protocol layer as well, particularly to handle long messages with large numbers of virtual spaces. Besides the PING message, which is a simple mechanism for testing the presence of the DSR, the rest of the messages are described in the context of either *INR Periodic Advertisements* or *Requesting Information* in the coming sections.

4.2.2 INR Periodic Advertisements

In the same way that services periodically advertise themselves to INRs with an expiration time so that they may be purged from the system when they go down, the robustness and self-healing properties of soft-state are also beneficial to virtual spaces. The DSR implementation is thus changed from the original where INRs registered themselves to the system indefinitely with the assumption of hosting all names into a periodic system that includes information on which virtual spaces are hosted.

To accomplish this, INRs send the AdvertiseVspaces message periodically to the DSR with the list of virtual spaces they support. The default advertisement period is longer than that of services, on the order of a half-hour, with the assumption the INRs change or go down less frequently than the services. This message may be sent at any time, which besides the periodic case includes whenever a vspace is added or removed from the server. A *thisComplete* flag is used to signal that the message completely lists all the virtual spaces hosted on the INR, which allows since-removed vspaces in the DSR's list for that INR to be purged. The $INRAddress$ includes the INR's IP address and port information, and virtual space names are sent as a list of regular strings. There is a single *ttl* time to live field for the entire INR entry rather than for each virtual space on the INR since the INR is a coherent unit that is either entirely up with all the vspaces it last advertised or entirely down. Otherwise, a last $wantVspaceData$ flag specifies whether an simple AdvertiseVspacesAck acknowledgement is returned or if a full VspacesResolvers message containing names of all the INRs supporting each vspace it hosts should be returned. Usually, the flag is set such that the full VspacesResolvers message is returned to help keep the INR's local neighboring data up to date, but in startup and when the VspacesResolvers was just otherwise returned, this flag is set off to save unnecessary DSR processing.

By aggregating the information from the periodic advertisements, the DSR can maintain a list of the unexpired INR nodes along with their hosted virtual spaces. An index of the existing virtual spaces, along with the INRs which support them, is also maintained for quicker access to some of the queries. This data is used to answer the requests that the DSR gets from both the INRs and clients.

4.2.3 Requesting Information

When asking which Intentional Name Resolvers support a virtual space, two messages may be used. The first, GetVspaceResolvers only requests a list of INRs for a single virtual space, whereas VspacesResovlers requests a list of INRs for each of a list of virtual spaces. The former is mostly intended for use in the Client Library and other lighter-weight situations, while the latter is used in the INR, which needs to keep better track of potentially many more virtual spaces. The received responses are used to keep the INR and Client Library INR lists updated, and the sequence number mechanism allows this to be done with a minimal amount of blocking or use of threads.

The other major request message is DiscoverVspaces, which returns a list of all the known virtual spaces on the DSR. This is just a list of strings, though it is plausible that additional information could be included or that a security mechanism on the DSR could eventually filter the names passed back to the application.

Beyond this, the messages are sent using UDP, so if there is no response within a small timeout, the INR and Client Library code resend the request. The class encapsulating the returned set of INRs has a PickNode method to return a single node from the list either for the Client Library to peer with or for the INR to forward an inter-vspace packet. This method currently selects the INR randomly, but it is possible to extend to choose the closest or otherwise better entry.

4.2.4 Information Expiration Mechanism

Keeping the data roughly consistent is a need which spans the DSR, INR, and Client Library. Since a periodic advertisement mechanism is used to disseminate this information, the other important half of this is ensuring that the information expires at the proper time. Since we have looked at the advertisement mechanism, we focus attention on the expiration mechanism, particularly on the boundary cases which need extra consideration.

As we see from Figure 4-8, there are a number of assumptions about what the different components of INS know about the virtual spaces. The DSR is the bootstrapping mechanism which is intended to contain all the vspace and INR information, but it may not have heard from all the INRs if it was recently restarted. Since there is an invariant that the AdvertiseVspaces message contains all the virtual spaces that an INR hosts, the DSR can assume to know about every vspace from a known INR, but that it may not know of every INR. Adding the latter part of the assumption

Virtual space knowledge and expiration assumptions

- **DSR**
  - If it knows about an INR n, it knows about all the vspaces that n hosts.
  - It may not immediately know about all INRs if it crashes and is restarted.
- **INR**
  - Knows all the vspaces it hosts.
  - In hosted vspaces: has a connection with all *neighbor* INRs, but not with *every* INR in the vspace. These neighbor INRs should not expire since they have direct contact, but other in-vspace INRs should.
  - Non-hosted vspaces are less critical, but attention needs to be paid to make sure they expire when the INR also has a hosted vspace.
- **Client Library**
  - Emphasis on simplicity and low overhead, thus less cached information.
  - Knowledge of which INRs host a vspace is entirely derived from the DSR. This info may be freely purged and re-requested.

Figure 4-8: Who knows what about which INRs host a virtual space? These assumptions shape the expiration mechanism.

allows the system to be more resilient to DSRs going up and down.

The Intentional Name Resolvers need to use the DSRs both to know INRs which host non locally-hosted vspaces and other INRs which host the same vspaces. The former is necessary to enable inter-vspace communication, whereas the latter is to help choose neighbors to form an overlay network. In the latter, a few of the resolvers are *neighbors* which form the advertisement network, and the balance are *ordinary*. The *neighbors* should not be pruned, except by the overlay network code, since the INR has a direct connection with them and will itself know if they have died. The other resolvers may be pruned from the list using two mechanisms: by a *response from the DSR* excluding them or by an *expiration time*.

