UIA: A Global Connectivity Architecture for Mobile Personal Devices

by

Bryan Alexander Ford

B.Sc. Computer Science
University of Utah, 1998

M.Sc. Computer Science and Engineering
Massachusetts Institute of Technology, 2002

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

Author . . . .
Department of Electrical Engineering and Computer Science
August 29, 2008

Certified by . . . .
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

UIA: A Global Connectivity Architecture for Mobile Personal Devices

by

Bryan Alexander Ford

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2008, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

The Internet's architecture, designed in the days of large, stationary computers tended by technically savvy and accountable administrators, fails to meet the demands of the emerging ubiquitous computing era. Nontechnical users now routinely own multiple personal devices, many of them mobile, and need to share information securely among them using interactive, delay-sensitive applications.

Unmanaged Internet Architecture (UIA) is a novel, incrementally deployable network architecture for modern personal devices, which reconsiders three architectural cornerstones: naming, routing, and transport. UIA augments the Internet's global name system with a *personal name system*, enabling users to build personal administrative groups easily and intuitively, to establish secure bindings between his devices and with other users' devices, and to name his devices and his friends much like using a cell phone's address book. To connect personal devices reliably, even while mobile, behind NATs or firewalls, or connected via isolated ad hoc networks, UIA gives each device a persistent, location-independent *identity*, and builds an *overlay routing service* atop IP to resolve and route among these identities. Finally, to support today's interactive applications built using concurrent transactions and delay-sensitive media streams, UIA introduces a new *structured stream* transport abstraction, which solves the efficiency and responsiveness problems of TCP streams and the functionality limitations of UDP datagrams.

Preliminary protocol designs and implementations demonstrate UIA's features and benefits. A personal naming prototype supports easy and portable group management, allowing use of personal names alongside global names in unmodified Internet applications. A prototype overlay router leverages the naming layer's social network to provide efficient ad hoc connectivity in restricted but important common-case scenarios. Simulations of more general routing protocols—one inspired by distributed hash tables, one based on recent compact routing theory—explore promising generalizations to UIA's overlay routing. A library-based prototype of UIA's structured stream transport enables incremental deployment in either OS infrastructure or applications, and demonstrates the responsiveness benefits of the new transport abstraction via dynamic prioritization of interactive web downloads. Finally, an exposition and experimental evaluation of NAT traversal techniques provides insight into routing optimizations useful in UIA and elsewhere.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgments

The design and implementation of UIA was a collaborative effort involving essential contributions from many people.

My MIT colleagues Jacob Strauss, Chris Lesniewski-Laas, and Sean Rhea were responsible for substantial portions of UIA's design and implementation, as detailed below. I can't possibly thank my advisor Frans Kaashoek enough for his constant guidance and invaluable intellectual insight, and for giving me just enough rope to satisfy my ambitions to explore a huge, many-faceted problem space without (quite) managing to hang myself. I also want to thank the other members of my thesis committee, Robert Morris and Hari Balakrishnan, for additional guidance and many stimulating technical discussions.

The UIA project greatly benefitted from a close collaboration with MyNet [127], a sister project at Nokia Research Center Cambridge (NRCC). I particularly wish to thank the MyNet team for believing in UIA enough to take our early, barely-functional code and dare to try building something real with it.

The design of UIA's personal naming system, described in Chapter 2, emerged from extensive brainstorming among the whole UIA team, and much of its prototype implementation is by Jacob Strauss. I would also like to thank Martín Abadi and Tom Rodeheffer at Microsoft Research, and the MyNet Team at NRCC, for extremely helpful feedback on early drafts of our OSDI paper on the naming system [84].

The UIA routing schemes presented in Chapter 3 are similarly collaborative products. Much of the Social Routing design and implementation was by Sean Rhea. The Compact Routing simulation framework was written mostly by Chris Lesniewski-Laas, and its evaluation largely by Chris Lesniewski-Laas and Jacob Strauss.

Chapter 4 on Structured Stream Transport (SST) benefitted greatly from the feedback of Craig Partridge and the anonymous reviewers of my SIGCOMM paper [83].

Chapter 5 on NAT traversal was joint work with Pyda Srisuresh and Dan Kegel. Pyda Srisuresh particularly deserves my gratitude for continuing to push our work towards standardization in the IETF BEHAVE working group [16, 101, 227], long after my attention had drifted to other research topics. I wish to thank Dave Andersen for his crucial support in gathering the results presented in Section 7.6. I also wish to thank Henrik Nordstrom, Christian Huitema, Justin Uberti, Mema Roussopoulos, and the anonymous reviewers of the USENIX paper [82]. Finally, I wish to thank the many volunteers who took the time to run NAT Check on their systems and submit the results.

I would like to thank my wife, Anna Lachowska, and my parents, Robert and Karen Ford, for their unending support and encouragement during these years. I wish to thank the entire PDOS group for creating a lively, supportive, and intellectually stimulating environment. And special thanks to my undergraduate research advisor, Jay Lepreau, for starting me on my current path, and for continuing to help and encourage me long after I'd become Somebody Else's Problem—Jay, you'll always have my best wishes and deepest gratitude.

Funding Attribution

This research was sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan, and by the National Science Foundation under Cooperative Agreement ANI-0225660 (Project IRIS) and FIND project 0627065 (User Information Architecture).

Prior Publication

Portions of this thesis were previously described in the following publications:

- Bryan Ford, *Scalable Internet Routing on Topology-Independent Node Identities*, MIT Laboratory for Computer Science Technical Report MIT-LCS-TR-926, October 2003 [80].
- Bryan Ford, *Unmanaged Internet Protocol: Taming the Edge Network Management Crisis*, 2nd Workshop on Hot Topics in Networks (HotNets-II), Cambridge, MA, November 2003 [81].
- Bryan Ford, *Peer-to-Peer Communication Across Network Address Translators*, USENIX Annual Technical Conference, Anaheim, CA, April 2005 [82].
- Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris, *User-Relative Names for Globally Connected Personal Devices*, 5th International Workshop on Peer-to-Peer Systems (IPTPS '06), Santa Barbara, CA, February 2006 [85].
- Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris, *Persistent Personal Names for Globally Connected Mobile Devices*, 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06), Seattle, WA, November 2006 [84].
- Bryan Ford, *Structured Streams: a New Transport Abstraction*, ACM SIGCOMM '07, Kyoto, Japan, August 2007 [83].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The original Internet architecture was designed to connect large mainframe computers, which were tended by trustworthy experts and whose primary communication applications were text-oriented messaging, remote terminal login, and bulk data transfer. The Internet has fundamentally changed since then, particularly since its commercialization: not only in scale but in many qualitative respects. Today anyone can own their own computer and connect it to the Internet, so the majority of Internet devices are now owned and managed by end users with minimal technical knowledge and limited accountability for their actions. Many devices are portable and move frequently with their users, such as laptops, personal digital assistants (PDAs), and smart cell phones. The application profile has changed as well: modern users are just as likely to use the Internet for streaming voice or video communication, or transaction-oriented web browsing or remote file system access, as for traditional text-oriented messaging and bulk file transfer. Modern users expect their personal devices to "just work," providing reliable and responsive performance across diverse applications, without their having to understand protocol details such as IP addresses and DNS records, and without necessarily having a clear understanding of how to protect themselves from now-rampant viruses and other malware. But the protocols underlying the Internet's original mainframe-oriented design are not up to this task.

This thesis takes a step toward addressing these problems by rethinking basic elements of the Internet architecture around the needs of modern users and their ubiquitous personal devices, to create *Unmanaged Internet Architecture* (UIA), a novel global connectivity architecture designed to support these personal devices more effectively. "Unmanaged" refers to the primary goal of making devices and protocols "just work" reliably and responsively in a wide variety of contexts and for a wide variety of applications, without requiring configuration or technical knowledge on behalf of the user. UIA is intended to augment the traditional Internet architecture, not replace it, thereby providing a practical and incrementally deployable approach to addressing some of the current architecture's shortcomings.

