A Self-Configuring Resolver Architecture for Resource Discovery and Routing in Device Networks

by

William Adjie-Winoto

B.S., Electrical Engineering and Computer Science
Cornell University (1998)

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

Author . . . .
Department of Electrical Engineering and Computer Science
May 19, 2000

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

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

A Self-Configuring Resolver Architecture for Resource Discovery and Routing in Device Networks

by

William Adjie-Winoto

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

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, rapid fluctuations in performance, and node failures. This combination of heterogeneity and dynamism makes it hard for applications to discover the network locations of services that best satisfy their needs.

This thesis presents a communication system that allows network applications to send messages by describing the *attributes* of the intended destinations for their messages, rather than by explicitly listing the network locations (e.g., IP addresses) of the message destinations. Senders are thus relieved from having to know in priori the destination network locations, and receiver nodes are determined during message delivery time by the system, allowing them to be mobile, grouped, or dynamically adapted to different nodes (based on load, failures, or other measures). We build this communication system in INS, an Intentional Naming System, using expressive names to describe the attributes of the intended destinations for a message. The system is designed to be responsive and adaptive, easily configurable, and robust. We describe the INS architecture, the discovery and routing mechanism, and the distributed self-configuring protocol used by INS name resolvers. We describe our Java-based implementation and several applications developed using this system.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Hari Balakrishnan, for the guidance, advice and kindness he provided during the years I worked with him. His enthusiasm for research and excellent writing and presentation skills have been a constant source of motivation and inspiration. He has been an ideal advisor a student could hope for. I have very much enjoyed pursuing my graduate research with his guidance.

I thank Professor Nancy Lynch for her comments and suggestions aiding my design of the self-configuration algorithm in this thesis. I am grateful to Stephen J. Garland and Professor John Guttag for providing insight and helpful feedback on the development of the system.

I would like to thank the members of the Networks and Mobile Systems group. Elliot Schwartz, Anit Chakraborty and Jeremy Lilley were my cohorts in developing and implementing an Intentional Naming System. Bodhi Priyantha provided useful and insightful comments and discussions on the self-configuration algorithm for the system. I am glad that I had the opportunity to share my days (and nights) in the lab with them, discussing academic and otherwise and making LCS a fun place to work. I would like to acknowledge the other members of the group: Suchitra Raman, Deepak Bansal, Allen Miu, David Andersen, Alex Snoeren, David Evans, Dorothy Curtis, John Ankcorn and Srinivasan Seshan. They were always an endless source of valuable information, answers and suggestions.

Some of the text in this thesis, especially parts in Chapters 2 and 5, were taken from the previous paper that I wrote as co-author with Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley, and appeared on the 17th ACM SOSP [1].

My research was primarily supported by a grant from the Nippon Telegraph and Telephone Corporation (NTT).

Finally, I would like to express my sincerest gratitude to my parents and family for providing me with this opportunity and for loving and supporting me for all these years.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

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, rapid fluctuations in performance, and node failures.

To understand the various issues involved, consider the following scenario. A user walks into a new building on a campus and discovers the map of the region. He easily retrieves the information of available services that are accessible to him and has them displayed on the map. As he moves around, the map gets updated and new services are discovered. In addition, he easily gains access to and communicates with those available resources. For example, he can send a print job to the closest (based on geographic location) and least-loaded printer, retrieve files from a replicated server based on the network performance and server load with automatic fail over mechanism, discover and interact with people in the region that are reachable for communication, or retrieve the current image from all the mobile cameras in a particular region of the building (to find an empty meeting room, for example).

Various network applications require the ability to seamlessly handle node mobility, service location and dynamic services, respond to performance variations such as network latency, server load, or any other application-specific performance factors. Network applications may also desire information or functionality from dynamic groups of services, of which group memberships are dynamically changing based on the specific information that the services provide and the clients require.

This combination of heterogeneity and dynamism makes it hard for applications to discover the network locations of services that best satisfy their needs. Applications often know the services they are looking for or providing (i.e., their intent), but do not always know the best network locations to find or deliver them.

There is usually no pre-configured support for describing, locating, and gaining access to available services in heterogenous, mobile networks. While providing network connectivity over mobile communication has been extensively researched [6, 39], the functionality 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 — as diminishing hardware costs make it inexpensive to network all sorts of devices, the cost of deploying and running such a network infrastructure will be dominated by software and service management.

Based on our target environment and applications, we propose a design of a network communication system that allows applications to send messages by describing the *attributes* or characteristics of the intended destinations for their messages, rather than by explicitly listing the network locations of the message destinations. Senders are thus relieved from having to know in priori the destination network locations, and receiver nodes are determined during message delivery time by the system, allowing them to be mobile, grouped, or dynamically adapted to different nodes (based on the load status, node-failure condition, or other measures).

We build this communication system in INS, an Intentional Naming System, using an expressive *name* to describe the attributes of the *intended* destinations of a communication message. This name is called *intentional name* and uses a flexible and expressive naming scheme and query language to handle of a wide variety of services, devices and applications and to allow application-specific attributes as part of the name. As a resource discovery system, INS accounts for performance metrics, failure conditions, or any other metrics under application control. INS performs routing and forwarding of messages among INS name resolvers for better responsiveness to mobility and dynamic conditions.

1.2 Intentional Naming System (INS)

The main contribution of this thesis is the design and implementation of an Intentional Naming System with the following properties:

- INS allows network applications to gain access to and communicate with services by specifying their attributes, rather than by specifying where to find them. INS applications can therefore seamlessly handle node mobility and service dynamism, and take advantage of a flexible group communication service using an expressive name as the group handle[1].

- INS uses a novel distributed self-configuring algorithm that enables its name resolvers to configure themselves into an overlay network that is adaptive and self-improving based on network latency metrics.

Routing and forwarding of messages in INS is based on intentional names, which describe the attributes of the intended destination end-nodes. In doing so, INS integrates name resolution and message routing, operations that have traditionally been kept separate in network architectures. By performing the forwarding decision based on the destination name of the message, Intentional Name Resolvers (INRs) make the binding between the name and network location(s) at the message delivery time, rather than at request resolution time. We call this delivery *late binding*. INS late binding delivery is "best-effort" since INS provides no guarantee on reliable message delivery. This integration of name resolution and message routing leads to a general method for performing application-level routing using names.

An important characteristic of our target network environment is the dynamism of end-nodes, which includes *node mobility* and *service dynamism*. Node mobility

[1]Note that we do not rely on any pre-installed support (e.g., Mobile IP [39] or IP Multicast [15]) in INS.

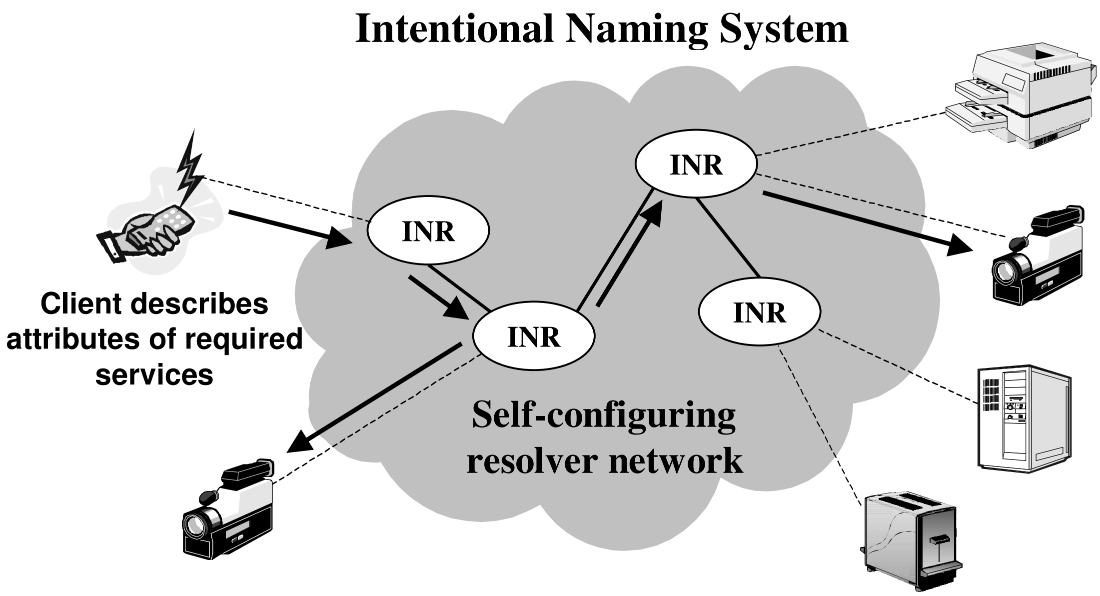

Figure 1-1: Intentional Naming System with its self-configuring network of name resolvers (INRs) allows applications to communicate by describing the attributes of the required services. INRs forward the message to the appropriate service nodes that satisfy the given attributes, allowing the applications to seamlessly handle node mobility and service dynamism.

occurs when the network location (e.g., IP address) of a node changes due to physical mobility or changes of the network interfaces used (e.g., from a wired Ethernet to a wireless radio frequency network). Service dynamism occurs when the end-node mapping to a service changes because of a change in the service availability (fail-over), or because of a change in the "optimality" of a service, such as server load (load balancing), or any other metrics under application control.

By allowing applications to specify a destination intentional name (rather than destination network locations) for their messages, INS provides a "level of indirection" useful for the applications to seamlessly continue communicating with end-nodes although the mapping from the name to network locations of end-nodes may change during the session.

Late binding delivery offers two basic types of message delivery service. An application may request that a message be delivered to the "optimal" service provider that satisfies a given name. The metric for optimality is under application control and has application-specific semantics that may correspond to certain measures, such as its current load. This kind of message delivery that forwards only to the best one is called *intentional anycast*. An application may also request that the message be delivered to *all* service nodes that satisfy a given name, in a message delivery method called *intentional multicast*. Intentional multicast enables data distribution, in which a sender pushes its data to all interested clients (identified by their intentional names) and clients advertise themselves as being interested in receiving the data. These two types of late binding delivery services allow INS to achieve *application-level* anycast and multicast.

In keeping with the end-to-end principle [48] and based on the difficulties experienced in deploying IP extensions (e.g., IP multicast [15], guaranteed services [13], active IP networks [58]), we provide the INS service as an overlay network over IP unicast, leaving the underlying network-layer addressing and IP routing architecture unchanged. INS handles network mobility and performs application-level intentional anycast and multicast over IP unicast; thus, the only network layer service that INS relies upon is IP unicast, which is ubiquitously available.

Another reason for leaving the core infrastructure unmodified is that often network-layer service alone does not completely satisfy the requirements of network applications at hand. Performing anycast delivery based on a specific network-layer metric such as hop-count or network latency alone, is ineffective from the point of view of many applications because it does not optimize the precise metric that applications require. For example, anycast delivery based on network-layer metric cannot locate the *least-loaded* printer, nor can it fail-over a failing service to another service node that advertises the same attributes. To enable anycast based on a metric that application cares about, INS allows an advertisement of an application-specific metric to be associated with a service name.

INS uses a decentralized network of INRs to discover names and route messages. INRs use *soft-state* [12] periodic advertisement from services, allowing applications to come and go without any explicit registration and deregistration. This design gracefully handles failures of end-nodes and services. Routing information for the overlay topology is treated as soft state, enabling network partitions and node failures to be detected in a timely fashion.

In constructing and maintaining the overlay network, INRs use a novel distributed self-configuration algorithm that forms an adaptive spanning tree amongst resolvers. Existing distributed algorithms that construct a spanning tree are not generally well suited to the INS operating environment because of the degree of dynamism due to node mobility, with nodes joining and leaving the system, and variations and fluctuations in network performance and reliability in this environment. The INS self-configuration algorithm adapts its topology to handle failures and performance variations in a scalable fashion, and attempts to evolve into a minimum spanning tree in the absence of change. The main feature of the algorithm is its capability to evolve any tree into an optimal tree in a distributed manner by using *relaxation* operations that asynchronously adapt the neighbor relationships between INS resolvers. Other distributed algorithms that construct an optimal spanning tree do not generally offer capability to relax neighbor relationships once the tree is completely constructed, which means that after the initial construction of the tree, changes of link metrics will require a significant amount of recomputation for the graph.

An important feature of our architecture is its incremental and easy deployment in the Internet, without modifying the existing Internet service model. INS service model is capable of performing application-level multicast and anycast delivery to mobile and dynamic nodes without requiring mobile IP, IP multicast, or any other IP extensions. 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). For a higher number of nodes, multiple disjoint INS overlay networks can be formed and used to handle different sets of namespaces.

1.3 Related Work

INS, to our knowledge, is the first system that integrates name resolution and routing, allowing sender applications to send messages by only describing the properties of the intended destinations (in an intentional name), requiring no knowledge of network locations of the destinations.

Since INS borders on many different areas, there exists a wide variety of related work. The areas that contain work that relates to INS are as following:

Naming and resource discovery systems. Several other network naming systems have been developed. The Internet Domain Name System (DNS) [34], provides mapping from hostnames to IP addresses. X.500 distributed directory by the CCITT (now the ITU-T) [7, 44] facilitates the discovery of resources by using a single global namespace with decentralized maintenance. Both DNS and X.500, as well as several classical literature on naming in distributed systems (e.g., Grapevine [4], Global Name Service [30], etc.) are all naming systems that provide a name lookup directory service that returns a network location. They differ from INS — in addition to providing a standard name directory service, INS performs message routing and forwarding based on intentional names, incorporating late binding for application-level anycast and multicast delivery. INS resolver network is tuned for dynamic and mobile networks with little pre-configured infrastructure.

There has been some recent activity in service discovery for heterogeneous networks of devices. Sun's Jini [29] provides a framework for spontaneous distributed computing by forming a "federation of networked devices" over Java RMI (Remote Message Invocation). Jini does not address how resource discovery will work in a dynamic environment or when services fail. Jini can benefit from INS as its resource discovery system. INS handles dynamism using late binding, provides intentional anycast and multicast services, and has self-configuring resolvers. Other architectures for object oriented distributed computing are OMG's CORBA [36] and ANSA Trading Service [16], where federated servers resolve client resolution requests.

Universal plug-and-play [55] uses a subset of XML to describe resources provided by devices and can benefit from INS as a discovery system. The Service Location Protocol (SLP) [57, 24, 40] facilitates the discovery and use of resources in a heterogeneous network using centralized Directory Agents. The Berkeley Service Discovery

Service (SDS) [14] extends this concept with secure, authenticated communications and a fixed hierarchical structure for wide-area operation. Unlike SLP and SDS, INS offers late binding with self-configuring resolvers, and does not rely on IP multicast to perform discovery. IBM's "T Spaces" [31] allow network applications to communicate by performing queries over a lightweight database that maintains tuple mappings. This system is optimized for relatively static client-server applications rather than for dynamic peer-to-peer communication and uses a central database.

ActiveNames [56] allows applications to define arbitrary computation that executes on names at resolvers. INS differs from ActiveNames in the way that INS does not require mobile code, relying instead on late binding to achieve responsiveness and flexibility. In addition, INS implements a self-configuring resolver network based on network performance.

Network-layer routing and forwarding. Mobile IP [39], anycasting [38], and IP multicast [15] are network-layer routing schemes that allow node mobility, anycast delivery, and group communication respectively. INS achieves a similar goal by using different approaches, not modifying the existing IP network infrastructure. INS offers a richer semantics by using a name as a handle for communication instead of a network address. A network group can be created on the fly more easily by wildcarding or ignoring certain parts of the name. Overhead of looking up a name during the forwarding in INS can be higher than the overhead of the other schemes, but as we will show in Section 3.4, an optimization by associating a label to a destination name and maintaining a cache of active labels for the forwarding can reduce much of the overhead of a name lookup.

In several early-developed network protocols (e.g., AppleTalk, NetBIOS) applications use name as their addressing scheme and the protocol on the node resolves the name into its corresponding network location before the packet is sent out, allowing changes of network locations to be transparent to the applications. These protocols, however, do not handle group communication and thus require each name to be unique. They utilize a large number of network broadcasts for name discovery and do not scale well beyond local area networks.

Generic application-layer services. Information Bus [37] shares a similar idea of abstracting the network locations of end-nodes and allowing applications to communicate by describing the subject of the desired data, without knowing who the providers are. Its target environment however is different. INS targets node mobility, group communications and dynamic environments beyond local area network and as such INS implements decentralized name resolvers running routing protocol and self-configuration protocol for the overlay network. Other projects similar in flavor include Salamander [33] and SmartSockets [54]. They have statically configured resolvers and topology. INS utilizes a self-configuring protocol to keep its overlay network adaptive and self-improving based on network performances.

More recent projects that explore application-level multicast delivery include Yallcast [21], End-system multicast [10], RMX (Reliable Multicast ProXies) [9] and RE-UNITE [52]. They offer wide-area multicast delivery service by tunneling multicast messages through IP unicast paths, which form the whole or some part of their multicast trees. Application-layer anycasting [3] introduces the idea of performing anycast delivery on the application layer.

Cisco's Distributed Director [11] resolves a URL request to the IP address of the "closest" server, based on some mapping function that accounts for among others client proximity and client-to-server network latency. Unlike INS, Distributed Director is not a general framework that integrates resolution and routing, and its resolvers are independent, in contrast to the cooperating INS resolvers that form an overlay network.

Intentional naming in other contexts. An early proposal for intentional naming was described in a paper by O'Toole and Gifford [35] and its application for file retrieval in the Semantic File Systems [23]. The *Discover* system [51] is a document discovery system that forwards a query to the server containing the result. Jacobson in [28] makes the case for intentional naming in the context of multicast-based self-configuring Web caching. Estrin *et al.* [27] in the context of sensor networks suggest a diffusion-based approach to data dissemination using data attribute to instantiate forwarding state at sensor nodes.

As a closing perspective, if we look at different switching technologies, networking has evolved from circuit switching that has a fixed path between end points of communication, to packet switching that allows a dynamic path but fixed end points for the communication. With late binding, INS allows not only a dynamic path (based on the underlying packet switching network), but also dynamic end points for the communication.

1.4 Roadmap

The rest of this thesis is organized as follows. We describe the design criteria of INS and our design of INS architecture in Chapter 2. We discuss the distributed name discovery and routing protocols used by INS and the message forwarding schemes, intentional anycast and intentional multicast, in Chapter 3. We describe our distributed algorithm and protocol for constructing a self-configuring overlay network in Chapter 4. The algorithm is used to form the INS resolver network and is general enough to be applicable to other networking systems that require an adaptive overlay topology that optimizes network performance. INS functionality and interfaces that are provided to the applications, as well as several applications that we have developed using INS, are described in Chapter 5. Then we turn to our implementation of INS name resolver, describe the INS message format and present several evaluation results in Chapter 6. We then conclude with a summary in Chapter 7.

Chapter 2

Design of INS Architecture

INS allows network applications to communicate and access services by specifying what their needs are, rather than where to find them. INS allows applications to seamlessly handle node mobility and service dynamism and provides a flexible group communication service using an intentional name as the communication handle. In Section 2.1 we present our design criteria for an Intentional Naming System. Section 2.2 describes our design of the INS architecture. We show some examples of possible naming languages used by INS in Section 2.3. Scalability issues are discussed in Section 2.4.

2.1 Design Criteria

We identify the following design criteria for INS based on our observations of target environments:

Responsiveness. Since INS is operating in a mobile and dynamic environment, it must adapt quickly to node and service mobility, performance fluctuations, and any other factors that can cause a change in the "best" network location of a service. INS must be responsive to handle dynamic name-to-address mappings that may change rapidly.

Self-configuration. The construction of the resolver network must take place in a fully distributed fashion, requiring no manual set-up or configuration. INS resolvers should self-configure, optimizing the network performance between them. The INS overlay network topology must adapt to network performance and evolve toward an optimal topology in a scalable fashion. The system should allow services and clients to join and leave the system without any manual registration and de-registration. Calculations of overlay paths for message delivery between applications should be dynamic and require no global knowledge of the overlay. The deployment of the system should avoid supplanting, and should minimize modification to, the underlying network infrastructure.

Robustness. The resolver architecture must avoid a single point of failure and is capable of recovering from inconsistencies in the internal state of the resolvers. The overall system must be resilient to application failures, such that failures of services and clients will not collapse the whole system. INS must be able to detect network partitions and node failures and recover from them in a timely fashion.

Allowing resource discovery and routing in heterogenous networks, INS employs an expressive name to describe the attributes of the intended destinations of a message. The name should be extensible by applications to include their own semantics as part of the names. For the exact format of a name language we leverage existing naming scheme and query language, such as *name-specifier* [50], or XML (Extensible Markup Language) [5]. It is an objective to design the resolver architecture to be largely independent of any specific name and query language used, such that different languages may be used as needed and future extensions to the system may allow interoperability of different name and query languages.

