An Architecture for Intentional Name Resolution and Application-level Routing

William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan

Laboratory for Computer Science
Massachusetts Institute of Technology
{wadjie,elliot,hari}@lcs.mit.edu

February 5, 1999

Abstract

Today's Internet naming scheme, the Domain Name System [28], implicitly assumes that applications want to reach an address, where the address signifies location in the network topology. Typically, applications desire either information or functionality, and do not often know the best network location that satisfies their needs. We argue that current efforts to efficiently enable new services such as mobility, group communication, resource discovery, service location, caching, etc. have been greatly hampered by the lack of a flexible naming system and the inability of the name resolution process to affect data routing decisions. Significant effort is spent in creating independent, but similar infrastructure for each situation.

This paper presents the design and implementation of an intentional network naming architecture, where applications describe what they are looking for (i.e., their intent), not where to find it. In this architecture, name resolvers can also route messages to the eventual destinations, leading to an integrated approach to resolution and routing. We present efficient data structures for maintaining intentional names based on attribute-value tuples, efficient algorithms for name lookups, and discovery protocols for disseminating name information among resolvers and end-hosts. We analyze the performance of the algorithms and present measurements of the system implementation, which show that our architecture is practical and feasible. We also describe a sample application—a mobile, wireless camera application for remote surveillance—demonstrating the utility of the architecture in supporting mobility, group communication, service location and data caching.

1 Introduction

In the last several years, we have seen the Internet grow phenomenally in terms of number of users, size, traffic, and applications. Thus, it now faces a different set of demands and challenges than it did originally—in particular, a demand for better support for the efficient deployment and performance of new services. People and the applications they use are demanding features such as mobility, caching, load balancing, replication, service location, resource discovery, and group communication. While many of these problems are receiving attention in the networking community, we believe that existing proposals to solve these problems tend to be overly specific or unnecessarily complex.

We argue that current efforts to efficiently enable new services have been greatly hampered by the inflexibility of the naming system and the inability of the name resolution process to affect data routing decisions. Significant effort is spent in creating independent, but similar infrastructure for each situation. We observe that for a number of these services and applications, a more flexible and integrated naming and routing architecture will greatly simplify and often solve the hard problems that arise, because at a fundamental level, all these problems are made easier with a "level of indirection" in the way messages are routed in the network. Motivated by these observations, we strive to provide the right, general level of indirection via the name resolution system.

Today, most network naming schemes such as the Domain Name System (DNS) [28] implicitly assume that applications want to reach an address, where the address signifies location in the network topology. Typically, applications desire either information or functionality, and do not often know the best network location that satisfies their needs. We therefore argue that what is needed is an *intentional* naming scheme and resolution architecture in which applications describe *what* they are looking for, not *where* to find it. Furthermore, we advocate that the name resolvers also participate in message routing based on intentional names, thereby integrating name resolution and routing, which until now have been kept separate in the Internet. This integration, implemented by including application payload with the name resolution request[1], allows the "late binding" (i.e., binding at packet delivery time) between the network routes to the end nodes and the name that maps on to those nodes. This enables users and applications to track changes easily, including host mobility, dynamic re-

[1] Name resolvers treat the application payload as opaque data.

sources, and changing data at network nodes.

The main contribution of our work is the design and implementation of an *Intentional Name System*, called *INS*. In this paper, we describe three aspects of INS in detail:

- **Intentional naming scheme.** We present a naming scheme where applications express the characteristics of the information or nodes they want to reach as query expressions in a restricted query language. In particular, we show that significant benefits can be obtained using an *exact match* operator in the language, which enables resolution based on matching variables in an incoming name (i.e., attributes) to one among a set of values by the resolvers. These variables are under application control and not pre-defined. The resolvers can perform resolution without understanding the semantics of specific applications.
- **Intentional Name Resolvers (INRs).** We present a resolution architecture composed of a self-organizing network of INRs to resolve intentional names and route messages. Our architecture incorporates efficient algorithms for name lookups based on exact matches and a low-latency discovery protocol for disseminating name information among end nodes and resolvers. We analyze these algorithms and discuss experimental performance results from our implementation to justify the feasibility of our ideas.
- **Application architecture.** We demonstrate the INS application architecture by describing the design and implementation of a mobile, wireless camera application for remote surveillance over INS. We discuss how this application easily leverages INS's API and automatic support for mobility, group communication, service location and data caching, gaining these advantages from INS without any other pre-installed support for these different services.

A key feature of our architecture is its potential for incremental and easy deployment in the Internet, without changing or supplanting the existing Internet service model. We achieve this by designing the resolvers to communicate with each other tunneled over an IP network, using well-understood Internet routing protocols to route messages between resolvers. Our experience with this demonstrates that a variety of new services can be deployed effectively by our extensions to Internet naming and resolution, without requiring active, general-purpose computation in either the routing [38] or naming [40] subsystems of the Internet architecture.

We hasten to note that the INS architecture presented in this paper is *not* intended for a network as large as the global Internet. Rather, it is intended for networks on the order of few hundred or few thousand nodes (e.g., inside an administrative domain), much like schemes for intra-domain unicast (e.g., OSPF [29]) or multicast routing (e.g., DVMRP [11]). We are actively developing a wide-area architecture to complement our intra-domain INS architecture, which will integrate INS with extensions to DNS.

The rest of this paper describes our design rationale and presents the details of INS. Section 2 discusses the INS architecture, describing the intentional naming scheme, name lookup algorithm, entity discovery protocol and self-organization protocol. It also describes various optimizations and benefits of INS. Section 3 discusses the INS API and our mobile camera application. Section 4 discusses implementation details and Section 5 presents the analysis of the algorithm and the results of performance experiments based on our implementation. We survey related work in Section 6 and then conclude.

2 System Architecture

The design of the INS architecture is motivated by our desire to enable applications to express the destination (and source) of their messages using an intentional name that describes the intent of the application, rather than a specific end-point. Towards this goal, we introduce the *name-specifier*, which is used in the message header instead of the traditional source and destination addresses, to describe the intent of the application. Section 2.1 describes the components of the name-specifier, how they are assembled into an intentional name, and the wire representation of the name-specifier.

The name-specifiers are resolved into their corresponding network locations by Intentional Name Resolvers or INRs. INRs communicate with each other and applications in an arbitrary topology of tunnels overlaid on the IP network. Rather than having statically configured relationships, as is common in other overlay networks [13, 17], a self-organization protocol is used to spawn and terminate INRs, and maintain neighbor relationships; this protocol is described in Section 2.4.