While many elements of the Internet's architecture ultimately need to be reconsidered around the requirements of personal devices, UIA focuses on three major functional areas: naming, routing, and transport. UIA introduces new networking abstractions and protocols into each of these major

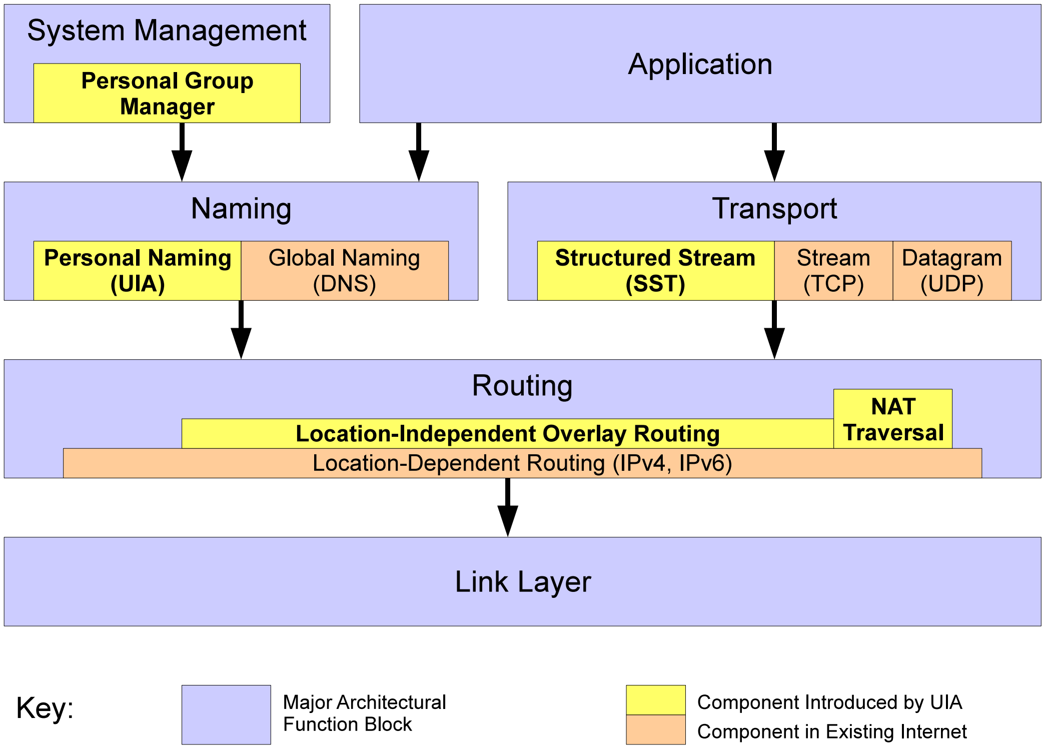


Figure 1-1: Architectural diagram of UIA and how it fits into the Internet architecture

function blocks, as illustrated in Figure 1-1. After first introducing the common theme of decentralization upon which many elements of UIA's design rests, subsequent sections briefly introduce UIA's new architectural components in each of these functional areas. Later thesis chapters will explore these functional areas in detail.

1.1 An Emphasis on Decentralization

A primary objective of UIA and a common theme throughout its design is the goal of building a *fully decentralized* architecture, in which personal devices do not inherently have to rely on the existence or accessibility of centralized services in order to communicate and function properly. This focus on decentralization has three primary motivations.

- **Management Autonomy:** First, a centralized connectivity architecture forces the user to register with or subscribe to some service (e.g., allocate a domain name and static IP address), talk to an expert who can set things up for him, or perhaps even deploy such a service himself (e.g., set up a Mobile IP [183] home server), before his devices will be able to communicate with each other properly. UIA's philosophy is that users should not be forced *by the architecture* to wade through bureaucratic "red tape" of one kind or another before their own devices will be able to communicate with each other in natural peer-to-peer fashion. A decentralized architecture can still allow the user to associate with and make use of centralized services—

indeed, for many high-level services such as airline reservations, a centralized architecture is indeed most natural—but a decentralized connectivity architecture makes such registration optional and not assumed, enabling the user to communicate among his personal devices even without having made some or any of them known to central services.

- **Availability:** Second, global Internet connectivity is not always available from every location we may need to use our personal devices, or equally fast or inexpensive even when it is available. UIA's philosophy is that two users sitting next to each other on a bus or plane without Internet access should still be able to communicate and work together via their devices over an ad hoc wireless LAN, without their connectivity or naming regime breaking or substantially changing just because they temporarily lack access to central Internet services. Even if "always-available" wireless Internet services eventually reach every corner of the planet, an ability to communicate and operate within a local area independently of central services may still be important whenever global communication is substantially slower or more expensive than local-area communication. Further, during times of crisis or natural disaster, a whole geographic region might be temporarily disconnected from the rest of the Internet, and it is at such times that personal device users within such a region are *most* in need of their devices to continue to function and communicate, to whatever extent is theoretically possible under the circumstances.

- **Privacy:** Third, a communication system that relies on centralized services unavoidably "leaks" information to the operators of those services, which can create security and privacy concerns for both individual and corporate users. Even a fully decentralized system cannot necessarily prevent such information leakage entirely, since global communication traffic still has to flow through untrusted intermediaries such as network providers and ISPs, and even encrypted traffic may be susceptible to traffic analysis. A fully decentralized design nevertheless minimizes the amount of information that needs to be explicitly exposed to third parties merely in order for the communication system to function.

Some parts of UIA's design could be made substantially simpler by giving up this emphasis on decentralization and making personal devices rely on central "authoritative" servers; these potential simplifications will be pointed out at the appropriate places.

1.2 Naming

In order to connect to remote devices, services, and users over the Internet, users first need to be able to *name* those remote entities in order to identify the desired communication target(s). There are many possible approaches to naming, many involving tradeoffs between different goals such as ease of use, name conciseness, portability, and implementation complexity. A crucial issue in any naming system is how names and namespaces are *managed*; different approaches to naming often imply different paradigms for name management. The Internet's traditional naming paradigm

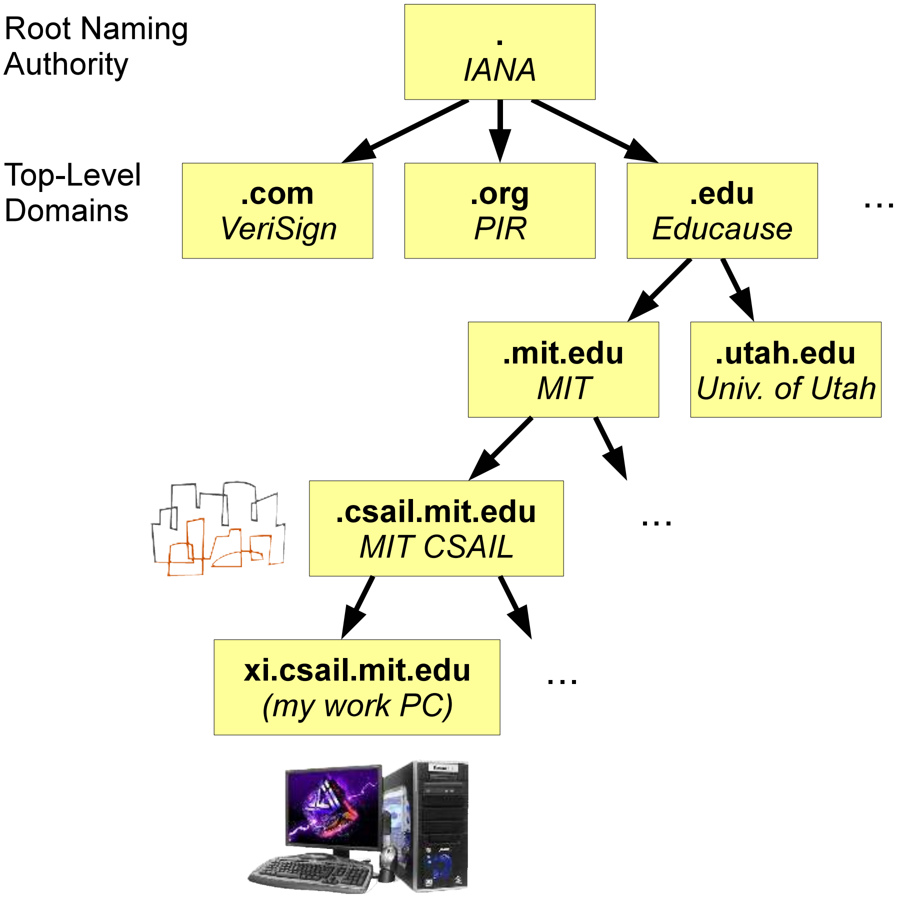


Figure 1-2: The Internet's traditional global naming hierarchy

is built around *global names*, but UIA enhances the Internet to support a new and complementary naming paradigm called *personal names*.