2.2 INS Architecture

INS architecture consists of INS applications and Intentional Name Resolvers (INRs). INS applications may be *services* or *clients*: services provide functionality or data and clients access and use them. INRs route client requests to the appropriate services, implementing simple distributed protocols that may be implemented even on computationally impoverished devices. Any device or computer 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, through which they exchange name information and maintain lazy-consistent cache of name information. There is a well known entity in the system, a Domain Space Resolver (DSR) that maintains a list of currently active INRs in the system. DSR can 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 that return the currently active INRs in the system.

2.2.1 INS Service Model

An application that would like to join the INS system communicates with one of the available INRs in the system. The list of available INRs can be obtained by contacting the DSR. Although potentially an application can attach to any available INR, it is beneficial for an application to attach to an INR to which it has the best connection (e.g., the one with the shortest round trip latency). An application advertises its names to an INR, describing its intent of being a client or provider for some services. The advertisement is in the form of an intentional name, which describes the attributes of the information they are providing (if a service) or seeking (if a client). An application can become a client of some services and a provider for other services at the same time.

Services, when advertising their names, can include an application-specific metric to be associated with the name. This metric will be used by INRs to select the "best" service node when multiple service nodes satisfy a given query. Applications can discover/interact with one another in the following ways:

An application can discover whether names having particular attributes (which correspond to some services/clients) exist in the system, by sending a request containing a query expression to an INR. Because name information is disseminated through

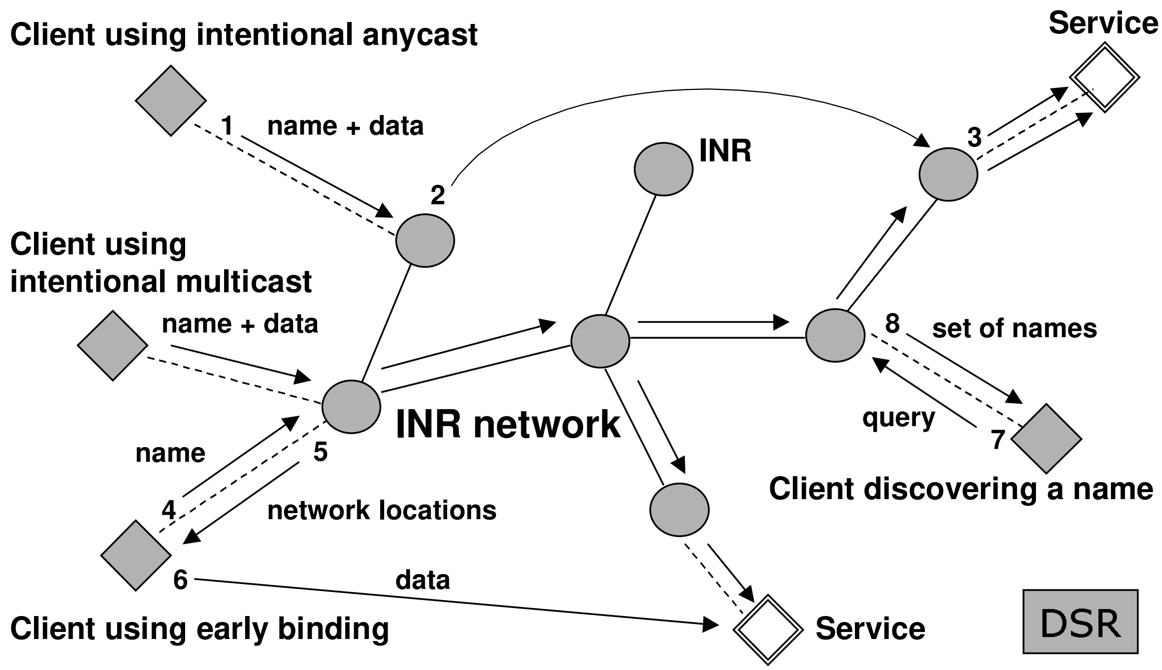

Figure 2-1: The architecture of an Intentional Naming System. The upper-left corner shows an application using intentional anycast: the application sends an intentional name and the data to an INR (1), which tunnels to the INR closest to the destination that has the least metric (2), which then forwards to the destination (3). The middle-left 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-left corner shows an application using early binding: the application sends an intentional name to an INR to be resolved (4), receives the network location (5), and sends the data directly to the destination application (6). The lower-right corner shows an application discovering names sends a query to an INR (7), receives a set of names that match the name in query (8). DSR (Domain Space Resolver) is a well-known entity in the system than maintains a list of currently active INRs in the system.

the INR network in a timely manner, a new service becomes known to other resolvers and through them to the clients.

Applications can use two late binding options — intentional anycast and intentional multicast — to handle dynamic situations. In late binding, the source application sends to an INR the message content (payload) together with an intentional name describing the properties of the desired destinations for the message content. The INR forwards the message and the associated payload directly to the end-nodes that match the destination name.

If the application requests intentional anycast, the INR selects exactly one of the end-nodes in its list that has the least metric and tunnels the message to the INR *closest* to that end-node, which in turn forwards it to the end-node. The closest INR is the INR to which the end-node attaches and advertises its names. In INS the metric determining the best end-node does not reflect a network-layer metric such as hop-count used in network-layer anycast [38]; rather, INS allows applications to advertise arbitrary application-specific numeric metrics such as average load.

In intentional multicast, INRs forward the message along the spanning tree overlay network to all the end-nodes satisfying the properties described in the destination intentional name of the message. Each INR knows its next-hop INR for the message delivery because of the routing protocol that is run on all INRs. INRs, in addition to exchanging information about names they know about, also send updates for routing information for each name, such that INRs know the shortest overlay route to get to every name in the system.

Since intentional multicast delivers to all the end-nodes that have their names matched, identical names that are announced by different applications need to be distinguished and recorded. INRs differentiate identical names advertised by different applications by associating a unique ID, the application's ID (*App-ID*), with each name. Every application is given an App-ID, which is constructed by concatenating the IP address of the node, the port number, and the start time of the application. The port number is used as part of the ID because multiple INS applications may reside at a single IP address but each using a different port number.

Each application is supposed to retain its App-ID for the duration of its lifetime, even in the case of mobility. That is, even though an application may obtain a new IP address because of its network mobility, it should retain and use its old App-ID. The start time, which is used as part of the App-ID, ensures that the ID stays unique; that is, if at the time being a different application comes up and takes the place of its old IP address and port number, the start time of this new application will be different, thus producing a different ID. This unique App-ID allows INS to handle node mobility seamlessly.

In the intentional anycast delivery, a sender has a choice of whether to include the App-ID of the destination name. If it does, INRs that receive the message will forward the message only to the destination name with the given App-ID. This is useful when a sender wants to maintain persistent communication to the same end-node under all conditions. If the sender does not include any App-ID for the destination name, INRs will select an optimal one based on the application-advertised metric and forward the message to that optimal end-node.

In addition to explicit name advertisements from applications, INRs also learn about new names by passively observing the headers of the messages they receive. When an INR receives a message from an application A that contains a destination name `d-name` and a source name `s-name`, in addition to forwarding this message toward the destination nodes matching `d-name`, it also adds to its *name table*[1] the necessary information about `s-name`, noting that the originator of `s-name` is application A. The INR can then distribute this new name information to other INRs in the system. This learning, called *inference*, enables INRs to forward any response from services back to the requesting clients without requiring the clients to explicitly advertise themselves (as clients). This inference mechanism is similar to that used by the *learning bridges* [41].

In addition to the late-binding delivery service, INS offers an early binding service that allows an application to lookup a destination name and obtain the corresponding network locations by sending to an INR a request for early-binding information of a particular name. Receiving the request, the INR returns a list of *early-binding records* corresponding to the name with each record consisting of an IP address, a port number and an additional set of [`port-number, transport-type`] pairs. Example of possible transport types in the additional set are HTTP [2], RTP [49], TCP [42], etc. This service model is useful when services and clients are relatively static. Early binding is similar to service provided by other naming systems, such as the Internet Domain Name System (DNS) [34]. In the case when multiple network locations satisfy a given

[1]We use the term *name table* to refer to a conceptual data structure used to store the correspondence between names and their associated information.

request, INRs return the network location that has the least application-advertised metric. This early binding service of INS is richer than the round-robin service of DNS, which does not account for any performance or failure conditions.

INS uses a decentralized network of INRs to discover names and route messages. INRs use *soft-state* [12] periodic advertisement from services, allowing applications to join and leave the system without any explicit registration and deregistration. This design gracefully handles failures of end-nodes and services. Routing information for the overlay topology is also treated as soft state, allowing network reachability and partition to be detected in a timely manner.

2.2.2 INS Resolver Network

INRs form a resolver network to exchange name and routing information, and to forward messages during intentional multicast delivery. The resolver network is formed as an overlay network over IP unicast, leaving the underlying network-layer addressing and IP routing architecture unmodified. In our current design, INRs self-organize into a spanning tree topology, providing loop-free connectivity. A list of active INRs is maintained by the Domain Space Resolver (DSR).

When a new INR joins the system, it contacts the DSR to obtain a list of currently active INRs. The new INR then picks one INR from the list to which it has the smallest round-trip latency and establishes a neighbor relationship (or *peers*) with it. If each INR does this, the resulting topology is a spanning tree.

Despite the local decision made by each new INR joining to minimize the latency, the resulting spanning tree will not in general be the minimum one. Hence, a new INR, after peering with an existing one, enters a *relaxation* phase, where it participates with other INRs in the system to adapt the neighbor relationships between INRs to evolve the spanning tree into a minimum one.

The algorithm for constructing and maintaining the INS MST topology is thus different from some existing algorithms [22, 25, 26] that construct a MST. The INS spanning tree algorithm is distributed and produces a spanning tree that is self-improving (based on the current latency metrics) toward an optimal spanning tree

(MST). The algorithm is distributed requiring no knowledge of global topology. The spanning tree is adaptive and self-improving, such that in the case where the measured round-trip latency between INRs change, the graph will adapt and turn itself back into a new MST based on the new latency metrics. The algorithm gracefully handles message losses; this is in contrast to some other algorithms (e.g. in [22]) that construct a MST by iteratively merging smaller components into a bigger one, in which case message losses may very well prohibit the tree construction. INS overlay topology is also in contrast to other overlay networks that maintain pre-configured, static neighbors such as the MBone [18] or the 6Bone [19].

The DSR can be implemented as an extension to a DNS server for the administrative domain, or may be assigned a well known DNS name in the domain (e.g., dsr.lcs.mit.edu). Detail implementations of a DSR can be found in [32].

2.3 Name Language for Intentional Names

An intentional name conveys the intent of an application in providing or accessing services. INS uses a simple language based on attributes and values for its names. Intentional names must be expressive, allowing applications to include attributes and values specific to them as part of the name. They should be extensible to describe a rich variety of network resources.

An example of an intentional name in the INS name-specifier format is shown in Figure 2-2 with its graphical view in Figure 2-3 [50]. This name refers to a camera service located in the oval-office of whitehouse (with resolution 640x400 and gif format) having a DNS name of bill.whitehouse.gov.

Alternatively, XML (Extensible Markup Language) [5] can be used to describe an intentional name. XML uses application-defined tags as "meta-data", allowing expressive names. One way to represent the previous intentional name in XML format is shown in Figure 2-4. Figure 2-5 shows another example of an XML expression that refers to any class or all classes in which *both* Jeff and Susan enroll. This XML expression is not directly mappable to the INS name-specifier format because

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

Figure 2-2: An example of an intentional name in name-specifier format. This name-specifier describes an object in the oval-office that provides a camera service (with 640-by-400 GIF images) and has a DNS name of bill.whitehouse.gov.

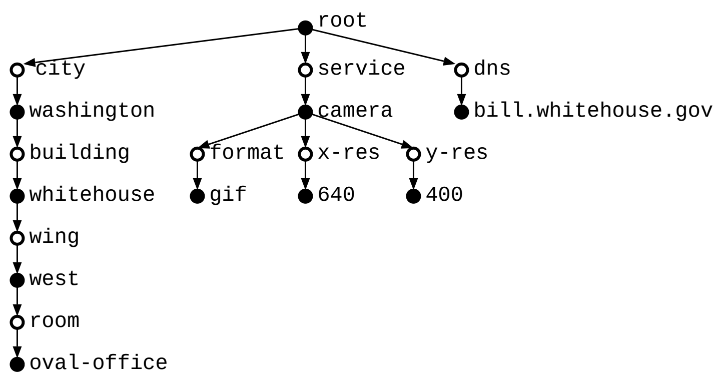

Figure 2-3: A graphical view of an example name-specifier shown in Figure 2-2 (from [50]).

it contains duplicate tags.

The query language currently supported in INS includes exact matches of attributes and values and wildcard (*) matches (which will match any values). The language also allows omissions of *don't-care* attributes.

We note that a detailed discussion of the INS name-specifier name and query language can be found in [50]. We also note that query languages that are available for XML include XML-QL [17], XQL [46], Quilt [45] and XSet [59].

The INS resolver architecture and protocols are designed to be decoupled from any specific name and query language and the architecture allows usage of features provided by each different name and query language. Of course, services described in one language may not be directly mappable to other languages, but future extension to

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

Figure 2-4: An intentional name in XML that is equivalent to the name-specifier shown in Figure 2-2.

```
<class>
     <name>Jeff<grade>A</grade></name>
     <name>Susan<grade>B+</grade></name>
</class>
```

Figure 2-5: Another example of intentional name in XML. This name refers to any class or all classes in which both Jeff and Susan enroll.

the system may incorporate some language inter-operability conversion. Our current implementation uses the name-specifier format and query language.

2.4 Scalability

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). In general, load processing (for query) and size of name table are two constraints that can limit the number of nodes that join the system at any particular time. To handle excessive loads from applications in name query and/or message forwarding, we allow new INRs to be spawn to join the INS resolver network. Of course, mechanisms to distribute existing loads or new loads among INRs are needed for the load balancing. The size of the name table grows proportional to the number of services running in the system, such that there is a particular upper limit for the size of name table where looking up a name from the table becomes prohibitive, due to the name lookup processing. To handle dynamic networks with a larger number of names, INS allows multiple overlay resolver networks to be formed, each with its own (disjoint) namespace. For example, we may partition services in the system based on geographic locations of services, or perhaps based on the accessibility of services (e.g., partitioning into public, group, and personal services). In this scheme, since namespaces are disjoint, applications in one INS overlay network are not able to discover names in another overlay network, unless of course it knows the exact INS overlay network to join to find the required services — i.e., there is no discovery mechanism between disjoint overlay networks. Applications, of course, can join multiple INS networks to access services on those networks. The current version of INS does allow clients and services to join multiple INS overlay networks. In addition, the system allows name discovery across multiple disjoint overlay networks and inter-communications between disjoint overlay networks; techniques to scale the INS in the local- and wide-areas are described in [32].

Chapter 3

Discovery, Routing and Forwarding in INS

Intentional name resolvers (INRs) replicate, creating a decentralized name resolution architecture, and form an overlay network among themselves, over which updates of valid names and routing information flow. Section 3.1 describes a distributed name discovery protocol used by INRs to disseminate and maintain up-to-date name information. The routing protocol run by INRs to determine the route on the overlay to each name in the system is discussed in Section 3.2. In Section 3.3 we discuss the desirable properties of soft-state name and routing information and present some possible optimization methods to lower the bandwidth consumption incurred by periodic refreshes of soft-state. Based on the name and routing information that they have obtained, INRs perform forwarding of messages based on intentional names, offering two basic types of late-binding delivery: intentional anycast and intentional multicast. Message forwarding is discussed in Section 3.4.

3.1 Name Discovery

Services periodically advertise themselves to describe what they provide to one of the available INRs in the system, usually to the local INR to which they have the best connection. Each INR listens to these periodic announcements to discover services running at different end-nodes. To make sure this service information available for system-wide resource discovery, each INR that receives a new name information propagates the information to all other available INRs in the system through the INR network.

The name discovery protocol treats name information as soft-state [12, 43], 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 outdated information. Using a soft-state mechanism for name information has two important consequences:

- Clients and services may join and leave the system without any explicit registration and de-registration, because new names are automatically disseminated and expired names automatically eliminated after a timeout. Of course advertisers may also explicitly remove outdated names that they have advertised before they time out.

- Soft-state information improves the robustness of the system against application failures. Any incorrect information associated with a name will be refreshed in the next cycle of periodic advertisements. Moreover, since names originate from and are refreshed by the applications that advertise them, soft-state introduces a notion of *fate sharing* [12] between names and the corresponding services — if a node providing a service crashes, it will also cease to announce that service and hence all its names will expire after a lifetime, preventing the system from announcing outdated name information.

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

- The IP address and the port number of the end-node announcing this name, and a set of [port-number, transport-type] pairs. The set of port-number and transport-type (e.g., HTTP [2], RTP [49], TCP [42], etc.) are returned to the client to allow it to implement early binding.

- An application-advertised metric for intentional anycast delivery and early binding. This metric may reflect any property that the service wants anycast routing on, such as current or average load. This metric is used to determine the optimal service node that satisfies a given request, providing a metric-based resolution that is richer than standard round-robin techniques.

- A unique identifier of the application announcing the name, called the *App-ID*, used to differentiate identical names that originate from two different applications on the same node. App-ID is created by concatenating the IP address, the port number and the start-time of the application.

- An identifier of the INR closest to the application announcing the name, i.e., the INR to which this application announces the name to. This identifier of an INR is called *INR-ID*. The use of the INR-ID of the closest INR in a name update to allow some optimizations in the routing and forwarding operations is discussed later in this chapter.

INRs use periodic name updates to refresh entries in neighboring INRs and to reliably disseminate name information. Triggered updates occur when an INR receives an update from one of its neighboring INRs or from an application that contains new information (e.g., a newly discovered name) or information that is different from the one previously known (e.g., due to node mobility a different INR-ID may be associated with a name).

Figure 3-1 shows the pseudocode used by INRs to process a soft-state name update. First, the INR checks what type of update it is. If the update is to merge a name, it adds or updates the name information in its name table accordingly. The INR-ID field in the name update is the ID of the INR to which the application announces its name. Hence, if the update is coming directly from an application, that INR-ID should be set to my-INR-ID. After that, the INR also sends a triggered update to all its neighbors (except the neighbor from where the update came). If the

```
if receive a name update (Name, App-ID, Metric, Early-binding,
INR-ID) from n
   let r be the result of looking up from name table for
      Name announced by App-ID

   if (name update is to MERGE the name)
      if (r = null) or
         (some information in name update is different from r)

         if (n is a neighbor INR)
            merge the following record: (Name, App-ID, Metric,
            Early-binding, INR-ID) to name table

         else if (n is an application)
            merge the following record: (Name, App-ID, Metric,
            Early-binding, my-INR-ID) to name table

         associate the record with a lifetime
         propagate name update to all neighbors, except n

   else if (name update is to REMOVE the name)
      if (r != null)
         remove the record containing both Name and AppID from
         name table

         propagate name update to all neighbors, except n
```

Figure 3-1: Soft-state name update processing

update is to remove a name, the INR simply removes the name and its name-record and propagates this update to its neighbors (except the neighbor from where the update came). The code assumes that there is another process that sends periodic refreshes of valid names to appropriate neighbors.

Here, we use the term *name table* to refer to a conceptual data structure used to store the correspondence between names and name-records. The actual implementation of a name table is usually a tree (rather than a table) (e.g., a *name-tree* in [50]) to speed up the name lookup process.

When clients make name resolution requests, INRs resolve them using the information obtained from service advertisements and neighboring INRs' name updates. In addition to sending name resolution requests, clients can send a query expression to discover whether names with particular attributes exist in the system. This mechanism is useful for clients to bootstrap in a new environment.

3.2 Routing Protocol

In addition to maintaining up-to-date name information, INRs maintain the overlay routing information to get to every name in the system. INRs use this information for intentional multicast forwarding to determine the appropriate next-hop INR when multicast messages arrive. Intentional anycast, on the other hand, does not require this information since it tunnels the message and does not forward the message across the INR network.

One possible way of obtaining the next-hop information of the overlay route toward a name is by simply observing the direction from where the name update (sent by the name discovery protocol previously discussed) is coming from, since that direction must be the one toward the source of the name. This is equivalent to running a routing protocol for each name. We observe, however, that many names share a single route — in particular, all names that are originally disseminated by a given INR have the exact same route on the overlay network in getting to other INRs in the system (since routes are based on the topology of the overlay). This suggests that performing routing on a per-INR basis is a natural approach, rather than routing on a per-name basis, thereby reducing the number of updates that need to be sent when the overlay topology changes. This approach is beneficial since the INS overlay topology is dynamic and adaptive to network performances (discussed in Chapter 4), which may change from time to time depending on the network status. We note that, as discussed in Chapters 4 and 6, topology change occurs only when there is some considerable performance benefits gained as the result.

Using this approach, in addition to running a name discovery protocol, INRs run a routing protocol that sends updates about overlay routes to reach different INRs in the system. The number of route updates is proportional to the the number of available INRs, and not to the number of names in the system. While the name discovery protocol disseminates the information associated with a given name (i.e., the name-record of the name) by using a name update, routing protocol disseminates the availability of a route on the overlay INR network to reach a given INR that disseminates a given name, by using a route update.