To learn and share information about names, the INRs communicate via a name discovery protocol. The protocol uses periodic updates to convey name information, and uses triggered updates for fast changes. In addition, we discuss a novel optimization to implicitly learn about names by inferring information from message headers. These issues are detailed in Section 2.3.

The central activity of INRs, of course, is to resolve name-specifiers into their corresponding network locations. INRs support two methods of name resolution: *early binding*, in which the INR returns a handle to the end-hosts (typically a set of IP addresses), and *late binding*, in which the INRs forward data on behalf of the application, deferring the binding of the name-specifier to the end-host until just before the data is delivered to its final destination. Late binding enables highly dynamic name bindings, since the application is never left with a stale binding even if bindings change while the message is in transit. We focus our attention on the late binding case in this paper. Section 2.2 describes the INR namespace, the *name-tree*—a data structure used to store name information, and the algorithm used to look up intentional names in the name-tree.

2.1 Name-Specifiers

There are many ways to implement intentional names; in INS we use query expressions called name-specifiers that replace traditional addresses in packet headers. Our design decisions are based on the idea that the name-specifier should provide a flexible and powerful, yet efficient method of selecting names; we were also motivated by the desire to keep name-specifiers simple and easy to understand.

The two main concepts of the name-specifier are the *attribute* and the *value*. An attribute is a category in which an object can be classified, for example its 'color.' A value is the object's classification within that category, for example, 'red.' Attributes and values are free-form strings that are defined by applications; name-specifiers do not restrict applications to using a fixed or predefined set of attributes and values. Together, an attribute and its associated value form an *attribute-value pair*.

Name-specifiers are a hierarchical arrangement of attribute-value pairs. The pairs are arranged in a tree such that a pair that is *dependent* on another is a descendant of it. For instance, in the example name-specifier shown in Figure 1, it only makes sense to talk about a building called the Whitehouse if you are referring to the city of Washington, so the attribute-value pair `building=whitehouse` is dependent on the pair `city=washington`. Pairs that are *orthogonal* to each other, but dependent on the same pair, are siblings in the tree. For example, a digital camera's data-type and resolution can be selected independently of each other, are meaningful only in the context of the camera service. Therefore, the pairs `data-type=picture` and `resolution=640x480` are orthogonal. This hierarchical arrangement narrows down the search space during name resolution, and makes name-specifiers easier to understand.

A simpler alternative would have been to construct a hierarchy of attributes, rather than one of pairs. This would result in `building` being directly dependent on `city`, rather than `city=washington`. However, it is also less flexible; our current hierarchy allows child attributes to vary according to their parent value. For example, `country=us` has a child that is `state=virginia`, while `country=canada` has a child that is `province=ontario`.

To include it in the header to describe the source and destination of a message, the name-specifier has a wire representation as shown in Figure 2. This string-based representation was chosen to be readable to assist with debugging, in the spirit of SMTP [35], HTTP [16], NNTP [23], etc. Levels of nesting are indicated by the use of brackets ([and]), and attributes and values are seperated by an equals sign (=). The arbitrary use of whitespace is permitted anywhere within the name specifier, except in the middle of attribute and value tokens.

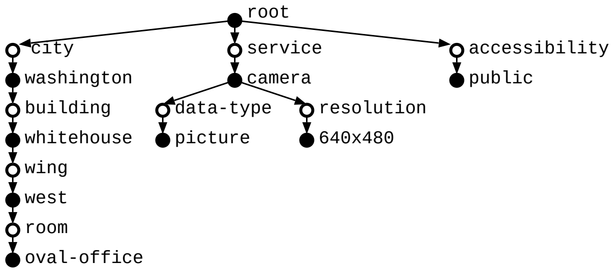

Figure 1: A graphical view of an example name-specifier. The hollow circles are used to identify attributes; the filled circles identify values. The tree is arranged such that dependent attributes are descendants, and orthogonal attributes are siblings. This name-specifier describes a public-access camera in the Oval office.

```
[city = washington [building = whitehouse
                   [wing = west
                   [room = oval-office]]]]
[service = camera  [data-type = picture
                       [format = jpg]]
                   [resolution = 640x480]]
[accessibility = public]
```

Figure 2: The wire representation of the example name-specifier shown in Figure 1, with line-breaks and extra spacing added to improve readability.

In addition to exact value matches, name-specifiers also permit wildcard matching of values. To do this, the value is simply replaced by the wildcard token, a star (*). Thus to construct a name-specifier that refers to *all* public cameras providing 640x480 pictures in the Whitehouse, not just the one in the Oval Office, an application replaces the value `oval-office` with '*' in the name-specifier shown in Figures 1 and 2. The application also sets the *anycast/multicast flag* to choose whether the packet should be sent to *all* cameras or just *any* one camera. We are currently investigating the use of inequality operators ($<$, $>$, $\leq$, and $\geq$) to provide range selection operations in name-specifiers.

2.2 Name Resolution and Message Routing

The central activity of an INR is to resolve name-specifiers to their corresponding network locations. When a message arrives at an INR, the INR performs a lookup on the destination name-specifier in its name-tree. The lookup returns information that includes a set of "routes" to next-hop INRs, as well as the IP addresses of final destinations. If the application has chosen to use early binding by setting the *early-binding flag*, the INR simply returns the IP addresses to the application. If the application desires late binding, the INR forwards the message to the next-hop INRs without making any changes to the name-specifiers or data. This forwarding continues until the message reaches its final destinations, providing a late binding between the destination name-specifiers and their respective IP addresses.

Name-trees. Name-trees are a data structure used to store the correspondence between name-specifiers and *name-info* records. The principal information that the name-info records contain are the routes to the next-hop INRs and the IP addresses of potential final destinations. The records also store additional information such as the metric for the routes and the expiration time of the record.

Not surprisingly, the structure of a name-tree bears a close resemblance to a name-specifier. Like a name-specifier, it consists of alternating levels of attributes and values; but unlike a name-specifer there can be multiple values per attribute, since the name-tree is a superposition of all the name-specifiers the INR knows about. Each of these name-specifiers has a pointer from each of its leaf-values to a name-info record. Figure 3 depicts an example name-tree, with the example name-specifier from Figure 1 in bold.