1.2.1 Global Names

The Internet's existing domain name system (DNS) [256] provides globally unique names based on a hierarchy of name registration services. The Internet Assigned Numbers Authority (IANA) delegates each *top-level domain* such as `.com` and `.edu` to a specific organization, each of which manages the allocation of sub-domains such as `meraki.com` or `mit.edu` according to various administrative policies. These authorities in turn delegate registration authority for sub-domains to the organizations that own them, and so on. Figure 1-2 illustrates this traditional global naming hierarchy.

To name a particular device in this scheme, so that the device can be accessed remotely from anywhere on the Internet, the device's owner must *register* the device: i.e., find a unique, not-yet-allocated name in some domain within this administrative hierarchy, and follow appropriate registration procedures defined by the owner of that domain. For example, to make my work PC globally reachable via the name `xi.csail.mit.edu`, I had to register the name `xi` with my work organization, CSAIL, which manages the domain .csail.mit.edu, and derives its authority to manage that domain from MIT (the authority for .mit.edu), which in turn derives from the top-level registrar for the .edu domain and ultimately from IANA. This naming architecture implies that names are *globally unique*, meaning that they can in theory be used from any device on the Internet and will always refer or *resolve* to the same target, but it also means that these names must be *administratively registered* in order to enforce this global uniqueness: i.e., CSAIL's registration process is responsible for ensuring that only one device can be named xi.csail.mit.edu at once. Ultimately, what this means to ordinary Internet users is that when naming their devices with DNS, they can use only names that are *not already taken* (and in these post-Internet-land-rush days most of the "good" ones are already taken), and users must obtain (often purchase) the use of those names from some registration authority in the DNS hierarchy.

Global uniqueness of names is a valuable property for certain types of names to have: particularly names corresponding to global brands or other trademarks such as amazon.com. As companies have realized since early in the ".com boom," having a unique, memorable "online brand name" such as amazon.com, which people can type into any web browser to reach the company's web site, is crucial to E-commerce and to "online presence" in general. And companies are willing to pay far more than the nominal registration fees to obtain "premium" names corresponding to recognizable words or phrases [236].

1.2.2 Personal Names

But when a user wishes to connect remotely not to a global online brand, but rather to one of his own personal devices—or to that of a friend—global uniqueness is less important than simplicity and convenience. For example, a casual tourist may wish to upload photos from his WiFi-enabled digital camera directly to his home PC or to a family member's PDA, and a business traveller may need remote access from her laptop to files on her work PC or a colleague's. In this situation, it is common for the device *initiating* the communication to be owned by or otherwise administratively associated with the user, which means that the initiating device's own identity can provide context for naming the remote device to be accessed. For example, if Alice has only one work PC, and she wishes to access it from *her own account* on *her own laptop or home PC*, it would be more convenient for her to be able to use a shorter, simpler name like work-pc instead of work-pc.alice.csail.mit.edu. Even though work-pc is probably not globally unique (there are many people in the world who have work PCs that they may wish to name work-pc), that may be fine if *Alice* has only one device named work-pc and the device initiating the connection knows that it is Alice who is requesting the connection. We refer to such a non-globally-unique name as a *personal name*, since its interpretation context is *personal* to the user.

A useful analogy is the address book on a modern cell phone, which allows the phone's owner to use short, convenient names like Bob and Mom instead of cryptic, globally unique phone numbers to call people the phone's owner dials regularly. Indeed, we might consider such an address book to be a *personal naming system*. The personal naming system that traditional address books provide are not adequate for personal Internet devices, however. Users increasingly have many more Internet-connected devices than they have cellular accounts or phone numbers, and separately managing address books on each of those devices quickly becomes cumbersome and impractical. Current address book synchronization software typically either operates between only a specific pair of personal devices—e.g., between *one* smart phone and *one* laptop or desktop PC, leaving other personal devices' address books unsynchronized—or it operates between one or more personal devices and a central server deployed by a device manufacturer or service provider, in which case this synchronization and personal naming is unavailable whenever devices are disconnected from the Internet or from their designated cellular network. To meet the demands of modern Internet users, personal devices must not only support convenient personal names, but must be able to keep them synchronized automatically across an arbitrary number of personal devices, and those names must be usable even in partial connectivity situations: e.g., when Bob and Alice are travelling together in a bus or plane and need to share information between their devices while they have no Internet access or cellular coverage.

A basic assumption underlying the idea of personal names is that the devices through which the user typically works or accesses the network—i.e., the *terminals* responsible for interpreting these personal names—are either owned by or somehow affiliated with the user, and thus have some *personal user context* in which to interpret these personal names. This assumption fails in the case of public-access terminals, such as those found in an airport or Internet café: since a public terminal has no prior affiliation with the user, it typically has no context in which to resolve personal names, so only global names are directly usable at such terminals. (A user could still use a public terminal to log into a remote machine or account via a global name, and from there use personal names to reach his other devices or services.) Given the trends of personal Internet devices becoming increasing cheap and readily available [151, 202], the ubiquitous availability of WiFi hotspots [1] and mobile broadband [41] for use with consumer-owned devices, and the well-known security risks inherent in the use of public-access terminals [216, 254], it appears safe to assume that private devices rather than public terminals will remain the primary means by which most users access the network, satisfying the main requirement for personal names to be useful.

1.2.3 UIA Personal Names and Personal Groups

The Unmanaged Internet Architecture is not designed to replace DNS, the Internet's existing global naming system, but rather to augment it with a personal naming system that can work alongside DNS. With UIA, the user can still use global names like `amazon.com`, but can also use personal names like `laptop` or `pda.alice` (meaning "Alice's PDA").

A key technical challenge in create such a personal name system is defining the context or *scope* in which these personal names are to be resolved: this context should be larger than a single device, so that the user does not have to administer one address book for each device he owns or operates, but it should be small enough that the user does not have to worry about his names conflicting with those assigned by other users. That is, the personal naming context needs to correspond to

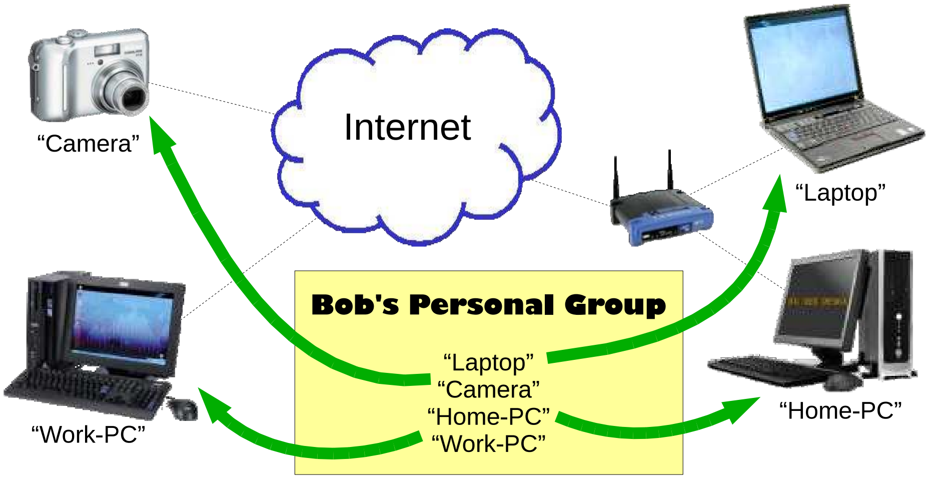


Figure 1-3: An example UIA personal group containing four devices, each with a personal name

the user's personal *administrative domain*: the complete set of personal devices (or accounts on shared devices or services) that the user manages. For this purpose, UIA introduces the concept of *personal groups*, which delineate a symmetric group of devices or accounts in which to scope the user's naming activities. Figure 1-3 illustrates a personal group containing four devices, which may be scattered across the Internet, each having its own personal name that may be used from any other device in the group.

It is crucial for personal groups to be both secure and easy to set up and manage. Since we expect users may wish to share sensitive personal information among their devices via UIA personal names, it is important that attackers not be able to "spoof" a user's device, causing the user to log in to or leak information to the wrong device. Similarly, an attacker must not be able to add devices to a user's personal group or modify the name bindings in the group without the user's permission, and if the user retires, sells, or loses a personal device, the user must be able to revoke that device's membership in the group so that its new owner does not obtain private information about or access to the user's personal group. But the user needs an easy way to set up and manage his personal group while satisfying all of these security requirements.