Each route update contains routing information for an INR-ID, consisting of the next-hop INR and the hop-count metric for the route toward the INR-ID. Since the topology of the INR network is a tree, a flooding of a route update initiated by an INR is enough to determine the path to the INR. Next-hop INR of the route is determined simply by observing from where the route update message is coming[1].

INRs treat the routing information (i.e., overlay paths to reach different INRs in the system) as soft-state. Similar to the soft-state name information, soft-state routing information requires periodic refreshes to prevent valid routes from expiring. Thus, each INR periodically initiates a flooding of a route update to the INR network to refresh its route.

Treating routing information as soft-state improves the robustness of the system against network partition and INR-node failures. That is, network reachability and INR-node failures will be detected in a timely manner. Without having periodic soft-state refreshes, detecting whether certain routes are still valid may not always be possible without requiring knowledge of global INR network topology. Aliveness of an INR can be detected in a timely manner only if there are periodic refreshes to validate its availability. Moreover, soft-state routing reduces the complexity of the overlay routing machinery; INRs build their routing table simply by listening to the route update messages passing by, rather than by requiring transactional operations.

A routing table is used to store the correspondence between an INR-ID and its route on the overlay network (i.e., the next-hop INR toward that INR-ID). Figure 3-2 shows the relationship between a name table and a routing table for the process of resolving a name to its next-hop INR (performed during intentional multicast forwarding). Determining the next-hop INR for a name requires first looking up the

[1]Hop-count metric in the route update is hence not necessary for determining route, but is included because the overlay topology management makes use of this information to perform a *quick healing* when a network partition occurs (discussed in Section 4.4).

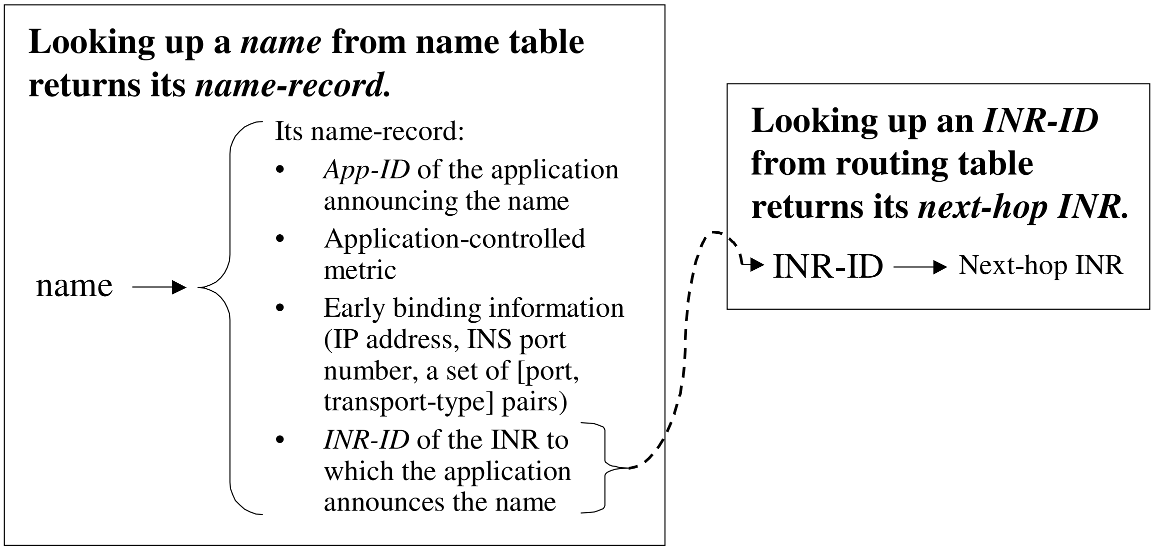

Figure 3-2: Name resolution to a next-hop route. Each entry in the name table provides information about a name, including application-controlled metric, early binding information, App-ID and INR-ID for the name. Each entry in the routing table provides the next-hop route information for INR-ID. Resolving a name to its corresponding next hop INR needs first a lookup in the name table to retrieve the associated INR-ID of the name, and then a lookup in the routing table to determine the next hop INR for that INR-ID.

name from the name table, which returns the INR-ID associated with the name (i.e., the INR-ID of the INR *closest* to the application announcing the name), followed by looking up that INR-ID from the routing table to determine the next-hop INR of the route. The implementation uses an optimization that removes the second lookup by incorporating "pointers" from entries in the name table to the entries in the routing table (see Section 6.2).

Mobility. INS name discovery and routing protocols keep track of changes due to network mobility and physical mobility. Because of node mobility, clients and services may change their network locations and may attach to a different INR after the mobility. Since applications periodically refresh their advertised names, INRs can track any changes that occur due to mobility. If an application's IP address and/or port number change because of mobility, INRs quickly update the information in their name table accordingly. However, if the mobile application now acquires a new IP address (and perhaps a port number), the INRs need to distinguish between an existing application from before the mobility and a new application coming up that happens to announce the same names. INRs distinguish each application from its unique *App-ID*[2]; a mobile application will retain its App-ID in the event of mobility and this ID is sufficient for INRs to discern whether the name refresh comes from a pre-existing mobile application or a new one joining.

If a mobile application with App-ID `id1` attaches to a different INR after the movement and sends a name refresh from its new location to the INR, the INR notices that the name refresh from `id1` is now sent directly to it, indicating the application node has moved and attached to it. Thus, the INR floods a triggered name update letting others know about this new route to App-ID `id1`.

3.3 Soft-State Discussion and Optimizations

Name discovery protocol treats name information as soft-state, associated with a lifetime. Such state needs refreshes to prevent it from expiring. This is in contrast to the hard-state information, which has no timeout and never expires; an explicit removal operation is required to remove hard-state information.

Soft-state names offer the benefits of robustness and fate sharing, allowing applications to join and leave the system without explicit registration and de-registration. This is very useful in the system where intentional names change rapidly, coming and going in a short interval. However, in the system where only a fraction of the names actually change, benefits of soft-state come at the cost of having to send name updates periodically, including the ones that have not changed, just to prevent them from expiring. One way to minimize the number of name periodic updates, while still attaining the benefits of fate sharing and implicit registration/de-registration, is to split the maintenance state into *soft-state* between applications and their corresponding first-hop INRs[3] (i.e., periodic advertisements), and *hard-state* between

[2] An *App-ID* is constructed by concatenating the IP address of the node, the port number, and the start time of the application (discussed in Section 2.2).

[3] A first-hop INR is the INR to which an application attaches for name advertisements, queries, and other INS services.

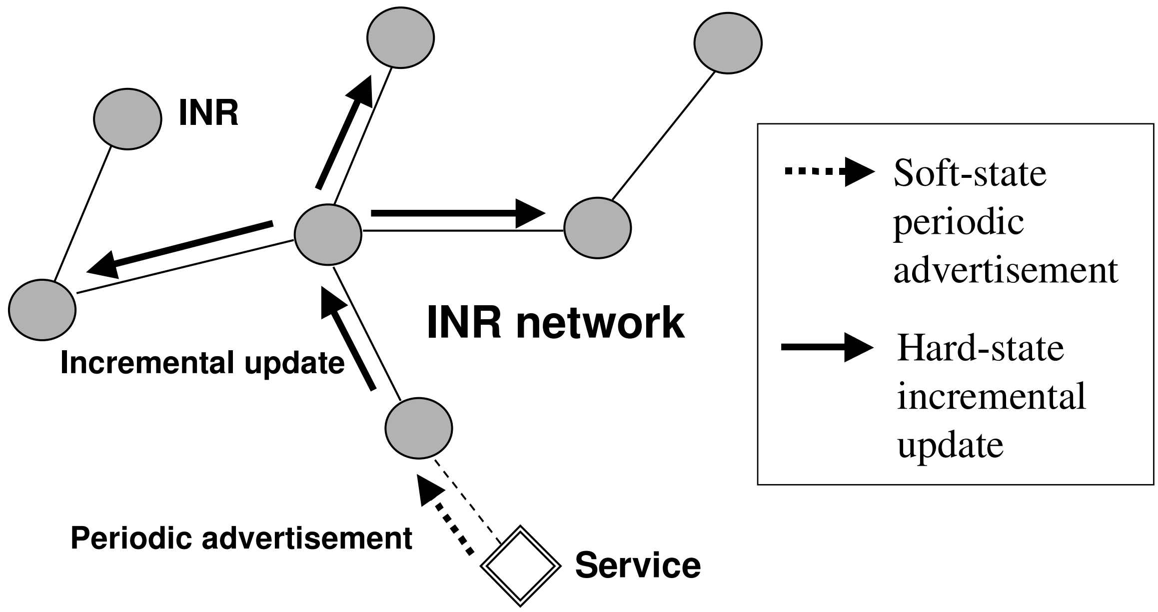

Figure 3-3: Minimizing number of name updates between INRs by splitting state maintenance of names into soft state (between applications and their first-hop INRs) and hard states (between INRs). Applications *periodically* advertise themselves and INRs send hard-state *incremental* updates (i.e., send only what has changed — *not* periodically) between each other.

INRs (using incremental updates). This means that only the first-hop INR maintains a lifetime of name information and other INRs never expire the information unless the first-hop INR tells them so. That is, other INRs follow whatever information the first-hop INR has for that name using *incremental updates*. Incremental updates mean to send only the name-records that change and using hard-state means to remove only when there is an incremental update to explicitly remove the name.

Figure 3-3 shows this optimization and Figure 3-4 shows the optimized algorithm for name-update processing. In this algorithm, some names are soft-state and some are hard-state. Names that are received directly from application advertisements are soft, but the ones received from neighbors are hard-state. The code assumes that there is another process that monitors which soft-state names have expired and then sends incremental updates to neighbors to remove those expired names.

In this design, the responsibility for maintaining the up-to-date mappings of a name is delegated to the first-hop INR, and the name updates between INRs are

```
if receive a name update (Name, App-ID, Metric, Early-binding,
INR-ID) from n
   let r be the result of looking up from name table for
      Name announced by App-ID

   if (name update is to MERGE the name)
      if (r = null) or
         (some information in name update is different from r)

         if (n is a neighbor INR)
             merge the following record: (Name, App-ID, Metric,
                Early-binding, INR-ID) to name table
             make the record never expire

         else if (n is an application)
             merge the following record: (Name, App-ID, Metric,
                Early-binding, my-INR-ID) to name table
             associate the record with a lifetime

         propagate name update to all neighbors, except n

   else if (name update is to REMOVE the name)
      if (r != null)
         remove the record containing both Name and App-ID from
         name table

         propagate name update to all neighbors, except n
```

Figure 3-4: Modified name update processing to handle soft-state and hard-state.

reduced. However, this approach introduces a new problem. Since name information in the non-first-hop INRs is stored as hard-state, when a network partition that splits the overlay network into multiple disconnected components occurs, how do we know which names are still valid and which are not? Using a soft-state mechanism, names that are unreachable will eventually time out and get removed, but there is no timeout mechanism for the hard-state case. How do we know which names are no longer valid?

The answer to this question lies in observing the network reachability information. This information is still soft-state because of the soft-state (overlay) routing information maintained by the routing protocol. By correlating name table and routing table, the name discovery protocol can determine which names are no longer valid. That is, if the route entry for some INR-ID is no longer valid in the routing table, then it must be that all names that have that INR-ID associated with them are no longer valid either in the name table.

The routing table may display transients during changes in the overlay topology. This might be a problem in the case when the partition occurs only for a brief period (such that the connectivity is restored before the route entries time out), but during that brief period of disconnectivity an explicit name removal update starts being flooded (but does not get to the disconnected INRs). Those INRs that are in temporary isolation never know about the name removals and correlating with the routing-table entries, which have never timed out as yet, is not helpful. How can INRs synchronize their name tables? One simple solution used here is that to keep any deleted names for a short period of time to ensure that INRs in some partitioned component have already removed unreachable names. If connectivity is restored before the routes time out, then by keeping deleted names for a while, the INR can inform the newly connected component of the explicit name removal.

Another logical question following the above discussion would be "is it also possible to use hard-state (overlay) routing information to reduce the number of periodic updates for the routes?". The answer to this is not obvious, and it is not clear whether it is either possible or even desirable! Hard-state routing information imposes a difficult synchronization process for routing tables when a network partition occurs. Using soft-state routing information, routes that are unreachable will eventually time out and get removed. However, when there is no timeout for route information (as with hard-state), an INR cannot be sure which routes are supposed to be removed when the partition occurs without having the knowledge of the global overlay network topology. Each INR in its distributed view of the network topology knows only the directional vector toward an INR. This directional information, however, may have transients when the overlay topology changes, during which the information is not accurate. Hence, relying solely on the directional vector information for a route to determine whether that route is supposed to be removed is neither robust nor accurate. Since everything is permanent in the hard-state case until an explicit "remove" message arrives, incorrect information may persist in an INR for a long period of time, thus reducing the responsiveness and robustness of the system.

Supposedly, we allow some inaccuracy of routes that is caused by network partitions, but require that as soon as the partition is healed all routing information must be synchronized. This relaxed requirement is often impractical, but even with this relaxed requirement, it can be shown that synchronizing routing tables after a partition is healed requires complex message logging and sequence numbering for each update, or otherwise special refresh messages need to be formed, by having all the INRs each broadcasts its availability again right after the connectivity is restored. Of course, allowing inaccuracy of routing information during network partition is not as robust. Hence, INS routing protocol prevents this situation and uses soft-state periodic refreshes for routing information.

To conclude the soft-state discussion, INS splits name information into soft-state names (between applications and their first-hop INR), and hard-state names (between INRs). This mechanism allows the system to attain the benefits of implicit registration/de-registration and fate-sharing properties with less network bandwidth consumption used for name updates between INRs. Hard-state names between INRs, however, introduce a difficulty in detecting unreachable names due to network partitions that split the INR network. The soft-state (overlay) routing information is the answer to detecting unreachable INRs, and thus detecting unreachable names disseminated by those INRs.

3.4 Message Forwarding: *Anycast* and *Multicast*

The central activity of an INR is to resolve an intentional names to their corresponding network locations. When a message arrives at an INR, the INR performs a lookup on the destination name in its name table to obtain the corresponding name-record of the name. By integrating name resolution and message routing in the late binding process, INS enables clients and services to continue communicating even if the name-to-location mappings change during the session.

Late binding offers two basic types of message delivery: intentional anycast and

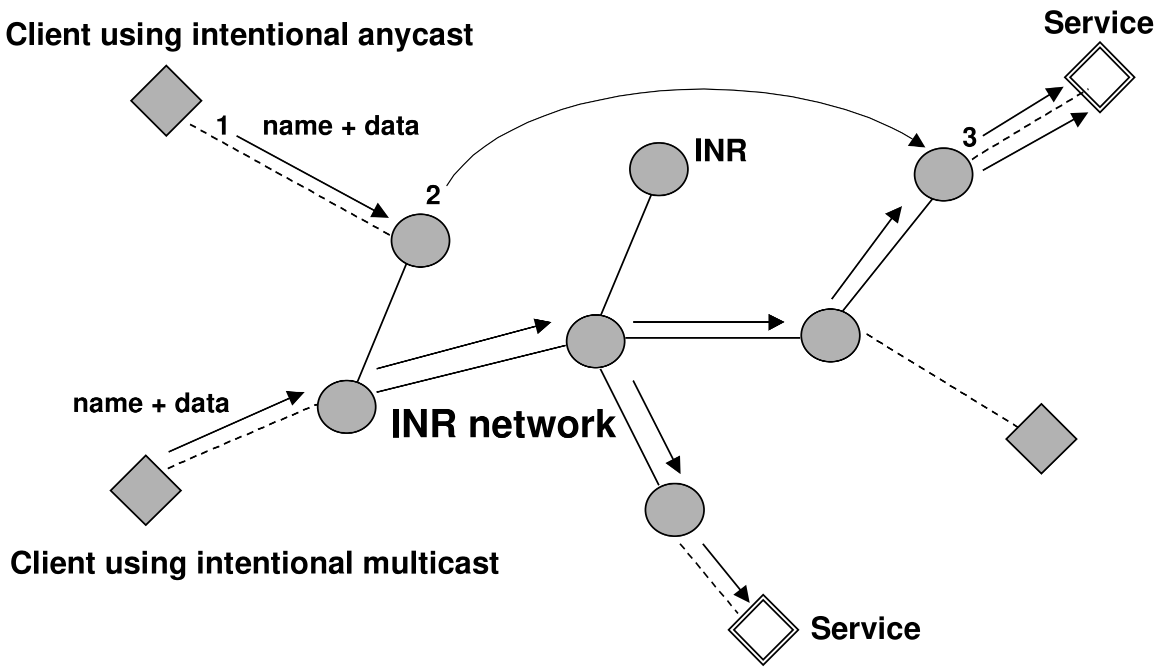

Figure 3-5: Intentional anycast and intentional multicast forwarding.

intentional multicast. Figure 3-5 shows the message forwarding performed by INRs in each type of delivery.

3.4.1 Intentional Anycast

In intentional anycast, when multiple end-nodes satisfy a given name request, the INRs forward the message only to the optimal one. Optimality is based on the application-advertised metric that is included when the application advertises its name. This metric is under application control, such as its current load. An application requesting intentional anycast sets the *Delivery* flag to *any* and includes an intentional name describing the attributes of the intended destinations for the message in the message header, and then sends the message to an INR. The INR performs anycast forwarding of the message using the pseudocode shown in Figure 3-6. To make the pseudocode more readable, several simplifications have been made, in particular, the handling of null result and the details of the inference mechanism are not shown.

First, the INR learns by *inference* about the source name of the message. It

```
anycast packet p that is received from n
{
   perform inference on the source name of p

   let nrset = the set of name-records returned by the look up
      process in looking up name table for destination name of p

   if (n is an INR)
      let filtered-set = all name-records in nrset that have
      INR-ID equal to my-INR-ID
   else if (n is an application)
      let filtered-set = nrset

   let best-record = name-record in filtered-set that has the
      least metric

   if (INR-ID in best-record = my-INR-ID)
      forward p to the application with App-ID in the best-record
   else
      forward p to the INR with INR-ID in the best-record
}
```

Figure 3-6: Pseudocode for intentional anycast.

then looks up all the end-nodes that satisfy the destination name of the message and picks the one that has the least metric. Before selecting the optimal one, it performs a filtering to prevent the tunneled message from circling around different last-hop INRs due to rapid changes of metrics determining the optimal end-node. Then, if the INR is already the one closest to that optimal end-node, it forwards directly to the end-node; otherwise, it forwards to another INR that is closest to it.

Intentional anycast performs anycast delivery based on the service-advertised metric, i.e., a metric under application control. This is in contrast to IP-level anycasting, which selects a route based on a network-layer metric, such as hop count or network latency, which may not optimize a metric useful to applications.

The decision to forward a message first to the INR *closest* to the end-node, rather than forwarding it directly to the optimal end-node is due to the dynamic *metric* of the network. In a number of cases, this method improves the responsiveness of the system, since changes to the name information are usually tracked more rapidly by the closest (local) INR rather than the one farther away; i.e., an INR usually has more up-to-date information about its directly connected end-nodes. The benefits of this are more apparent when the INRs span a wide-area heterogeneous network.

3.4.2 Intentional Multicast

In intentional multicast, when multiple end-nodes satisfy a given name request, the INRs forward the message to all those end-nodes. An application requesting intentional anycast sets the *Delivery* flag to *all* and includes an intentional name describing the attributes of the intended destinations for the message in the message header, and then sends the message to an INR. The INR will perform the multicast forwarding of the message using the pseudocode as shown in Figure 3-7. Again, to make the pseudocode more readable, details of the handling to null result and the exact inference on source name are not shown.

First, the INR learns about the source name of the message using *inference*. It then looks up all the end-nodes satisfying the destination name of the message and forwards the message along the overlay spanning tree of the INR network to all end nodes satisfying the destination name. The `alreadySentTo` set is used to aggregate paths that are common to some of the end-nodes, such that the message is sent only once across that common path (instead of multiple times equal to the number of end-nodes sharing that common path).

Forwarding messages along the overlay tree of the INR network with the aforementioned overlay path aggregation enables aggregations of message forwarding on the physical links to some degree as well. However, the actual gain is in the "aggregation" of network latency, since INS optimizes its overlay topology based on a network performance metric, in particular the network latency, and not on how aligned it is to the underlying network topology (details of the overlay topology are discussed in Chapter 4). What this means is that local replications of a message by the INR closest to the matched end-nodes in most cases incur less delay than re-sending the message again from the source. Similarly, INR replicating a message to neighbor INRs in normal cases incurs less delay than the neighbor INRs requesting the message directly from the source INR again. Intentional multicast enables a data distribution mecha-

```
multicast a packet p that is received from n
{
   perform inference on the source name of p

   let nrset = the set of name-records returned by the look up
      process in looking up name table for destination name of p

   alreadySentTo = {}

   for each record r in nrset
      if (INR-ID in r = my-INR-ID)
         forward p to the application with App-ID in r

      else
         let nexthop = the next hop INR for overlay route toward
            INR-ID in r (found by looking up routing table)

         ; prevent sending it back to where we receive it from
         if (nexthop == n) continue

         ; send to a nexthop only once
         if (nexthop is in alreadySentTo) continue

         add nexthop to set alreadySentTo
         forward p to nexthop
}
```

Figure 3-7: Pseudocode for intentional multicast.

nism where a source pushes its data toward all interested clients and clients advertise themselves as being interested in receiving the data by specifying their intentional names.