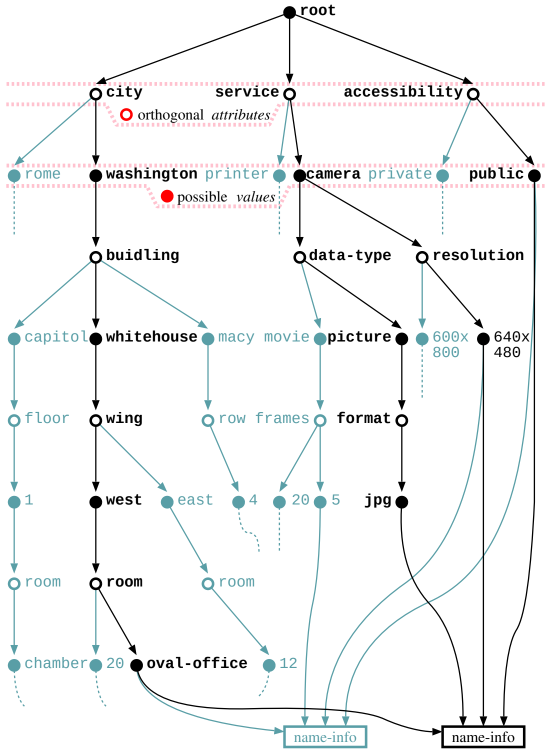

Figure 3: A partial graphical view of an example INR name-tree. The name-tree consists of alternating layers of orthogonal attributes and their possible values. Leaf-values contain pointers to all the routes they correspond to. The part of the name-tree corresponding to the example name-specifier shown in Figure 1 is in bold.

Name lookups. The LOOKUP algorithm, shown in Figure 4, is used to retrieve the name-info records for a particular name-specifier n from the name-tree T. The main idea behind the algorithm is that a series of recursive calls reduce the candidate name-info set S by intersecting it with the name-info set consisting of the records pointed to by each leaf-value. When the algorithm terminates, S contains only the relevant name-info records.

The algorithm starts by initializing S to the set of all possible name-info records. Then, for each attribute-value pair of the name-specifier, it finds the corresponding attribute in the name-tree. If the value in the attribute-value pair is a wildcard, then it computes S' as the union of all name-info records in the subtree rooted at the corresponding attribute, and intersects S with S'. If not, it finds the corresponding value in the name-tree. If it reaches the leaf of either the name-specifier or the name-tree, the algorithm intersects S with the name-info records pointed to by the corresponding value. If not, it makes a recursive call to compute the relevant set from the subtree rooted at the corresponding value, and intersects that with S.

Section 5.1 analyses this algorithm and discusses the experimental results of our implementation.

2.3 Name Discovery

INRs learn about names by participating in a name discovery protocol with their neighbor INRs and end-nodes. Through this peer-to-peer protocol, the associations between name-specifiers and name-info records propagate throughout the INR network and to end-nodes and applications. These associations are obtained from the name-tree using an extraction algorithm, and are then disseminated either during periodic or triggered updates. The INRs also employ a novel optimization to learn new names, which we call *inference*. The following paragraphs detail these topics.

Name extractions. Since the INR name-tree is a superposition of all the name-specifiers the INR knows about, extracting a single name-specifier to advertise is non-trivial. The EXTRACT algorithm, shown in Figure 5, is used to retrieve the name-specifiers for a particular name-info record r from the name-tree T. The main idea behind the algorithm is that a name-specifier can be reconstructed while tracing upwards to the root of the name-tree from each pointer to a name-info

```
LOOKUP(n,T)
  S := the set of all possible name-info records
  for each attribute-value pair p := (n_a, n_v) in n
    T_a := the child of T such that
             name(T_a) = name(n_a)
    if n_v = *                        ▷ wildcard matching
      S' := ∅
      for each T_v which is a child of T_a
        S' := S' ∪ all of the name-info records in the
                       subtree rooted at T_v
      S := S ∩ S'
    else                              ▷ normal matching
      T_v := the child of T_a such that
               name(T_v) = name(n_v)
      if T_v is a leaf node or p is a leaf node then
        S := S ∩ the name-info records of T_v
      else
        S := S ∩ LOOKUP(p, T_v)
  return(S)
```

Figure 4: The LOOKUP algorithm. This algorithm looks up the name-specifier n in the name-tree T and returns all appropriate name-info records.

```
EXTRACT(r,T)
  set all PTRs in the tree rooted at T to null
  T.PTR := a new, empty name-specifier
  for each v which is a parent value element of r
    EXTRACT-TRACE(v, null)
  return(T.PTR)

EXTRACT-TRACE(v,s)
  if v.PTR != null            ▷ something to graft onto
    if s != null              ▷ something to graft
      graft(s, v.PTR)
  else                        ▷ nothing to graft onto; make it
    v.PTR := a new attr.-value pair consisting of
               this value and its parent attribute
    if s != null              ▷ something to graft
      graft(s, v.PTR)
    EXTRACT-TRACE(parent value of v, v.PTR)
```

Figure 5: The EXTRACT algorithm. This algorithm extracts and returns the name-specifier for the name-info record r in the name-tree T. EXTRACT-TRACE implements most of the functionality, tracing up from a leaf-value until it can graft onto the existing name-speficier.

record, and grafting on to parts of the name-specifier that have already been reconstructed.

All the values in the name-tree, T, are augmented with a "PTR" variable, which is a pointer to the corresponding attribute-value pair in the name-specifier being extracted. Initially, all the PTRs are set to null, since they have no corresponding attribute-value pairs; the root pointer (T.PTR) is set to point to a new, empty name-specifier. Then, for each parent value of r, the algorithm traces upwards through the name-tree. If it gets to part of the name-tree where there is a corresponding attribute-value pair (v.PTR != null), and it has a name-specifier subtree to graft on to (s != null), it does so. If not, it creates the corresponding part of the name-specifier, sets v.PTR to it, grafts on s if applicable, and continues the trace with the parent value of v and the new subtree. Figure 6 illustrates the progress of the algorithm.

Updates. INRs use updates to keep each other informed of the name-specifiers they know about. Triggered updates occur when an INR receives an update from one of its neighbors (either an INR or an application) that causes a change in its name-tree; this allows new advertisments to propagate through the network rapidly. Periodic updates are used to prevent the aging out of data that has not changed and to refresh entries in neighboring INRs. This combination of periodic and triggered updates enables us to treat the disseminated name state as *soft* [9], and therefore does not require a fully reliable transport protocol such as TCP.

INRs use the Bellman-Ford algorithm [4] to calculate the shortest distance to the end-nodes. Unlike traditional routing protocols that use the algorithm [20, 27], the INS architecture does not require unique end-nodes—if a name is advertised from more than one location, the algorithm computes the best overall metric based on INR hop count.

