Proceedings of the 17th ACM SOSP,
Kiawah Island, SC, December 1999.

The Design and Implementation of an Intentional Naming System

William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan
and Jeremy Lilley

MIT Laboratory for Computer Science
{wadjie, elliot, hari, jjlilley}@lcs.mit.edu

Abstract. This paper presents the design and implementation of the Intentional Naming System (INS), a resource discovery and service location system for dynamic and mobile networks of devices and computers. Such environments require a naming system that is (i) expressive, to describe and make requests based on specific properties of services, (ii) responsive, to track changes due to mobility and performance, (iii) robust, to handle failures, and (iv) easily configurable. INS uses a simple language based on attributes and values for its names. Applications use the language to describe what they are looking for (i.e., their *intent*), not where to find things (i.e., not hostnames). INS implements a *late binding* mechanism that integrates name resolution and message routing, enabling clients to continue communicating with end-nodes even if the name-to-address mappings change while a session is in progress. INS resolvers self-configure to form an application-level overlay network, which they use to discover new services, perform late binding, and maintain weak consistency of names using soft-state name exchanges and updates. We analyze the performance of the INS algorithms and protocols, present measurements of a Java-based implementation, and describe three applications we have implemented that demonstrate the feasibility and utility of INS.

1 Introduction

Network environments of the future will be characterized by a variety of mobile and wireless devices in addition to general-purpose computers. Such environments display a degree of dynamism not usually seen in traditional wired networks due to mobility of nodes and services as well as rapid fluctuations in performance. There is usually no pre-configured support for describing, locating, and gaining access to available services in these heterogeneous, mobile networks. While the packet routing problem in mobile networks has been extensively researched [6, 32], the important functions of resource discovery and service location are only recently beginning to receive attention in the research community. We believe that this is an important problem to solve because the cost of deploying and running such a network infrastructure is dominated by software and service management, while diminishing hardware costs are making it cheap to network all sorts of devices. Examples of applications in such environments include sending a job to the closest (based on geographic location) and least-loaded printer, retrieving files from a mobile, replicated server based on network performance and server load, and retrieving the current image from all the mobile cameras in a particular section of a building.

Based on our target environment and applications, we identify the following important design goals for a naming system that enables dynamic resource discovery and service location:

- **Expressiveness.** The naming system must be flexible in order to handle a wide variety of devices and services. It must allow applications to express arbitrary service descriptions and queries.
- **Responsiveness.** The naming system must adapt quickly to end-node and service mobility, performance fluctuations, and other factors that can cause a change in the “best” network location of a service.
- **Robustness.** The naming system must be resilient to name resolver and service failures as well as inconsistencies in the internal state of the resolvers.
- **Easy configuration.** The name resolvers should configure themselves with minimal manual intervention and the system should not require manual registration of services. The resulting system should automatically distribute request resolution load among resolvers.

The main contribution of our work is the design and implementation of INS, an Intentional Naming System that meets the above goals. Because applications in our environment (as in many distributed environments) often do not know the best network location that satisfies their needs for information or functionality, we argue in favor of an *intentional* naming scheme and resolution architecture in which applications describe *what* they are looking for, not *where* to find it. Name resolvers in the network route requests to the appropriate locations by maintaining a mapping between service descriptions and their network locations.

INS achieves expressiveness by using an intentional name language based on a hierarchy of attributes and values, allowing nodes that provide a service to precisely describe what they provide and consumers to easily describe what they require. Names based on attributes and values have been suggested before in other contexts [5, 7, 13, 45] and we draw upon previous work in this area in designing our naming language. While several complex query languages based on attributes and values exist in the literature, ours is particularly simple and has a small set of commonly used operators, which makes it lightweight and easy to implement even on impoverished devices. We also design the INS resolution architecture to be independent of the specific language used to perform queries, so that it can also be used in the context of other service description languages.

An important characteristic of our target environment is mobility, where the network location (e.g., IP address) of an end-node changes. *Node mobility* may occur due to physical mobility or when a node changes the network used to communicate (e.g., changing from an wired Ethernet to a radio frequency network). Another form of mobility is *service mobility,* where the network addresses of a service does not change, but the end-nodes mapping to a service change because of a change in the values of the attributes sought by clients. In addition, our environment is dynamic because of performance fluctuations—as the load on service nodes and paths in the network changes, so does the location of the best node to serve each client request. Hence, INS should reflect performance changes in the results of name resolution.

In INS, clients use an intentional name to request a service without explicitly listing the end-node(s) that ultimately serve the request. This “level of indirection” provided by an intentional name allows applications to seamlessly continue communicating with end-nodes even though the mapping from name to end-node addresses may change during the session, transparent to the client. Thus, INS supports mobile applications, which use intentional names rather than IP addresses.

INS achieves responsiveness by *integrating* name resolution and message routing, operations that have traditionally been kept separate in network architectures. INS applications benefit from this abstraction using a *late binding* option, where the binding between the intentional name and network location(s) is made at message delivery time rather than at request resolution time. This binding is “best-effort” since INS provides no guarantees on reliable message delivery. Thus, INS uses an intentional name not only to locate services, but also to route messages to the appropriate end-points. This integration leads to a general method for performing application-level routing using names, effected by applications including data with the name resolution query.

Our integrated routing and resolution system provides two basic types of message delivery service using late binding. An application may request that a message be delivered to the "optimal" service node that satisfies a given intentional name, called *intentional anycast*. Here, the metric for optimality is application-controlled and reflects a property of the service node such as current load. A second type of message delivery, *intentional multicast,* is used to deliver data to *all* service nodes that satisfy a given name, for example, the group of sensors that have all recorded sub-zero temperatures. These two delivery services allow INS to achieve *application-level* anycast and multicast.

In keeping with the end-to-end principle [37], we leave the underlying network-layer addressing and routing of the IP architecture unchanged. Rather, our approach to providing these services is to layer them as an overlay network over unicast IP. The only network layer service that we rely upon is IP unicast, which is rapidly becoming ubiquitous in mobile and wireless environments[1].

Another reason for leaving the core infrastructure unmodified is that often, a network-layer service does not perfectly match the requirements of the application at hand. Indeed, performing anycast on a specific network-layer criterion such as hop-count, network latency or available bandwidth, is ineffective from the point of view of many applications because it does not optimize the precise metric that applications require. For example, a network-layer anycast [31] to find the "best" printer on a floor of a building cannot locate the *least-loaded* printers. To remedy this, INS allows intentional anycast based on *application-controlled* metrics, where resolvers select the least value according to a metric that is meaningful to and advertised by applications.

Despite allowing application-controlled routing metrics, INS presents a simple and well-defined service model for intentional anycast and multicast. In contrast to the active networks architecture [41, 46] and their naming counterpart, ActiveNames [43], where arbitrary code and services may be injected into the data path to customize the functions of an IP router or name resolver, INS resolvers do not run arbitrary code nor embed any application-specific semantics in the routing and resolution architecture. Instead, our system relies on structured names to express application parameters. This decision to leave IP unicast unmodified is based on the difficulties encountered in deploying other IP

[1] Note that we do not rely on Mobile IP [32] in INS.

extensions, for example, IP multicast [12], guaranteed services [10], and more recently, active IP networks [46]. In this sense, one may view the INS architecture as similar in philosophy to application-level anycast [3] and Web server selection, which have recently gained in popularity.

INS uses a decentralized network of resolvers to discover names and route messages. To ease configuration, INS resolvers self-configure into an application-level overlay network and clients can attach to any of them to resolve their requests and advertise services. These resolvers use *soft-state* [9] periodic advertisements from services to discover names and delete entries that have not been refreshed by services, eliminating the need to explicitly de-register a service. This design gracefully handles failures of end-nodes and services. They also implement load-balancing and load-shedding algorithms, which allows them to scale well to several thousand services.

The INS resolution architecture presented in this paper makes three key contributions: (i) it integrates resolution and routing, allowing applications to seamlessly handle node and service mobility, and provides flexible group communication using an intentional name as the group handle; (ii) its resolvers self-configure into an overlay network and incorporate load-balancing algorithms to perform well; and (iii) it maintains weak consistency amongst replicated resolvers using soft-state message exchanges. These features distinguish it from other service discovery proposals made in the recent past, including the IETF Service Location Protocol (SLP) [44, 33], Sun's Jini service discovery [21], the Simple Service Discovery Protocol [19], universal plug-and-play [42], and Berkeley's service discovery service [11].