Intentional multicast uses an intentional name as the group handle, providing a high degree of flexibility to handle dynamic group communication. By composing different attributes and values as part of the name, a new group is created "on the fly" without having to perform address allocation in advance as is the case for IP multicast [15]. Some example applications, such as retrieving data from all temperature sensors that currently read a sub-zero temperature within a particular region, requires the ability to handle dynamic groups responsively, since the members of the group are directly dependent on the exact information the clients are seeking (i.e., sub-zero temperature in this example) and the current data the services supply (i.e., current temperature degree that each sensor reads). Furthermore, the flexibility in composing a new group enables more than just receiver-initiated group communication.

3.4.3 Optimizations

In general, looking up a name incurs a higher overhead than looking up a fixed-length number or label. One possible optimization to reduce the amount of name-lookup overhead in the forwarding hop-by-hop across overlay network for intentional multicast delivery is to associate a fixed-length label to each destination name of multicast messages. Similar to Multi-Protocol Label Switching (MPLS) [47] in lieu of IP lookups, label switching in INS speeds up a name lookup process on the downstream INRs. Only the first-hop INR that receives a message from an application needs to lookup a name to find the associated label, which will then be inserted back into the message for downstream INRs to immediately use it in determining its next-hop INR. Here, each INR maintains a cache of labels for forwarding (i.e., a routing table for labels). The same name-to-label mappings must be used among all INRs to enable this, implying that some label distribution mechanism must be employed. Using the above forwarding scheme however, no explicit label distribution is necessary. When the first-hop INR inserts a label back into the message during message forwarding for use by downstream INRs, the label is indirectly being distributed. That is, when the first-hop INR receives a message from an application, it checks whether there exists a label for the destination name already in the cache. If it does not exist, it creates a new label for the name, appends the label to its cache of labels, and inserts the label back into the message. Downstream INRs that forward the message will then use the same name-to-label mapping as the one inserted in the message if no such label exists yet in their caches of labels.

This optimization reduces the overhead of using a name as a group communication handle. By creating an "internal address" for the group on which the routing and forwarding of messages within INR network are based on, INRs remove much of the overhead of looking up name, while at the same time allowing applications to still an expressive and flexible handle for communication.

In this chapter, we have described the algorithms and protocols used by INS name resolvers to disseminate name and routing information. We explored the benefits of soft-state names and soft-state overlay routing information. In addition, we introduced an optimization to gain the benefits of soft-state with much less network bandwidth consumption, i.e., by splitting the state maintenance of names into soft-state (between applications and their closest INRs) and hard-state with incremental updates (between INRs). The soft-state overlay routing information ensures that network partition and INR-node failures are detected in a timely fashion. Finally, we described the INS message forwarding schemes, which include intentional anycast and intentional multicast. We presented an optimization to reduce the overhead of looking up a name during message forwarding by associating a fixed-length label to the destination name of multicast messages and performing forwarding decision based on the label.

Chapter 4

Self-Configuring Resolver Network

INS uses a decentralized network of cooperating name resolvers to provide a system-wide resource discovery service. INRs form a resolver network to exchange name and routing information, and also to forward messages during intentional multicast delivery. The resolver network is formed as an overlay network over IP unicast, leaving the underlying network-layer addressing and IP routing architecture unmodified. In our current design, INRs self-organize into a spanning tree topology, providing loop-free connectivity.

Since updates and multicast messages will flow along this spanning tree, its "links" should be aligned to the paths having good network performance, such as low network latency paths. In other words, the peering (neighbor relationships) between INRs should be aligned to optimize network performance. Existing distributed algorithms that construct a spanning tree are not generally well suited to our operating environment because of the degree of dynamism due to node mobility, nodes joining and leaving the system, and variations and fluctuations in network performance and reliability in this environment. INS uses a novel distributed self-configuring algorithm to keep the spanning tree of the resolver network adaptive to optimize network performance in a scalable fashion, and self-improving such that it eventually evolves into a minimum spanning tree in the absence of mobility. The main feature of the algorithm is its capability to evolve any tree into an optimal tree in a distributed manner by using *relaxation* operations that incrementally and asynchronously adapt the neighbor relationships between INRs.

Although the self-configuring algorithm presented here is for constructing and maintaining an adaptive spanning tree of the INS resolver network, the algorithm is general enough to be applicable to other systems that desire an adaptive and self-improving network topology that evolves toward optimality.

In Section 4.1 we give an overview of the algorithm, including some assumptions used by the algorithm and an example showing how the algorithm evolves any spanning tree into a minimum spanning tree using *relaxation* operations. Section 4.2 describes the detailed algorithm for relaxation operations, including several conditions to regulate concurrent relaxation operations and some analysis. Section 4.3 discusses the messages used in relaxation and some optimization to speed up INRs in adapting routing information due to topology changes. When a node or link failure occurs, INRs heal the spanning tree by using a healing mechanism presented in Section 4.4.

4.1 Spanning Tree Algorithm: Overview

To propagate updates and forward data to services and clients, the INRs must be organized as a connected network. Our self-configuring algorithm constructs a spanning tree in a fully distributed fashion based on metrics that reflect the INR-to-INR round-trip latency, and continually modifies the tree to optimize the latency metric. The spanning-tree algorithm includes a mechanism by which INRs can detect which parts of the tree are sub-optimal and hence can be improved. This mechanism of improving the tree by detecting and replacing a sub-optimal peering between INRs with a better one is called *relaxation*. Unlike our relaxation-based approach, other distributed algorithms that construct a MST do not generally offer the capability to relax neighbor relationships once the tree is completely constructed, which means that after the initial construction of the MST, changes of link metrics will require a significant amount of recomputation to obtain a new MST.

A list of active INRs is maintained by a well-known entity in the system, called the *Domain Space Resolver* (DSR). As discussed in Section 2.2, DSR supports a query that returns a list of currently active INRs in the system. Before discussing the algorithm, we would like to mention our assumptions about the underlying graph.

Assumptions. The algorithm uses the following assumptions about the overlay graph across which the spanning tree of resolver network is constructed:

- The underlying graph is a fully connected graph (in normal cases, when no network partition occurs), since the INS resolver network is formed as an *overlay* network.

- Each INR has a unique ID, the *INR-ID* (discussed in Section 2.2). In the subsequent discussion we use the term *node* to mean an INR node.

- A link in the overlay graph is a network *path* between overlay nodes, which may consist of multiple physical-layer links. We use the term *tree links* for links that are part of the spanning tree, and *non-tree links* for other links in the graph that are not part of the spanning tree.

- Tree links are reliable. However, since routing information may be in transient state when overlay topology changes, packets sent across multi-hop links may not always get delivered to destinations. Tree links are assumed to be bidirectional and FIFO. For non-FIFO links, we can easily add a sequence number to achieve FIFO. (The implementation uses a long-lived TCP connection between nodes for the *tree* links.) Non-tree links are not reliable.

- Each link has a metric (cost), which reflects the INR-to-INR round-trip latency. Since network latency performance may fluctuate, link metrics in this environment are dynamic.

The algorithm works in (asynchronous) stages. First, a spanning tree is constructed amongst the INRs based on the local minimization decision performed by each INR when it joins the system. Then, the relaxation protocol of the algorithm evolves the constructed spanning tree into an optimal one.

Initial construction of a spanning tree. When a new INR joins the system, it contacts the DSR to obtain a list of currently active INRs. The new INR then conducts a set of *INR-pings*[1] to the currently active INRs and picks the one to which it has the smallest round-trip latency and establish a neighbor relationship (or *peer*) with it. 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 INRs obtain the list from the DSR, race conditions are avoided and one can impose a linear order amongst the active INRs based on the order in which they contacted the DSR. Each INR on the 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, creating n-1 links in an n-node network. Any connected graph with n nodes and n-1 links must be a tree, which implies that this initial construction leads to a spanning tree.

Evolving toward optimality using *relaxations*. Of course, despite each INR making a local minimization decision from the INR-pings, the resulting spanning tree will not in general be the minimum one. Hence, a new INR, after peering with an existing one, enters a *relaxation* phase, where it participates with other INRs in the system to adapt the neighbor relationships to evolve the spanning tree to a minimum one.

The idea of the relaxation is to replace a high-cost link with a lower cost-link without disconnecting the graph. An INR can probe for the existence of a better spanning tree by sending a `PROBE` message along the current tree path to some destination. The `PROBE` message discovers the highest-cost link of the path it has traversed. The other end of the path, which receives this `PROBE` message, knows about the *bottleneck* (i.e., the highest-cost link) of the path and can decide whether it can improve the tree by comparing the cost of the bottleneck with the cost of a direct link between the source of the `PROBE` message and itself (see Figure 4-1). If the cost of direct link is better

[1]The experiments conducted by the INRs to obtain the INR-to-INR latency metric are called *INR-pings*, which consist of sending a small message between INRs and measuring the time it takes to process this message and get a response.

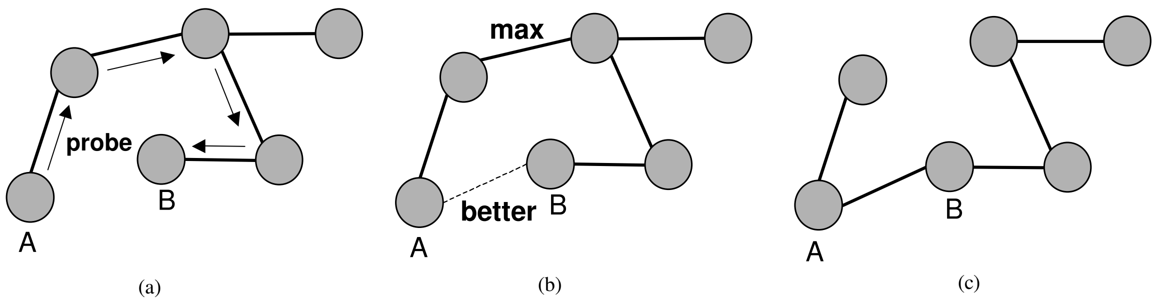

Figure 4-1: Illustration of a relaxation operation. (a) Node A sends a `PROBE` message to B along the current tree path to discover the highest-cost link of the path. (b) Node B receives this message, compares the highest cost with the cost of direct link between A and B, and finds that direct link between A and B has lower cost. (c) Node B replaces that highest-cost link with a direct link between A and B. This results in a new spanning tree with better cost overall.

than the bottleneck cost, it replaces the bottleneck link with a direct link between the source of the `PROBE` message and itself. This replacement is called a *relaxation* operation.

Since the graph is fully connected (in normal cases, when no network partition occurs), a node can compare any of its non-tree links for possible relaxations, and hence a node can select any node in the graph as its destination for a `PROBE` message. Potentially, multiple nodes may select different destinations and send their `PROBE` messages at the same time; thus, multiple concurrent relaxations may occur. Section 4.2 describes the relaxation operation in more detail, describing how concurrent relaxations may violate tree properties and how to overcome this. Relaxation gracefully handles message losses as well. Message losses will not harm the connectivity of the graph; they may slow down the evolution toward optimality, but no degradation of the tree will occur due to those losses. This is in contrast to other distributed algorithms (e.g., the one in [22]) that construct a minimum spanning tree by iteratively merging smaller components into a bigger one, in which case message losses may stall the construction of the tree.

Example. Figure 4-2 shows an example of five nodes joining the graph one by one. Node 1 is the first one joining the system. Then node 2 comes in, and since node 1 is the only available, it peers with node 1. Node 3 comes in and peers with node

2 since the link metric between 2 and 3 is smaller than the link metric between 1 and 3. At this point, the local minimizations performed by each node still result in an optimum tree. Next, node 4 comes in and peers with a node whose link metric to it is the lowest, i.e., node 3. The resulting tree happens to be an optimum tree already — thus any relaxation operations cannot improve the tree. Finally, node 5 comes in and peers with node 2 because of the same local minimization as before. At this point, however, the resulting spanning tree is not an optimum one because node 5 "bridges" several previously higher-cost links; in particular, replacing both links (1, 2) and (2, 3) with links (1,5) and (4,5) will result in a better tree. Hence, the relaxation operations performed by the nodes can improve the tree. At some point either node 4 or node 5 sends a `PROBE` message. The figure shows the case where node 4 sends a `PROBE` message to node 5. Node 5 receiving this probe knows that link (2,3) has the maximum cost among links on the tree path between 4 and 5. Node 5 then compares this maximum cost (i.e., 18) with the cost of the link between 4 and itself (i.e., 8) and finds the latter is better. Hence, it replaces link (2,3) with link (4,5). Similarly, at some point there will be a probe from 5 that will reach node 1 (or a probe from 1 to 5). Either one will detect link (1,2) as the maximum-cost link of the tree path between 1 and 5 and find direct link (1,5) is better. Hence, link (1,2) is replaced with link (1,5). The resulting tree is a minimum spanning tree and hence no other relaxation can improve the tree. The order of the relaxation operations in this scenario is only an example; of course, replacement of link (1,2) with link (1,5) may also occur before the other replacement.

4.2 The *Relaxation* Protocol

After peering with an existing node in the graph, a new node enters a relaxation phase, in which from time to time it probes for the existence of a better spanning tree. A node may select a destination for its `PROBE` in some controllable way, or it may flood the message to compare all of its non-tree links for relaxations.

In the description that follows, we assume that a node floods its `PROBE` message

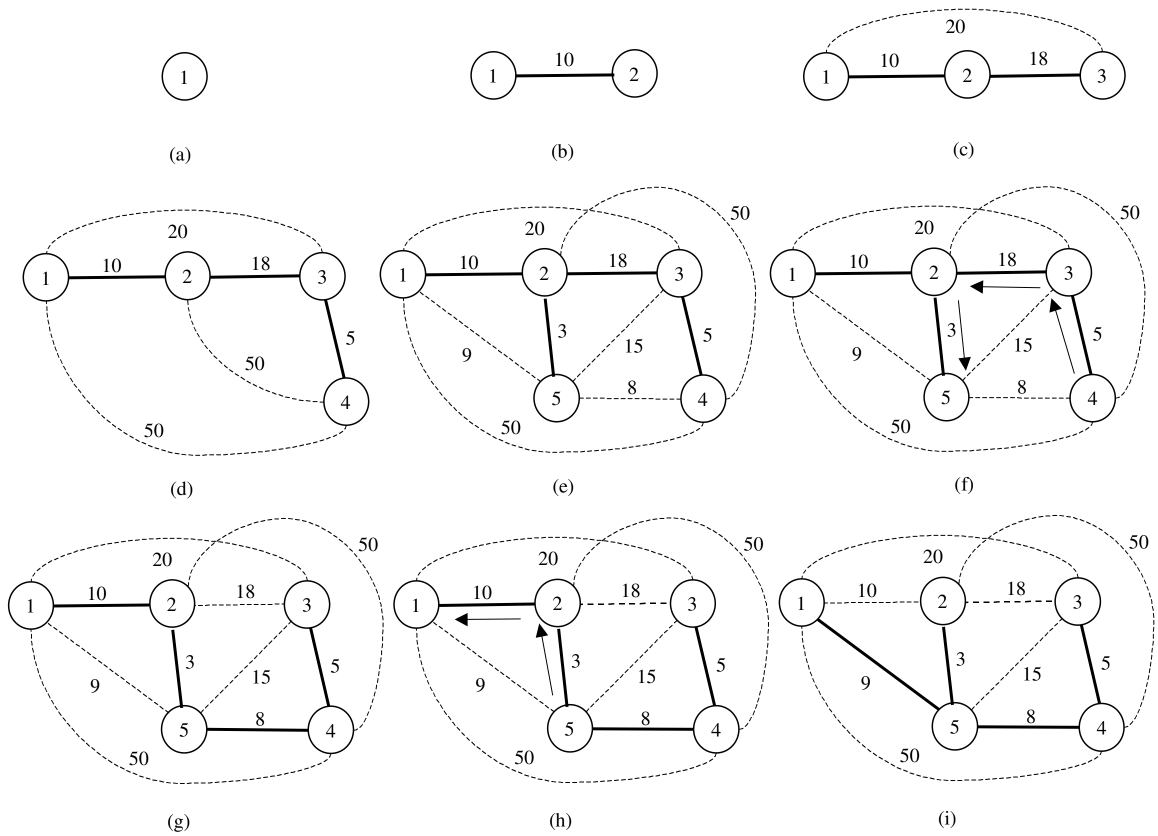

Figure 4-2: An example of a spanning tree construction and its evolution toward a minimum spanning tree. The links in bold are *tree* links, while dotted-lines represent *non-tree* links. Five nodes join the system one by one (a, b, c, d, e) and each performs local minimization in selecting another node to peer with. All those local minimizations result in an optimal tree until the fifth node peering, in which case a better tree may still be attained. The relaxation operations performed by the nodes in the system evolve the tree (in (e)) into a minimum spanning tree (in (i)).

for faster evolution toward optimality. In the relaxation phase, every node sends and listens to periodic `PROBE` messages. This probe message contains information about the highest-cost link of the path that it has traveled through. The purpose of `PROBE` messages is to determine the largest-cost link along the path between two nodes. By comparing this cost with the cost of a direct link, the nodes determine if a more optimal tree can be obtained.

A node i that receives a probe message originated by a node s can determine the highest cost link of the path between s and i (see Figure 4-3). Suppose the link between nodes a and b is the highest-cost link of the path. Upon receiving a probe,

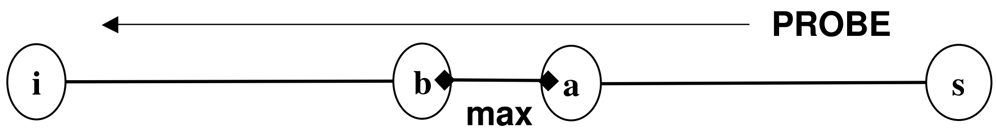

Figure 4-3: A probe message flowing from node s to node i through bottleneck (a,b). A line with diamonds at the end-points indicates a *link*, while a line without diamonds indicates a *path*.

node i calculates whether it can improve the optimality of the tree by replacing link (a, b) with the direct link between i and s. If doing so improves the tree, it starts a relaxation *experiment*.

Node i starts a relaxation experiment by sending a `REPLACE-RQ` message to node b along current overlay tree path to remove link (a, b) from the spanning tree. In this case, we say that node i *initiates* an experiment and the experiment *starts* at the time the `REPLACE-RQ` message is sent out. The experiment *ends* when the link (a, b) is removed (for a successful experiment), or when node i receives a failure notification (for a failed experiment). An *experiment path* is defined to be the path from node i to node s using the edges of the current spanning tree.

Two experiments are defined to be *concurrent* if one experiment starts before the other ends. Two experiment paths are said to be *overlapping* if they share a common fragment. There are a number of conditions to be followed to prevent concurrent experiments from interacting in a way detrimental to the tree property (e.g., resulting in a disconnected graph or a cycle). Before describing these conditions, let us look at several examples in which two experiments interact in a way that violates the tree properties at the end.