For this purpose UIA builds on the idea of *device introduction* [65, 231]. Users build their personal groups by *introducing* them, typically "in person" over a local-area network; the introduction process includes a simple random phrase comparison procedure that allows the devices to exchange cryptographic keys securely while ensuring that the devices involved in the procedure are in fact the ones the user(s) believe to be involved. One the user has introduced more than one device into a group, UIA uses optimistic replication [105, 133, 243] to keep the user's personal namespace synchronized among all the devices in the group. UIA personal groups can name not only devices but other personal groups: thus, if Bob's personal group has a name `alice` referring to Alice's personal group, and Alice's personal group includes a device named `pda`, then Bob can refer to Alice's PDA as `pda.alice`.

Group membership revocation presents a particular challenge in UIA, because there may not be a clear central point of control, such as a "master device" or an online service, which can always uniquely determine the set of members of a group at a particular point in time. If one or more of Alice's personal devices are stolen, for example, then she needs to revoke the lost devices' membership in her group so that the thief will not be able to use those devices to authenticate as her or remotely access her personal information, and so that Bob's name `alice` will henceforth refer only to Alice's remaining devices and not to the stolen ones. But whatever revocation mechanism Alice may invoke on her remaining devices, the thief could also invoke that mechanism on the stolen devices (assuming the thief breaks whatever local login/account security those devices may provide), in order to try to steal the identity of Alice's group and make Bob's name `alice` refer to the thief's stolen devices instead of Alice's remaining ones. Although a variety of approaches to this revocation and identity management problem are possible, the prototype UIA design explores a highly decentralized approach in which all personal devices in a group are considered "equals" and irreconcilable "revocation dispute" must in the worst case be resolved by manual re-introduction.

1.2.4 Centralized Designs for Personal Groups and Personal Naming

UIA's personal group management mechanism could be substantially simplified by de-emphasizing the goal of full decentralization and making devices reliant on a central service of some kind. Personal devices could for example connect to a user's account on Facebook [76] or another Web-based social networking service, and rely on that service to define the user's identity and his namespace of personal devices and other users. The client-side software for such a mechanism would probably be substantially simpler than UIA's because it could always rely on the central Facebook servers to maintain the "authoritative" state about the user's identity and personal names in one central location. The issues of conflicting namespace changes and revocation disputes would disappear since there would be only one "master" state repository.

As pointed out already in Section 1.1, however, such a design would require the user to register with Facebook before his devices can communicate with each other at all, it would render the personal naming system inoperative whenever the central Facebook service is inaccessible, and it would create security and privacy concerns by forcing users to disclose complete information about their personal administrative domains to Facebook and anyone else with (legitimate or otherwise) access to Facebook's servers. And replacing the risk of revocation disputes in UIA's decentralized design, the centralized design would create the danger of the user's Facebook account being hacked and the user's entire personal administrative domain being effectively stolen all at once, leaving the user powerless to recover or even safely disable his identity until he can successfully navigate Facebook's customer support system and somehow "prove" his rightful ownership of the account.

1.3 Routing

The Internet's original routing architecture was designed around the assumption that each Internet host would be given a unique *IP address* within a single global address space [91, 120, 199], and that every host would be able to communicate with every other. A number of evolutionary pressures have since compromised these assumptions, however.

1.3.1 Eroding Any-to-Any Connectivity

Since all Internet hosts are no longer run by trustworthy (or at least accountable) system administrators, and security systems built into end hosts are frequently inadequate, users and organizations have pervasively deployed firewalls [49,88,108,170] to protect private networks from unwanted traffic originating from external sources. While specific firewall policies vary widely, the most common *default policy*, which typically applies to all traffic in the absence of application-specific rules, is to deny "unsolicited" incoming connections originating from the external side of the firewall, while allowing outgoing connections originating from within the private network. While this default policy has the benefit of protecting the internal network from a wide variety of remote packet-level attacks while allowing typical client/server applications to operate normally—at least when the client is on the private network and the server is on the public network—it prevents users from remotely accessing even their own devices (e.g., a work PC) located behind such a firewall, without making special administrative provisions.

Worse, the gradual depletion of the 32-bit address space of IPv4 [120], coupled with the administrative challenges and costs of allocating and managing IP subnets within the global address space, have led to pervasive deployment of network address translators (NATs) [70, 112, 226, 228], which create private IP address spaces [110, 196] for corporate or home networks, within which hosts may be assigned IP addresses independently of the Internet's "official" hierarchical of IP address allocation scheme [91, 197, 199]. NATs typically multiplex many private IP addresses onto one or a small number of public IP addresses, using the IP addresses and TCP/UDP port numbers of individual sessions to determine how to translate the addresses in traffic flowing in both directions across the NAT. Whereas the "deny unsolicited incoming connections" rule for firewalls is an administrative choice (albeit a particularly common one), for NATs this rule is *technically essential*: it is a side-effect of the fact that the NAT simply does not know to what internal host it should route incoming packets unless a *binding* for the flow was previously established by earlier packets flowing from the appropriate internal host to the external network.

IP version 6 [62] may eventually eliminate one motivation for NAT deployment—namely IP address space depletion—but it does not eliminate other incentives such as the administrative convenience of managing private IP address spaces independently of upstream Internet providers, and in the near term it even creates the additional incentive of providing convenient interoperability between IPv4 and IPv6 address domains [106, 249]. And firewalls will remain commonplace in the IPv6 world even if NATs eventually are not.

1.3.2 Host Mobility and IP Address Instability

Independently of firewalls and NATs, the fact that an increasing percentage of Internet-attached hosts are small, *mobile* devices such as laptops and PDAs creates another complication not addressed by the Internet's original design: Internet routing protocols depend on the structure of IP addresses [199] for efficient routing [91], which means that IP addresses must be assigned according to the topological network location at which the device is attached at a given time. Unfortunately this means that whenever the device moves, its address changes: if a DNS record associates a host name with a particular IP address, that host name will no longer work if the host moves to a different network attachment point and changes its address.

Further, IP address autoconfiguration mechanisms such as Dynamic Host Configuration Protocol (DHCP) [5, 68], intended to simplify the administration of IP address spaces, make IP addresses even less stable, in that a host may change IP addresses merely by virtue of being turned off or disconnected from the network for some time period even if it hasn't physically moved. Since transport and application protocols typically rely on IP addresses to define end-to-end communication endpoints, not only does IP address instability complicate device naming, but already-established transport connections and application sessions generally fail whenever a host's IP address changes for any reason.

Various partial solutions to these problems have been developed. Dynamic DNS [256] enables a host to change its name record whenever it changes IP addresses, but existing transport sessions still break at this point, and the host generally remains unreachable whenever it is behind a firewall or NAT. Mobile IP [183] gives each mobile host a static "home" IP address that can tunnel traffic to the host's actual, dynamic location: this solution can traverse firewalls and NATs, and can preserve transport sessions across location changes. Mobile IP's approach has several downsides, however: setting up the host's "home" IP address and the tunneling association requires considerable manual setup, permanently stable IP addresses to act as these "home" locations are increasingly scarce and costly, the tunneling indirection adds communication latency to all traffic and increases load on the host's home network whenever the device is roaming, and the host is unreachable from anywhere if its home network or tunneling server become unavailable. The Host Identity Protocol (HIP) [163] and shim6 [218] address mobility by introducing a layer between the network and transport layers that translates between the network layer's hierarchically structured, location-dependent *locators* and flat, location-independent *host identities*, but neither HIP or shim6 have defined general mechanisms to resolve and route on these location-independent identities.