An important feature of our architecture is its potential for incremental and easy deployment in the Internet, without changing or supplanting the existing Internet service model. INS is intended for dynamic networks on the order of several hundred to a few thousand nodes, many of which could be mobile (e.g., inside a single administrative domain, building, office or home network). We note, however, that the architecture presented in this paper is not directly applicable in the wide-area Internet. We are currently developing a wide-area architecture to complement this intra-domain INS architecture. However, despite this cautionary note, our performance results show that our resolution algorithms and load-balancing strategies permit a network of INS resolvers to scale to several thousand names and services. Our experimental results show that the time to discover new names is on the order of tens of milliseconds. We find that the time to process name updates is the performance bottleneck in many cases, and describe a technique to partition the namespace amongst resolvers to alleviate this problem.

To demonstrate the utility of INS, we describe its programming interface and the implementation of three applications: *Floorplan*, a map-based service discovery tool for location-dependent services, *Camera*, a mobile camera network for remote surveillance, and *Printer*, a load-balancing printer utility that sends user print requests to the best printer based on properties such as physical location and load. These applications use the INS API and support for mobility, group communication, and service location, gaining these benefits without any other pre-installed support (e.g., Mobile IP [32], IP multicast [12], SLP [44], etc.) for these features.

The rest of this paper is organized as follows. We describe the INS architecture in Section 2, the API and some applications in Section 3, our implementation in Section 4, its performance in Section 5, related work in Section 6, and our conclusions in Section 7.

2 System architecture

INS applications may be *services* or *clients*: services provide functionality or data and clients request and access these. *Intentional Name Resolvers (INRs)* route client requests to the appropriate services, implementing simple algorithms and protocols that may be implemented even on computationally impoverished devices. Any device or computer in an *ad hoc* network can potentially act as a resolver, and a network of cooperating resolvers provides a system-wide resource discovery service.

INRs form an application-level overlay network to exchange service descriptions and construct a local cache based on these advertisements. Each service attaches to an INR and advertises an attribute-value-based service description and an application-controlled metric. Each client communicates with an INR and requests a service using a query expression. Because service descriptions are disseminated through the INR network in a timely manner, a new service becomes known to the other resolvers and through them to the clients.

When a message arrives at an INR, it is resolved on the basis of the destination name. The INR makes a resolution/forwarding decision depending on the specific service requested by the client application. If the application has chosen early binding (selected using the *early-binding flag* in the request), the INR returns a list of IP addresses corresponding to the name. This is similar to the interface provided by the Internet Domain Name System (DNS) [27] and most other existing service discovery systems, and is useful when services are relatively static. When

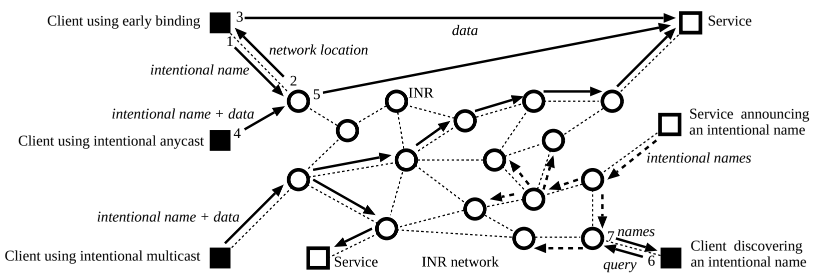

Fig. 1. The architecture of the Intentional Naming System. The upper-left corner shows an application using early binding: the application sends an intentional name to an INR to be resolved (1), receives the network location (2), and sends the data directly to the destination application (3). The middle-left shows an application using intentional anycast—the application sends an intentional name and the data to an INR (4), which tunnels to exactly one of the destinations that has the least metric (5). The lower-left corner shows an application using intentional multicast: the application sends an intentional name and the data to an INR, which forwards it through the INR network to all of the destination applications. The lower-right corner shows an application announcing intentional names to an INR. The intentional names are beginning to propagate throughout the INR network. An application discovering names sends a query to an INR (6), receives a set of names that match the name in query.

there are multiple IP addresses corresponding to a name, the client may select an end-node with the least metric, which is available from the result of the resolution request. This metric-based resolution is richer than round-robin DNS resolution.

INS applications use the two late binding options—intentional anycast and intentional multicast—to handle more dynamic situations. Here, the network addresses are not returned to the client, but instead, the INR forwards the name and the associated application payload directly to the end-nodes (e.g., services). If the application requests intentional anycast, the INR tunnels the message to exactly one of the end-nodes in its list that has the least metric. In INS, this metric does not reflect a network-layer metric such as hop-count used in network-layer anycast [31]; rather, INS allows applications to advertise arbitrary application-specific numeric metrics such as average load. In intentional multicast, the INR forwards each message to all next-hop INRs associated with the destination name. The message is forwarded along the INR overlay to all the final destination nodes that match the name.

INRs self-configure into a spanning-tree overlay network topology, optimizing the average delay between neighboring INRs. In constructing this overlay topology, we use measurements of INR-to-INR round-trip time. The spanning tree is used to disseminate service descriptions as well as receiver subscriptions. Unlike other overlay networks that maintain pre-configured, static neighbors such as the MBone [14] or the 6Bone [17], INRs can be spawned or terminated and automatically adjust their neighbor relationships based on network conditions. They also implement load-balancing algorithms to perform better, by spawning new resolvers on other nodes when the incoming request rate is high and delegating portions of the namespace to the newly spawned instances.

Figure 1 summarizes the architecture of INS, illustrating how applications and INRs interact.

2.1 Name-specifiers

INS implements intentional names using expressions called *name-specifiers*. Clients use name-specifiers in their message headers to identify the desired destinations (and sources) of messages. Name-specifiers are designed to be simple and easy to implement. The two main parts 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[2] that are defined by applications; name-specifiers do not restrict applications to using a fixed set of attributes and values. Together, an attribute and its associated value form an *attribute-value pair* or *av-pair*.

A name-specifier is a hierarchical arrangement of av-pairs such that an av-pair that is *dependent* on another is a descendant of it. For instance, in the example name-specifier shown in Figure 2, a building called the Whitehouse is meaningful only in the context of the city of Washington, so the av-pair `building=whitehouse` is dependent on the av-pair `city=washington`. Av-pairs that are *orthogonal* to each other but dependent on the same av-pair, are siblings in the tree. For example, a digital camera's data-type and resolution can be selected

[2] Attributes and values being free-form strings is not a fundamental property; fixed length integers could be used just as easily if the bandwidth or processing power required for handling names is a concern. Not having human readable strings makes debugging more difficult, but does not affect normal use of the system, since applications still need to understand the semantics of attribute and values to present them to users.

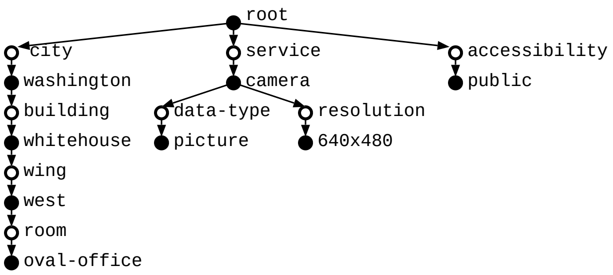

Fig. 2. A graphical view of an example name-specifier. The hollow circles are used to identify attributes; the filled circles identify values. The tree is arranged such that dependent attributes are descendants, and orthogonal attributes are siblings. This name-specifier describes a public-access camera in the Oval office.

independently of each other, and are meaningful only in the context of the camera service. Therefore, the av-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 av-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`.

Name-specifiers have a representation (Figure 3) that is used when they are included in a message header to describe the source and destination of the message. This string-based representation was chosen to be readable to assist with debugging, in the spirit of other string-based protocols like HTTP [16] and NNTP [22]. Levels of nesting are indicated by the use of brackets (`[` and `]`), and attributes and values are separated 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.