The circumstances by which a response from the DSR may be used to prune resolvers from the list for virtual space v are as follows:

- The INR asked for virtual space v

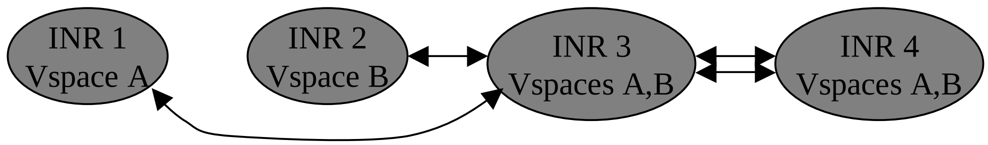

Figure 4-9: Example topology for neighbors and expirations: there are two virtual spaces, A and B, and four INR nodes. The arrows depict the INR neighboring.

- The DSR response's list for virtual space v did not contain INR r
- INR r is found on some other vspace's list in the DSR response

The DSR response method is fairly restrictive and applies mostly to INRs which stop hosting one of their virtual spaces. Otherwise, there is a time expiration mechanism. There is a single expiration timer for simplicity, and this timer applies only to non-neighbors since, again, the overlay network has additional information about neighbors. Whenever a list of INRs for a vspace is requested from the internal $VSResolvers$ structure, a expiration-checking method is called to ensure that all the resolvers to be returned are still valid. Otherwise, they are pruned from the returned result.

As an example of this shown in Figure 4-9, there may be two virtual spaces, A and B. At first, INR 1 hosts A, INR 2 hosts B, and both INR 3 and 4 host vspaces A and B. There are separate advertisement networks for virtual spaces A and B. In these, INR 1 might be a *neighbor* in vspace A with 3, but not with 4, whereas 3 would have to be a *neighbor* in vspace A of both 1 and 4 in order to connect the virtual space. Since the DSR may be restarted and have incomplete information, a DSR response to 1 without its connected neighbor 3 should not case the neighbor to be pruned because 1 empirically knows that 3 exists.

As a complication to the situation, an INR may dynamically rebalance its load, so INR 3 may drop its support for vspace B. Our goal is to make sure INR 1 eventually finds out that 3 no longer supports B, if it has the pointer to that vspace cached. This requires the ability to do selective purging of an INR's virtual spaces, and we want to do this without inelegant and bulky per-vspace expiration timers. To do this,

```
ProcessVspacesResolvers(DSRResponse)
    for vspace ← each virtual space in the DSRResponse
        nodes ← DSRResponse.GetINRsForVspace(vspace)

        for node ← each INR in nodes
            if node ≠ self
                AddToAllINRsList(vspace, node, node.TTL)

        toRemove ← ∅
        for node ← INRs hosting vspace that we know about
            if isNeighbor(node) continue

            if DSRMessage.ContainsINRAnywhere(node)
                toRemove.Add(node)

        for node ← INRs on toRemove list
            RemoveFromAllINRsList(node, vspace)
    return
```

Figure 4-10: ProcessVspaceResolvers Pseudo-code: code on the INR to process the hosting information returned by the DSR about a set of virtual spaces. This both adds new INRs and prunes INRs which the DSR implies no longer exist.

INR 1's vspace request must include vspaces A and B. Upon receiving the response, INR 1 will note that B does not have 3 as a host and that 3 is included elsewhere in the message. This information satisfies the constraints listed for knowing that 3 no longer supports vspace B and thus 1's view of the world will converge to correctness. The pseudo-code for receiving a DSR response and doing this type of pruning is in Figure 4-10.

The second expiration method is more applicable to entire INRs going down or simply to the information being unnecessary to request again. For a neighbor, such as INR 4 relative to 3, the overlay network code [2] will detect and deal with the problem. For a non-neighbor, such as INR 4 relative to INR 1, an expiration check in the internal $VSResolvers$ data structure will purge the expired data.

The Client Library's expiration mechanism is simpler because there is less information to keep consistent. In its simplest form, the client only needs to contact a DSR once and peer with a single INR for life. This will work since the INR can forward non-local vspace requests to other INRs but is not optimally efficient or robust. Otherwise, the Client Library keeps around a single "picked" node from each virtual space that it is interested in and may renew that data once it expires. The simplifying factors are not needing to keep extra data for neighbor formation and self-organization, not needing to keep around many INRs for a given virtual space, and not needing to maintain extra data indexes that are useful in the INR.

Chapter 5

Evaluation

In this chapter, we evaluate the effectiveness of the virtual space implementation in the Intentional Naming System. There are two main areas by which to examine such an addition: quantitatively by performance, and qualitively by usability. In the first section, we look at performance numbers indicating the cost and quantifiable benefits of the mechanism. The second section delves into a more qualitative evaluation of the system; we examine applications for INS which were implemented to be vspace-aware to show the system is in fact usable.

5.1 Performance

To evaluate the performance of the virtual space system, a number of experiments were run on the Java implementation of INS. The goals were to determine and better understand the overhead of the mechanism. Thus, we first observe at the routing performance and the reduction in advertisement bandwidth. The latter also gives indications about the overhead of the vspace mechanism. Then, we look at the performance of specifically aggregate vspaces. This both indicates how well the unions of vspaces feature works and gives performance information about INS in general. Finally, we look at the performance of the DSR and see how scalable this part of the system is to thousands of INR nodes.