4.2.1 Examples

Example 1. This example shows a case in which two experiments interact in a way that produces a disconnected graph with a cycle in one component. Figure 4-4 shows node s sending a `PROBE` message to node i flowing across the maximum-cost link (a, b). Node i receives the probe and determines that replacing link (a, b) with its link (i, s) will improve the tree. However, before node i sends out its `REPLACE-RQ`

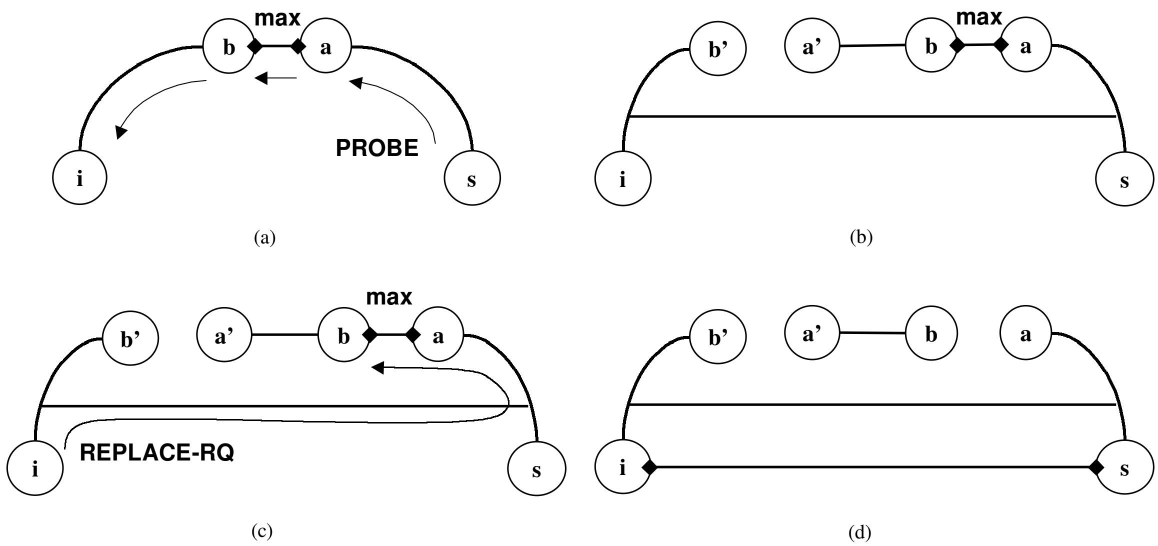

Figure 4-4: An example of two relaxations interacting in such a way that the result is not a tree. A line with diamonds at the end-points indicates a *link*, while a line without diamonds indicates a *path*. (a) Node s sends its `PROBE` to node i. (b) Before i sends its `REPLACE-RQ` to node b, the path changes as shown due to some other relaxation initiated by a different node. (c) Node i sends a `REPLACE-RQ` to b through the new path. (d) Link (a, b) is replaced with link (i, s), but this new graph is not a tree.

message to node b to break the link (a, b), a second experiment changes the tree path between nodes i and s such that the path from i to s is not through nodes a and b anymore (part (b) of the Figure). Node i, however, is not aware of this path change, and hence still sends its `REPLACE-RQ` message to node b. The result is that link (a, b) is replaced by link (i, s) (part (d) of the Figure), producing a disconnected graph with a cycle in the component containing i and s. These two experiments are not necessarily concurrent (by definition of an *experiment*), because an experiment is defined to start when node i sends its `REPLACE-RQ` message and in this example the other experiment that changes the path to the one shown in part (b) of the Figure may finish before the `REPLACE-RQ` from i is sent out.

The next two examples show cases in which two experiments if run *concurrently* may produce a disconnected graph and/or a cycle.

Example 2. This example (Figure 4-5) shows a case in which two concurrent experiments happen to disconnect the same maximum-cost link, but each adds its own

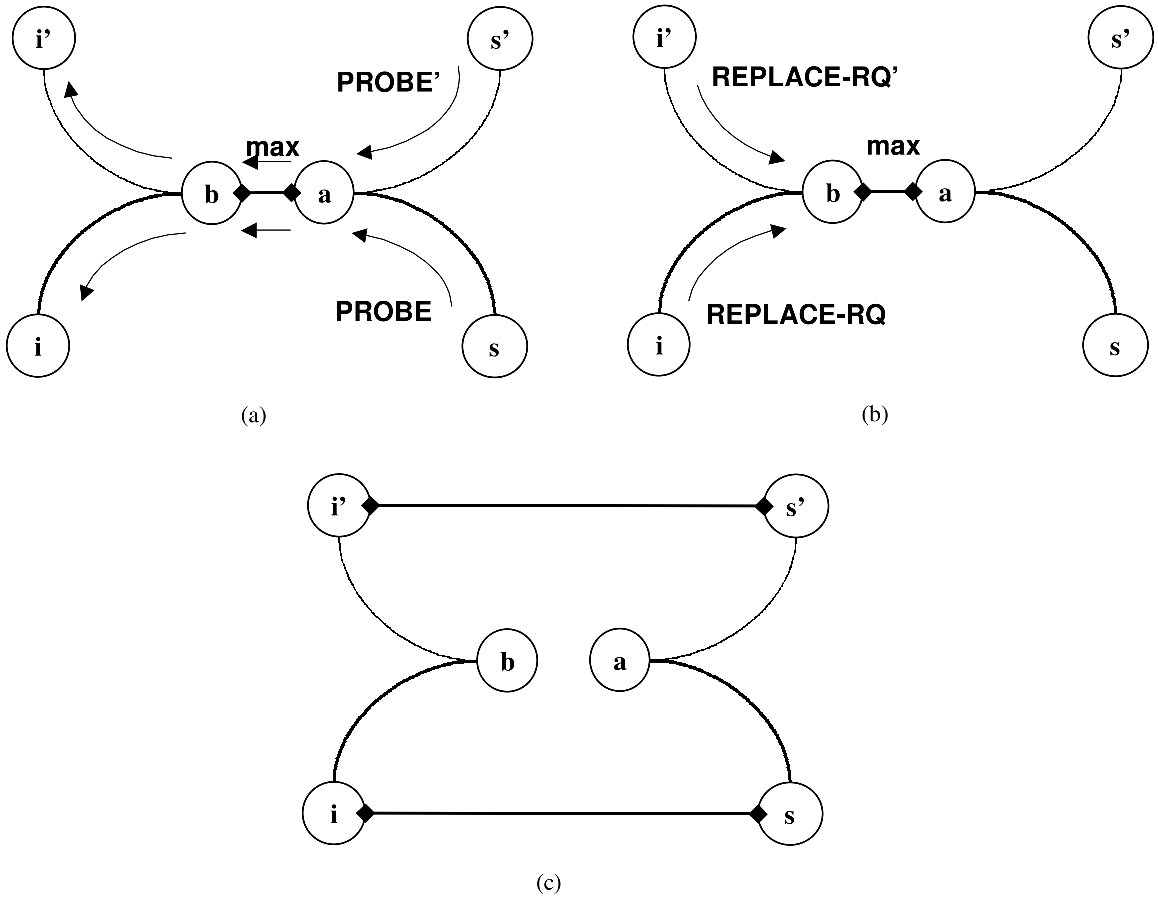

Figure 4-5: An example of two *concurrent* relaxations interacting in such a way that results in a cycle. (a) Node i receives a `PROBE` from s and node i' receives a `PROBE` from s'. (b) Both i and i' send their `REPLACE-RQ` to node b. (c) The result is that link (a, b) is removed from the tree but two new links, (i, s) and (i', s'), are added, producing a cycle.

new link to tree. Since only one link is removed from the tree and two new ones are added, the result contains a cycle involving both the new links.

Example 3. This example (Figure 4-6) shows a case in which two *concurrent* experiments have *overlapping* experiment paths. But unlike the case in Example 2, each experiment identifies a *different* maximum-cost link. This happens because link metrics are dynamic (reflecting the INR-to-INR round-trip latency in INS); link (a, b) may at one time have a higher cost than the cost of link (a', b'), but the situation may be reversed at some later point in time . Hence, node i replaces link (a, b) with (i, s), and node i' replaces link (a', b') with (i', s'), producing a disconnected graph and a

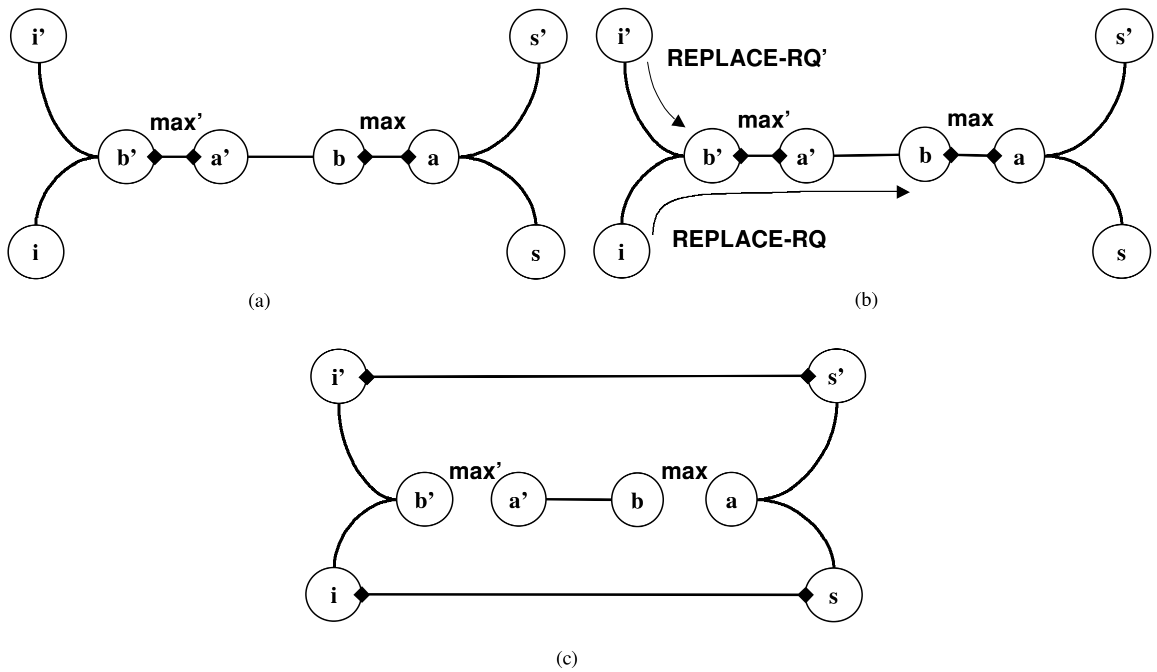

Figure 4-6: Another example of two *concurrent* relaxations interacting in such a way that the result is not a tree. (a) Node i receives a `PROBE` from s and identify link (a, b) as the maximum-cost link to be replaced. Node i' on the other hand receives a `PROBE` from s' and identify link (a', b') as the maximum-cost link to be replaced. (b) Node i sends a `REPLACE-RQ` to node b, while node i' sends a `REPLACE-RQ` to node b'. (c) The result is that link (a, b) is replaced by (i, s) and link (a', b') is replaced by (i', s') producing a disconnected graph and a cycle in the component containing both i and i'.

cycle in the component containing both i and i'.

In general, two concurrent experiments will violate the tree property if they are replacing links that both lie on the common fragment of their overlapping experiment paths. However, if only one of the replaced links is from the common fragment and the other is from a different fragment, or if none of the replaced links is from the common fragment, then the result will be another spanning tree (see Figure 4-7). For more than two concurrent experiments, the same property applies. That is, if more than two concurrent experiments replace links that all happen to fall in the common fragment of their overlapping experiment paths, they will not produce a spanning tree.

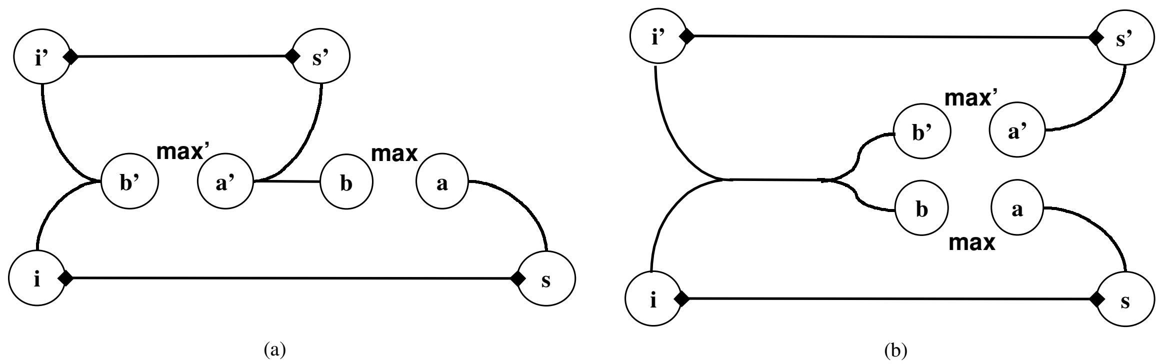

Figure 4-7: Examples of two *concurrent* relaxations that do not violate any tree property.

4.2.2 Conditions

There are four conditions to be met for an experiment to successfully replace a link without violating the tree properties:

C1. Since the path along which a PROBE traverses may change by the time node i (which receives the PROBE) starts a relaxation experiment (shown by Example 1 above), node i must verify that the overlay tree path from i to s is still *valid* before the actual link replacement occurs. A path from i to s is valid if the link (a, b) replaced in an experiment is an intermediate link on the path from i to s. In other words, if the path is valid, disconnecting link (a, b) will produce two disconnected components, one containing i and a, and the other containing b and s.

One way to ensure a valid path for an experiment is to enforce the REPLACE-RQ message to travel from i to s through exactly the same path as the path through which the PROBE that induced the experiment traveled. A possible mechanism for this is to use "source routing"; that is, each intermediate node that forwards the PROBE message appends its ID to the message. At the end, the PROBE contains a list of all intermediate nodes in the order they forwarded it, which can be used by node i to source-route the REPLACE-RQ message back to node s through link (a, b). An intermediate node between i and s forwards the REPLACE-RQ message to its next hop based on the source-routing information. If the source-route contains a link that is no longer a valid tree link (i.e., it has already been removed from the tree by another experiment), the immediate node of that link fails the experiment by sending a `REPLACE-FAIL` to i.

Using source routing to verify whether a path is *valid* for an experiment works well, but we can actually relax the requirement further, such that the `REPLACE-RQ` does not have to travel through the exact same path as the `PROBE` message. The path is valid as long as the `REPLACE-RQ` message travels through a path that meets the following two requirements:

(I) The tree path between i and b does not pass through node a (i.e., node a is not an intermediate node between i and b).

(II) The tree path between a and s does not pass through node b (i.e., node b is not an intermediate node between a and s).

We can understand these two requirements because a path for the experiment is valid only if disconnecting link (a, b) produces two disconnected components — one containing i and a, and the other containing b and s (see Figure 4-3). Node i can verify the validity of the path by sending a *path-verification* message along the experiment path from i to s with each intermediate node participating in enforcing the above two requirements. This *path-verification* message can be piggybacked on the `REPLACE-RQ` message. Section 4.3 will further discuss the messages used for path verification of an experiment.

C2. Because link metrics of the graph are dynamic, upon receiving a `REPLACE-RQ` to replace link (a, b), node b has to examine first whether the link (a, b) indeed has a cost higher than the cost of its replacement link.

The rest of the conditions are to regulate concurrent experiments. There are several ways to prevent concurrent experiments from violating a tree property. One option is of course to completely disallow concurrent experiments so that the cases in examples 2 and 3 above will not occur. A better option is to allow concurrent experiments to continue as long as their experiment paths do not overlap, i.e., each tree link can be part of at most one experiment path. If two or more experiment paths overlap, then only the first one will continue and the other stops.

This option does work, but as it turns out, we can actually relax the requirement further to allow concurrent experiments with overlapping experiment paths to continue as long as they follow the following two requirements, **C3** and **C4**. We know that two concurrent experiments may violate a tree property only if they both replace links on the common fragment of their overlapping experiment paths. Condition **C3** below is to regulate the case in which they all try to replace a single maximum-cost link. **C4** is to regulate more general cases in which they all try to replace different links but all those links happen to fall in the common fragment of their overlapping experiment paths.

C3. Concurrent experiments may all try to replace a single maximum-cost link (a, b). Without any conditions to regulate them, they may end up producing a cycle (as shown in Example 2 above). To avoid such race conditions, the immediate nodes of the link to be replaced must filter and arbitrarily select one of the experiments to be the *winner* for the link replacement. In particular, upon receiving a `REPLACE-RQ` message, node b needs to check whether node a is also in the middle of some other experiment that is also trying to remove link (a, b). If both a and b are involved in concurrent experiments from two different experiment initiators, and both try to remove the same link (a, b) from the tree, then both a and b have to agree on a single experiment that will win the replacement. The protocol arbitrarily selects the one that has the best replacement among those concurrent experiments for the link.

C4. Every experiment needs to perform a path verification as indicated by **C1**. However, this process of verification is not instantaneous; it takes the propagation delay of the *path-verification* message, therefore, concurrent experiments with overlapping experiment paths may influence the validity of one verification. That is, one fragment of the path may have been verified to be valid, but before the whole path is verified, the verified fragment suddenly changes because of some other concurrent experiments replacing some links on that fragment. Here, we have the option of either starting the verification process over if the verified path has changed, or using another mechanism to regulate these types of concurrent experiments. We choose the later approach in our algorithm.

In our scheme, each link keeps track of the experiments going across it. If a link sees the *path-verification* message of an experiment, it adds this experiment to the set of experiments having *access* to the link (this verification message indicates that the link is part of the experiment path). Two or more experiments may simultaneously access a single link. Removing a link, however, requires that no other experiments are currently accessing the link, i.e., *exclusive* access is necessary for link removal. A link indicates that it has been exclusively accessed by an experiment by setting its *exclusive-access* flag.

Using the above policy, selected concurrent experiments are allowed to proceed concurrently, but others are not. An experiment that needs to remove some link that has been accessed by some other experiments will fail. Similarly, an experiment that needs to access a link that has been exclusively accessed by another experiment will find that its path-verification fails.

Another complexity this policy introduces occurs when two concurrent experiments with overlapping experiment paths send their path verification messages from two opposite directions. If both experiments try to replace two different high-cost links that both lie on the common fragment of their overlapping experiment path, both experiments may fail. However, the next probe message will determine either one of the links to be the highest-cost link of the common fragment (thus, both experiments try to remove that single link and either experiment will succeed based on $C3$), unless of course the metrics change faster than the time needed to propagate the path-verification message across the experiment path. In the latter case, there is no definite better spanning tree within that short interval for the tree to converge toward. This additional complexity of messages coming from two opposite directions happens only when metrics are dynamic (which is the case in our environment), and will not happen with static metrics.

Summary. In summary, after a node i receives a probe message generated by node s and detects that it can improve the tree by replacing the highest-cost link (a, b) of the path from s to i with its immediate link (i, s), i starts a relaxation *experiment* to replace it. First, node i verifies the validity of the current tree path from i to s (condition **C1**) by sending a *path-verification* message along the path. While forwarding the message to b, each intermediate node checks whether it is the "node a" of the experiment (i.e., the downstream end-point of the link to be replaced) following the first condition of **C1**. In addition, it records which of its immediate tree links that is part of the experiment-path (**C4**). Upon receiving the message, node b performs dynamic link check (**C2**) and examines whether node i can gain exclusive access to link (a, b) (**C4**) by *negotiating* with node a (**C3**). If node i succeeds, the path verification continues for the path from a to s. As before, each intermediate node checks whether it is the "node b" of the experiment following the second condition of **C1** and also records which link is part of the experiment path (**C4**). If the path verification is successful all the way through s, a notification is sent back to node i, and i can then replace link (a, b) with link (i, s). This notification also informs all the intermediate nodes that the path verification is complete, so that they can remove the experiment from the sets of experiment currently accessing their links.

4.2.3 Analysis

In this section we informally describe our analysis of the algorithm, in particular, the number of relaxation operations required to evolve a spanning tree, in its worst case, into a minimum spanning tree (MST). Since the link metrics are assumed to be dynamic, there will be no definite MST if the metrics are always changing before the tree converges to a MST. Hence, the following analysis assumes that there is some *steady-state* period, in which metrics do not change, that is long enough for the tree to converge. In other words, we assume the link metrics to be static for the duration of this steady-state.

We show that it takes at most E relaxation operations to turn any spanning tree into a MST, where E is the number of overlay links in the graph. Since the graph is normally fully connected, the number of edges is in the $\mathrm{O}(n^2)$, where n is the number of nodes in the graph. What this means is that it will take at most $\mathrm{O}(n^2)$ relaxation operations to turn any tree into a MST after the *last* link-metric changes.

To show that it takes at most E relaxation operations to turn any spanning tree into a MST, we need to show that in the steady-state after a link is removed (replaced) from the tree, it will never be added to the tree again. We show this by contradiction. Since the metrics can be treated as static during the steady-state, the following will hold:

The only reason a link (a, b) is removed from the tree is because a node i receives a probe from some other node s that identifies link (a, b) as the highest-cost link of the path between s and i, and i has a better link to replace (a, b). After link (a, b) is replaced by link (i, s), there exists a new path that connects a and b through link (i, s). Let's call this path $p_{(a,b)}$ (see Figure 4-8 part (a)). Since link (a, b) is removed, all the links on the path $p_{(a,b)}$ must have costs less than or equal to the cost of the (now non-tree) link (a, b) (by construction, the relaxation operation always removes the highest-cost link).

Suppose link (a, b) gets added again at some later time. We have two cases depending on whether $p_{(a,b)}$ has been transformed into some new path or not by the time link (a, b) gets added again.

- Case 1: No changes on the path $p_{(a,b)}$ by the time link (a, b) is added. Since link (a, b) gets added again, there must be a link e' on $p_{(a,b)}$ that has a cost higher than the cost of link (a, b) causing link (a, b) to be favored over e'. However, this is a contradiction since link (a, b) was removed because all links on $p_{(a,b)}$, *including* e', have costs less than or equal to the cost of link (a, b).