In addition to exact-value matches, name-specifiers also permit wild-card matching of values. To do this, the value is simply replaced by the wild-card token (`*`). Thus to construct a name-specifier that refers to *all* public cameras providing 640x480 pictures in the West Wing of the Whitehouse, not just the one in the Oval Office, an application replaces

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

Fig. 3. The wire representation of the example name-specifier shown in Figure 2, with line-breaks and extra spacing added to improve readability.

the value `oval-office` with '`*`' in the name-specifier shown in Figures 2 and 3. We are currently incorporating inequality operators ($<$, $>$, $\leq$, and $\geq$) to provide range selection operations in name-specifiers, to augment the matches described above.

2.2 Discovering names

Services periodically advertise their intentional names to the system to describe what they provide. Each INR listens to these periodic announcements on a well-known port to discover services running at different end-nodes. INRs replicate and form an overlay network among themselves, over which they send updates of valid names in the system.

The name discovery protocol treats name information as soft-state [9, 35], associated with a lifetime. Such state is kept alive or refreshed whenever newer information becomes available and is discarded when no refresh announcement is received within a lifetime. Rapid changes due to node mobility quickly propagate through the system and new information automatically replaces old, outdated information. Soft-state enables robust operation of the system since INRs can recover from errors and failures automatically without much disruption because incorrect information is updated by new messages. This choice allows a design where applications may join and leave the system without explicit registration and de-registration, because new names are automatically disseminated and expired names automatically eliminated after a timeout.

When clients make name resolution requests, INRs use the information obtained using service advertisements and INR updates to resolve them. In addition to sending resolution requests, clients can discover particular types of names or all known names in the system by sending a *name discovery* message to an INR, specifying an intentional name for the INR to match with all the names it knows about. This mechanism is useful for clients to bootstrap in a new environment.

INRs disseminate name information between each other using a routing protocol that includes *periodic* updates and *triggered* updates to their neighbor INRs. Each update contains the following information about a name-specifier:

- The IP addresses for this name-specifier and a set of `[port-number, transport-type]` pairs for each IP address. The port number and transport type (e.g., HTTP [2], RTP [38], TCP [34], etc.) are returned to the client to allow it to implement early binding.
- An application-advertised metric for intentional anycast and early binding that reflects any property that the service wants anycast routing on, such as current or average load.
- The next-hop INR and the metric, currently the INR-to-INR round-trip latency in the overlay network for the route, used for intentional multicast.
- A unique identifier for the announcer of the name called the *AnnouncerID*, used to differentiate identical names that originate from two different applications on the same node.

INRs use periodic updates to refresh entries in neighboring INRs and to reliably flood names. Triggered updates occur when an INR receives an update from one of its neighbors (either an INR or a client or service) that contains new information (e.g., a newly discovered name-specifier) or information that is different from the one previously known (e.g., better metric)[3].

For applications requiring intentional multicast, INRs forward the name and payload message through the overlay network to all of the network locations that announce a given name. In our current implementation, INRs use the distributed Bellman-Ford algorithm [1] to calculate shortest path trees to those end-nodes announcing the name. Unlike traditional routing protocols that use the algorithm [26], the INS architecture allows multiple identical names to exist in the system. The unique AnnouncerID ensures that routes to identical names can be differentiated. In our implementation, applications generate an AnnouncerID by concatenating their IP address with their startup time, allowing multiple instances to run on the same node.

[3] For inter-INR communications we could have had the INRs use reliable TCP connections and send updates only for entries that change, perhaps eliminating periodic updates at the expense of maintaining connection state in the INRs. We do not explore this option further in this paper, but intend to in the future.

2.3 Name lookup and extraction

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 a *name-record*, which includes the IP addresses of the destinations advertising the name as well as a set of "routes" to next-hop INRs. The late binding process for anycast and multicast do not change the name-specifiers or data in any way. By integrating resolution with routing in the late binding process, INS enables seamless communication between clients and services even if the name-to-location mapping changes during the session.

The rest of this section describes how names are stored at an INR, lookups are performed on an incoming name, and how names are extracted for periodic or triggered updates in the name discovery protocol.

Name-trees Name-trees are a data structure used to store the correspondence between name-specifiers and name-records. The information that the name-records contain are the routes to the next-hop INRs, the IP addresses of potential final destinations, the metric for the routes, end-node metrics for intentional anycast, and the expiration time of the name-record.

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-specifier 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-record. Figure 4 depicts an example name-tree, with the example name-specifier from Figure 2 in bold.

Name lookup The LOOKUP-NAME algorithm, shown in Figure 5, is used to retrieve the name-records for a particular name-specifier n from the name-tree T. The main idea behind the algorithm is that a series of recursive calls reduce the candidate name-record set S by intersecting it with the name-record set consisting of the records pointed to by each leaf-value-node. When the algorithm terminates, S contains only the relevant name-records.

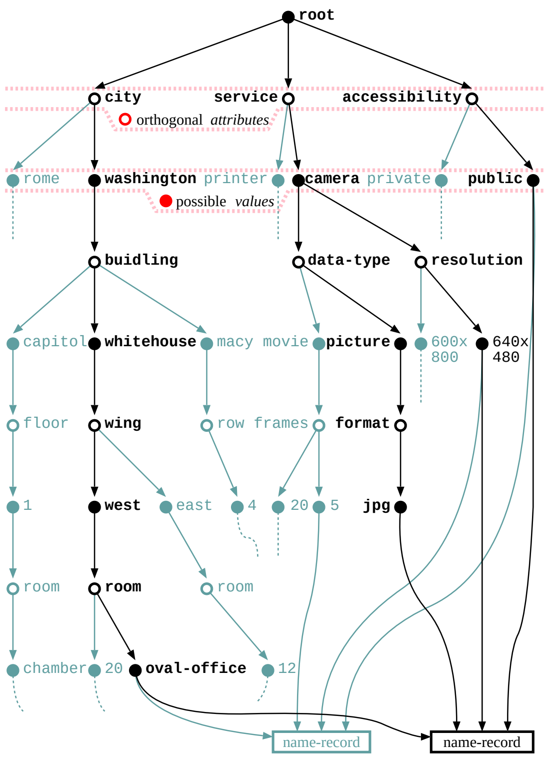

Fig. 4. A partial graphical view of an example INR name-tree. The name-tree consists of alternating layers of attribute-nodes, which contain orthogonal attributes, and value-nodes, which contain possible values. Value-nodes also contain pointers to all the name-records they correspond to. The part of the name-tree corresponding to the example name-specifier shown in Figure 2 is in bold.

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

This algorithm uses the assumption that omitted attributes correspond to wild-cards; this is true for both queries and advertisements. A nice property of the algorithm is that it executes in a single pass without any backtracking. This also means that wild-cards should be used only on the leaf values (any av-pairs after a wild-card will be ignored).

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

Name extraction To send updates to neighboring INRs, an INR needs to get name-specifiers from its name-tree to transmit. Since the name-tree is a superposition of all the name-specifiers the INR knows about, extracting a single name-specifier is non-trivial. The GET-NAME algorithm, shown in Figure 6, is used to retrieve the name-specifiers for a particular name-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 parent of the name-record, and grafting on to parts of the name-specifier that have already been reconstructed.

All the value-nodes in the name-tree, T, are augmented with a "PTR" variable, which is a pointer to the corresponding av-pair in the name-specifier being extracted. Initially, all the PTRs are set to *null*, since they have no corresponding av-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 av-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 7 illustrates an in-progress execution of the algorithm.

```
LOOKUP-NAME(T,n)
  S ← the set of all possible name-records
  for each av-pair p := (n_a, n_v) in n
    T_a ← the child of T such that
            T_a's attribute = n_a's attribute
    if T_a = null
      continue

    if n_v = *                    ▷ wild card matching
      S' ← ∅
      for each T_v which is a child of T_a
        S' ← S' ∪ all of the name-records in the
                    subtree rooted at T_v
      S ← S ∩ S'
    else                          ▷ normal matching
      T_v ← the child of T_a such that
              T_v's value = n_v's value
      if T_v is a leaf node or p is a leaf node
        S ← S ∩ the name-records of T_v
      else
        S ← S ∩ LOOKUP-NAME(T_v, p)
  return S∪ the name-records of T
```

Fig. 5. The LOOKUP-NAME algorithm. This algorithm looks up the name-specifier n in the name-tree T and returns all appropriate name-records.