The tests were performed on Pentium II computers running the Blackdown port of

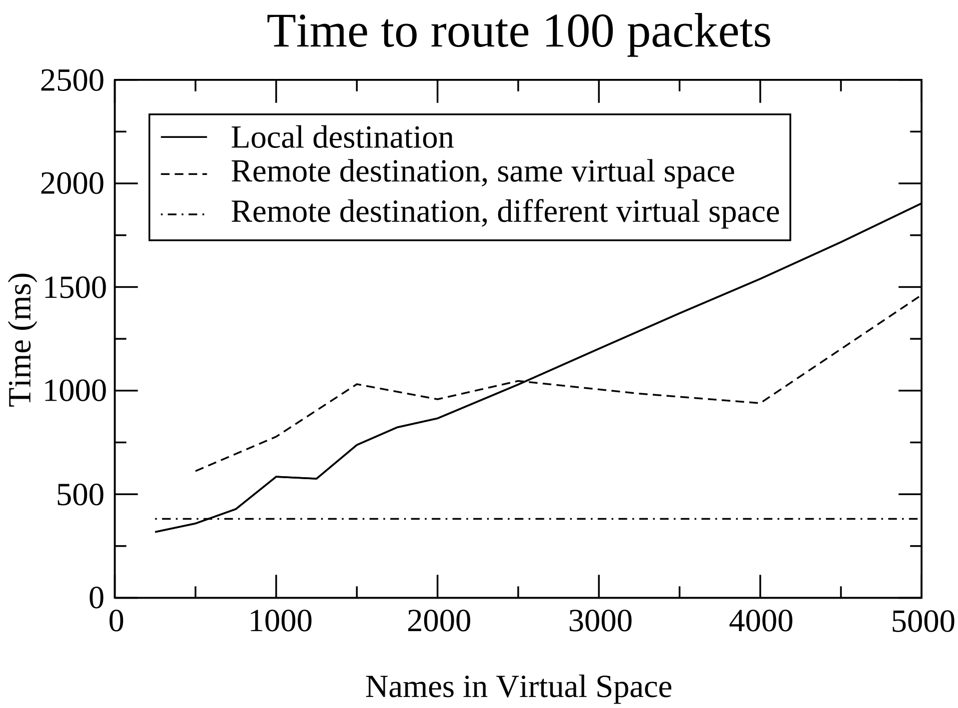

Figure 5-1: Processing and routing time for a 100-packet burst in the intra-INR, inter-INR, and inter-virtual space cases.

Sun's JDK version 1.1.7 with the Redhat 5.2 Linux distribution. When host-to-host tests occurred, they were connected with a 100 Mbps Ethernet.

5.1.1 Routing Performance

First, we look at the relative performance of routing with these virtual spaces. We sent a burst of one hundred 586-byte messages with different numbers of names in the virtual spaces. The name-specifier source and destination addresses were randomly generated, on average 82 bytes long. The results are shown in Figure 5-1.

The most important case is intra-virtual space communication, since it reflects the most common communication. There are two cases to consider here, which are reflected in the two sloping lines. In the first, the message may be delivered within the same INR node. The processing and routing time here vary somewhat with the virtual space's name-tree size, from 3.1 ms per packet with 250 names to 19 ms per packet with 5000 names. This figure is heavily influenced by the speed of the name-tree lookups, and it also counts in some of the end-delivery overhead. This extra overhead varies somewhat linearly with the number of name-tree entries. The other case is without this end-delivery overhead, which is more typical with the Client library in place. Here, we observe a flatter line, where the scalability bottleneck in routing comes mostly from the lookup speed of the name-tree. For the most part, this processing time is about 9.8 ms per packet during the burst.

The other major case is inter-virtual space communication, where the virtual space is not hosted locally. Here, we find a flat line at 381 ms, which does not vary at all with the number of names in the destination vspace. The remote vspace's INR information is cached from the DSR in this case. This type of result is to be expected; the INR is simply forwarding the packet to the proper vspace. The performance of this portion of the trip is not affected by the number of names on other INRs or performance in future hops.

5.1.2 Advertisement Reduction and Overhead

Otherwise, we observe the periodic advertisement bandwidth and how it is reduced by the system. In Figure 5-2, we see the size of the periodic messages required to maintain the state for a given number of names. There are three relevant lines.

The solid line gives the basic case where a given number of names are placed in a single virtual space on a single host. This gives an idea of the "normal" requirements of the system. When the names are evenly divided into two distinct vspaces but kept on the same machine, some virtual space-related overhead is encountered. We see this in the difference between the top two lines, but we note that it is a minor level of overhead. More importantly, when the partitioned vspaces are placed on separated machines, we find that the required time does indeed drop in half.