- Case 2: By the time link (a, b) gets added again, another relaxation has replaced some link (a', b') on the path $p_{(a,b)}$ with another link (i', s') causing $p_{(a,b)}$ to be transformed into some new path $p_{new(a,b)}$, which connects a and b through (i', s'). Figure 4-8 part (b) shows one possible path for $p_{new(a,b)}$. It can be shown that all links on $p_{new(a,b)}$ also have costs less than the cost of the (non-tree) link

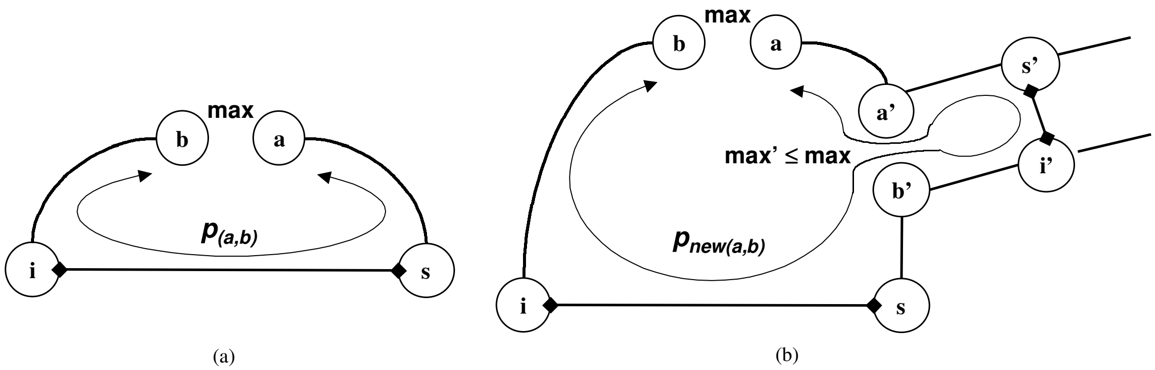

Figure 4-8: An illustration used in the analysis of spanning tree algorithm. (a) After node i replaces link (a, b) with its link (i, s), a new path $p_{(a,b)}$ connects a and b through link (i, s). (b) Path $p_{(a,b)}$ is transformed into $p_{new(a,b)}$ by another relaxation that replaces link (a', b') with a new link (i', s').

(a, b): the second relaxation replaces (a', b') only if (a', b') is the highest-cost link of the second relaxation's experiment path. Equivalently, all links on the second relaxation's experiment path must have costs less than or equal to the cost of (a', b'), which is less than or equal to the cost of (a, b) (since (a', b') is on $p_{(a,b)}$). Hence, all links on the new path $p_{new(a,b)}$ (consisting of some/all links of $p_{(a,b)}$, and link (i', s') and perhaps some/all links of the second relaxation's experiment path) will also have costs less than or equal to the cost of link (a, b).

Since link (a, b) gets added again, there must be a link e' on $p_{new(a,b)}$ that has a cost higher than the cost of link (a, b), causing link (a, b) to be favored over e'. However, this is a contradiction since all links on the $p_{new(a,b)}$, including e' have costs less than or equal to the cost of link (a, b).

Since once a link is removed from the tree, it will never be added again, the tree monotonically converges to a better tree. Since there are E links in the graph, we can perform at most E relaxation operations, implying that a minimum spanning tree will be attained using at most E relaxation operations.

4.3 Messaging in the Relaxation Protocol

Probing. Initially, after peering with an existing node in the initial spanning tree construction, a node is in a *probing* mode. In this mode, a node sends and listens to periodic `PROBE` messages. It sends a `PROBE` message to all of its neighbors, which in turn forward the message to all their neighbors except to the neighbor from which they receive the message. While forwarding a `PROBE` originated by a node s, an intermediate node includes information about the highest-cost link of the path that the `PROBE` has traveled so far.

We use the notation `PROBE`(s, (a, b):C_{ab})
for a `PROBE` originated by node s, with the highest cost link seen so far on the path being (a, b) with cost metric C_{ab}. Initially, node s itself sends `PROBE`(s, ():0) to its neighbors.

The actions of a node i that receives a `PROBE`(s, (a, b):C_{ab}) from its neighbor j are as follows:

(1) First, i determines the highest cost link (a', b') of the path from s to i by comparing whether the cost of the tree link between i and j is greater than the highest cost information contained in the `PROBE` message. The new highest-cost link of the path is (a, b) if $C_{ab} > C_{ij}$ and (j, i) otherwise. Let's call this new highest-cost link (a', b').

(2) Then, i determine whether it can improve the optimality of the spanning tree, by comparing $C_{a'b'}$ with C_{is} (i.e., the cost of the non-tree link between i and s). If the tree can be improved (i.e., $C_{is} < C_{a'b'}$), i performs step (4) below (skipping step (3)). Otherwise, it performs step (3) only.

(3) Node i forwards the `PROBE` message with the new highest cost link information as calculated before, i.e., `PROBE`(s, (a', b'):$C_{a'b'}$). After forwarding, node i listens for other `PROBE` messages (and does not perform any of the steps below).

(4) However, if node i knows that it can improve the tree, it starts an experiment to replace the highest-cost link of the path (a', b') with a new link (i, s), and it

sets its mode from *probing* to *experiment*. The communication messages for an experiment are described in the subsequent paragraphs and are shown in Figure 4-9.

Relaxation experiment. When a node i starts an experiment to replace a high cost link (a, b) with a lower cost link (i, s), it sends out a `REPLACE-RQ`((a, b):C_{ab}, (i, s):C_{is}) message to node b along the current overlay tree. This message is also used for path verification; intermediate nodes that forward the message participate in verifying that the experiment path of the experiment is valid, conforming to the condition **C1**. To verify that the path between i and b does not go through a, every intermediate node between i and b, before forwarding the `REPLACE-RQ` message toward b, checks whether the "node a" of the experiment is itself. If it is, then the first condition of C1 is violated and hence the intermediate node fails the experiment by sending a `REPLACE-FAIL`((a, b):C_{ab}, (i, s):C_{is}) back to node i along the tree. Otherwise, it forwards the message to the next-hop node toward b.

In addition to performing path verification for the experiment, every intermediate node also keeps track of which experiments are currently accessing its tree link(s). This step is to follow requirement C4. That is, if an intermediate node receives a `REPLACE-RQ`((a, b):C_{ab}, (i, s):C_{is}) and is supposed to forward the message across a link l (toward b), the node first has to check whether another experiment is currently having an exclusive access to link l. If another experiment does, it fails the experiment initiated by i. Otherwise, it adds the experiment initiated by i to the set of experiments currently having a shared access to link l. All intermediate nodes perform the same forwarding behavior as above until the `REPLACE-RQ` message reaches node b.

A `REPLACE-FAIL` message, which is sent when an experiment fails, will remove the experiment from the set of experiments having shared access to the link on the intermediate nodes. Node b, upon receiving this message, checks whether the cost of the its link (a, b) to be replaced is indeed worse than the cost of the replacing link (i, s), since costs may change dynamically (**C2**). If condition **C2** is met, node

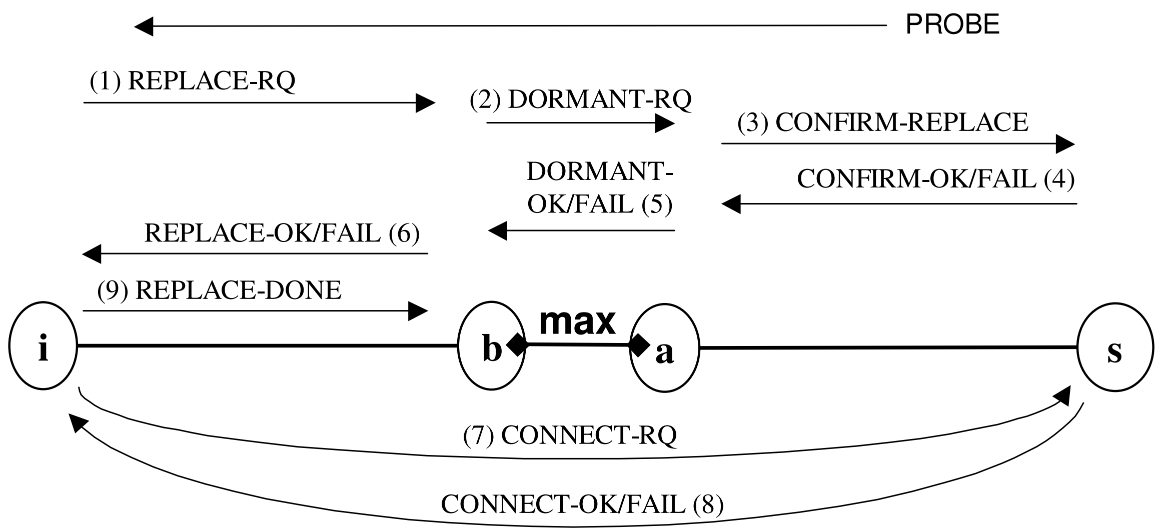

Figure 4-9: Messages for relaxation protocol. The order of messages is noted; however, the first intermediate node that detects the experiment path to be *invalid* intercepts the *handshake* and notifies node i of the failure immediate using the appropriate *fail* message.

b communicates with node a to avoid the concurrency problem where both a and b receive multiple requests from different experiments to break link (a, b) at the same time (**C3**). Node b sets the status of the link (a, b) to *in-progress* and sends out a `DORMANT-RQ` to a. Setting the link status to *in-progress* is equivalent to acquiring exclusive access to the link. If multiple requests received, a and b arbitrarily select the experiment that has the most optimal replacement to be the winner (**C3**).

Suppose experiment initiated by i is the winner. Node a then sets the status of link (a, b) from its side to *in-progress* and continues the path verification by sending a `CONFIRM-REPLACE` message to s along the current tree. As before, in conforming to condition **C1**, all intermediate nodes that forward this message have to participate in verifying that the path between a and s does not go through b. Hence, every intermediate node between a and s, before forwarding the `CONFIRM-REPLACE` message toward b, checks whether the "node b" of the experiment is itself. If it is, then the C1 is violated and it fails the experiment by sending a `CONFIRM-FAIL`((a, b):C_{ab}, (i, s):C_{is}) back to node a along the tree, which in turn informs b, causing it to send a `REPLACE-FAIL` to i. Otherwise, each intermediate node forwards the message to the next-hop node toward s.

Similar to a `REPLACE-FAIL` message, `CONFIRM-FAIL` will remove the experiment from the set of experiments having a shared access to the link on the intermediate nodes.

Upon receiving a `CONFIRM-REPLACE(`(a,b)`:`C_{ab}`,` (i,s)`:`C_{is}`)`, node s replies with a `CONFIRM-OK(`(a,b)`:`C_{ab}`,` (i,s)`:`C_{is}`)` sent to node a across the overlay network. This `CONFIRM-OK` will flow through the same path as `CONFIRM-REPLACE` has traveled since the links on this path have been accessed by the experiment and no disconnection can happen to those links. The `CONFIRM-OK` message tells the intermediate nodes that they can now remove this experiment initiated by i from the set of experiments accessing the link (experiment by i is not using that link for its path verification anymore).

Upon receiving a `CONFIRM-OK` message, node a sets the status of link (a,b) from *in-progress* to *dormant*, and sends a `DORMANT-OK` to b. Node b also sets the link status from its side to *dormant* and sends a `REPLACE-OK` message to node i along the tree path. Similar to the `CONFIRM-OK` message, `REPLACE-OK` causes the intermediate nodes between b and i to remove the experiment initiated by i from the set of experiments accessing the link.

Once it receives the `REPLACE-OK`, node i performs a handshake to node s to establish a neighbor relationship, by sending a `CONNECT-RQ` to and waiting for a `CONNECT-OK` from s. Once neighboring is established, node i sends a `REPLACE-DONE(`(a,b):C_{ab}, (i,s):C_{is}`)` to node b along current tree path to inform b that it can safely remove the link (a,b) now and remove any state information associated with this experiment.

Why do we need a *dormant* state with `REPLACE-DONE` at the end, rather than having node a and b remove link (a,b) right away after receiving the `CONFIRM-OK`? The dormant state is needed to handle retransmissions; suppose that a `REPLACE-OK` message is lost on the way, and link (a,b) has been disconnected while link (i,s) is not yet established due to `REPLACE-OK` not getting to i. Node i at some later time re-transmits a request. Node b will fail the request. At this point, node i will never establish the link (i,s) as a replacement for (a,b) and the resulting graph is disconnected. Hence, nodes b and a need to maintain some state (here, by dormant state) before the new link is successfully established. The above algorithm incorporates timeout with retransmit mechanism for `REPLACE-RQ`, `CONFIRM-REPLACE`, `REPLACE-OK`, and `CONNECT-RQ`.

Our algorithm also incorporates a lifetime for shared/exclusive access, such that access to a link (either shared or exclusive) can not exceed a certain time period. This mechanism is to prevent some failing node from stalling other experiments, thereby improving the robustness of the system.

Routing optimization for relaxation. When the spanning tree changes due to some relaxation operations, the routing information will be in some transient. By using soft-state routing information, nodes are able to obtain the new routes easily just by listening to and propagating periodic route update messages that flow through, and after some period will have all entries of their routing table reflect the new routes. In addition to this waiting for the periodic updates to come by, nodes involved the relaxation experiment can actually perform some optimization in distributing the new route information, since they have enough information about which routes are changed (i.e., routes that are originally through (a, b) are changed to go through the new link (i, s)). One way to enable this optimization is as following:

The end-points of the link to be removed (i.e., node a and b) can deduce from its routing table which nodes will be in a different component if link (a, b) is removed. Node a and b then send the information about these nodes to node s and i respectively. Node s receiving this information updates its routing table such that routes to all these nodes (which will be in different component when link (a, b) is removed) are set with next-hop to node i, and also floods a route update to all its neighbors (except to i) about the new routes to all those nodes. Similarly, node i will perform the same based on the information it receives from node b.

4.4 Failures and Healing Mechanisms

This section discusses several mechanisms used to heal the spanning tree when a node terminates (either on purpose or not) and when a link fails. Planned termination is easier to heal since the terminating node, right before it quits, can post a notification to all its neighbors about its intent to quit. Unplanned terminations (failures) need a more complex healing mechanism.

Planned termination. If a node plans to quit, it posts a `QUIT` message to all of its neighbors, containing the notification that it is about to terminate, and waits for their replies before dropping the connections to all its neighbors. The `QUIT` message also contains a list of neighbors it currently peers with, of which information will be used by the surrounding neighbors to quickly heal around the spanning tree by forming another spanning tree just among the neighbors of the quitting node themselves. The resulting (overall) spanning tree is not generally an optimal one, but this healing quickly restores the tree connectivity and the relaxation operations that are running *on the background* eventually turn the tree back into a new optimal tree.

Unplanned termination. Now suppose that a node fails (crashes) and thus it has no way of posting a `QUIT` message to its neighbors before it terminates. Without the `QUIT` message, the surrounding neighbors of the terminating node eventually detect the node termination because of the soft-state routing information which will time out after a lifetime. Even though the surrounding neighbors do not have an explicit list of neighbors of the terminating node, they can deduce the information from their routing table (assuming the routing table is in steady state) which nodes are two-hop away in the direction of the terminating node. They can then heal around the spanning tree as before.

The tricky part now is that only from this soft-state information, a node will not be able to differentiate whether the other node actually terminates, or only the link to that node goes down, in which case that node may still have connections to some of its other neighbors. In the latter case, forming a spanning tree with all two-hop away nodes in the direction of the unreachable node is not a solution since it may end up producing a cycle. The algorithm incorporates two types of healing mechanisms for this:

- **Quick healing**, which is used to quickly heal a link failure or a node failure. If quick healing can not successfully heal the whole graph (i.e., there are some nodes that are still disconnected), incremental healing will be used.

- **Incremental healing**, which is used after quick healing to ensure all nodes are connected, or when the quick healing method can not be applied.

Quick healing. In the quick healing, the following steps must be taken by a node i that detects an unreachability to its neighbor j:

(i) Node i queries all neighbors of j for information whether any of them can still reach node j. If i finds any such node, it peers with it. Otherwise, it asks all neighbors of j to form a spanning tree among themselves to heal around the connectivity. This last step of creating a spanning tree among themselves is taken only if i has successfully contacted *all* neighbors of j and finds that none of them can reach j. If, on the other hand, i is able to contact only a subset of them (due to possible link failures between i and some of j's neighbors), step (ii) below is taken first.

(ii) Node i sends out a second round of queries to all neighbors of j to find whether any of them has successfully found and peered with another neighbor of j that can reach j. If i finds any such node, it peers with it. Otherwise, as the last step, node i asks all reachable neighbors of j to form a spanning tree among themselves to heal around the connectivity.

After performing the above step, node i figures out whether there are some other nodes that are still unreachable, in which case it enters the incremental healing mode.

Incremental healing. When a node detects the existence of some unreachable nodes (by comparing the entries of its routing table with a list of available nodes in the system from the DSR), and it can not use the quick healing to heal the connectivity of the spanning tree, it initiates an incremental healing. It broadcasts a query to find out whether any node in its component can reach any of those nodes to which it has no connectivity (i.e., whether any outgoing link of its component can connect to any of those nodes). Similar to the *slotting and damping* mechanism used in [53, 20], the node waits for a random time before sending the query and refrains from sending the query if it sees another incremental-healing query from a different node.

If multiple nodes reply with a positive answer about the links to connect to any of those nodes, it selects the link with the least cost and asks the immediate node of the link to add the link into the spanning tree. Since potentially the other component is also performing the same search and finds a connecting link, but possibly a different one (due to dynamic metrics, relaxation interference, or other causes), a cycle may be formed. Hence, after adding a new link to the tree, the component waits for a period of time (in the order of the diameter of the network in time) to find out if there are still some other unreachable nodes and also to detect any cycle in the graph. Cycle-detection operation can be performed easily by sending the same relaxation's `PROBE` message; if a node receives its own probe we know a cycle exists. To get back into a spanning tree, we need to remove a link from the cycle — the maximum-cost link of the cycle, of which information is contained in the `PROBE`. Similar conditions to the ones for relaxation can be applied to regulate concurrent cycle-detection operations. After waiting for some period, if a node still finds some unreachable nodes, it initiates another incremental healing operation as before.

4.5 Summary

In this chapter, we have described the self-configuring algorithm used by the INS name resolvers to configure themselves into a spanning tree topology that is adaptive and self-improving based on network latency metrics. Although the algorithm presented was for constructing and maintaining the INS resolver network, the algorithm was general enough to be applicable to other systems that desire an adaptive and self-improving network topology. The algorithm works in stages. First, a spanning tree is constructed amongst the nodes based on the local minimization decision performed by each node when it joins the system. Then, the relaxation protocol of the algorithm improves the tree to eventually attain a minimum spanning tree. The algorithm allows concurrent relaxations to proceed as long as they satisfy several conditions described. Messages used in the relaxation protocol were also described. Finally, this chapter discussed the healing mechanisms used to overcome node failures and network partitions, which include quick healing and incremental healing.

Chapter 5

Applications

This chapter describes how applications use INS and presents three demonstration applications that we have developed leveraging INS supports for resource discovery, mobility, and group communication. In Section 5.1 we describe the features and functionality of INS that are provided to clients and services as part of the INS service model. Section 5.2 presents *Floorplan*, a graphical service discovery tool; Section 5.3 presents *Camera*, a mobile camera service; and Section 5.4 presents *Printer*, a load-balancing printer utility.

5.1 INS Application Interface

INS provides interfaces that enable applications to take advantage of features such as transparent node mobility, service dynamism, and flexible group communication by using an intentional name as the communication handle. Several basic functions for advertising and querying names to a name resolver are provided:

- Services can periodically advertise their soft-state name information to an INR, which will be disseminated to the other INRs in the system. Services, when advertising their name, can include an application-specific metric to be associated with the name, which is used by the INRs to select the "best" location when multiple service nodes satisfy a given query.

- Services can explicitly update or remove name information they have previously advertised, without having to wait for the name to time out. This is useful for example, when a service physically moves to a different location and it wants its new location attribute to be reflected immediately. In general, when an attribute in a service description or an application-specific metric for it changes such that it causes a change in the "best" network location of a service, the service wants to update its name and any information in its name-record immediately.

- Clients can discover whether names containing particular attributes (which correspond to some services/clients) exist in the system, by sending a request containing a query expression to an INR. Because name information is disseminated through the INR network in a timely manner, a new service becomes known to other resolvers and through them to the clients.

Clients and services can request a late-binding delivery service for their messages, by sending a destination intentional name (describing the attributes of the intended destinations) together with the data to an INR. The INR determines the appropriate destination nodes that satisfy the given name and performs the rest of the delivery, which may involve other INRs in the process. Clients and services that use late binding can choose to send their messages by either intentional anycast or intentional multicast by setting the *Delivery* bit-flag in the message header (message format is discussed in Section 6.1).

There are some optional functions that enable clients to select whether to allow an INR to *infer* the source name of the message. If inference is enabled, the INR that receives the application's message will infer from which application node the source name is and record the information in the name table. The INR then disseminates the information to the other INRs using a triggered update, if the information is new or different. This *inference* enables INRs to forward any response from services back to the requesting clients without having the clients explicitly advertise themselves.