2.4 Resolver network

To propagate updates and forward data to services and clients, the INRs must be organized as a connected network. In our current design, this application-level overlay network is constructed in a distributed way by INRs self-configuring to form a spanning tree based on metrics that reflect INR-to-INR round-trip latency. The experiments conducted by the INRs to obtain this metric are called *INR-pings,* which consist of sending a small name between INRs and measuring the time it takes to process this message and get a response.

A list of active and candidate INRs is maintained by a well-known entity in the system, called the *Domain Space Resolver* (DSR). The DSR may be thought of as an extension to a DNS server for the administrative domain in which we currently are, and may be replicated for fault-tolerance. DSRs support queries to return the currently active and candidate INRs in a domain.

```
GET-NAME(T,r)
  n ← a new, empty name-specifier
  T.PTR ← n
  for each T_v which is a parent value-node of r
    TRACE(T_v, null)
  reset all PTRs that have been set to null
  return n

TRACE(T_v,n)
  if T_v.PTR ≠ null            ▷ something to graft onto
    if n ≠ null                ▷ something to graft
      graft n as a child of T_v.PTR
  else                         ▷ nothing to graft onto; make it
    T_v.PTR ← a new av-pair consisting of
                 T_v's value and its parent's attribute
    if n ≠ null                ▷ something to graft
      graft n as a child of T_v.PTR
    TRACE(parent value-node of T_v, T_v.PTR)
```

Fig. 6. The GET-NAME algorithm. This algorithm extracts and returns the name-specifier for the name-record r in the name-tree T. TRACE implements most of the functionality, tracing up from a leaf-value until it can graft onto the existing name-specifier.

When a new INR comes up, it contacts the DSR to get a list of currently active INRs. The new INR then conducts a set of INR-pings to the currently active INRs and picks the one with the minimum value to establish a neighbor relationship (or *peer*) with. If each INR does this, the resulting topology is a spanning tree. Because the list of active INRs is maintained by the DSR, and all the other INRs obtain the same list, race conditions are avoided and one can impose a linear order amongst the active INRs based on the order in which they became active in the overlay network. Each INR on the active list, except the first one, has at least one neighbor ahead of it in the linear order, and the resulting graph is clearly connected by construction. Furthermore, each time a node arrives after the first one, it peers with exactly one node, so the number of links formed in an $n-$node network is $n-1$. Any connected graph with n nodes and $n-1$ links must be a tree.

Of course, despite each node making a local minimization decision from the INR-pings, the resulting spanning tree will not in general be the minimum one. We are currently working on improving this configuration algorithm by designing a relaxation phase that asynchronously changes neighbor relationships to eventually converge to an optimal

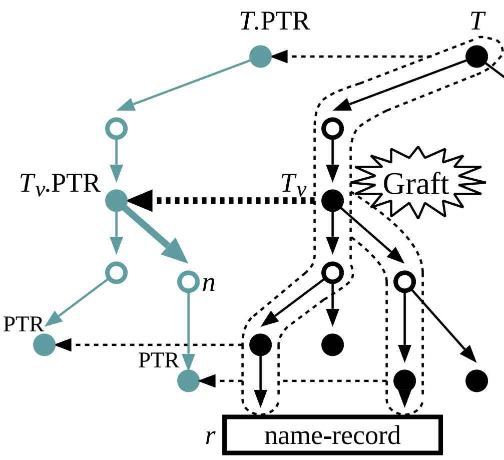

Fig. 7. An illustration of an in-progress execution of the GET-NAME algorithm. The name-specifier being created is shown in gray on the left, while the name-tree it is being created from is shown in black on the right. The parts of the name-tree that are circled with dotted lines are the paths through the name-tree that have been traced. The dotted arrows are used to illustrate the assignments of the PTR variables. The thick arrows indicate the parts of the data structures that are currently being manipulated. In this example, the name-specifier fragment rooted at n is being grafted onto T_v.PTR, which is part of the main name-specifier.

tree in the absence of mobility. We also note that a spanning tree may not be a sufficiently robust overlay topology to exchange names and perform intentional multicast, because it has several single points of failure. We are currently exploring other algorithms for constructing more redundant overlay structures.

2.5 Load balancing and scaling

There are two potential performance and scaling bottlenecks in the system—lookups and name update processing. To handle excessive lookup loads, we allow INRs to spawn instances on other candidate (but currently inactive) resolvers, and kill themselves if they are not loaded. To spawn an INR on a candidate node, an INR obtains the candidate-node information from the DSR. An INR can also terminate itself if its load falls below a threshold, informing its peers and the DSR of this. The spanning tree overlay algorithm then adjusts to these changes in the active INR list.

Since INRs exchange name information with other resolvers on a periodic basis and also via triggered updates, update scalability is a serious concern. That is, after a point, the volume of name updates will start to saturate either the available bandwidth or processing capacity of a given resolver node. We conducted several experiments to understand the bottlenecks in our design. While the link bandwidth and processing time required for the name update protocol depends on the size of the name-specifiers and the complexity of the name tree, we found that the process was CPU-bound in all our experiments. On our Java implementation between various Pentium II machines running Linux RedHat 5.2 over 1-5 Mbps wireless links, we found that for a relatively rapid refresh interval of 15 seconds with randomly-generated 82-byte intentional names, the processor was saturated before the bandwidth consumption was 1 Mbps (Figure 8). We also found that the name processing in the name dissemination protocol dominated the lookup processing in most of our experiments. This occurs because in this design, all the resolvers need to be aware of all the names in the system and have to process them.

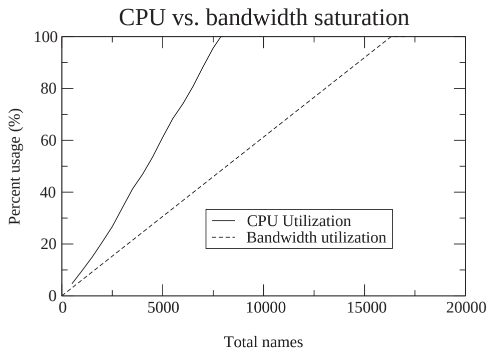

Fig. 8. An example of a CPU-bound configuration of INS. The Pentium II processor is saturated well before a 1mbit/s link. Numbers are shown with refreshes happening every 15 seconds.

Based on these experiments and a better understanding of the scaling bottleneck, we describe a solution that alleviates it. The idea is to partition the namespace into several *virtual spaces*, ensuring that each resolver only needs to route for a subset of all the active virtual spaces in the system. Conceptually, there is now one resolver overlay network per virtual space (however, the overlays for different virtual spaces may span the same resolver nodes).

More formally, we define a virtual space to be an application-specified set of names that share some attributes in common. For instance, all the cameras in building NE-43 at MIT could form the *camera-ne43* virtual space, and all the devices in the building NE43 could form the *building-NE43* virtual space. In the first case, the names (services) in the space might share the "service" (equal to "camera") and "location" (equal to NE-43 in MIT) attributes in common, while in the second case, they all share the same location. INS does not assume particular virtual space names in the system, but does require that each service name the virtual spaces it belongs to (it may belong to multiple virtual spaces too). Clients and applications may interact with services in any virtual space.

An INR knows which virtual space an advertisement or query belongs to because it standardizes a well-known attribute, "vspace" by which applications can express the name of their virtual space. The names of two virtual spaces for different sets of services must not collide, and we are currently exploring ways of handling this issue. Internally, an INR stores the names of different virtual spaces in separate, self-contained name trees.

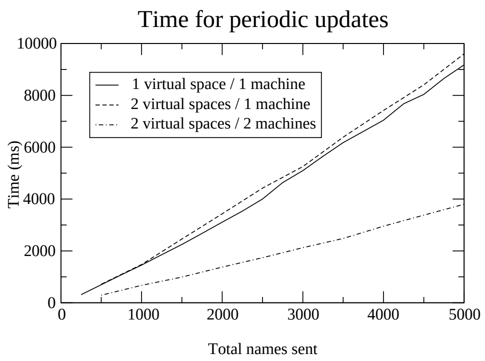

Fig. 9. Periodic update times when the names are divided into two equally-sized spaces.