Inference. Here, INRs learn about new names by passively observing the headers of messages they receive. When an INR receives a message that is travelling from a to b, in addition to forwarding it towards b, it also adds a name-info record to the name-tree for a, noting that its next-hop INR is the INR the message arrived from. The metric for the name-info record is found by looking at the *up-counter* field in the header, which is incremented as the message travels from source to destination. Learning about routes via inference is especially important for learning about clients who have just made a request in large networks. Using inference, only the INRs on the path from the source to the client need to learn about this client.

2.4 INR Self-Organization

INR machines are not static, pre-configured servers, but are dynamic, in order to reflect load, node locality and the need for efficient routing in the face of mobility. We achieve this with a self-organization protocol that spawns INRs as needed, forms neighborhoods of active nodes, and kills existing INRs when they are no longer useful. Being highly distributed with no

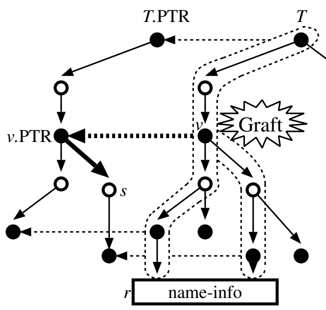

Figure 6: An illustration an in-progress execution of the EXTRACT algorithm. The subtree s is being grafted onto v.PTR. The traced paths and PTRs are shown with dotted lines; the PTR being used and graft are shown in bold.

centralized control, it has the potential to scale with increasing load and perform well even when failures or partitions occur.

INRs run on machines in the general computing infrastructure of an organization; typically there is a pool of machines that are candidate INRs, of which a subset are active at any time. The administrators of these machines can limit the amount of processing and communication expended by INR on a node. This means that nodes can join and leave the active INR set at any stage based on external conditions

Bootstrapping is based on a list of candidate INRs that is available to applications (e.g., from the DNS). If an application detects that none of these candidate nodes are active, it can spawn an INR on any of these nodes. This is especially useful for efficient *ad hoc* mobile networking in remote locations, such as a meeting room or network isolated from the rest of the Internet.

Neighbors are dynamically maintained by a SOLICIT/ACCEPT protocol. The neighbor relationship in our design is explicitly designed to be symmetric. This simplifies routing by eliminating the need to deal with uni-directional paths and allowing the use of inferred routes.

As the load increases on a resolver, additional INRs are spawned on other candidate nodes, and INR functionality can be terminated at any time (e.g., if the load is too light or heavy). Our design uses a probabilistic birth/death algorithm with local load monitoring for this; this algorithm is similar to the one described by Amir *et al.* in the context of an Active Service framework [1]. It has the desirable property that all nodes make autonomous decisions to achieve good global behavior.

We have currently not implemented the self-organization protocol, but expect to do so shotly.

2.5 Benefits and Optimizations

In this section we discuss some of the benefits that our architecture offers and the optimizations one can make because of

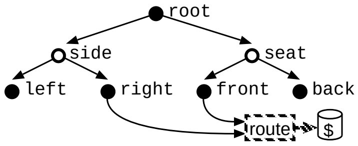

Figure 7: A name-tree with a cached copy of packet data. The cached data is indicated by a '$'.

them. These include caching, group communications, mobility, and resource discovery.

Caching. The main difficulty with implementing middle-of-the-network caching in today's Internet is that packets aren't named in a way that is both application-independent and reusable. For example, an IP address / TCP port number / TCP sequence number can be used to identify a packet in the middle of an HTTP transfer, but it isn't reusable since there is no permanent mapping from sequence number to the source data. To make this reusable, one has to sacrifice the application-independence of the name by interpreting the HTTP transfer.

Intentional names give each packet a reusable, application-independent name—the source name-specifier. Thus, adding caching requires only two modifications to the INR behavior:

1. When transmitting a packet, make a copy of its data and store a pointer to it under the route entry for the *source name-specifier* in its name-tree.
2. When looking up the *destination name-specifier* of a packet, check to see if there is a cached copy of the data which could be sent instead.

An illustration of a name-tree from such an INR is shown in Figure 7; in addition to the regular route, it also has a cached copy of the data. The *cache-TTL* field of the packet indicates how long the data is valid and is also stored in the name-tree. INRs may also inform their neighbors that they have a cached copy by using regular name advertisements.

Group communication. Despite the development of multicast routing (detailed in [10, 11, 3, 12, 24]) and a proposal for anycast routing ([32]), unicast transmission remains the predominant communication paradigm of the Internet. In contrast, INS makes no assumption that any service resides on any particular host, or even on only a single host. In particular, our name lookup algorithm uses set operations to determine the correct routes, rather than just looking for a single route. This allows INS to easily support features akin to multicast and anycast, with unicast merely being a special, single-host case of either of those.

Since intentional names represent a service rather than an end-point, multiple hosts can announce that they are providing the service. Rather than just choosing the best route towards a name-specifier, the INRs maintain a list of all of the routes towards that service. When the INR forwards a packet, it checks the *anycast/multicast flag* of the packet header to decide how to handle it. If the flag is set to *anycast*, it forwards it to the neighbor with the *best* metric; if it is set to *multicast*, it forwards it to *all* neighbors for whom it has a route. This anycast/multicast feature is useful for implementing mirrored or redundant services and load balancing. It is further enhanced by the ability to use wildcards in intentional names, as described in Section 2.1.

Mobility. Mobility in the Internet relies upon a complex Mobile IP scheme that involves the deployment of Home and Foreign Agents [33]. This compexity arises because of the overloading of the IP address as both a permanent endpoint identifier, and a topologically sensitive address that must change as a host moves through the network. Another possible solution is to use a naming system to provide a permanent identifier, and then update the naming system as the host moves. DNS cannot easily accomplish this since it is statically configured, though dynamic updates in the DNS improve the situation [42].

INS, on the other hand, is explicitly designed for rapid updates. Periodic updates ensure that the name-tree maintains long-term consistency, while triggered updates allow announced changes to occur almost immediately. Thus when a host moves, all it has to do is announce the intentional names for the services it provides to its new neighbors, and these are quickly propagated through the INR network. This feature can also be used to implement *service mobility*, which can be used to move services from one machine to another (perhaps to allow upgrades), or even among many machines (perhaps to have a service follow a particular person). To accomplish this, the hosts just coordinate the passing of the intentional name announcement among each other.

Resource discovery. Intentional names are inherently a method of resource discovery: rather than specifying the host it wants to access, users and applications convey their intent by supplying an intentional name that describes the service they desire. INRs learn about the services that exist in the network via the name discovery protocol, and either pass this information on to applications in the form of a handle, or simply pass the application data on to the service provider on its behalf. The passing of a handle is similar to the operation of the Service Location Protocol (SLP) [41, 34]. INS is designed for a similar scale network, but operates without the use of a centralized Directory Agent or any other single point of failure.