1.3.3 UIA Routing

The Unmanaged Internet Architecture provides an *overlay routing layer* that operates on top of the Internet's existing routing layer to provide robust peer-to-peer connectivity between personal devices even when those devices are mobile and/or behind firewalls or NATs, as illustrated in Figure 1-4. UIA's routing layer generalizes HIP's notion of location-independent *host identities* into

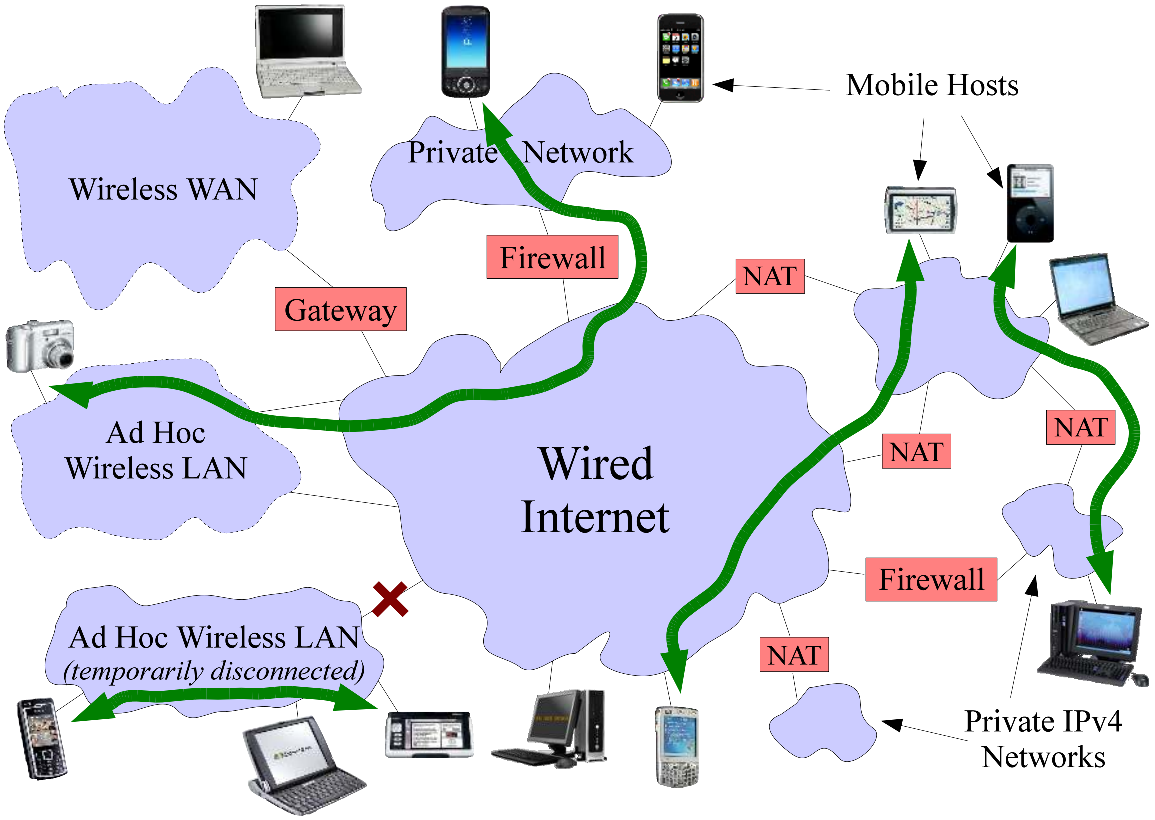


Figure 1-4: Global connectivity challenges for the UIA overlay routing layer

a notion of location-independent *endpoint identities* (EIDs) that can refer not only to hosts but to individual users or user accounts on a particular host, so each user of a shared host can have a separate EID with independent application endpoint and service namespaces. The UIA routing layer furthermore includes a self-organizing distributed protocol allowing cooperating hosts to search for and connect with arbitrary EIDs on demand, even when the target EID is on a mobile and/or NATted host.

There are many possible approaches to designing a specific routing protocol to serve as UIA's overlay routing layer, and the best approach is not yet clear since the design space is large and involves a wide variety of pragmatic constraints and tradeoffs. We therefore explore three different approaches to UIA routing: a simple but easily deployed routing protocol optimized for the "common case" of communication within a social network; and two more general routing schemes, based on distributed hash tables and compact routing theory, respectively.

Social Routing

UIA's *social routing* protocol leverages the user's social network, as defined by the UIA naming layer, to optimize for the common case in which the user communicates most frequently with friends and colleagues. Each UIA host tracks and regularly communicates with some number of other hosts in order to maintain the overlay routing structure, and prioritizes tracked hosts based on social proximity or *friendship distance*, so that peer-to-peer connections to primary communication partners will succeed as quickly and reliably as possible. Leveraging the user's social network in this way also takes advantage of the implicit trust and incentive structure provided by the social network: users are more likely to want to permit their hosts to serve as rendezvous points or forwarding intermediaries for their trusted friends than for random strangers.

In the basic, currently deployed version of the UIA routing protocol, a host finds a given target EID by *token-limited flooding*. The search initiator first allocates some number of *tokens* to the search, which constrains the total amount of network bandwidth (in terms of number of messages) that can be consumed by the search, and divides these tokens up among the immediate friends (social neighbors) it can contact. The initiator forwards a search request to each of these friends with an indicator of the appropriate token count; each of these friends responds directly if has information about the current location of the target EID, otherwise it further subdivides the tokens it received among *its* friends and re-broadcasts the request to those friends, and so on. The original initiator uses a small token count for the first iteration of the search, but increases the token count for subsequent search attempts if earlier attempts fail. Thus, successful searches for EIDs close to the initiator in the social graph tend to complete very quickly and inexpensively, while searches for EIDs farther away are more expensive but still have a chance of completing.

Identity Hash Routing

A second overlay routing scheme for UIA, *identity hash routing*, builds on distributed hash table (DHT) concepts [146, 148, 156, 194, 209, 234, 271]. Each node uses its pseudorandom EID to determine its logical position within a distributed structure, and registers itself with other nodes having high *identifier affinity* with its own EID (i.e., some number of bits of common identifier prefix), so that searches for a given EID can be guaranteed to succeed in a correct, fully-functioning n-node network after contacting at most $O(log\ n)$ identifier lookup stages.

While classic DHTs only provide a scalable *lookup* service and assume the underlying network provides full any-to-any connectivity, UIA's scalable routing protocol makes the more pragmatic assumption that some pairs of nodes can communicate directly via IP routing and others cannot, or require NAT traversal techniques [30, 82, 99, 100] involving an intermediary rendezvous node in order to do so. To handle such asymmetries in connectivity, UIA's scalable routing protocol allows DHT routing table entries to contain not just "direct" pointers (i.e., IP addresses) but also *indirect* pointers representing either "flat" source routes or *virtual links* that build hierarchically on combinations of other routing table entries.

Compact Routing

The third overlay routing scheme explored here is based on recent theoretical results in the area of *compact routing* [2, 15, 56, 245]: schemes that use routing tables provably sublinear in the number of nodes, while provably maintaining a bound on *stretch*, or the inefficiency of chosen paths with respect to a non-compact shortest-path scheme. Compact routing schemes present a promising basis for UIA routing because of their formal scalability and stretch properties, and because they work between arbitrary nodes on arbitrary topologies. Unfortunately, these formal schemes have so far been developed only as centralized *algorithms* to compute a set of routing tables from a network graph description; converting these algorithms into practical distributed *protocols* remains to be done. Also, compactness and stretch—the properties of primary focus in the compact routing literature—are only two of the many properties of a routing scheme that are important in a practical system; other properties, such as congestion caused by the route selection mechanism, robustness to failure, and the messaging cost of maintaining routing tables, have been largely ignored so far.

We therefore explore techniques and challenges for converting one particular compact routing scheme—Thorup and Zwick's variable k-level scheme [245]—into an overlay routing protocol for UIA. One major issue with this Thorup/Zwick scheme is a tendency to "focus" global traffic through a few central, randomly chosen "landmark" nodes, which can create congestion bottlenecks: we explore ways to modify the routing scheme to mitigate this bottleneck effect while preserving the scheme's formal compactness and stretch bounds. To better understand the scheme's viability for use in large-scale UIA overlay networks, we also examine the scheme's behavior in terms of stretch, congestion, and robustness to failure, on a variety of simulated network topologies up to 192,000 nodes in size.

1.3.4 Centralized Alternatives to Overlay Routing

As with the naming layer, reliance on centralized services represents a seductively simple but potentially problematic alternative design for overlay routing. Personal devices could simply connect to a cluster of well-known rendezvous and forwarding servers operated by the device or software vendor, keeping a communication session open with at least one such server to receive notifications of incoming connection attempts even when the device is behind a restrictive NAT or firewall, and forward traffic through the central server whenever direct peer-to-peer connection attempts fail. Indeed, such a centralized routing/rendezvous service for peer-to-peer communication between personal devices is the basis of Apple's MobileMe [13] and Microsoft's Live Mesh [160] services.