Partitioning virtual spaces appears to be a promising way to shed load and significantly improve the scalability of INS, especially up to several thousand services. Based on several experiments, we found that the processing time required for periodic updates reduces proportionally when we partition the names into different virtual spaces and then distribute them on to separate resolvers, as shown in Figure 9. If an INR gets a request from a client to resolve for a virtual space it does not route for, it needs to forward the request to a resolver that does. This can be done by caching the resolvers for a small number of popular virtual spaces, and if a cache miss occurs, sending the request to the DSR to be resolved by an appropriate resolver.

In summary, two simple techniques hold promise for scaling the current performance of INS. If an INR is heavily loaded because of name lookups, it can obtain a candidate-INR list and spawn a new INR to handle some of its current load. The configuration protocol used by clients to pick a default INR will cause some of them to move to the newly spawned INR. If an INR is loaded because of update processing, it is likely that all the INRs in that virtual space are also loaded. Therefore, the solution is not to spawn another one for the same space, but to *delegate* one or more virtual spaces to a new INR network. Our experimental results indicate that this is a promising approach to take and we have started implementing this idea.

3 Applications

This section describes the INS API and three of the applications we have developed using it that leverage its support for resource discovery, mobility, and group communication. We describe *Floorplan*, a map-based discovery tool for location-dependent services, *Camera*, a mobile camera network, and *Printer*, a load-balancing printer utility.

An application uses the API to create a name-specifier for a service and to periodically advertise it to the INR network. To discover new services, an application uses the API to send a *discovery* message to an INR to find out what services matching a given name-specifier have been discovered by it. After discovering name-specifiers, the application communicates with the corresponding services by using the API functions to construct a message. Applications choose intentional anycast or intentional multicast by setting the *Delivery* bit-flag in the message header, and early or late binding by setting the *Binding* bit-flag.

3.1 *Floorplan*: a service discovery tool

Floorplan is a service discovery tool that shows how various location-based services can be discovered using the INS. As the user moves into a new region, a map of that region pops up on her display as a building floorplan. *Floorplan* learns about new services by sending a *discovery* message to an INR. This message contains a name-specifier that is used as a filter, and all the name-specifiers that match it are sent back to the application. Floorplan uses the location and service information contained in the returned name-specifiers to deduce the location and the type of each service and display the appropriate icon.

An important component of *Floorplan* is *Locator*, a location server. Rather than directly incorporate maps of regions, *Floorplan* retrieves them as needed from *Locator*. This retrieval is done by sending a request using a name-specifier such as:
`[service=locator[entity=server]][`*location*`]`.
In response, *Locator* retrieves the desired map and sends it back to the requesting *Floorplan* instance, using the requestor's intentional name to route the message.

As services are announced or timed out, new icons are displayed or removed. Clicking on an icon invokes the appropriate application for the service the icon represents. The implementation of *Floorplan* deployed in our building allows users to discover a variety of services including networked cameras (Section 3.2), printers (Section 3.3), and device controllers for TV/MP3 players. These service providers advertise name-specifiers specifying several of their attributes, including their location in the building. For example, a camera in Room 510 advertises the following name-specifier:
`[service=camera[entity=transmitter][id=a]][room=510]`

3.2 *Camera*: a mobile camera service

We have implemented a mobile camera service, *Camera*, that uses INS. There are two types of entities in *Camera*: transmitters and receivers. A receiver requests images from the camera the user has chosen (in *Floorplan*) by sending requests to an intentional name that describes it. These requests are forwarded by INRs to a *Camera* transmitter, which sends back a response with the picture.

There are two possible modes of communication between camera transmitters and receivers. The first is a request-response mode, while the second is a subscription-style interaction that uses intentional multicast for group communication. In the request-response mode, a receiver sends an image request to the transmitter of interest by appropriately naming it; the corresponding transmitter, in turn, sends back the requested image to the receiver. To send the image back to only the requester, the transmitter uses the `id` field of the receiver that uniquely identifies it. *Camera* uses this to seamlessly continue in the presence of node or camera mobility.

For example, a user who wants to request an image from a camera in room 510 can send out a request to INRs with destination name-specifier:
`[service=camera[entity=transmitter]][room=510]`
and source name-specifier:
`[service=camera[entity=receiver][id=r]][room=510]`
The transmitter that receives this request will send back the image with the source and destination name-specifiers inverted from the above. The `room` attribute in the destination name-specifier refers to the *transmitter's* location; the `id` attribute allows the INRs to forward the reply to the interested receiver.

When a mobile camera moves to a different network location, it sends out an update to an INR announcing its name from the new location. The name discovery protocol ensures that outdated information is removed from the name-tree, and the new name information that reflects the new network location will soon come into effect. Thus, any changes in network location of a service is rapidly tracked and refreshed by INRs, allowing applications to continue.

In addition to such network mobility, INS also allows applications to handle service mobility. Here, a service such as a mobile camera moves from one location to another, and its network location does not (necessarily) change. However, its intentional name may change to reflect its new location or any new properties of the new environment it has observed, and it may now be in a position to provide the client with the information it seeks. With intentional names, such application-specific properties such as physical location can be expressed and tracked.

Camera uses intentional multicast to allow clients to communicate with groups of cameras, and cameras to communicate with groups of users. It takes advantage of the property that an intentional name can be used not only for rich service descriptions, but also to refer to a group of network nodes that share certain properties that are specified in their names.

To use this feature, the *Camera* transmitter sends out an image destined to all users subscribing to its images by setting the D bit-flag to *all*. When an INR looks up a name-specifier, it finds all of the network locations that match it. Rather than forwarding the data to just the best one of them, it sends the data to each next-hop INR for which there is an associated network location. Similarly, a user can also subscribe to *all* cameras in the building (or a subset of them named by certain criteria).

For example, a camera transmitter located in room 510 sends out its images to all of its subscribers at once using the following destination name-specifier:

`[service=camera [entity=receiver][id=*]][room=510]`

and set the *Delivery* bit-flag to *all*. The use of wild card `[id=*]` refers to all subscribers, regardless of their specific IDs.

When implementing *Camera*, we noticed that it would be useful to cache pictures at various places in the network, so that requests do not have to go back to the original server every time. To achieve this, we designed an *application-independent* extension to INS that allows INRs to cache data packets. Intentional names made the design of application-independent caching rather simple. With traditional naming schemes each application provides its own opaque names for its data units, and today's distributed caching schemes are tied to specific applications (e.g., Web caches). In contrast, intentional names give applications a rich vocabulary with which to name their data, while keeping the structure of these names understandable without any application-specific knowledge. Thus, the intentional names can be used as a handle for a cached object. Of course, it is still necessary to provide additional information to describe if or for how long the object should be cached; we therefore added a small number of additional fields to the INS message header to convey this information to the INRs.

3.3 *Printer*: a load-balancing printer utility

The printer client application starts when the user clicks on a printer icon on the floorplan display. The printer client application has several features. It can retrieve a list of jobs that are in the queue of the printer, remove a selected job from the queue provided the user has permission to do so, and allow the user to submit files to the printer. Job submissions to *Printer* can be done in two ways, one of which uses intentional anycast to discover the "best" printer according to location and load characteristics.

The first submission mode is the straightforward "submit job to *name*," where the *name* is the printer's intentional name. The second mode, which is one we find useful in day-to-day use, is to submit a job based on the user's location. The printer servers, which are proxies for the actual printers in our implementation, change the metrics that are periodically advertised to the INRs taking into account the error status, number of enqueued jobs, the length of each one, etc. The INRs forward jobs to the currently least-loaded printer based on these advertisements, and inform the user of the chosen printer. Advertising a smaller metric for a less loaded printer and using intentional anycast allows *Printer* to perform automatically balance their load.

For example, to submit a file to the least-loaded printer in room 517, the printer client sends the file with the following destination name-specifier:

```
[service=printer [entity=spooler]][room=517]
```

and sets the *Delivery* bit-flag to *any*. Note that the name of the printer is omitted on purpose. Using intentional anycast, INRs automatically pick the route that has the best metric for the specified printer name-specifier, which corresponds to the least-loaded printer in room 517.