3 Using the System

3.1 Application Programming Interface

The INS API provides a flexible framework for developing applications that take advantage of its application-controlled name resolution. It provides functions for creating and manipulating intentional names (name-specifiers), advertising and discovering new services, and leveraging INS support for caching, anycast, group communication and late binding.

In the application, a name-specifier is represented as pairs of attribute and value objects with an implicit assumption that the resolution operator is an exact match or wildcard operator[2]. The API provides functions to link these objects and connect them to other similar objects to form a complete name-specifier.

INS provides name-specifier functions to:

- add a query clause,
- retrieve a component,
- search for a query clause,
- generate a readable text representation, and
- compare another name-specifier to it.

After creating a name-specifier describing a service, the application can advertise the new service to the network using an INS function. Similarly, to discover new services, the application can use an INS function to find out whether services matching a given name-specifier have been discovered; if they have, it may communicate with them by using INS functions to construct the appropriate name-specifiers.

The INS API also allows applications to enable caching, simply by setting the length of time the message should be cached by intermediate nodes. Applications choose whether anycast or multicast is used by setting a flag in the message header.

3.2 A Mobile Camera Application

In order to evaluate the utility of our system for enabling applications that are difficult to create in today's Internet, we have implemented a mobile camera application for remote surveillance using INS. We present an example scenario from it here to reinforce system concepts and demonstrate how an application uses the API. The camera service consists of a number of physically mobile nodes equipped with cameras, each running a *transmitter* application, and a number of nodes running *receiver* applications displaying images from the remote cameras. Figure 8 shows the network topology used in this example.

The application uses name-specifiers with four orthogonal attributes: service (`svc`) identifies that it is a camera application, location (`loc`) describes its physical location, entity

[2] As we incorporate other operators such as range checks, this assumption will change.

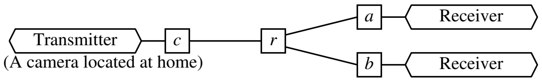

Figure 8: Network topology for the camera application example. INRs a and b run a receiver, c runs a transmitter, and r is an intermediate INR without an application.

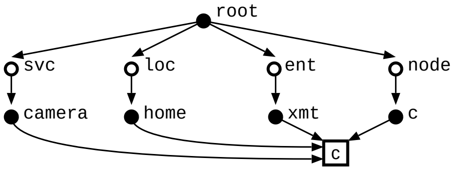

Figure 9: Name-tree at INR r after Step 2.

(`ent`) indicates whether it is a transmitter (`xmt`) or a receiver (`rcv`), and `node` is a unique identifier for each node.

The application operates as follows: First, receivers learn about available camera locations via discovery protocol updates sent by transmitters. Then users of the receiver application may request an image from a particular location. The request message will be routed to the transmitter(s) that can service a request for that location, independent of their network location and the transmitter that announced the service. Inference is used by intermediate nodes to create a path back to the receiver. Finally, the transmitter sends back a response containing the image requested.

The sequence of INS events in an example operation is:

1. The transmitter on c is started, and informs c's INR that it wants to receive all messages destined to it. The transmitter registers a route to it for name-specifier `[svc=camera][loc=home][ent=xmt][node=c]` and also tells the INR to announce this to its neighbors.

2. INR c sends a triggered-update to INR r with `[svc=camera][loc=home][ent=xmt][node=c]` in it. INR r updates its name-tree to include the new name, as shown in Figure 9. It then sends the update to INRs a and b, which update their name-trees.

3. When the receiver on a is started, it asks INR a to let it know about any name-specifiers match `[svc=camera][ent=xmt]`.

4. INR a, which has received an update from INR r for `[svc=camera][loc=home][ent=xmt][node=c]` passes this name-specifier to the receiver, which informs the user that a new camera has been discovered at `home`.

5. The user requests an image from `home`. The receiver sends a message with destination name-specifier: `[svc=camera][loc=home][ent=xmt]` and source name-specifier: `[svc=camera][loc=home][ent=rcv][node=a]`. Note that the `node` attribute is omitted from the destination name-specifier, since the receiver only wants an image from camera at home, and doesn't care which particular node supplies the image.

6. While handling the message, INR a infers that `[svc=camera][loc=home][ent=rcv][node=a]` (the *source* name-specifier of the message) is coming from the application receiver. INR a then forwards it to INR r.

7. INR r knows to forward this message to INR c since its name-tree (Figure 9) contains this route information. Using inference as before, INR r knows that the source name-specifier of the message must be in the direction of INR a, and adds this information to its name-tree. This is shown in Figure 10.

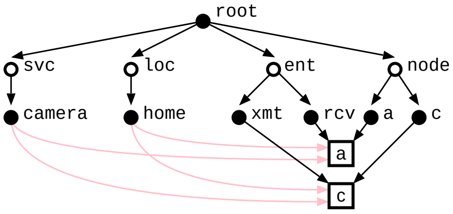

Figure 10: Name-tree at INR r after Step 7.

8. Upon receiving the message from INR r, INR c passes it to the transmitter. INR c also adds an inferred route for the source name-specifier. The transmitter notices this request and sends back a message with the requested image, destination name-specifier `[svc=camera][loc=home][ent=rcv]` and source name-specifier: `[svc=camera][loc=home][ent=xmt][node=c]`. Note that the `node` attribute is omitted from the destination name-specifier. This can be used to send it to *all* receivers who request the image, not just node a. INS uses this to perform group communication.

9. Node c receives the message from the transmitter application and forwards it to INR r. Similarly, r forwards the message to a, which then passes it on to the receiver application. All these nodes know the route to the receiver because of the inferred routes acquired during the flow of the request message from the receiver to the transmitter.

4 Implementation

The INS architecture has been fully implemented and tested using the mobile network of cameras as the test application. Our implementation of INR is in Java, to take advantage of

Message: | Header | Data |

Header: | v | p1 | p2 | p3 | IP | id | TTL | ct | af | cTTL | eb | s-ns | d-ns |

where:

v = version	ct = up-counter
p1 = pointer to s-ns	af = anycast flag
p2 = pointer to d-ns	cTTL = cache-TTL
p3 = pointer to data	eb = early-binding flag
IP = IP address	s-ns = source name-specifier
id = packet ID	d-ns = source name-specifier
TTL = time to live (TTL)	

Figure 11: The INS message format.

its easy cross-platform portability. User applications are not constrained to be written in Java.

In this section, we present the implementation details of two key aspects of INS: the architecture of an INR node, and the packet formats for intentional names.