While the reliance of centralized services such as these would indeed simplify the client-side part of UIA's routing layer, doing so would bring the usual attendant issues pointed out in Section 1.1. First, the user would have to register with the central service before his devices could communicate at all. Second, devices would be unable to communicate with each other when disconnected from the Internet, or while part of a temporarily disconnected fragment of the Internet such as a disaster zone. Third, since all the user's personal devices would have to maintain constant connectivity with the central service in order to maintain global reachability, the central service would always have to "know" the user's location (or at least network attachment point) at any given moment, which may be a significant privacy concern for his users.

1.4 Transport

Once devices have been given usable names and a mechanism is available to find and establish contact with named hosts, applications still need a convenient way to exchange both control information and application data between those hosts, in some fashion appropriate to the specific application. In the early days of the Internet, the most common applications were remote text-oriented terminal access via protocols like Telnet [186], Rlogin [128], and eventually the Secure Shell [264–267]; text-oriented messaging systems such as E-mail [58,135,168,201] and USENET [77,114,175]; and bulk data transfer protocols such as FTP [113,187]. For most of these applications, the reliable byte stream abstraction provided by TCP [6, 242] generally proved quite suitable, and for applications like DNS name resolution [162] that needed a lighter-weight transport, UDP [185] was available as an alternative.

1.4.1 The Demands of Modern Interactive Applications

The evolution of modern personal Internet devices has placed many new demands on application protocols, however, with which existing Internet transport protocols have not adequately kept up. In particular, modern applications are frequently much more delay-sensitive than traditional text-oriented or bulk transfer applications. In multimedia streaming applications [214, 215], unexpected delay translates to user-visible glitches, quality degradation, or temporary interruptions in audio/video service. Interactive applications such as web browsing and remote file system access rely on transaction-oriented protocols such as HTTP [78] and NFS [217], in which the delay of a transaction's completion is often easily perceptible to the user as an annoying "freeze" in the system's normal operation. Wireless broadband technologies are increasingly used for Internet connectivity and not just voice communication [41], but the high latency and/or lossiness of these networks exacerbate the challenges faced by modern delay-sensitive applications [7,17,21,23,179,263]. Furthermore, modern interactive applications frequently need to perform many independent transactions concurrently, such as downloading the many individual pictures and other embedded objects on a web page; the common practice of multiplexing these many transactions onto one TCP stream, as done in HTTP [78], IMAP [58], or iSCSI [212], for example, causes a delay in processing one transaction to block all the other transactions queued up behind it in the stream [51, 213, 262], unnecessarily turning a minor freeze in one transaction into a much more obvious and annoying freeze affecting many transactions or the whole application.

TCP has seen numerous extensions over the years to improve its performance in various ways, such as large windows and timestamps [122], rapid connection renewal for transactions [36], selective acknowledgment [154], and duplicate acknowledgment [33, 79], but these partial solutions do not solve the fundamental limitation of TCP's byte-stream abstraction: that it unnecessarily imposes a total order on all communication even when individual transactions or "application-layer frames" [51] are naturally order-independent.

Other transports, such as UDP [185], RDP [177, 178, 255], SCTP [232, 233], UDP-Lite [142], and DCCP [136, 137], forsake the convenience, generality, and conceptual simplicity of TCP's stream abstraction for a transport abstraction based on limited-size *datagrams*. All of these transports force the application into building atop a different and more restrictive abstraction, in which the application is responsible for breaking up all its logical transactions or application-layer frames into reasonably small (datagram-size) units, and (in the case of UDP-Lite and DCCP) for handling any necessary reliability provisions. While SCTP can handle multiple concurrent "streams," these are streams of *datagrams* rather than streams of bytes, and SCTP uses datagrams as its atomic units for reordering and traffic scheduling purposes: thus, if an application transmits a datagram that is too large, the transmission of that datagram will "block" the entire SCTP connection until its entire content is completely transmitted and acknowledged, before datagrams from other (perhaps higher-priority) streams may be sent on the connection.

1.4.2 Structured Streams in UIA

UIA takes a different approach to supporting the modern applications prevalent on today's personal devices. Instead of forsaking or drastically changing TCP's byte stream abstraction, UIA's *structured stream transport* (SST) merely attempts to make classic byte streams easily and efficiently usable at finer (but completely application-defined) granularity. Thus, SST enables transaction-oriented interactive applications like HTTP, IMAP, and NFS to create a brand-new byte stream for *every transaction*. Similarly, a media streaming application using SST can place each media frame in a separate *ephemeral stream*. Since SST preserves TCP's byte stream semantics, each of these streams provides strict data ordering *within* a transaction or frame, while allowing transactions or frames to be of any size ranging from a few bytes to long-running bulk transfers of megabytes or gigabytes of data. Since each transaction or frame is in a separate stream, however, different transactions or frames are ordered and delivered fully independently of each other: thus, a temporary delay in one transaction (caused for example by a server-side processing delay or a packet loss in the network) need not prevent other ongoing transactions from completing promptly.

To make it easy for applications to create and manage the many per-transaction or per-frame streams they will require in order to take advantage of this natural transaction/frame independence, SST augments the traditional stream API with a "fork" operation, which allows the application to create a new stream from an existing stream at any time. A new stream created this way is considered a "child" of the existing stream, and child streams can likewise be forked, giving SST streams a *hereditary structure*—hence the name "structured streams." A child stream is semantically independent from its parent, in that its data transmission is unordered with respect to data in its parent stream, and child streams can even outlive their parent streams. Nevertheless, SST maintains this hereditary relationship during the process of initiating a new stream, so that the host accepting the new stream has a suitable context (namely the parent stream) in which to interpret the meaning and purpose of the new child stream. Structured streams make general "out-of-band" communication very simple, for example: if two hosts are in the process of transferring data related to one trans-

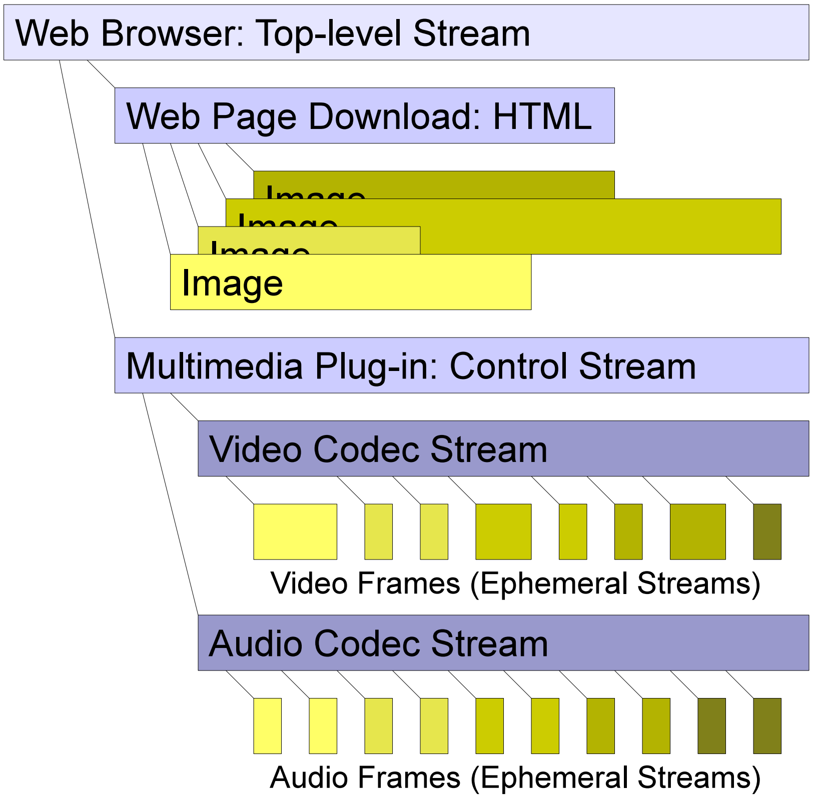

Web Browser: Top-level Stream
Web Page Download: HTML
Image
Image
Image
Image
Multimedia Plug-in: Control Stream
Video Codec Stream
Video Frames (Ephemeral Streams)
Audio Codec Stream
Audio Frames (Ephemeral Streams)

Figure 1-5: Example structured stream hierarchy in SST

action, such as a long HTTP download, they can fork the stream on which that long transaction is occurring in order to communicate out-of-band messages relating to that transaction, such as status updates, priority changes, or graceful cancellations. Figure 1-5 illustrates an example structured stream hierarchy that an application such as a web browser might create.