4 Implementation

We have implemented INS and tested it using a number of applications, including those described in the previous section. Our INR implementation is in Java, to take advantage of its cross-platform portability; clients and services, however, are not constrained to be written in Java. In this section, we present the details of two aspects of INS: the architecture of an INR node, and the packet formats for intentional names.

INRs use UDP to communicate with each other. At an INR, the `Node` object manages all network resources. It maintains the `NameTree` that is used to resolve an intentional name to its corresponding name-record, a `NodeListener` that receives all incoming messages, and a `ForwardingAgent` to forward messages to INRs and applications. In addition, a `NameDiscovery` module implements the name discovery protocol, and a `NetworkManagement` application provides a graphical interface to monitor and debug the system, and view the name-tree. At the client, a `MobilityManager` detects network movement and rebinds the UDP socket if the IP address changes, transparent to applications.

The INR implementation consists of approximately 8500 lines of Java code, of which about 2500 lines are for the INS API. The API significantly eases application development—for instance, the *Floorplan* and *Camera* applications presented in Section 3 were each implemented in less than 400 lines of Java code (including both service and client code, but excluding the graphical user-interface), and the *Printer* application in less than 1000 lines.

Figure 10 shows the INS packet format for intentional names. The binding bit-flag (B) is used to determine whether early or late binding should be used, while the delivery bit-flag (D) is used to determine whether intentional anycast or multicast delivery should be used. Because name-specifiers are of variable length, the header contains pointers to the source name-specifier, destination name-specifier, and data, which give offsets from the beginning of the packet. This allows the forwarding agent of an INR to find the end of name-specifiers without having to parse them. INRs do not process application data. In addition, a *hop limit* field decrements at each hop and limits the number of hops a message can traverse in the overlay. The *cache lifetime* field gives the lifetime the data of this packet may be cached for, with a value of zero disallowing caching.

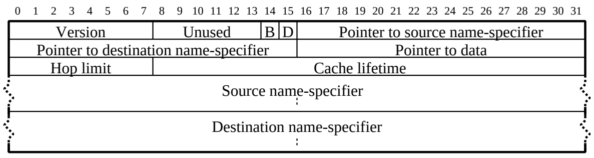

Fig. 10. The INS message header format.

5 Performance analysis and evaluation

In this section, we analyze the performance of the INS name lookup algorithm and present the results of our experiments with the lookup algorithm and name discovery protocol. These experiments were all conducted using off-the-shelf Intel Pentium II 450 MHz computers with a 512 kb cache and 128 Mb RAM, running either Red Hat Linux 5.2 or Windows NT Server 4.0, with our software built using Sun's Java version 1.1.7. The network nodes were connected over wireless RF links ranging between 1 and 5 Mbps.

5.1 Name lookup performance

Analysis To understand how INS scales with increasing lookup load, it is important to analyze the performance of the lookup algorithm. We analyze the worst-case run-time of the algorithm as a function of the complexity of the incoming name-specifier and the name-tree. To simplify the analysis, we assume that name-specifiers grow uniformly in the following dimensions (illustrated in Figure 11):

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

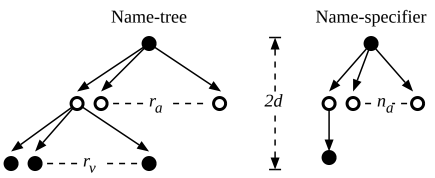

Fig. 11. A uniformly grown name-tree. Note that $d =$ (tree depth)/2 = 1 for this tree.

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:

$$T(0) = b$$

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

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

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:

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

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, 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 d will be near-constant and relatively small; indeed, all our current applications have this property in their name-specifiers.

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, if the two sets are sorted (as in our implementation). In the worst case the value of b is of 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 perform analytically since it depends on the number and distribution of names.

Experiments To experimentally determine the name lookup performance of our (untuned) Java implementation of an INR, we constructed a large, random name-tree, and timed how long it took to perform 1000 random lookup operations on the tree. The name-tree and name-specifiers were uniformly chosen with the same parameters as in the analysis in Section 5.1. We varied n, the number of distinct names in the tree, and measured lookup times. We limited the maximum heap size of the Java interpreter to 64 Mb and set the initial allocation to that amount to avoid artifacts from other memory allocation on the machine. The range of our experiments was limited by the memory required to store the distinct names to be looked up (part of the experimental apparatus) rather than the name-tree itself (which is much more compact).

We fixed the parameters at $r_a = 3$, $r_v = 3$, $n_a = 2$, and $d = 3$, and varied n from 100 to 14300 in increments of 100. Our results are shown in Figure 12. For this name-tree and name-specifier structure, our performance went from a maximum of about 900 lookups per second to a minimum of about 700 lookups per second. This experiment gave us a practical idea of how the base case b affects performance.

For the same experiment, we also recorded the amount of memory allocated by Java to the experiment; this amount should be greater than the actual name-tree size by only a constant amount. The strings we used for attribute and value names were only one (Unicode) character or 16 bits long, thus the memory is representative of what a more compact encoding of attributes and values would achieve. However the growth of the name-tree would remain the same, since after the first thousand names are in the name tree (where the graph curves up from zero) all of the attributes and values that exist are stored in the name-tree, and additional memory usage comes only from additional pointers and name-records. Our results are shown in Figure 13. The amount of memory allocated to the name-tree went from approximately 0.5 megabytes to 4 megabytes as the number of names was increased. We believe that this order-of-magnitude of lookup performance is adequate for intra-domain deployment, because of the load balancing provided by having multiple INRs and the parallelism inherent in independent name lookups.

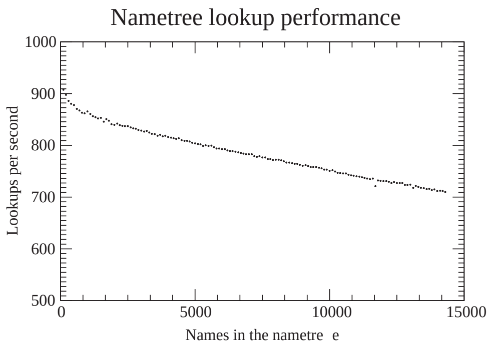

Fig. 12. Name-tree lookup performance. This graph shows how the name-tree lookup performance of an INR varies according to the number of names in its name-tree.

5.2 Name discovery performance

This section shows that INS is responsive to change and dynamism in services and nodes, by discussing the performance of the name discov-

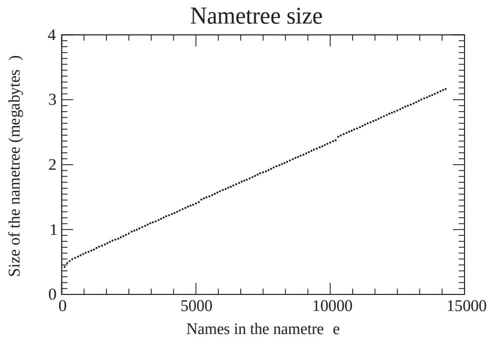

Fig. 13. Name-tree size. This graph shows how the name-tree size varies according to the number of names in its name-tree.

ery protocol. We measured the performance of INS in discovering *new* service providers, 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.

When an INR observes a new name-specifier from a service advertisement, it processes the update message and performs a lookup operation on the name-tree to see if a name-specifier with the same AnnouncerID already exists. If it does not find it, it grafts the name-specifier on to its name-tree and propagates a triggered update to its neighbors. Thus, 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 should be linear in the number of hops. The experimental question is what the slope of the line is, because that determines how responsive INS 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.

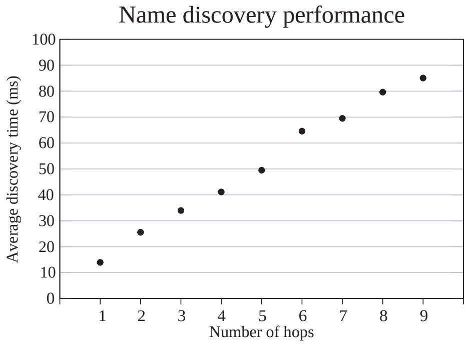

Fig. 14. Discovery time of a new network name. This graph shows that the time to discover a new network name is linear in the number of INR hops.