INR node architecture. INRs use UDP to communicate with each other. At an INR, the `Node` is the manager of all network resources and running applications at a resolver. It maintains the `NameTree` that is used to resolve an intentional name to its corresponding route information, a `ForwardAgent` to forward messages, and a `NodeListener` that receives all incoming packets. In addition, there are two useful applications that run at each INR: an `EntityDiscovery` application, which implements the name dissemination protocol, and a `NetworkManagement` application that provides a graphical interface to monitor and debug the system and view the name-tree.

The INR implementation consists of approximately 5000 lines of Java code. Using the INS API, applications are relatively easy to develop. For example, the camera application was implemented in less than 1000 lines of Java, of which over 60% was for the user-interface and image display.

Packet format of intentional names. Figure 11 shows the INS packet format for intentional names. Because name-specifiers are of variable length, three pointers (`p1, p2, p3`) point to the start of the source name-specifier, destination-name-specifier and data fields of the message. INR nodes do not process application data.

The `IP` field contains the IP address of the source node, and is used by applications to perform "early binding" of intentional name to address. `id` is a monotonically increasing 32-bit unique ID for the message, used in conjunction with `IP` to detect routing loops. In addition to the standard `TTL` field, messages contain a `ct` field that counts up from zero at each INR. When an INR performs route inference, it uses `ct` as a hint to initialize the metric for the route.

The `af` field indicates whether the destination specified by the `d-ns` field is meant to be an individual ("any") entity or a group ("all") of entities. `cTTL` field stores the TTL of (optional) cached data.

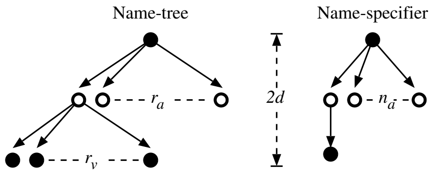

Figure 12: A uniformly grown name-tree. Note that d = (tree depth)/2 = 1 for this tree.

5 Performance Analysis and Evaluation

In this section, we analyze performance of the INS name lookup algorithm and present the results of our experiments with the lookup algorithm and name discovery protocol. Our results are encouraging and demonstrate the practical feasibility and deployability of INS.

5.1 Name Lookup Performance

Analysis. Since INS scalability with load is a major concern, it is important to analyze the performance of the lookup algorithm as the demands on it increase. While many of the tasks involved in resolving a name take the same amount of time (e.g., copying the data, transmitting it over the network), the time to perform a name lookup depends on a number of factors. It is therefore important to determine the worst-case run-time of the algorithm.

To simplify the analysis of our lookup algorithm, we assume that name-specifiers grow uniformly in the following dimensions (illustrated in Figure 12):

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

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

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

where t_a and t_v represent the time to find the attribute and value respectively. For now, assume that it takes time b for the base case such that:

$$T(0) = b$$

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

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

$$= \quad \Theta(n_a^d \cdot (t + b))$$

If linear search is used to find attributes and values, the running time would be:

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

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

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

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

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

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

Experiment. To experimentally determine the name lookup performance of our (untuned) Java implementation of an INR, we created a number of randomly constructed name-trees, and timed how long it took to perform 1000 random lookup operations on the tree. The name-tree and name-specifiers were chosen to be uniform with same parameters as the analysis in Section 5.1. We varied n, the number of distinct names in the tree, and measured lookup times. We performed our experiment on an off-the-shelf PC with an Intel Pentium II processor running at 450 MHz with 512 KB cache and 128 MB RAM. The machine was running Red Hat Linux 5.2, and the code was compiled and run under Sun's Java version 1.1.7.

We fixed the parameters at $r_a = 3$, $r_v = 3$, $n_a = 2$, and $d = 3$, and varied n from 1 to 2500. Our results are shown in Figure 13. For this name-tree and name-specifier structure, our performance went from a maximum of about 1060 lookups per second to a minimum of 220 lookups per second. We did see occasional large variations in similar trials, and conjecture that it is a consequence of quirks in Java's memory allocation. We also found that lookup performance improved significantly for narrower name-specifiers, reaching several thousand lookups per second.

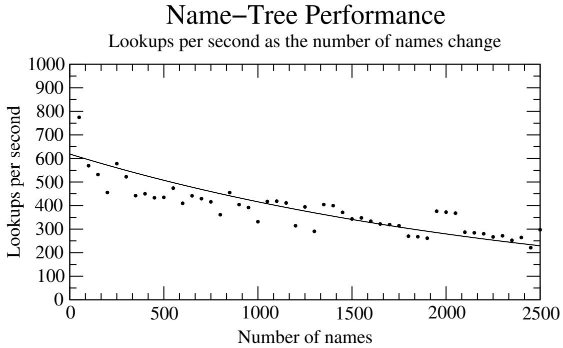

Figure 13: Performance of name-tree lookups. This graph shows how the name-tree lookup performance of an INR goes down as the number of names in its name-tree increases.

These experiments give us a practical idea of how the base case b affects performance. We believe that this order-of-magnitude of lookup performance is adequate for intra-domain deployments, because of the load balancing provided by the INS self-organization protocol and the parallelism inherent in independent name lookups.

5.2 Name Discovery Performance

One of the claims we made about INS was that it was capable of tracking rapid change and dynamism in services and hosts. This section substantiates this claim by discussing the performance of the name discovery protocol.

We measured the performance of INS in discovering *new* network entities, which advertise their existence via name-specifiers. Figure 14 shows the average discovery time of a new name-specifier as a function of n, the number of hops in the INR network from the new name. The machines used in the experiments off-the-shelf Intel Pentium II 450 MHz PCs running Red Hat Linux 5.2 and Windows NT Server 4.0. The network nodes were connected over 100 Mbps Ethernet and 10 Mbps wireless RF links.

When an INR observed a new name-specifier from a periodic node announcement, it processes the update message and performs a lookup operation on the name-tree to see if a route already exists. When it does not find the route, it grafts the name-specifier on to its name-tree and propagates a triggered-update to its neighbors. Thus, it is easy to see that the name discovery time in a network of identical INRs and links, $T_d(n) = n(T_l + T_g + T_{up} + d)$, where T_l is the lookup time, T_g is the graft time, T_{up} is the update processing time, and d is the one-way network delay between any two nodes. That is, name discovery time is to first-order linear in the number of hops. The key experimental question is what the slope of the line is, because that determines how agile INS

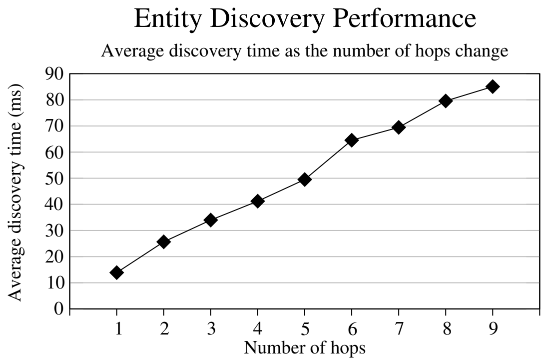

Figure 14: Discovery time of a new network name. This graph shows that the time to discover a new network name is linear in the number of INR hops.

is in tracking changes.