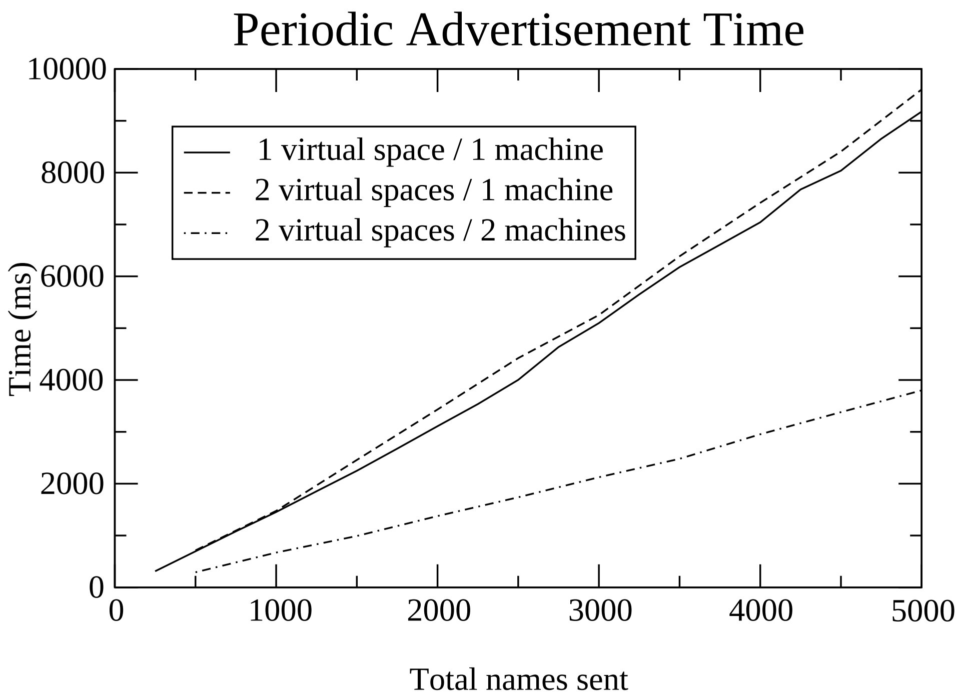

Figure 5-2: Periodic advertisement times when the names are divided into two equally-sized virtual spaces.

5.1.3 Aggregate Vspace Performance

We can get a better idea of the overhead of searching separate nametrees by looking at the performance of aggregate vspaces. These aggregate vspaces, as introduced in Section 2.2.1, are defined as the union of several child virtual spaces. When a request is performed on an aggregate vspace, work typically needs to be performed on each of the child vspaces.

As we vary the number of these child virtual spaces, we note the performance associated with a given number of simultaneous virtual spaces. Testing how this performance changes as more child virtual spaces enter the picture can give insights not only into how well the aggregate vspace mechanism works but also into the overhead of virtual spaces in general.

The metric we used for examining aggregate vspace performance is the round-trip anycast latency. This is simply the total time required for a given service A to anycast a message to a service B and for B then to return the response to A. Anycast

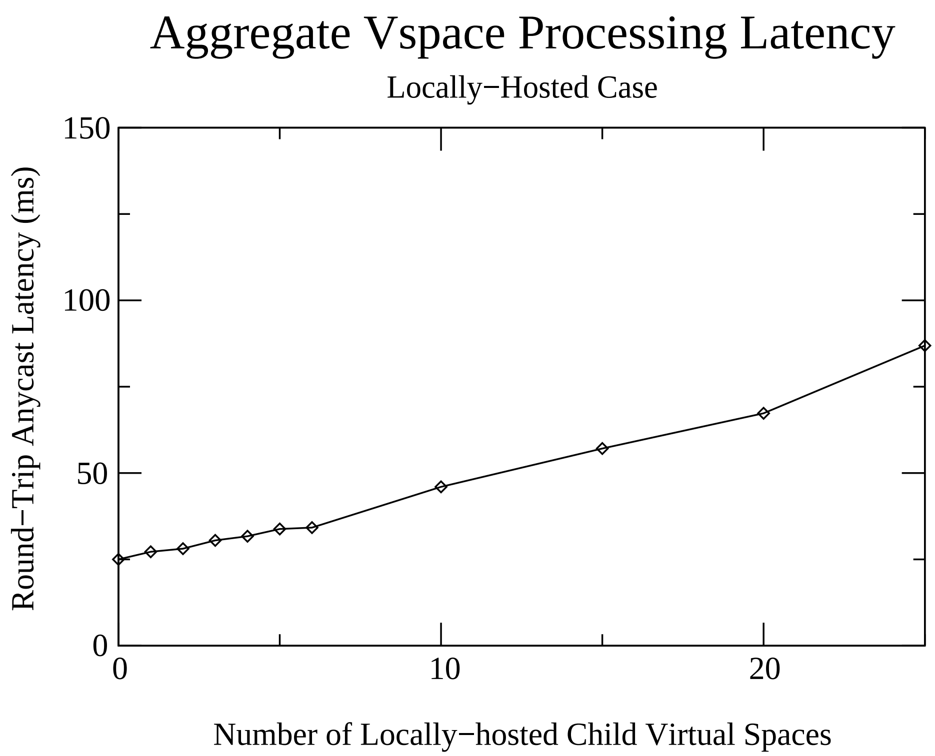

Figure 5-3: Aggregate vspaces must search all their child vspaces for many routing functions, including anycasting, to maintain proper application-level semantics. This graph shows how the anycast latency increases as the number of locally-hosted child vspaces grows.

in INS typically requires searching the proper virtual space's nametree for matching name records. However, in the aggregate case, it involves checking every child virtual space's nametree so that the message may be sent to the "best" service out of the entire aggregate set. This type of aggregate child searching also needs to be done for *discovery* and *early-binding* requests, so data collected for anycast provides a good picture for how aggregate vspaces scale in general.