In the intentional anycast delivery, applications have an option to specify the application ID (*App-ID*) of the destination name in the message header, causing the

INRs that receive the message to forward the message only to the destination name with the given App-ID. This is useful when a sender wants to maintain persistent communication to the same end-node under all conditions.

Late-binding deliveries — intentional anycast and intentional multicast — use names as communication handles, providing a "level-of-indirection" that enables clients to continue communicating with end-nodes, even if the name-to-address mappings change while a session is in progress.

In addition to the late-binding delivery service, INS offers an early binding service, which allows an application to lookup a destination name and obtain the corresponding network locations. To use this service, an application sends to an INR a request for early-binding information of a particular name. Receiving the request, the INR returns a list of *early-binding records* that match the given name. Each record consists of an IP address, a port number and an additional set of [`port-number, transport-type`] pairs. This service model is useful when services and clients are relatively static.

INS service model allows clients and services to join and leave the system without any explicit registration and de-registration. The soft-state mechanism used by INRs in storing names facilitates the management of up-to-date names.

The details of INS API implementation for clients and services are described in [8]. The API functions for creating and manipulating intentional names in *name-specifier* format are described in [50].

5.2 *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 the user's display as a building floorplan. *Floorplan* learns about new services by sending a query expression to an INR about all services in the current region. The INR then replies with the query result containing all the services in that region. *Floorplan* uses the location and service attributes contained in the returned

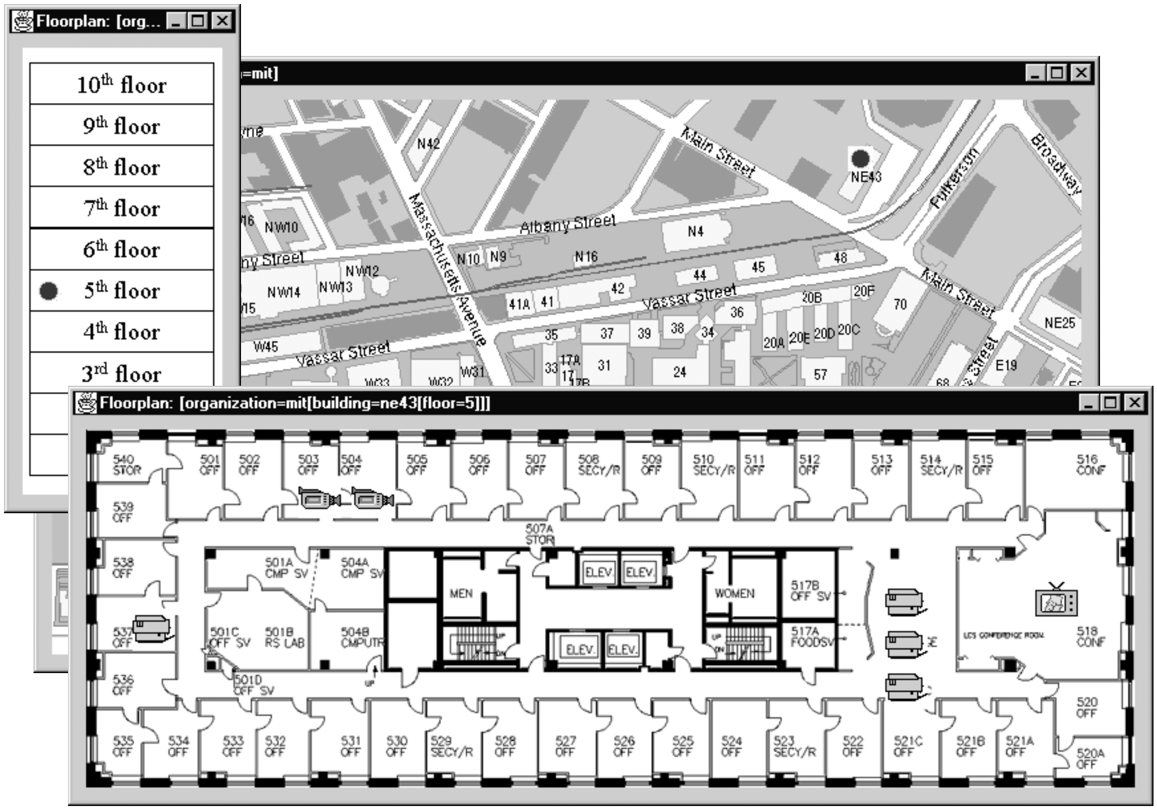

Figure 5-1: A screenshot from *Floorplan*. It starts by displaying the MIT map with a circle by NE43 to indicate that a service has been discovered there. Clicking on the circle brings up a zoomed-in map of building NE43. Clicking on the circle by the 5th floor brings up a floorplan of the 5th floor. The floorplan shows all services that have been discovered: cameras in room 503 and 504, three printers in the lounge and one in room 537, and a TV stream broadcaster in room 518. Clicking on a service icon accesses or requests functionality from that service.

names to deduce the location and the type of each service and display the appropriate icon. Figure 5-1 shows a screenshot from *Floorplan*.

Using INS for discovering resources allows *Floorplan* to specify what information or functionality it is looking for, rather than where to find it. *Floorplan*, when it starts, obtains the map of the region by describing to an INR the properties of the map it requires. The INR then forwards the request to the optimal service node (i.e., optimal map server) that has the map information satisfying the given properties. By using INS late-binding in requesting the desired map, *Floorplan* does not need to know the network location of the map server. In addition, late binding also provides a useful abstraction for the map server design, allowing the map server to be replicated for automatic load-balancing and transparent fail-over of a server. Since INRs are the ones that determine the appropriate service nodes that have the requested map, the map server service can also be designed as a network of *distributed* local map servers that cooperate to provide a system-wide map service, rather than having all replicated map servers keep the same amount of information.

In our implementation, *Floorplan* sends a request using the following intentional name: `[service=map [entity=server]][`*location*`]`, where *location* is the region of which map is requested.

In response, the map server sends the requested map back to the requesting *Floorplan* instance, using the requestor's intentional name to route the message, allowing the requestor to be mobile (e.g., because the user using the *Floorplan* moves around exploring the region).

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 5.3), printers (Section 5.4), and real-time video stream for TV channel broadcast and music jukebox. These service providers advertise intentional names specifying several of their attributes, including their location in the building. For example, a camera in Room 510 advertises the following intentional name:

`[service=camera[entity=transmitter][id=a]][room=510]`.

If a service moves to a different geographic location, it refreshes the location value in its intentional name to reflect the new location, causing the *Floorplan* to update and move the icon of the service correspondingly to the new location on the user's display.

5.3 *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

Figure 5-2: A screenshot from *Camera*. The display shows the latest image that has been received. Clicking the "Update" button will request the latest image from the camera, used in the *request-response* mode. The "Subscribe" and "Unsubscribe" buttons are for *subscription-style* interaction, where the camera sends an image using intentional multicast destined to all subscribers.

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 communicating in the presence of node or camera mobility. One possible value for the `id` is the application ID, *App-ID* (discussed in Section 2.2).

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 comes into effect. The camera may also explicitly remove the old name before it expires. 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. For example, a client can request current image from all mobile cameras in the west wing of the buildings. Since there can be a different subset of cameras that happen to be in the west wing at every instant of time, INRs responsively determine the appropriate camera nodes during the message delivery by using late binding.

To enable the subscription-style feature, the *Camera* transmitter sends out an image destined to all users subscribing to its images by setting the *Delivery* bit-flag to *all*. 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 sub-

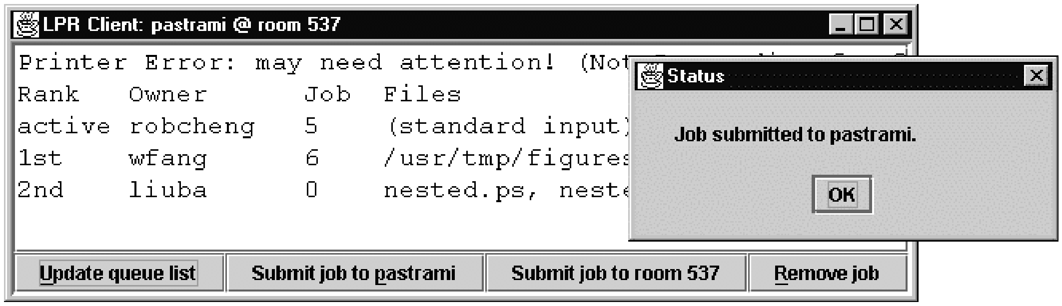

Figure 5-3: A screenshot from *Printer*. The display shows the jobs currently in the printer queue for the named printer (pastrami). "Submit job to pastrami" will submit a new job to the printer named pastrami, while "Submit job to room 537" will submit a new job to the least-loaded printer in room 537. It uses the INS intentional anycast feature to discover the "best" printer node (*Printer* advertises a better metric for a less loaded printer).

scribers, regardless of their specific IDs. Using INS late-binding service, camera applications handle node mobility and enable flexible group communication.

5.4 *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 complete 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 and load status. 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 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:

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

and sets the *Delivery* bit-flag to *any*. Note that the name of the printer is wildcarded 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.

Another feature, which increases the robustness of the *Printer* application, is to have automatic fail-over mechanism for printers in the same room. Suppose one of the printers in room 517 fails. Soft-state name information used by INRs ensures that the intentional name of the failing printer will be eliminated after a lifetime. Hence, any subsequent requests to send a print job to a printer in room 517 (using the same destination name as before) will be transparently re-routed to the non-failing printer in room 517. Using intentional anycast to the "best" printer in the room, clients transparently handle printer failures and receive automatic fail-over availability.

Chapter 6

Implementation

In order to demonstrate the utility of INS, we have implemented INS name resolver (INR), evaluated and tested it. Our implementation supports all features of INS service model, including late-binding delivery (intentional anycast and multicast), early-binding service, name advertisements and queries, and uses a distributed self-configuring protocol for the construction and maintenance of the INS resolver network. Our current INR implementation is in Java, to take advantage of its cross-platform portability and modularity. Clients and services, however, are not constrained to be written in Java. We have tested this implementation using a number of applications, including those described in Chapter 5.

In Section 6.1 we show the INS message format that is used by both applications and INRs. Section 6.2 describes the architecture of an INR in our modular implementation, with description of functionality for each module. We present our Java classes that compose each module of an INR in Section 6.3. Section 6.4 evaluates the performance of our implementation.

6.1 INS Message Format

Figure 6-1 shows the INS message format. The version field indicates the version of INS protocol used. The hop-count field limits the number of hops a message can traverse in the overlay. It is initialized by the sender to some value and decremented

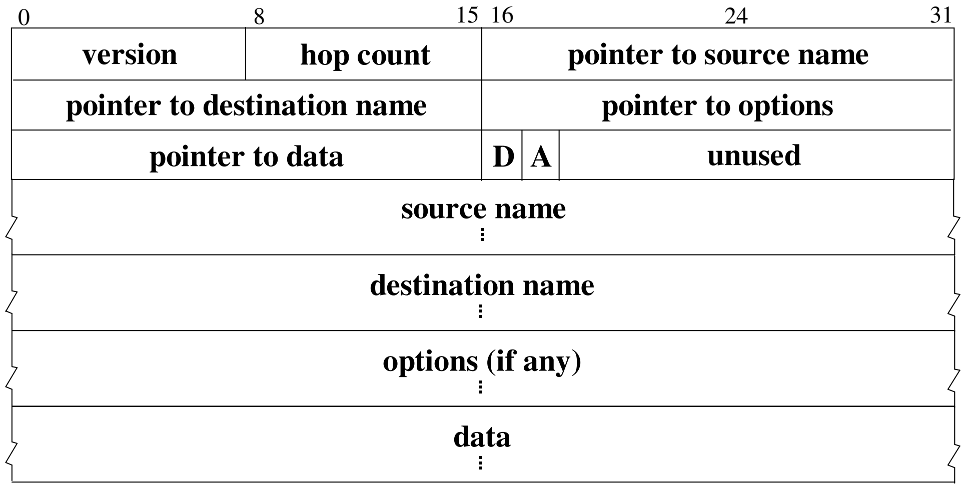

Figure 6-1: INS message format.

by one by every INR that forwards the message. Because intentional names are of variable length, the header contains pointers to the source name, destination name, options, and data. INRs do not process application data.

The *Delivery* big-flag (D) is used to determine whether intentional anycast or intentional multicast is used in the message forwarding. The *Application* bit-flag (A) is used to indicate whether the sender of a message is an application or another INR. This bit-flag is used among others to learn the source name by *inference* if the source name is from an application.

The option field is used for several options in the late-binding delivery. Option field contains a sequence of options, each with the following format:

kind	length	option-data

Two options are currently provided to applications: (i) an option to enable/disable INR performing *inference* about the source name of the message, and (ii) in the intentional anycast delivery, applications may specify an optional application ID (*App-ID*) of the destination name in the message, causing the INRs that receive the message to forward only to the destination name with the given App-ID.

Control messages in INS use the same message format. Control messages are identified by their destination names containing attribute `control-msg`. This attribute is reserved for INS control messages, such as name and route updates, overlay tree

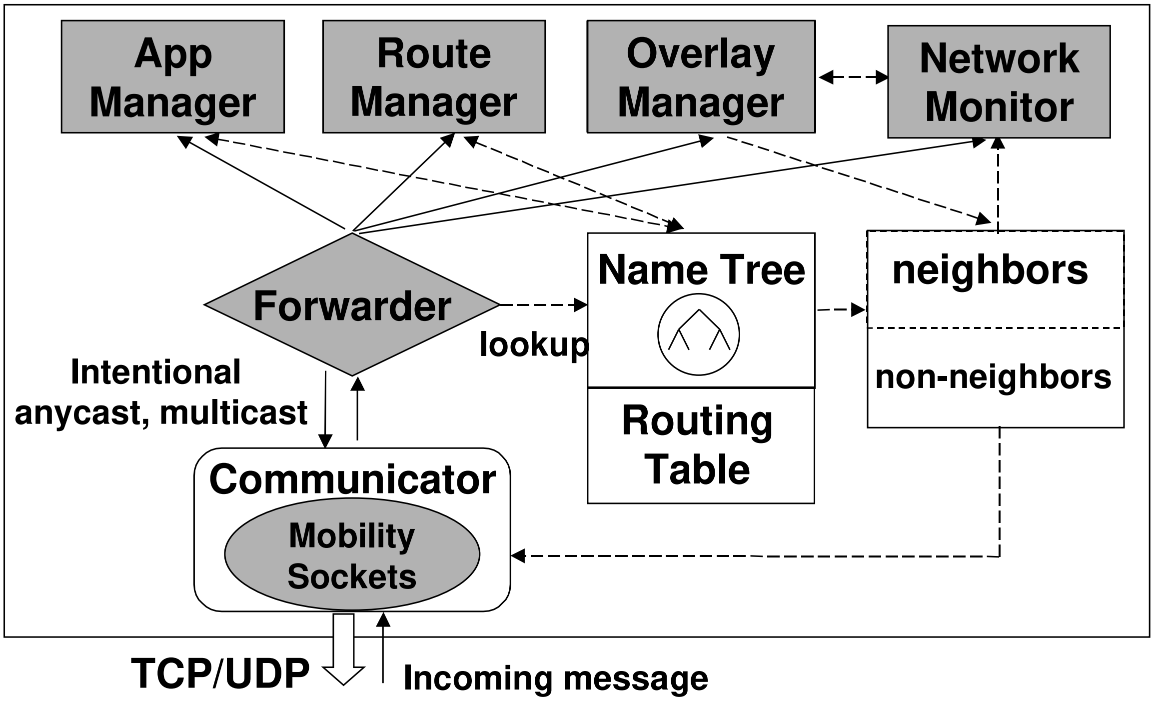

Figure 6-2: INR node architecture.

probing and relaxations, as well as control messages to/from applications for name advertisements, name queries and early binding requests. The following section describes each type of control messages in more detail in the context of each module of an INR.

6.2 INS Resolver Node Architecture

INRs communicate with one another, forwarding application messages via UDP and exchanging name updates and route updates via long-lived TCP connections that are established during the spanning-tree construction of the resolver network.

INR is structured and implemented in a modular fashion. Figure 6-2 shows the structure and modules of an INR, which include the following:

- **Name-tree**, a data structure that stores the mapping between a name and its corresponding *name-record*. Each name-record contains all associated information for a name, including the *App-ID* of the application announcing the name, the application-advertised metric, the early-binding record, and the *INR-ID* of the INR to which the application attaches. Since name-tree is dependent on the exact name language used, different languages may use different data structures for the maintenance of names. Current implementation uses name-specifier format and implements the name-tree as shown in Figure 6-3 [50]. Looking up a name in the name-tree will return its corresponding name-record.

- **Routing-table**, a data structure that stores the routing information to get to every active INR in the system through the overlay network. The table maintains a mapping between INR-ID and the next-hop INR for the route toward INR-ID.

- **Neighbors** and **non-neighbors**, data structures that maintain the information about current neighbors that the INR-node peers with and the rest of the currently active INRs in the system. An INR obtains a list of active INRs in the system from the DSR. Neighbors are selected and maintained by the **OverlayManager**.

- **RouteManager**, a module that implements the name discovery and routing protocols. It has a reference to the name-tree and routing-table data structures. The name discovery protocol maintains the names in the name-tree using soft-state and hard-state mechanisms as described in Section 3.3. That is, periodic advertisements from applications are stored as soft-state, while incremental updates from neighbor INRs are stored as hard state. Name discovery protocol employs a dedicated thread that periodically checks for the expiration of soft-state names and then sends an incremental update to neighbors to remove any expired names that it found. Routing protocol processes route updates and periodically disseminates updates to refresh the routing-table entries of the neighboring INRs. Routing protocol employs a dedicated thread for the periodic refreshes. Since RouteManager is a handler for name-update and route-update control messages, during its initialization it registers itself to the name-tree as the handler for those control messages.

- **OverlayManager**, a module that implements the self-configuring spanning tree. When the INR-node starts, OverlayManager contacts the DSR to retrieve the list of currently active INRs in the system and selects the *best* INR to peer with based on the INR-to-INR round-trip latency. NetworkMonitor in the INR is the one responsible for obtaining measurements of network performance, including the network latency to other INRs. After peering with an existing active INR for the initial construction of the spanning tree, OverlayManager enters a relaxation mode, in which it listens to and sends out periodic `PROBE` messages to/from neighbors. OverlayManager is the one that keeps the spanning tree adaptive and self-improving using the relaxation protocol; it handles the relaxation messages specified in Section 4.3. In performing relaxations, OverlayManager incorporates some hysteresis with thresholds, such that it replaces a high-cost link with a lower-cost link only if the benefit obtained by this replacement exceeds a certain threshold.

- **AppManager**, a module that serves clients and services. It handles name advertisements and queries from applications and replies to early-binding requests from applications. Name advertisements from applications will be stored by the AppManager to the name-tree as soft-state names.

- **NetworkMonitor**, a module that determines the network performance, particularly the round-trip latency, to all other active INRs in the system. It responds and performs the *INR-ping* experiments, of which measurements are used by the OverlayManager during the initial construction of the tree and from time to time when it is in the relaxation mode.

- **Forwarder**, a module that decides where to forward an incoming message. It bases its forwarding decision on the name-tree data structure. When an incoming message is received, Forwarder looks up the name tree to retrieve its name-record. The name-record contains the information of where to forward the message to. If the message is a control message, the name-record contains the information of which module is responsible for that control message, and hence the Forwarder passes up the message to that module. Otherwise, if the message is for another application, it checks the *Deliver* flag to determine either intentional anycast or intentional multicast forwarding to be used. Intentional anycast forwarding follows the algorithm in Figure 3-6, while intentional multicast follows Figure 3-7. Hence, in intentional multicast after obtaining the matched name-records, the Forwarder needs to look up the routing table to determine the next-hop INRs it should forward the message to. The "pointer" mechanism implemented avoids this second lookup of routing-table, i.e., by creating a new field in the name-record that stores the "pointer" to the routing-table entry corresponding to the next hop INR toward the name. The Forwarder decrements the *hop count* field in the message header before forwarding the message to the application node or to another INR node.

- **Communicator**, is a module that abstracts away the socket implementations to neighbors and non-neighbors. It also implements **MobilitySocket** that automatically re-binds the sockets whenever changes to its IP address are detected.

Four modules of an INR: RouteManager, OverlayManager, AppManager, and NetworkMonitor, are *handlers* to INS control messages. Each handles different types of control messages corresponding to its functionality. RouteManager handles control messages that have a destination name containing the following attributes and values:

- `[control-msg=name-update]`: This control message is for name update, used by the name discovery protocol to maintain consistent name information among INRs in the system.

- `[control-msg=route-update]`: This control message is for route update, used by the routing protocol to refresh network reachability in the overlay network to other INRs in the system.

OverlayManager handles all control messages for spanning tree construction and relaxation protocol. These control messages, including the ones shown in Figure 4-9, have a destination name containing the attribute and value `[control-msg=overlay]`.