In our experiments the structure of the name-tree on each INR was relatively constant (except for the new grafts), since we were not running any other applications in the system during the measurements. Thus, the lookup and graft times at one INR and the others were roughly the same. As shown in Figure 14, $T_d(n)$ is indeed linear in n, with a slope of less than 10 ms/hop. This implies that typical discovery times are only a few tens of milliseconds, and dominated by network transmission delays.

6 Related Work

We are unaware of an application-controlled network architecture that integrates naming and routing the way the INS architecture does. We believe that the flexible naming and resolution provided by INS is well-suited to the future Internet infrastructure because it enables a variety of network services and applications in an easily deployable manner.

There has been significant research in wide-area naming and resolution, including some recent proposals. Vahdat *et al.* [40] present scheme for *active names.* Similar in spirit to active networks that incorporate general purpose computation into the routing infrastructure [38, 44], active names allow applications to define arbitrary computation that executes on names at resolvers. We believe that active names are overly general and therefore complex; for many applications, the benefits they obtain can be accomplished using a flexible but non-Turing-complete naming system with a carefully chosen set of operators such as the one we have proposed. Furthermore, the active names scheme does not specify a resolution protocol that incorporates message routing or name dissemination for resource discovery.

To our knowledge, the first proposal to decouple names from object locations was described in a paper by O'Toole and Gifford [30], where they describe a content naming scheme and its application to Semantic File Systems [18]. The design of content names is very different from ours and so is its application, but the underlying philosophy is similar.

Another suggestion similar to our intentional naming scheme was made by Jacobson [21]. Presented in the context of multicast-based self-configuring Web caching, the proposal was to use the URL namespace and "instead of asking X to send you Y, simply ask for Y." More recently, as part of the Simple Systems DARPA ISAT group, Estrin *et al.* [14] suggested a naming scheme based on attributes to enable diffusion-based sensor tracking and control applications. Our intentional naming scheme has some features in common with that proposal, but differs in the details of the resolution and message routing processes and in the self-organization mechanisms.

Cisco's DistributedDirector [7] intelligently resolves service names (in the URL namespace) to the IP address of the closest server, based on client proximity and client-to-server link latency. Unlike our system, DistributedDirector is not a general framework for naming and resolution and it does not integrate resolution and routing the way INS does using intentional names.

IBM's "T Spaces" [26] enable communication between applications in a network by providing a lightweight distributed database model. Network entities can perform queries on pieces of data that are described by tuples (similar to attribute-value pairs in name-specifier expressions) and have been set by other entities. However, this system has been optimized for client-server applications rather than for (ad hoc) peer-to-peer communication, and uses a central database to maintain tuple mappings. Sun Microsystems' Jini project [22] aims to provide a framework for users to discover and access local services, by forming a "federation of networked devices" over Java's Remote Message Invocation (RMI). Jini does not address either how naming should be done or the name resolution process.

In the past few years, several schemes for distributed Web caching have been proposed including Harvest [6], Squid and the Internet Cache Protocol (ICP) [43], Adaptive Web Caching [2], diffusion-based caching [19], Summary Cache [15], Cisco's Cache Engine [8], Web caching using active networks [25], multicast-based caches [39], hierarchical Web caching [36], meta-data caching, etc. We believe that by being able to incorporate data caching into the name resolution framework as an important optimization, INS has the potential to simplify the complex problem of Web caching. Using INS would lead to an infrastructure similar to that described by Jacobson [21].

The Service Location Protocol (SLP) [41, 34] is a protocol designed to facilitate the discovery and use of heterogeneous network resources using centralized Directory Agents. In contrast, INS enables highly robust, dynamic, and flexible entity (service) discovery.

Retaining network connectivity while mobile requires a level of indirection so that all traffic to the mobile host can be redirected to its current location. Mobile IP [33] achieves this using a Home Agent in the mobile host's home domain. With INS, the required level of indirection is obtained using the intentional naming system, since all traffic to the mobile host would go through a name resolution process. The tight integration of naming and routing enables continued network connectivity in the face of mobility. Furthermore, INS system is a highly distributed and fault tolerant architecture avoiding central points of failure that Mobile IP suffers from. A number of protocols for ad hoc or infastructureless routing have recently been proposed [5, 31, 37]. These protocols, while very useful to enable IP connectivity, do not support routing of queries via name-specifiers like INS does.

7 Concluding Remarks

In this paper, we established the need for an intentional naming scheme, where applications describe *what* they are looking for, not *where* to find data. We presented the design, implementation and evaluation of an Intentional Name System, called INS, to realize this vision. The components of INS include the an intentional naming scheme using name-specifiers, which are query expressions in a restricted query language, an intentional name resolver architecture that resolve names and integrates naming and message routing, and the INS API and application architecture.

We presented the design and analysis of an efficient algorithm for name lookups and measurements of our implementation, which show that an (untuned) Java implementation can perform several hundred to a few thousand lookups per second. We also presented the design and evaluation of an entity discovery protocol, demonstrating INS' agility in tracking highly dynamic changes in network topology or application data. We discussed the details of an INR self-organization protocol that we use to bootstrap the system, form dynamic neighborhoods, and perform load management.

Finally, we presented the design of the INS application architecture and detailed a sample mobile camera application for remote surveillance, which we have had experience with over the past many weeks. We detailed how this application used intentional names and leveraged INS' automatic support for mobility, group communication, service location and data caching.

Our experience with INS has convinced us that using intentional names in the naming system provides the right level of indirection over which to implement a variety of services. Furthermore, although we have not explicitly detailed it, INS allows applications to efficiently track *dynamic data attributes,* because the choice of attributes to use in name-specifiers are completely under application-control. For example, a sensor application can use INS to send a message to all the temperature sensors in a building that have recently observed a temperature greater than some threshold, to actuate an action such as turning on air vents. We believe that INS has the potential to become an integral part of future device and sensor networks.