In Figure 5-3, we observe the effects of increasing the number of child vspaces in an aggregate virtual space. In this case, a sender anycasted 128-byte packets every 100 ms to a receiver, which in turn anycasted the message back. The data points represent the average round-trip latency across 500 samples, and the tests were conducted on a single machine by sending to the loopback interface. For each aggregate vspace, all the child virtual spaces are hosted locally on the same INR. Thus,

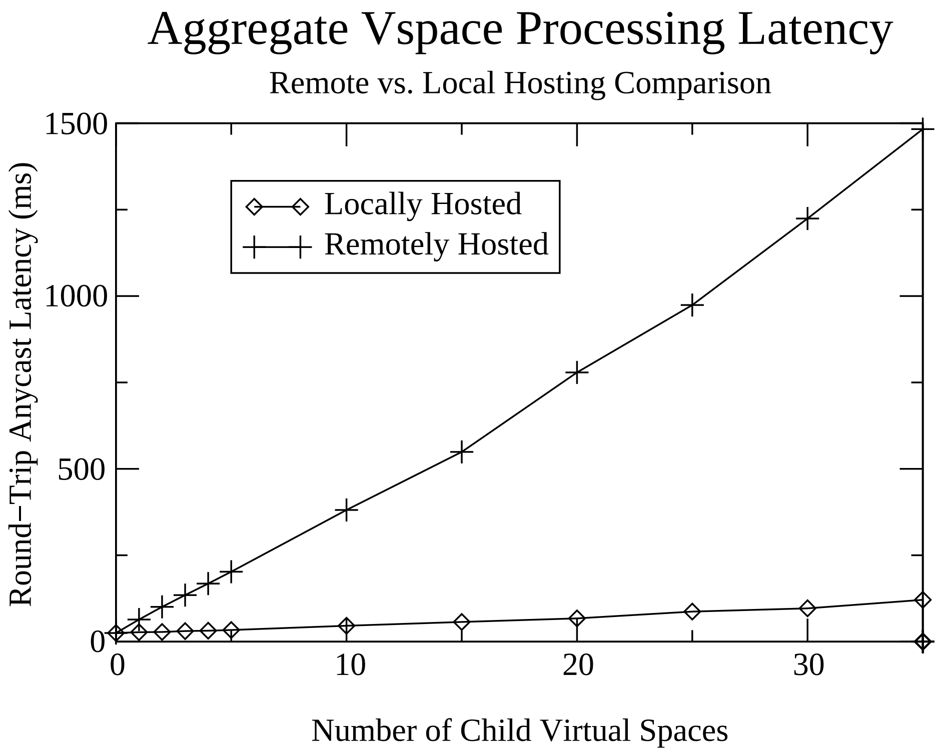

Figure 5-4: When the child vspaces of an aggregate virtual space are hosted on separate machines, requests to each child vspace need to be sent over the network. The latency grows much faster for the remote than for the local case. (The local-hosted line on the bottom is the scaled version of the Figure 5-3 graph.)

the increased time represents only the extra overhead of checking another nametree. These child nametrees are minimal in size, so that essentially all that is counted is the overhead of cycling through the different children.

We note both the starting point and the slope of the line in the graph. The figure starts at 25.0 ms with a slope of approximately 2.1 milliseconds per additional child vspace. From the standpoint of aggregate vspaces, we see there is a cost to the convenience of vspace unions, since in this case adding 12 minimal child vspaces doubles the routing cost. But, if using such a aggregate vspace could cut advertisement bandwidth by the same factor of 12 and at least $\frac{2}{12} = \frac{1}{6}$ of the traffic consisted of ad data, this could provide a net gain performance-wise.

This also gives an idea of the overhead of virtual spaces. Searching an extra child virtual space's locally-hosted nametree adds 2.1 ms per round trip, or about

1.1 ms per single lookup. This shows the relative magnitude of the time it takes to go through the INR's internal vspace table and minimal nametree compared with the rest of the routing process.

In Figure 5-4, we look at the results from a similar experiment, except that the child vspaces are hosted remotely on different INRs. Here, in order to send an anycast message, all of the child vspaces must be queried over the network, rather than locally. The latencies are much greater for the remote case than the local case, which is amplified by observing the locally-hosted line on the botton which was scaled from Figure 5-3.

The graph's slope for the remotely-hosted case is approximately 35.6 ms per extra virtual space, which dwarfs the previous 2.1 ms local figure. The ad traffic relative to usage needs to be comparatively much more significant than with the locally-hosted case to make a strict performance justification. But, the extra functionality of being able to view a number of vspaces as a coherent set, particularly when the set grows very large, may very well be worth it in some situations.

This slope number of 35.6 ms also gives an idea of how long the remote early-binding or discovery message takes, since anycast uses a standard early-binding message to figure out the best entry in a child vspace. When this 35.6 ms number is compared to the base 25.0 ms number of an anycast round trip, the difference, about 10.6 ms, indicates that a fair amount of the early-binding processing time is spent doing name lookups as well as packing and unpacking data.

5.1.4 DSR Performance

As with the INR, we examine the DSR's performance quantitatively. The Domain Space Resolver has the advantage that it is much simpler than the INR, updates less frequently, and does not require expensive partial matches or an especially rich naming syntax. We looked at two areas: the time required to integrate an INR's advertisement into the system and the response latency for a query.

These tests were performed with different numbers of virtual spaces and INRs. For example, the system might have 200 virtual spaces, 5 vspaces hosted on each