5.3 Routing performance

In addition to the lookup and discovery experiments, we also measured the performance of the overall system when both occur simultaneously. For these experiments, we sent a burst of one hundred 586-byte messages, gathered from the *Camera* application, between 15-second periodic update intervals. The name specifier source and destination addresses were randomly generated, on average 82 bytes long. The results are shown in Figure 15.

For the case in which the sender and receiver are on the same node, the processing and routing time varies somewhat with the name-tree size for the given virtual space, from 3.1 ms per packet with 250 names to 19 ms per packet with 5000 names. This is partially due to the speed of the name-tree lookups, but is also an artifact of the current end-application delivery code, which happens to vary linearly with the number of names. We observe a flatter line by when examining the data for packets destined to a remote INR in the name-tree of the same virtual space. For the most part, the next-hop processing time is about 9.8 ms per packet during the burst. In this case, name-tree lookups still occur, but the end-application delivery code is not invoked. This gives a better indication of the pure lookup and forwarding performance.

When the recipient resides in a different virtual space on another node, we observe a nearly constant time of 381 ms to resolve and route the burst of 100 messages. This steady time comes from the node having no knowledge of the end virtual space except for a next-hop INR address, which is requested and cached from the DSR on the first access, to which it can forward packets.

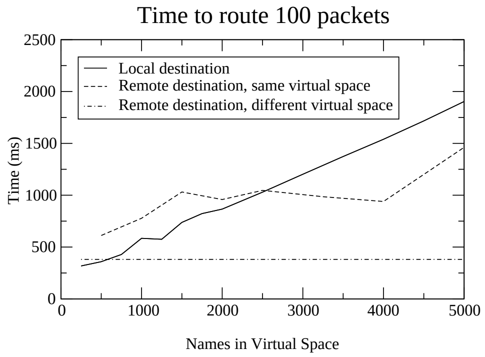

Fig. 15. Processing and routing time per INR for a 100-packet burst, in the intra-INR, inter-INR, and inter-virtual-space cases.

6 Related work

A flexible naming and resolution system for resource discovery, such as that provided by INS, is well-suited to dynamic network environments. INS uses a simple, expressive name language, late binding machinery that integrates resolution and routing for intentional anycast and multicast, soft-state name dissemination protocols for robustness, and a self-configuring resolver network.

INS is intended to *complement,* not replace the Internet DNS, which maps hostnames to IP addresses. DNS names are strictly hierarchical, whereas INS names use a more expressive attribute-based language. Unlike DNS, name propagation in INS resembles a routing protocol, tuned to perform rapid updates. In INS, names originate from and are refreshed by applications that advertise them. This enables *fate sharing* [9] between names and the corresponding services—if a node providing a service crashes, it will also cease to announce that service. In DNS, resolvers form a static overlay network consisting of the client's nameserver, the root server, and the owner domain's nameserver to route and resolve requests, unlike the INS self-configuring overlay.

There has been some recent activity in service discovery for heterogeneous networks of devices. Sun's Jini [21] provides a framework for spontaneous distributed computing by forming a "federation of networked devices" over Java's Remote Message Invocation (RMI). Jini does not address how resource discovery will work in a dynamic environment or when services fail, and can benefit from INS as its resource discovery system. Universal plug-and-play [42] uses a subset of XML to describe resources provided by devices and, like Jini, can benefit from INS as a discovery system. The Service Location Protocol (SLP) [44, 33] facilitates the discovery and use of heterogeneous network resources using centralized Directory Agents. The Berkeley Service Discovery Service (SDS) [11] extends this concept with secure, authenticated communications and a fixed hierarchical structure for wide-area operation. Unlike Jini, SLP, and SDS, INS handles dynamism via late binding, provides intentional anycast and multicast services, has self-configuring resolvers, and does not rely on IP multicast to perform discovery.

Numerous attribute-based directory services have been proposed in the past. The X.500 distributed directory [7, 36] by the CCITT (now the ITU-T) facilitates the discovery of resources by using a single global namespace with decentralized maintenance. INS differs from X.500 in its goals and mechanisms to achieve responsiveness and expressiveness; INS enables late binding and uses soft-state name dissemination. The INS resolver network is also different from the static X.500 hierarchy. These differences arise from differences in our environment, which is a dynamic and mobile network with little pre-configured infrastructure.

In addition to the wealth of classical literature on naming in distributed systems (e.g., Grapevine [4], Global Name Service [23], etc.), there has been some recent research in wide-area naming and resolution. For example, Vahdat *et al.* [43] present a scheme for *ActiveNames,* which allow applications to define arbitrary computation that executes on names at resolvers. INS and ActiveNames share some goals in common, but differ greatly in how they achieve them. In particular, INS does not require mobile code, relying instead on a simple but expressive naming scheme to enable applications to express intent, and late binding to be responsive to change. In addition, INS implements a self-configuring resolver network based on network performance.

An early proposal to decouple names from object locations was described in a paper by O'Toole and Gifford [28], where they describe a content naming scheme and its application to Semantic File Systems [18]. Their design and application of content names is very different from ours, but the underlying philosophy is similar. The *Discover* system [39] is an HTTP-based document discovery system that uses *query routing* to forward a query to the servers that contain the result. Discover is document-centric and uses parallel processes to search servers and merge the results.

Oki *et al.* introduce the *Information Bus* [30] to allow applications to communicate by describing the subject of the desired data, without knowing who the providers are. Other projects with a similar flavor include Malan *et al.'s* Salamander [25] and Talarian's SmartSockets [40]. These use a flat naming scheme, do not support late binding, and have statically configured resolvers. The idea of separating names from network locations was also proposed by Jacobson in the context of multicast-based self-configuring Web caches [20]. Estrin *et al.* build on this, exploring a diffusion-based approach to data dissemination in sensor networks using data attributes to instantiate forwarding state at sensor nodes [15]. Our intentional naming scheme has some features in common with these proposals, but differs in the details of the resolution, late binding and name dissemination processes, as well as the overall resolver architecture.

Cisco's DistributedDirector [8] resolves a URL to the IP address of the "closest" server, based on client proximity and client-to-server link latency. Unlike INS, DistributedDirector is not a general framework for naming and resolution and does not integrate resolution and routing. Furthermore, each resolver is independent in DistributedDirector, whereas they form a cooperating overlay network in INS.

IBM's "T Spaces" [24] enable communication between applications in a network by providing a lightweight database, over which network nodes can perform queries. However, this system has been optimized for relatively static client-server applications rather than for dynamic peer-to-peer communication, and uses a central database to maintain tuple mappings. Other architectures for object-oriented distributed computing are OMG's CORBA [29] and the ANSA Trading Service [13], where federated servers resolve client resolution requests.

Retaining network connectivity during mobility requires a level of indirection so that traffic to the mobile host can be redirected to its current location. Mobile IP [32] uses a Home Agent in the mobile host's home domain for this. With INS, the level of indirection to locate mobile services and users is obtained using the intentional naming system, since all traffic to the mobile service would go through the name resolution process. The tight integration of naming and forwarding enables continued network connectivity in the face of service mobility, and the decentralized INS architecture and name discovery protocols enhance robustness. A number of protocols for ad hoc or infrastructure-free routing have recently been proposed (e.g., [6]). These protocols, are essential to enable IP connectivity, but do not provide resource discovery functionality.

7 Conclusions

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. Our design goals were expressiveness, responsiveness, robustness and easy configuration. We presented the design, implementation and evaluation of an Intentional Naming System (INS) that meets these goals. INS uses a simple naming language based on attributes and values to achieve expressiveness, integrates name resolution and message routing to allow applications to be responsive to mobility and performance changes, uses periodic service advertisements and soft-state name dissemination protocols between replicated resolvers to achieve robustness, and deploys self-configuring name resolvers to ease configuration. The INS service model supports three types of resolution: early binding, where an application can obtain a list of IP addresses corresponding to a name, and two forms of late binding: intentional anycast and intentional multicast. Intentional anycast forwards a message to the "best" node satisfying a query while optimizing an application-controlled metric, and intentional multicast forwards a message to all names satisfying a query.