The SST protocol is itself broken into two layers: a *channel layer* that provides transport services that are shareable among many application streams, such as sequencing, acknowledgments, congestion control, authentication, and encryption; and a *stream layer* that implements the structured stream abstraction itself atop the channel layer's services. The channel layer's sequencing and acknowledgment services enable the the stream layer to create and destroy streams with very little per-stream state or on-the-wire overhead, avoiding TCP's 3-way handshake delay on stream creation and its four-minute TIME-WAIT state retention period after stream close, for example. SST's stream protocol also subsumes the functionality of datagram-oriented transports such as UDP [185], RDP [177, 178], and DCCP [136, 137] by providing applications the ability to send small *ephemeral streams* efficiently with optional "fire-and-forget" (unreliable) delivery semantics.

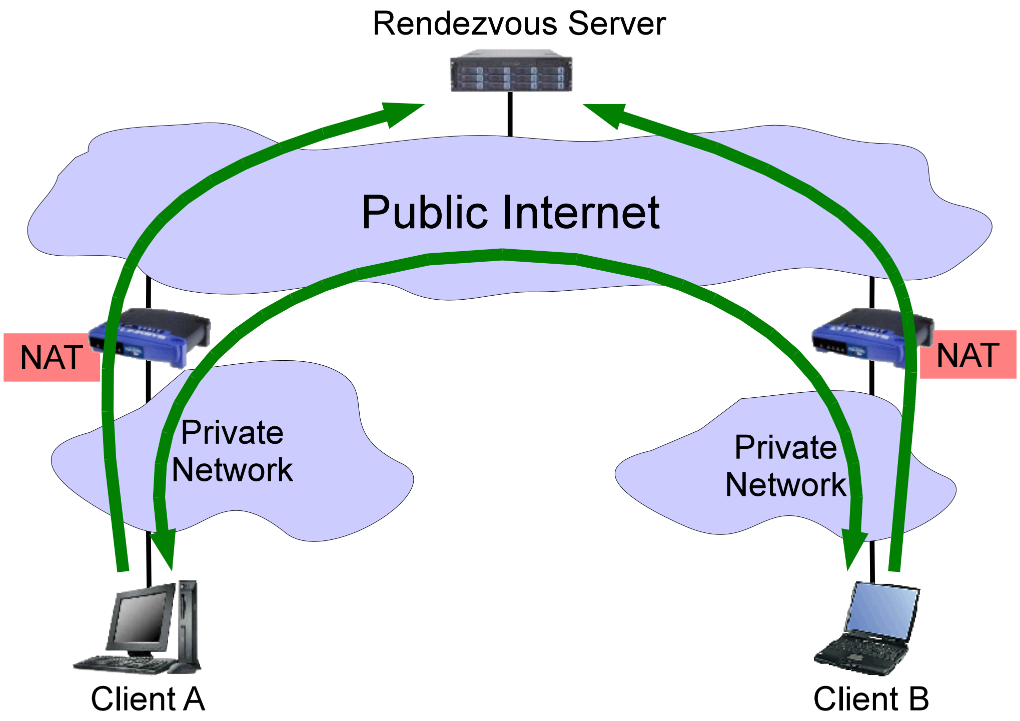


Figure 1-6: Implicit NAT traversal via "hole punching"

1.5 NAT Traversal

Personal mobile devices are often be located behind corporate or home NATs and firewalls, and although the UIA overlay routing layer inherently supports communication between such devices by forwarding traffic through some intermediate node on the public network, this UIA-level forwarding has costs in terms of both network delay and load on these intermediate nodes. The costs of UIA-level forwarding can sometimes be avoided through *NAT traversal* techniques, in which an intermediate node helps to establish a direct IP-layer path between two nodes that may both be located behind NATs. Since NAT traversal is useful both as an optimization within the UIA overlay routing layer, and to standalone non-UIA-based Internet applications, we explore the design space, behavior, and reliability of NAT traversal independently of a particular routing protocol.

NAT traversal can be implemented using either *explicit* or *implicit* communication with NATs and firewalls on the path. Protocols supporting explicit NAT traversal, such as UPnP [253] and NAT-PMP [48], provide the greatest flexibility and control, but are supported only on some NATs and generally do not address increasingly-common scenarios involving multiple levels of NAT. We therefore focus on implicit NAT traversal techniques, in particular a technique known as *hole punching*. With hole punching, illustrated in Figure 1-6, two "client" devices both behind NATs or firewalls can collaborate using a public *rendezvous server* on the Internet to open ("punch") a pair of forwarding sessions ("holes") by sending outgoing traffic through their respective NATs, to form an efficient peer-to-peer data path that thereafter requires no further traffic forwarding or other assistance from the rendezvous server.

Hole punching for UDP-based applications has been informally known and in use for a while, but hole punching for TCP connections is more challenging and less well-understood, and scant prior analysis or experimental research has been done on the generality and robustness of hole punching in the context of either transport. This thesis attempts to address this knowledge gap by laying out hole punching techniques that are robust and efficient as long as NATs in the path satisfy certain common and generally reasonable requirements, by exploring and identifying solutions for the problem of NAT traversal over multiple levels of NAT, and by experimentally evaluating the applicability of these techniques to the NATs commonly found "in the wild" on the Internet.

1.6 Prototyping UIA

Like any general systems architecture, there are many possible ways to implement UIA, different approaches necessarily offering different tradeoffs. One particularly important practical tradeoff for our purposes is driven by the goal of incremental deployment: how can we make UIA useful with as few changes to existing system and application components as possible, and should we prefer changes to the operating system or to applications?

UIA could be implemented as part of the operating system, for example, which might facilitate deployment by making it immediately usable by existing, unmodified applications running on that operating system. Such a UIA implementation may be tied to that particular operating system, however, limiting its portability to other operating systems, and the requirement of installing privileged operating system extensions can be a pragmatic deterrent to use by more casual (or merely unprivileged) users. Alternatively, UIA might be implemented as part of specific applications that wish to use UIA's communication facilities, in which case the UIA implementation need not require special privileges or any explicit installation step separate from that of the application itself, making it easily deployable by casual users. Such an implementation may unfortunately become application-specific and functionally isolated from similar (or even identical) instances of UIA in use by other applications running on the same system, however.

Since there is no clear "best" implementation strategy, we explore a combination of strategies in several working prototypes that demonstrate various elements of UIA both together and separately, and provide an experimental base on which to develop the architecture further. We use a system-level approach to implementing the personal naming and social routing protocols, allowing both new and legacy applications to benefit from UIA's personal naming and overlay routing facilities. We use simulation frameworks to explore identity hash routing and compact routing. Finally, since the transport layer is more closely tied to the application than the naming and routing layers are, we use an application link library approach to prototype UIA's structured stream transport.

1.6.1 System-level Naming and Routing Prototype

The main UIA prototype is designed to be installed on existing operating systems as a new set of system-level services implementing the UIA architecture in a modular fashion. This modular prototype includes a basic social routing layer written in C++ [237], a personal naming layer written in Python [189], a graphical user interface (GUI) written in C++ using the Qt toolkit [248] for managing personal groups and personal names, and system daemons to provide compatibility with unmodified legacy applications. All components of the prototype are designed to run in user space for relatively easy portability to multiple platforms: the prototype currently runs in Linux, Mac OS X, and FreeBSD. The primary advantage of this system-level design is that it allows unmodified legacy applications to use UIA, resolve UIA personal names, and communicate within and between personal groups. The main disadvantage of this system-level approach is that deployment requires a somewhat invasive installation of a set of privileged daemons.

1.6.2 Routing Simulation Frameworks

To provide a basis for developing and analyzing more general overlay routing protocols for UIA, we develop a set of simulation frameworks in C++ and Python. In order to provide insight on the behavior of a routing scheme when scaled to networks containing large numbers of interconnected UIA nodes, these simulation frameworks are designed to support simulation of a very large number of nodes on one physical machine: the C++ framework has been run on simulated topologies up to 192,000 nodes, for example. The downside of this streamlined approach to routing simulation is detail and hence, possibly, fidelity: our simulators do not emulate all aspects of the routing protocols or of realistic network environments, leaving uncertainty about the extent to which these simulations may be relied upon to predict real-world behavior. More accurate (but perhaps necessarily smaller-scale) simulations and real-world deployments of the more general routing schemes developed here remain for future work.