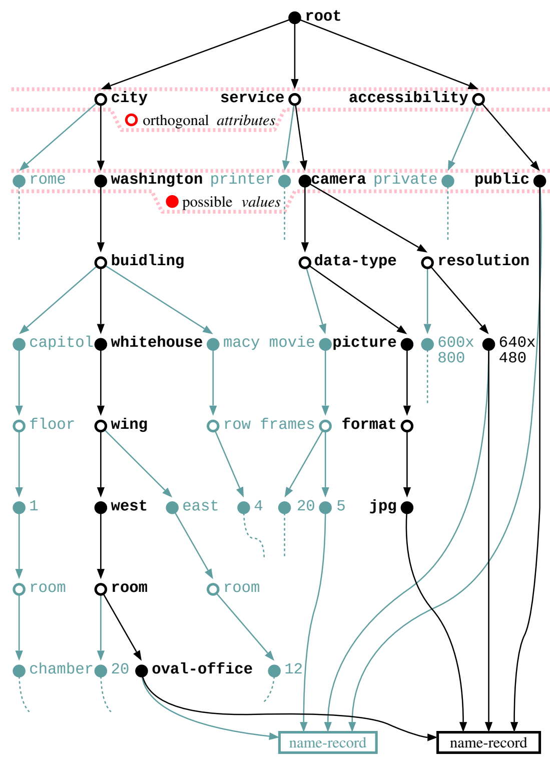

Figure 6-3: A name-tree data structure for storing and looking up names in *name-specifier* format to retrieve its name-record (from [50]).

Communication between applications and AppManager uses the following control messages:

- `[control-msg=announcement]`, used by applications to advertise a new name together with its name record to an INR, or to update the information in its name record (e.g., changes of application-advertised metric), or to explicitly remove names they previously advertised.

- `[control-msg=discovery]`, used by applications to query for names in the system; the query is dependent on the name language used and current implementation supports exact matches and wildcard entries, allowing omissions of *don't-care* attributes.

- `[control-msg=early-binding]`, used by applications to request the early-binding records of a name.

NetworkMonitor handles control messages that relate to the network performance measurements, in particular, the *INR-ping* messages, which have a destination name containing `[control-msg=ping]`.

In summary, a message that arrives on an INR node will be received by the Communicator module from the network-layer (UDP/TCP) handler. Communicator passes up the message to the Forwarder, which will lookup the name to retrieve its name-record. If the message is an INS control message, the name-record indicates which handler (i.e., RouteManager, OverlayManager, AppManager, or NetworkMonitor) the message is for. Otherwise, based on the type of delivery — intentional anycast or multicast — Forwarder forwards the message using the information contained in the name-record. Forwarder decrements the *hop count* field in the message header and passes down the message to the Communicator, which will then forward to the application node or to another INR node.

6.3 Implementation Components

Our Java implementation is developed following the modular design of an INR as shown in Figure 6-2. All the modules and data structures are each implemented by one or more Java classes. Figure 6-4 shows a list of classes that implement an INR.

The classes that make up the RouteManager modules are *RouteManager*, *NameUpdate*, *RouteUpdate*, and *RouteUpdateThread*. RouteManager class implements the name discovery and routing protocols, which perform the processing of name updates and route updates respectively. NameUpdate class is used for incremental updates between INRs and periodic advertisements from applications, while RouteUpdate class is used to disseminate and refresh routing information to neighboring INRs. RouteUpdateThread is a dedicated thread that sends periodic route refreshes to neighbors.

The classes that make up the OverlayManager module are *OverlayManager* and *DSRManager*. DSRManager class implements the communication protocol to a DSR, including retrieving the list of currently active INRs in the system. This list is then used by the OverlayManager class to constructs an initial spanning tree. OverlayManager implements the self-configuring spanning tree algorithm; it performs relaxation operations to evolve the spanning tree into an optimal one.

AppManager class implements the AppManager module, while *NetMonitor* implements the NetworkMonitor module. RouteManager, OverlayManager, AppManager, and NetMonitor classes implement the handling of INS control messages, and are all sub-class of the *IHandler* class.

Communicator module is composed of the *Communicator* class that uses the *MobilitySocket* class for its socket implementation, and the *Message* class that is used for communication with the *Forwarder* class.

The classes that make up the Forwarder module are *Forwarder* class that implements intentional anycast and intentional multicast forwarding, and *UDPForwardThread* and *TCPForwardThread* classes that wait for messages being passed up by the *Communicator* class.

Neighbors and non-neighbors data structures use the following classes: *Neighbors*, *Node*, and *NodeSet*. Routing-table is implemented as a hash table using the *RouteTable* and the *RouteEntry* classes. Name-tree uses the *NameRecord* class as well as a number of classes from [50] (shown in Figure 6-5) to implement a data structure that stores and looks up intentional names in *name-specifier* format.

6.4 Evaluation

In this section, we evaluate the performance of our INS implementation, in particular the responsiveness in handling change and dynamism in services and nodes, and the performance of the message routing and forwarding for late-binding service. 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.

6.4.1 Name Discovery Performance

We evaluated the responsiveness of our INS implementation to change and dynamism in services and nodes by measuring the performance of the name discovery protocol. We measured the performance of INS in discovering *new* services, which advertise their existence via intentional names. Figure 6-6 shows the average discovery time of a new name as a function of n, the number of hops in the resolver network from the new name.

When an INR observes a new name from a service advertisement, it processes the update message and performs a lookup operation on the name-tree to see if a name with the same *App-ID* already exists. If it does not find it, it adds the name into its name-tree and propagates an update immediately to its neighbors. Thus, the name discovery time in a network of identical INRs and links, $T_d(n) = n(T_l + T_m + T_{up} + d)$, where T_l is the lookup time, T_m is the time to merge a new name into the name-tree, 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 those merge operations, since we were not running any other applications in the system during the measurements. Thus, the lookup and merge times at one INR and the others were roughly the same. As shown in Figure 6-6, $T_d(n)$ 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.

6.4.2 Routing Performance

In addition to the discovery experiment, we also measured the performance of the message routing and forwarding by an INR. 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 source and destination names were randomly generated, on average 82 bytes long. The results are shown in Figure 6-7.

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, from 3.1 ms per message with 250 names to 19 ms per message 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 when examining the data for messages destined to a remote INR. For the most part, the next-hop processing time with several thousand names in the name-tree is between 9 ms and 10 ms per message 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.

In this chapter, we have described our INS implementation. We showed the format of the INS message and the architecture of an INR node in our implementation. The INR node is designed in modules with a well-defined function for each. We described the Java classes that made up each module of an INR. Finally, we presented experiments that evaluated the performance of the name discovery protocol and the performance of the message routing and forwarding in INS late binding.

Java class	**Used by INR module**	**Lines**
AppManager	AppManager	447
Communicator	Communicator	409
DSRManager	OverlayManager	416
Forwarder	Forwarder	502
HostRecord	Early-binding information	273
IHandler	All *handlers*	33
Message	Communicator, Forwarder	38
MobilitySocket	Communicator	70
NameRecord	NameTree	673
NameUpdate	RouteManager	238
Neighbors	Neighbors	66
NetMonitor	NetworkMonitor	310
Node	Neighbors	372
NodeSet	Neighbors	71
OverlayManager	OverlayManager	2124
Packet	INS message	439
Resolver	Entry (main) module	375
RouteEntry	Routing Table	138
RouteManager	RouteManager	574
RouteTable	Routing Table	68
RouteUpdate	RouteManager	68
RouteUpdateThread	RouteManager	47
TCPForwardThread	Forwarder	159
UDPForwardThread	Forwarder	24
Conversion	(auxiliary class)	87
LoggedPrintStream	(auxiliary class)	180
MutableBoolean	(auxiliary class)	15
SendWithRetransmit	(auxiliary class)	69
MultipleOverlays	(auxiliary class)	865
Total		9150

Figure 6-4: Java classes in the INR implementation and their use in the INR module. The last column gives the number of lines of the source code, including comments and whitespace.

Java class	Used by INR module	Lines
NameTree	Name-tree	140
AttributeNode	Name-tree	123
ValueNode	Name-tree	299
NameRecordSet	Name-tree	274
Total		836

Figure 6-5: Other Java classes from [50] that are used to implement a name-tree. The last column gives the number of lines of the source code, including comments and whitespace.

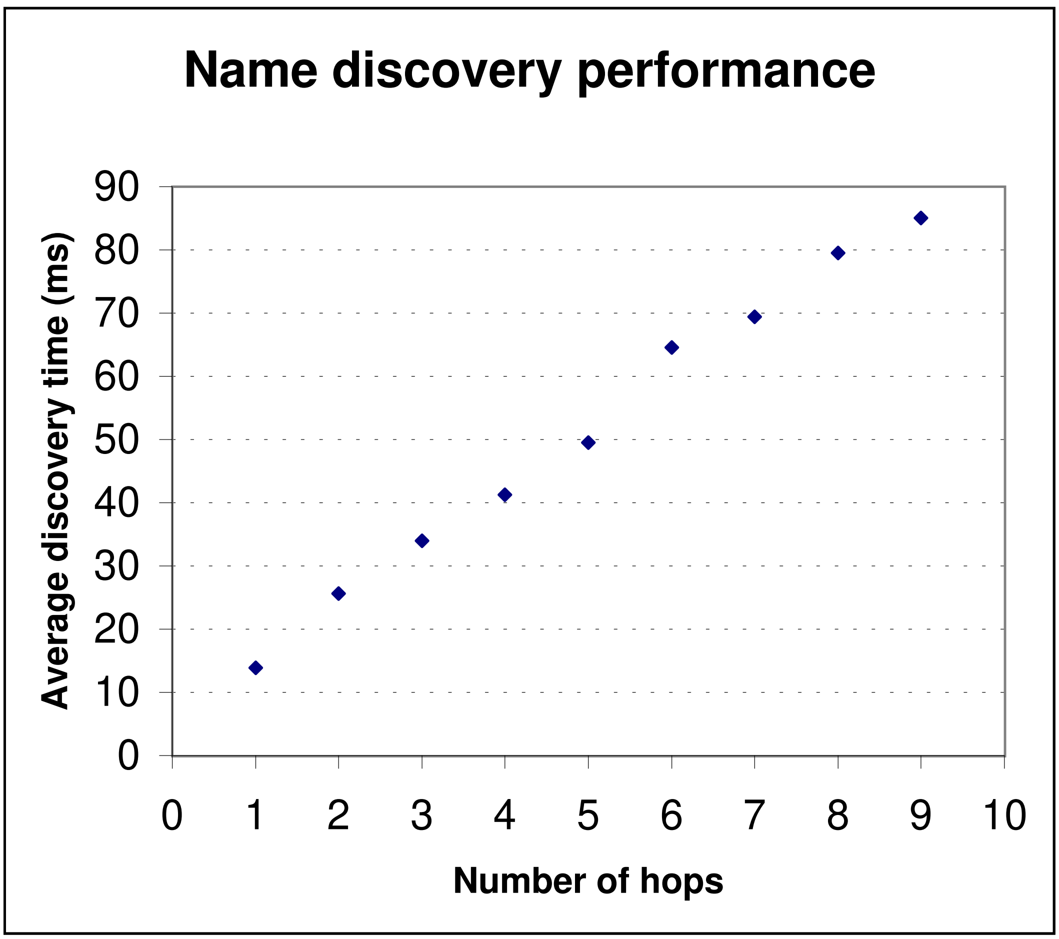

Figure 6-6: Discovery time of a new network name. This graph shows that the time to discover a new network name is linear in the number of INR hops.

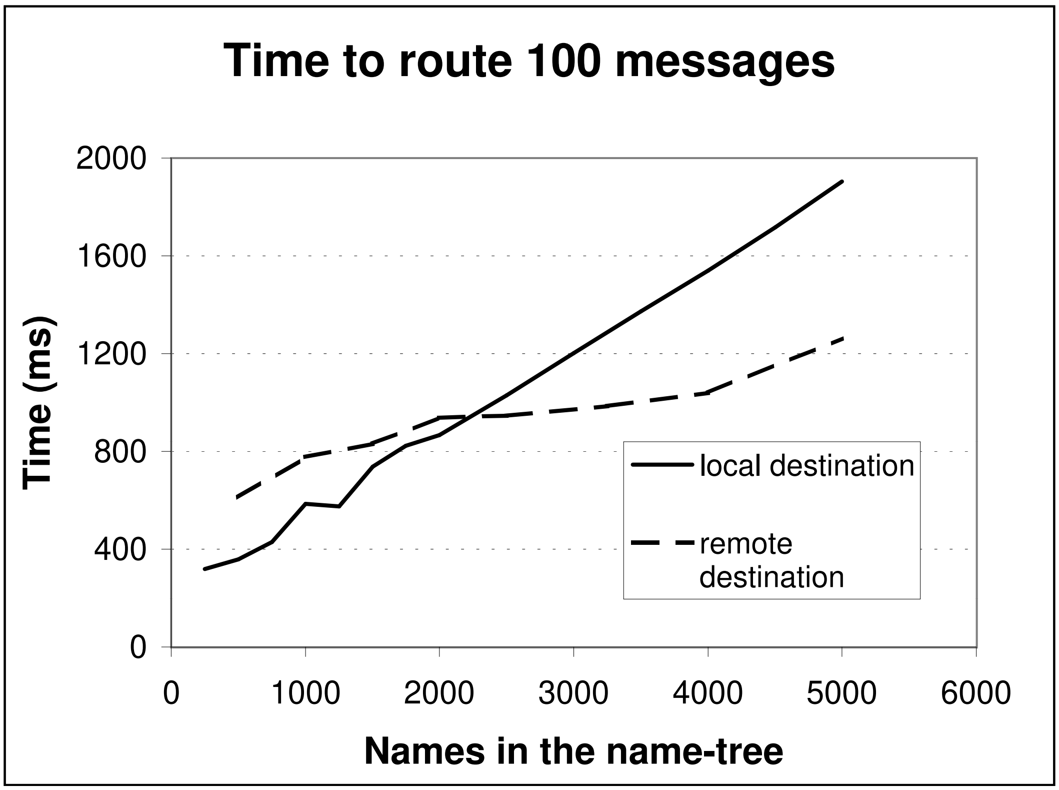

Figure 6-7: Processing and routing time per INR for a 100-message burst, in the intra-INR and inter-INR cases.

Chapter 7

Conclusions

The combination of heterogeneity and dynamism in the future network environment makes it hard for applications to discover the network locations of services that best satisfy their need. Applications often know the services they need, but do not always know where to find them. In this thesis, we presented an Intentional Naming System (INS), which allows network applications to gain access to and communicate with services by specifying their attributes, rather than by specifying where to find them. INS applications can therefore seamlessly handle node mobility and service dynamism, and take advantage of a flexible group communication service using an expressive name as the group handle.

INS uses a simple naming language based on attributes and values to achieve expressiveness, allowing applications to describe a wide variety of devices and services. INS is designed to be responsive, easily configurable, and robust.

To achieve responsiveness to mobility and performance changes, INS integrates name resolution and message routing in an operation called *late binding*. Late binding offers two forms of delivery: 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 nodes satisfying a query.

INS uses a novel distributed self-configuring algorithm that enables its name resolvers to configure themselves to form a resolver network without any manual configuration. The algorithm constructs a spanning tree in a fully distributed fashion based on the INR-to-INR round-trip latency, and continually modifies the tree to optimize the latency metric. The algorithm includes a mechanism by which INRs can improve the tree and evolve it into an optimal tree, i.e., by using the *relaxation* protocol, which incrementally and asynchronously adapts the neighbor relationships between INRs.

INS uses soft-state periodic service advertisements and soft-state routing information between replicated resolvers to achieve robustness. This choice allows a design where applications can join and leave the system without explicit registration and de-registration. Soft-state improves the robustness of the system against internal errors and inconsistency, as incorrect information will be refreshed in the next cycle of updates, and node failures and network partitions can be detected in a timely manner.

Based on our experience in using INS, we believe 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 names 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.

Bibliography

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. 17th SOSP*, pages 186–201, December 1999.

[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. H. Ammar, E. W. Zegura, V. Shah, and Z. Fei. Application Layer Anycasting. In *Proc Infocom '97*, pages 1388–1396, April 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] A. Chakraborty. A Distributed Architecture for Mobile Location-Dependent Applications. Master of Engineering Thesis, Massachusetts Institute of Technology, June 2000.

[9] Y. Chawathe, S. McCanne, and E. Brewer. RMX:Reliable Multicast in Heterogeneous Networks. In *Proc. IEEE INFOCOM*, March 2000.

[10] Y. Chu, S. Rao, and H. Zhang. A Case For End System Multicast. In *Proc. ACM Sigmetrics*, June 2000.

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

[12] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM*, pages 106–114, August 1988.

[13] 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.

[14] 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.

[15] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2):136–161, May 1990.

[16] 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.

[17] A. Deutsch, M. Fernandez, D. Florescu, Levy A., and D. Suciu. A Query Language for XML. `http://www.research.att.com/~mff/files/final.html`.

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

[19] B. Fink. 6bone Home Page. `http://www.6bone.net/`, January 1999.

[20] S. Floyd, V. Jacobson, S. McCanne, C. G. Liu, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proc. ACM SIGCOMM*, Boston, MA, September 1995.

[21] P. Francis. Yallcast: Extending the Internet Multicast Architecture. `http://www.yallcast.com/`, September 1999.

[22] R. G. Gallager, P.A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and System*, 5(1):66–77, January 1983.

[23] 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.

[24] E. Guttman, C. Perkins, J. Veizades, and M. Day. *Service Location Protocol, Version 2*, June 1999. RFC 2608 (`http://www.ietf.org/rfc/rfc2608.txt`).

[25] M. R. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proc. 24th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 594–604, 1997.

[26] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. ACM Symposium on Theory of Computing (STOC) 98*, pages 79–89, May 1998.

[27] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proc. ACM/IEEE MOBICOM*, August 2000.

[28] 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.

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

[30] B Lampson. Designing a Global Name Service. In *Proc. 5th ACM Principles of Dist. Comput.*, pages 1–10, August 1986.

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

[32] J. Lilley. Scalability in an Intentional Naming System. Master of Engineering Thesis, Massachusetts Institute of Technology, June 2000.

[33] 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.

[34] P. V. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of SIGCOMM '88 (Stanford, CA)*, pages 123–133, August 1988.

[35] 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.

[36] Object Management Group CORBA/IIOP 2.3. `http://www.omg.org/corba/corbaiiop.html`, December 1998.

[37] 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.

[38] C. Partridge, T. Mendez, and W. Milliken. *Host Anycasting Service*, November 1993. RFC 1546 (`http://www.ietf.org/rfc/rfc1546.txt`).

[39] C. Perkins. *IP Mobility Support*, October 1996. RFC 2002 (`http://www.ietf.org/rfc/rfc2002.txt`).

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

[41] R. Perlman. *Interconnections: Bridges and Routers.* Addison Wesley, Reading, MA, 1992.

[42] J. B. Postel. *Transmission Control Protocol.* Internet Engineering Task Force, September 1981. RFC 793 (http://www.ietf.org/rfc/rfc0793.txt).

[43] 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.

[44] J. Reynolds. *Technical Overview of Directory Services Using the X.500 Protocol*, March 1992. RFC 1309 (http://www.ietf.org/rfc/rfc1309.txt).

[45] J. Robie, D. Chamberlin, and D. Florescu. Quilt: an XML Query Language. http://www.w3.org/XML/Group/2000/03/Quilt/, March 2000.

[46] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, September 1998.

[47] E. C. Rosen, A. Viswanathan, and Callon R. Multiprotocol Label Switching Architecture. http://www.ietf.org/internet-drafts/draft-ietf-mpls-arch-06.txt, August 1999. Internet Draft, expires February 2000.

[48] J. Saltzer, D. Reed, and D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2:277–288, Nov 1984.

[49] 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).

[50] E. Schwartz. Design and Implementation of Intentional Names. Master of Engineering Thesis, Massachusetts Institute of Technology, June 1999.

[51] 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.*, April 1995.

[52] I. Stoica, T. S. Eugene, and H. Zhang. REUNITE: A Recursive Unicast Approach to Multicast. In *Proc. IEEE INFOCOM*, March 2000.

[53] W. T. Strayer, B. J. Dempsey, and A.C. Weaver. *XTP: The Xpress Transfer Protocol.* Addison-Wesley, Reading, MA, 1992.

[54] Rapid Infrastructure Development for Real-Time, Event-Driven Applications. `http://www.talarian.com/collateral/SmartSocketsWP-1.html`, 1998.

[55] Universal Plug and Play: Background. `http://www.upnp.com/resources/UPnPbkgnd.htm`, 1999.

[56] 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.

[57] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol*, June 1997. RFC 2165 (`http://www.ietf.org/rfc/rfc2165.txt`).

[58] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proc. 17th SOSP*, pages 64–79, December 1999.

[59] Y. B. Zhao. XSet. `http://www.cs.berkeley.edu/~ravenben/xset/`.