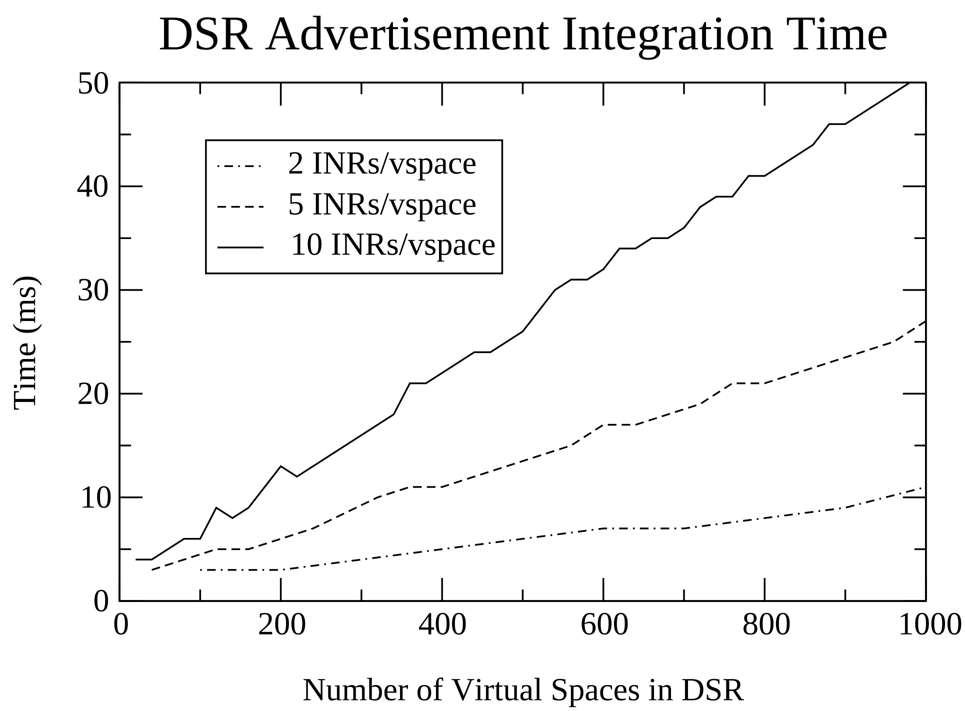

Figure 5-5: The amount of time the DSR takes to integrate a periodic vspace advertisement message from an INR, given various load characteristics.

INR, and 5 times as many INRs as vspaces. The last two figures, vspaces-per-INR and INRs-per-vspace, were parameterized with the same number to simplify graphing the problem, though this tends to produce a greater than normal DSR load.

First, a number of AdvertiseVspaces advertisement messages were sent to the DSR to build up usable state. The metric to evaluate here is the advertisement integration time—how much of the DSR's time is occupied processing each INR announcement? These tests were conducted on a single machine using the loopback interface so that the figure would primarily indicate server load rather than network-induced latency. The advertisement messages are generated to simulate INRs, so if the parameters were 200 virtual spaces and 5 vspaces per INR, each advertisement would consist of 5 randomly chosen vspaces out of the 200. Since INRs-per-vspace is parameterized the same as vspaces-per-INR, there would be $200 \times 5 = 1000$ simulated INRs giving these messages to the DSR.

We observe the performance of this advertisement test in Figure 5-5. The parameters included up to 1000 virtual spaces and values of 2, 5, and 10 INRs-per-vspace and vspaces-per-INR, with drastically outlying results discarded. From the first glance, it appears that the number of virtual spaces drives the growth in query time, from under 5 ms for 100 vspaces to 50 ms for 1000 virtual spaces in the 10 INRs-per-vspace case. However, we find that the number of INRs is a more accurate predictor for the query time than the number of vspaces. For instance, at the 1,000 vspaces mark, we find that the performance is something like a 2:5:10 ratio, but we also note that there are in fact 2,000, 5,000, and 10,000 INRs being simulated in the system that the time. And, looking across the three data sets, the times are nearly equivalent for the same number of effective INRs; when the 5 and 10 INRs-per-vspace lines are at he points which yield them 5,000 INRs, their times are 27 ms and 26 ms, respectively.

This performance limit, which is linear with the number of INRs, could probably be improved upon by adding more associative data structures to the DSR. However, the current times, particularly when the tests go to the scope of 10,000 INRs, are reasonable for any forseeable deployment of INS.

Additionally, we examined the actual lookup times once this advertise information had been built up in the DSR. Here, we find that the number of virtual spaces does matter, but not as much as the amount of the information being returned. In Figure 5-6, we plotted the time for a VspacesResolvers request to return the INRs that host a randomly-chosen vspace. The slopes of the lines are very small—on the order of 1.6 μs per additional vspace. The time is instead dominated by the amount of information in the virtual space. In other words, the query performance follows the 2:5:10 ratio that we would expect from having 2, 5, or 10 times as many INR entries in a given virtual space.

Having the latency vary linearly with the amount of information in the virtual space is a good sign for the scalability of the DSR unit, especially when it is not overly affect by increasing numbers of virtual spaces. These times, as with the advertisement analysis, are more than sufficient for any single administrative domain's forseeable INS deployment.