Although our current experiences with INS are very encouraging, there remain some important areas of fruitful research before the full benefits of this intentional naming architecture can be realized in the Internet. First, we need to carefully expand the set of supported operators in the resolution process, such as incorporating range matches. Second, the current INS architecture is intended for intra-domain deployment. We are actively developing a wide-area architecture to complement INS, which will integrate INS with extensions to DNS for ease of deployment. Ultimately, the benefits of such a system are in enabling or facilitating the development of new applications and services; to this end, we are designing new services (e.g., transparent performance-based server selection, location-dependent services, etc.) using INS. This will demonstrate the benefits of INS and help us characterize the class of applications that INS facilitates.

Acknowledgments

This work was supported by a grant from NTT Corporation. We would like to thank Dr. Ichizo Kogiku of NTT for his interest in this effort and helpful technical comments. We are grateful to John Guttag and Frans Kaashoek for useful comments and suggestions on INS that greatly influenced our research and focus.

References

[1] AMIR, E., MCCANNE, S., AND KATZ, R. An Active Service Framework and its Application to Real-time Media Transcoding. In *Proc. ACM SIGCOMM* (Sept. 1998).

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

[3] BALLARDIE, T., FRANCIS, P., AND CROWCROFT, J. Core Based Trees (CBT) An Architecture for Scalable Inter-Domain Multicast Routing. In *Proceedings of SIGCOMM '93 (San Francisco, CA)* (Aug. 1993).

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

[5] BROCH, J., MALTZ, D., JOHNSON, D., HU, Y., AND JETCHEVA, J. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proc. ACM/IEEE MOBICOM* (Oct. 1998).

[6] CHANKUNTHOD, A. AND DANZIG, P. AND SCHWARTZ, M. AND WORRELL, K. A Hierarchical Internet Object Cache. In *Proc. USENIX Technical Conf.* (Jan. 1996).

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

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

[9] CLARK, D. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM* (Aug. 1988).

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

[11] DEERING, S., AND CHERITON, D. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems 8*, 2 (May 1990).

[12] DEERING, S., ESTRIN, D., FARINACCI, D., JACOBSON, V., LIU, C.-G., AND WEI, L. An Architecture for Scalable Inter-Domain Multicast Routing. In *Proceedings of SIGCOMM '94 (London, UK)* (Sept. 1994).

[13] ERIKSSON, H. Mbone: The multicast backbone. *Communications of the ACM 37*, 8 (1994), 54–60.

[14] ESTRIN, D., ET AL. Simple Systems. ISAT study group, 1998.

[15] FAN, L. AND CAO, P. AND ALMEIDA, J. AND BRODER, A. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. ACM SIGCOMM* (Sept. 1998).

[16] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., AND BERNERS-LEE, T. *Hypertext Transfer Protocol – HTTP/1.1*, Jan 1997. RFC-2068.

[17] FINK, B. 6bone Home Page. `http://www.6bone.net/`, Jan. 1999.

[18] GIFFORD, D., JOUVELOT, P., SHELDON, M., AND O'TOOLE, J. Semantic File Systems. In *13th ACM Symp. on Operating Systems Principles* (Oct. 1991).

[19] HEDDAYA, A., MIRDAD, S., AND YATES, D. Diffusion-based Caching along Routing Paths. In *Proc. 2nd International WWW Caching Workshop* (June 1997).

[20] HEDRICK, C. *Routing Information Protocol*, June 1988. RFC-1058.

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

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

[23] KANTOR, B. ARE LAPSLEY, P. *Network News Transfer Protocol*, Feb. 1986. RFC-977.

[24] KUMAR, S., RADOSLAVOV, P., THALER, D., ALAETTINOĞLO, C., ESTRIN, D., AND HANDLEY, M. The MASC/BGMP Architecture for Inter-Domain Multicast Routing. In *Proceedings of SIGCOMM '98 (Vancouver, BC)* (Sept. 1998).

[25] LEGEDZA, U., AND GUTTAG, J. Using Network Level Support to Improve Cache Routing. In *Proc. 3rd International WWW Caching Workshop* (June 1998).

[26] LEHMAN, T., MCLAUGHRY, S., AND WYCKOFF, P. T Spaces: The Next Wave. `http://www.almaden.ibm.com/cs/TSpaces/`, 1998.

[27] MALKIN, G. *RIP Version 2*, Nov. 1988. RFC-2453.

[28] MOCKAPETRIS, P. V., AND DUNLAP, K. Development of the Domain Name System. In *Proceedings of SIGCOMM '88 (Stanford, CA)* (Aug. 1988).

[29] MOY, J. *OSPF Version 2*, Mar. 1994. RFC-1583.

[30] O' TOOLE, J., AND GIFFORD, D. Names should mean What, not Where. In *ACM 5th European Workshop on Distributed Systems* (Sept. 1992).

[31] PARK, V., AND CORSON, S. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Netowrks. In *Proc. INFOCOM'97* (Apr. 1997).

[32] PARTRIDGE, C., MENDEZ, T., AND MILLIKEN, W. *Host Anycasting Service*, Nov. 1993. RFC-1546.

[33] PERKINS, C. *IP Mobility Support*, Oct. 1996. RFC-2002.

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

[35] POSTEL, J. *Simple Mail Transfer Protocol*, August 1982. RFC-821.

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

[37] SINHA, P., SIVAKUMAR, R., AND BHARGHAVAN, V. CEDAR: A Core-Extraction Distributed ad hoc Routing Algorithm. In *Proc. IEEE INFOCOM* (Mar. 1999).

[38] TENNENHOUSE, D., SMITH, J., SINCOSKIE, W., WETHERALL, D., AND MINDEN, G. A Survey of Active Network Research. *IEEE Communications Magazine 35*, 1 (Jan. 1997).

[39] TOUCH, J. The LSAM Proxy Cache—A Multicast Distributed Virtual Cache. In *Proc. 3rd International WWW Caching Workshop* (June 1998).

[40] VAHDAT, A., ANDERSON, T., AND DAHLIN, M. Active Naming: Programmable Location and Transport of Wide-Area Resources, 1998.

[41] VEIZADES, J., GUTTMAN, E., PERKINS, C., AND KAPLAN, S. *Service Location Protocol*, June 1997. RFC-2165.

[42] VIXIE, P., THOMSON, S., REKHTER, Y., AND BOUND, J. *Dynamic Updates in the Domain Name System*, Apr 1997. RFC-2136.

[43] WESSELS, D., AND CLAFFY, K. ICP and the Squid Web Cache. `http://ircache.nlanr.net/~wessels/Papers/icp-squid.ps`, Aug. 1997.

[44] WETHERALL, D., GUTTAG, J., AND TENNENHOUSE, D. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proc. IEEE OPENARCH* (Apr. 1998).