We presented the design and analysis of an efficient algorithm for name lookups and measurements of our implementation, which show that a Java implementation can perform between several hundred lookups per second (for complex name-specifiers) to a few thousand lookups per second. We evaluated the name discovery protocol and demonstrated that INS could disseminate information about new names in tens of milliseconds. We also measured the the processing time for name updates, analyzed the scaling bottlenecks, and found that namespace partitioning is a practical technique to improve the scalability of INS.

Our experience with INS has convinced us that using intentional names with late binding is a useful way of discovering resources in dynamic, mobile networks, and simplifies the implementation of applications. We emphasize that INS allows applications to efficiently track dynamic data attributes, because the choice of attributes to use in name-specifiers is completely under application-control. We therefore believe that INS has the potential to become an integral part of future device and sensor networks where decentralized, easily configurable resource discovery is essential.

There remain some important areas of research before the full benefits of INS can be realized. First, we need to carefully expand the set of supported operators in the resolution process, incorporating range matches in addition to exact matches of attributes and values. Second, the current INS architecture is intended for intra-domain deployment. We are actively developing a wide-area architecture to scale INS to wide-area networks. Third, the name discovery protocols need to be tuned to use bandwidth efficiently while disseminating names; some names are more ephemeral or more important than others, implying that all names must not be treated equally by the soft-state dissemination protocol [35]. And perhaps most importantly, we need to incorporate security mechanisms in the naming architecture before a more wide-scale deployment. Ultimately, the benefits of INS are in facilitating the development of useful applications and services, and we are implementing more applications to demonstrate the benefits of INS and to characterize the class of applications that INS facilitates.

8 Acknowledgments

This work was supported by a research grant from the NTT Corporation. We would like to thank Minoru Katayama and Ichizo Kogiku for their interest in this effort. We are grateful to John Guttag, Frans Kaashoek, and Suchitra Raman for useful suggestions on the design of INS, and to Dave Andersen, Frans Kaashoek, Barbara Liskov, Robert Morris, and Suchitra Raman for commenting on earlier drafts of this paper. This paper also benefited from the many comments of our "shepherd," Jean Bacon, as well as the SOSP reviewers.

References

1. R. Bellman. On a routing problem. *Quarterly of Applied Mathematics, 16(1):87-90*, 1958.
2. T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol–HTTP/1.0.* Internet Engineering Task Force, May 1996. RFC 1945 (`http://www.ietf.org/rfc/rfc1945.txt`).
3. S. Bhattacharjee, M. Ammar, E. Zegura, V. Shah, and Z. Fei. Application-Layer Anycasting. In *Proc. IEEE INFOCOM*, pages 1388–1396, March 1997.

4. A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An exercise in distributed computing. *Comm. of the ACM*, 25(4):260–274, April 1982.
5. T. Bray, D. Hollander, and A. Layman. Namespaces in XML. `http://www.w3.org/TR/1998/WD-xml-names-19980327`, March 1998. World Wide Web Consortium Working Draft.
6. J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proc. ACM/IEEE MOBICOM*, pages 85–97, October 1998.
7. CCITT. *The Directory—Overview of Concepts, Models and Services*, December 1988. X.500 series recommendations, Geneva, Switzerland.
8. Cisco—Web Scaling Products & Technologies: DistributedDirector. `http://www.cisco.com/warp/public/751/distdir/`, 1998.
9. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM*, pages 106–114, August 1988.
10. D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms. In *Proc. ACM SIGCOMM*, pages 14–26, August 1992.
11. 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.
12. S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2):136–161, May 1990.
13. J. P. Deschrevel and A. Watson. A brief overview of the ANSA Trading Service. `http://www.omg.org/docs/1992/92-02-12.txt`, February 1992. APM/RC.324.00.
14. H. Eriksson. Mbone: The multicast backbone. *Communications of the ACM*, 37(8):54–60, 1994.
15. D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proc. ACM/IEEE MOBICOM*, pages 263–270, August 1999.
16. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, Jan 1997. RFC 2068 (`http://www.ietf.org/rfc/rfc2068.txt`).
17. B. Fink. 6bone Home Page. `http://www.6bone.net/`, January 1999.
18. D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Semantic File Systems. In *13th ACM Symp. on Operating Systems Principles*, pages 16–25, October 1991.
19. Y. Goland, T. Cai, P. Leach, Y. Gu, and S. Albright. Simple Service Discovery Protocol/1.0. `http://search.ietf.org/internet-drafts/draft-cai-ssdp-v1-02.txt`, June 1999. Internet Draft, expires December 1999.
20. V. Jacobson. How to Kill the Internet. Talk at the SIGCOMM 95 Middleware Workshop, available from `http://www-nrg.ee.lbl.gov/nrg-talks.html`, August 1995.

21. Jini (TM). http://java.sun.com/products/jini/, 1998.
22. B. Kantor and P. Lapsley. *Network News Transfer Protocol.* Internet Engineering Task Force, February 1986. RFC 977 (http://www.ietf.org/rfc/rfc977.txt).
23. B Lampson. Designing a Global Name Service. In *Proc. 5th ACM Principles of Dist. Comput.*, pages 1–10, August 1986.
24. T. Lehman, S. McLaughry, and P. Wyckoff. T Spaces: The Next Wave. http://www.almaden.ibm.com/cs/TSpaces/, 1998.
25. G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A Push-based Distribution Substrate for Internet Applications. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 171–181, December 1997.
26. G. Malkin. *RIP Version 2.* Internet Engineering Task Force, November 1998. RFC 2453 (http://www.ietf.org/rfc/rfc2453.txt).
27. P. V. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of SIGCOMM '88 (Stanford, CA)*, pages 123–133, August 1988.
28. J. O' Toole and D. Gifford. Names should mean what, not where. In *5th ACM European Workshop on Distributed Systems*, September 1992. Paper No. 20.
29. Object Management Group CORBA/IIOP 2.3. http://www.omg.org/corba/corbaiiop.html, December 1998.
30. B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus (R) – An Architecture for Extensible Distributed Systems. In *Proc. ACM SOSP*, pages 58–78, 1993.
31. C. Partridge, T. Mendez, and W. Milliken. *Host Anycasting Service*, November 1993. RFC 1546 (http://www.ietf.org/rfc/rfc1546.txt).
32. C. Perkins. *IP Mobility Support*, October 1996. RFC 2002 (http://www.ietf.org/rfc/rfc2002.txt).
33. C. Perkins. Service Location Protocol White Paper. http://playground.sun.com/srvloc/slp_white_paper.html, May 1997.
34. J. B. Postel. *Transmission Control Protocol.* Internet Engineering Task Force, September 1981. RFC 793 (http://www.ietf.org/rfc/rfc0793.txt).
35. S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proc. ACM SIGCOMM*, pages 15–25, September 1999.
36. J. Reynolds. *Technical Overview of Directory Services Using the X.500 Protocol*, March 1992. RFC 1309 (http://www.ietf.org/rfc/rfc1309.txt).
37. J. Saltzer, D. Reed, and D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2:277–288, Nov 1984.
38. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications.* Internet Engineering Task Force, Jan 1996. RFC 1889 (http://www.ietf.org/rfc/rfc1889.txt).

39. M. Sheldon, A. Duda, R. Weiss, and D. Gifford. Discover: A Resource Discovery System based on Content Routing. In *Proc. 3rd Intl. World Wide Web Conf.*, 1995.
40. Rapid Infrastructure Development for Real-Time, Event-Driven Applications. `http://www.talarian.com/collateral/SmartSocketsWP-1.html`, 1998.
41. D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.
42. Universal Plug and Play: Background. `http://www.upnp.com/resources/UPnPbkgnd.htm`, 1999.
43. A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proc. USENIX Symp. on Internet Technologies & Systems*, October 1999.
44. J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol*, June 1997. RFC 2165 (`http://www.ietf.org/rfc/rfc2165.txt`).
45. M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (Version 3).* Internet Engineering Task Force, December 1997. RFC 2251 (`http://www.ietf.org/rfc/rfc2251.txt`).
46. D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proc. 17th SOSP*, pages 64–79, December 1999.