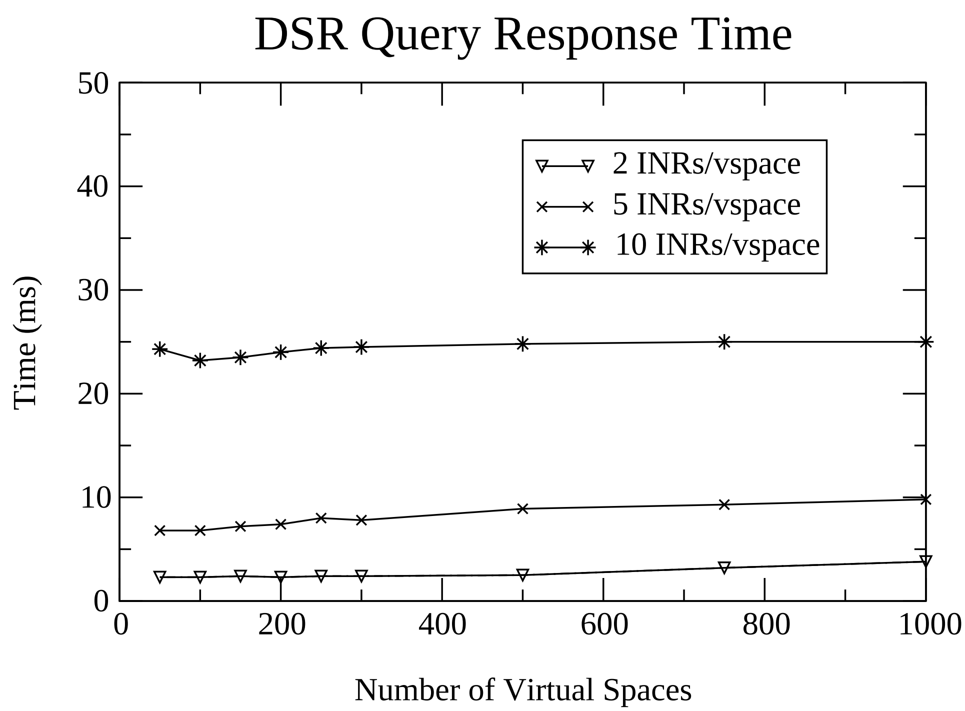

Figure 5-6: Measurements of how long the DSR takes to return a vspace hosting query, given the number of vspaces in the system and the number of INRs per vspace/vspaces per INR. Queries and INR-vspace assignments were randomly generated.

5.2 Applications

The qualitative side of evaluating the virtual space mechanism is determining its whether it is usable and whether applications can be written to take advantage of it. To this end, a number of the applications from the original INS implementation [24] were ported to take better advantage of vspaces.

The main task of integrating vspaces into applications is to add enough flexibility so that the applications can join and use multiple virtual spaces. Since only the default vspace can be accessed if the application has no knowledge of vspaces, applications will still work without modifications, but the scope of the services which they can use is limited.

Telling applications which virtual spaces they should join is accomplished in two ways: manually and automatically using the *LocationManager*.

On the manual side, many INS service providers, such as the printer gateway or the camera transmitters, already have a configuration file to store information they need. This includes data such as which printers should be monitored or how often to transmit video images. For them, the configuration files were extended so that whoever sets up the system is able to specify which vspaces the service should join. An example of an extended printer configuration file is in Figure 5-7. For a certain class of applications, particularly system-level ones, this type of approach makes sense.

#	**name**	**location hierarchy**	**vspace name**
printer	turkey	location/lcs/floor/5/room/517	lcsprinters
printer	turkey	location/floor5/room/517	lcsfloor5
printer	turkey	location/mit/building/ne43/room/517	mitprinters:mit.edu
printer	gi	location/lcs/floor/5/room/517	lcsprinters
printer	gi	location/floor5/room/517	lcsfloor5
printer	gi	location/mit/building/ne43/room/517	mitprinters:mit.edu

Figure 5-7: Some applications, such as the printer gateway, require some manual configuration, such as which printers to monitor. These configurations can be extended so that services may manually join the applicable vspaces.

The LocationManager is a more automatic means for telling the application which virtual space it should join. If the manager is running on the local machine, it can be queried for a name-specifier that describes the host's location. This includes a branch starting with the `location` attribute and the local virtual space. For instance, a query could find the following location:

```
[location=lcs[floor=5[room=517]]][vspace=lcs]
```

From this location, the application can tell that the local virtual space is *lcs*, join the vspace, and look for nearby services. Even if no physical location-finding infrastructure [4] is in place, the LocationManager on a machine can be told the current location and vspace to avoid the need for per-application configuration. Changes in the Client Library allow easily getting this location information.

With these means of obtaining vspace configuration information, the Floorplan application suite [24] was ported to take advantage of the virtual spaces and to use the Client Library. A screenshot of the Floorplan application viewing and using some

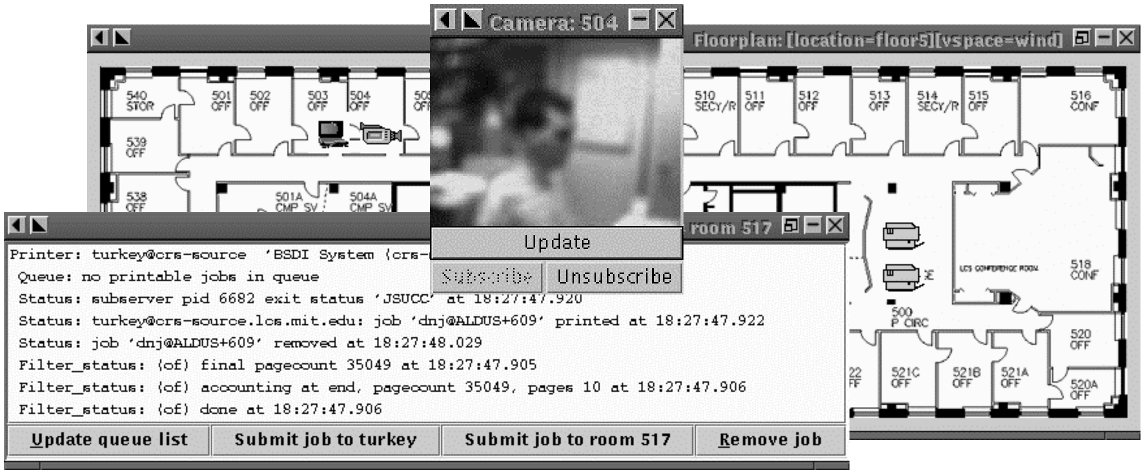

Figure 5-8: The Floorplan application, including mobile camera and printer services, implemented in the virtual space architecture.

of the nearby services is in Figure 5-8. This graphical floorplan browser depicts local services, including the mobile camera application and printer application that are shown. Different floors and vspaces may be hyperlinked as well.

Like the previous version, this floorplan application must be started in some initial location. This information can, however, be obtained from the LocationManager. Unlike the previous version, instead of assuming the location is defined by a single, global name-specifier branch starting with `[organization=mit[...]]`, the location is a combination a virtual space and a location within that vspace. This adds more scalable indirection to the location concept, yet avoids too tightly coupling location and vspaces. Once part of the floorplan is loaded, nearby regions can be in effect "hyperlinked" and browsed, allowing the user to experience all the elements of a connected web of services.

5.2.1 Summary

In the process of working virtual spaces, we find that they are a useable mechanism for writing applications, including the Floorplan browser and some related applications. For the purposes of Floorplan, the idea of vspaces integrated well with location and the automatic LocationManager could be written to take advantage of that, but this does not need always be true. With an application-defined partitioning mechanism like vspaces, the partitioning lines can be set up in a way that does not unduly interfere with how application writers build the system. Because of this, we observe that an application-defined partitioning scheme like the vspace mechanism is a usable way to scale a system.

Chapter 6

Conclusions

Service discovery systems are becoming increasingly necessary and important as mobile services and devices continue to be deployed at a frantic rate and as resource sets become more dynamic. They allow services to be efficiently discovered and used even as the available services frequently change.

However, most existing systems are only designed to work well in the local area with a small to medium number of services. As more sets of services and larger regions are encompassed by the system, it eventually starts to break. Name advertisement bandwidth to maintain state across large numbers of nodes and internal data structures become limiting factors, and the schema may become too large and unstructured to be manageable.

We explored different approaches for scaling such a system in and between domains, including building a master global namespace, keying off rigid factors such as location or service type, and using compression algorithms for transferring the data more efficiently. However, we found a scaling scheme based on application-defined namespace partitioning is best adapted to the specifics of this situation. In this case, it involves treating the `vspace` attribute as special and partitioning the namespace into autonomous virtual space communities. Given that these virtual spaces follow "natural boundaries" of the world, they function as a usable building block in a system of services. In doing this, they can cut total advertisement bandwidth by an order of $O(k)$ to $O(k^2)$, depending on topology, and they have a minimal overhead. Usability features such as virtual space hierarchy, default vspaces, and different vspace discovery mechanisms provide a greater level of flexibility in the INS system.

In addition, with domain-qualified vspaces, it is possible to use INS services in the wide area. Administrative domains may be bridged with a minimum of cooperation and may each have independent namespaces. Future work in this area includes potentially adding further layers of abstraction to reference services outside the local scope.

This virtual space mechanism provides a simple and elegant solution to many of the scalability problems in INS. It improves intra-domain scalability by reducing advertisement bandwidth and enables wide-area operation, with a low overhead, by partitioning the namespace. We believe this type of approach shows promise in other service discovery and naming systems.

Bibliography

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, SC, December 1999.

[2] William Adjie-Winoto. A self-configuring resolver architecture for resource discovery and routing in device networks. Master's thesis, MIT, May 2000.

[3] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[4] Anit Chakraborty. Distributed architecture for mobile, location-dependent applications. Master's thesis, MIT, May 2000.

[5] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM*, August 1988.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1997.

[7] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An Architecture for a Secure Service Discovery Service. In *Proc. ACM/IEEE MOBICOM*, pages 24–35, August 1999.

[8] W. Keith Edwards. *Core Jini.* Sun Microsystems Press, 1999.

[9] L. Fan, P. Cao, J. Almeida, and A Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proc. of SIGCOMM '98*, 1983.

[10] Google. http://www.google.com/, 2000.

[11] A. Gulbranssen, P. Vixie, and L. Esibov. *A DNS RR for specifying the location of services (DNS SRV)*. Internet Engineering Task Force, February 2000. RFC 2782.

[12] E. Guttman, C. Perkins, J. Veizades, and M. Day. *Service Location Protocol, Version 2*. Internet Engineering Task Force, November 1998. RFC 2608.

[13] The IEEE 802.11 Wireless LAN Standard. http://www.wlana.com/intro/standard/.

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

[15] Riku Mettala. Bluetooth Protocol Architecture. http://www.bluetooth.com/developer/whitepaper/whitepaper.asp, July 1999.

[16] P. V. Mockapetris and K. Dunlap. Development of the domain name system. In *Proc. of ACM SIGCOMM '88*, August 1988.

[17] Northern Light. http://www.northernlight.com/, 2000.

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

[19] J. B. Postel. *Transmission Control Protocol*. Internet Engineering Task Force, September 1981. RFC 793.

[20] S. Raman and S. McCanne. Scalable Data Naming for Application Level Framing in Reliable Multicast. In *Proc. ACM Multimedia*, September 1998.

[21] J. Reynolds. *Technical Overview of Directory Services Using the X.500 Protocol*. Internet Engineering Task Force, March 1992. RFC 1309.

[22] J. Rosenberg, E. Guttman, R. Moats, and H. Schulzrinne. WASRV Architectural Principles. Internet Draft draft-rosenberg-wasrv-arch-00.txt, February 1998. Work in progress.

[23] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[24] Elliot Schwartz. Design and implementation of intentional names. Master's thesis, MIT, May 1999.

[25] Socks home page. `http://www.socks.nec.com/`, 1997.

[26] R. Stevens. *TCP/IP Illustrated, Volume 1.* Addison-Wesley, Reading, MA, 1994.

[27] Universal Plug and Play. `http://www.upnp.org/`, 2000.

[28] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol.* Internet Engineering Task Force, June 1997. RFC 2165.