1.6.3 Library-based Structured Stream Transport Prototype

The working prototype of UIA's structured stream transport (SST) is not yet integrated into the system-level UIA system, but instead operates as a Qt-based library intended to be linked directly into applications. This library implementation can be linked into (and distributed with) applications for existing unmodified operating systems, making it immediately and incrementally deployable on a per-application basis. The library provides an API similar to traditional sockets APIs, but uses names that do not conflict with the system's, and does not require any special (e.g., administrator-level) privileges in order to operate.

Although the SST prototype is not yet integrated into the prototype of UIA's naming and routing layers, they can still be used together, since an SST-based application will appear to the underlying OS as a UDP application, and an OS with the UIA naming and routing prototype installed will be able to forward that "UDP" traffic transparently to the application within the user's UIA personal group and social network using UIA personal names. This implementation is far from ideal, however, since the naming and routing prototype's current use of SSL over TCP for authentication and communication security defeats many of the responsiveness advantages that SST could otherwise provide by forcing a total order on all UIA traffic between a given pair of hosts. Integrating SST properly into the main UIA prototype, so that the UIA routing layer uses SST instead of SSL/TCP for its transparent tunneling and forwarding functions, remains for future work.

1.7 Contributions

UIA's primary architectural contribution is a **cross-layer redesign of basic network abstractions and protocols around the requirements of modern personal devices**. Since personal devices have become mainstream commodities in the emerging world of ubiquitous computing, nontechnical users now frequently own several devices, and these devices need to work together to "manage themselves," operating efficiently under a wide variety of network conditions and application requirements without constant help from administrative authorities or technical experts. Although many networking abstractions throughout the entire Internet protocol suite will ultimately need to be reconsidered around these new usability requirements, UIA takes a few steps in this direction by identifying and developing new naming, routing, and transport abstractions that are more suited to modern personal devices than the traditional, "managed" abstractions the Internet evolved with.

UIA demonstrates that rethinking network abstractions across architectural boundaries in this way makes it possible to address problems that are impossible to solve by tweaking a single layer alone. UIA's personal names depend on the personal group abstraction to provide a scope within which to interpret these names, and depend on the overlay routing layer to provide devices with location-independent identities that can be securely authenticated to a named device regardless of how or where on the network the device is attached. UIA's distributed personal group management system in turn depends on the overlay routing layer to enable devices to keep their personal groups synchronized regardless of how or where they are connected. UIA's transport layer similarly depends on the overlay routing layer to support peer-to-peer communication directly between mobile devices on public or private networks. UIA's overlay routing layer in turn can make use of social network information provided by the personal name and personal group management system in order to increase its efficiency, security, and reliability, and can leverage advanced NAT traversal techniques to minimize unnecessary forwarding of traffic through the overlay.

Within each of the main functional areas it addresses, UIA makes additional technical contributions as summarized below.

1.7.1 Naming

UIA's primary naming contributions are its personal name and personal group abstractions:

- **Personal groups** provide users with a simple, intuitive model for administering a collection of devices they own and operate, based on pairwise device introduction and automatic synchronization of group management state across devices.

- **Personal names** provide a more concise and convenient alternative to traditional global DNS names for purposes of naming devices or other users from within the scope of a user's own administrative domain.

In addition to the basic networking abstractions above, UIA makes the following technical contributions in its design for a prototype protocol suite implementing personal groups and personal names:

- UIA introduces an **always-available namespace consistency model**, which ensures that a user's personal namespace is always accessible and modifiable on any device regardless of the device's connectivity, but that namespace changes always propagate automatically among devices whenever connectivity permits.

- UIA introduces a **decentralized device ownership and revocation algorithm**, allowing a user to remove a device from his group, either gracefully or forcefully, providing assurance that lost or stolen devices cannot be misused to gain unwarranted access to the user's personal group.

1.7.2 Routing

UIA's primary routing contribution is **a development and analysis of several approaches to scalable overlay routing among devices with self-configuring, location-independent identities**. In particular, UIA introduces:

- **A routing scheme that leverages the user's social network**, to provide secure, robust connectivity among devices in a user's personal group and social neighborhood with minimal protocol complexity or maintenance overhead.

- **A routing scheme that leverages distributed hash table (DHT) concepts**, to build a more general routing and traffic forwarding mechanism directly on top of UIA's flat endpoint identifier (EID) space.

- **A routing scheme that builds on compact routing theory**, which provides provable bounds on routing table size and stretch, but also addresses congestion, fault tolerance, and other practical issues ignored in the theoretical groundwork.

In support of this exploration of specific overlay routing schemes, UIA introduces **the first detailed exposition and experimental analysis of practical NAT traversal techniques**. The exposition analyzes the previously unaddressed problem of **traversal of multiple levels of NAT**, and identifies a crucial behavioral property for NATs, dubbed **hairpin translation**, which enables automatic and transparent multi-level NAT traversal. UIA's experimental analysis of NAT traversal techniques includes data covering a wide variety of NATs deployed in the field, examining NAT behavioral properties required for both single-level and multi-level NAT traversal via both UDP and TCP.

1.7.3 Transport

UIA's primary contribution to the Internet's transport layer is the **structured stream abstraction**, a novel extension to TCP's traditional byte-stream abstraction that enables modern interactive applications to create and manage many concurrent streams easily by "forking" new streams from existing ones. To demonstrate that this transport abstraction can be implemented efficiently, UIA introduces **structured stream transport** (SST), a prototype transport protocol based on structured streams. The SST protocol design makes two additional technical contributions:

- SST introduces a **division of the transport layer into two sublayers**, where the "bottom half" or *channel layer* implements network-oriented services such as sequencing, acknowledgments, congestion control, and security, and the "top half" or *stream layer* implements application-oriented services such as logical byte streams and receiver-directed flow control.

- SST's stream sublayer introduces a **datagram optimization** that provides either reliable or unreliable transmission of datagrams as "ephemeral streams," providing the efficiency benefits of datagram transports without the attendant datagram size limitations or other restrictions.

1.8 Limitations

While UIA explores some promising approaches to rethinking the Internet architecture around the requirements of personal devices, in terms of a "grand solution" to the Internet's current difficulties with personal devices, UIA is still incomplete in many ways—necessarily so due to the breadth of the topic and the design space. UIA's naming system explores one particular design for personal groups and personal names, using a particular introduction mechanism for building groups and a particular revocation mechanism for pruning them, but it is easy to envision other alternative naming designs that might be more user friendly or effective in some usage scenarios (and perhaps less so in others). The three routing schemes explored in this thesis cover some breadth in in design space but do not yet yield a single obvious "winner"—the deployed social routing scheme is simple and efficient in most cases but cannot support all types of applications effectively, whereas the other two schemes have only been evaluated under simulation and have known practical issues that might deter full implementation and large-scape deployment. UIA's structured stream transport is similarly promising for what it does, but there are many ways its design could be evolved further, such as by supporting multihoming as SCTP [233] already does. Finally, there may be are a variety of potential technical, economic, and political barriers to the widespread adoption of an architecture like UIA. Chapter 9 will describe these limitations and potential barriers to adoption in more detail.

1.9 Thesis Organization

In terms of overall organization, this thesis first develops and explores each of the main functional areas in the network stack that UIA addresses—naming, routing, and transport—fairly independently of each other in Chapters 2–4, then after a brief diversion into NAT traversal techniques in support of UIA's routing functionality, each of the remaining chapters addresses all of the functional areas together to take a broader perspective in evaluating the architecture. A quick summary of each chapter follows:

- **Chapter 2** introduces UIA's personal group management and naming paradigm, and describes protocols and algorithms to implement that naming model.

- **Chapter 3** addresses the challenge of overlay routing among UIA's location-independent endpoint identifiers, exploring three specific schemes: social routing (Section 3.2), identity hash routing (Section 3.3), and compact routing (Section 3.4).

- **Chapter 4** describes UIA's structured stream transport abstraction, and presents a protocol design representing one approach to implementing this transport abstraction efficiently.

- **Chapter 5** describes NAT traversal background, hole punching techniques for UDP (Section 5.2) and TCP (Section 5.3), and delineates the requirements NATs and firewalls must satisfy in order to support these techniques (Section 5.4).

- **Chapter 6** describes the current prototype implementations of UIA's personal naming (Section 6.1), overlay routing (Section 6.2), and structured stream transport (Section 6.3).

- **Chapter 7** evaluates UIA experimentally based on both real-world use and simulation, and informally through pragmatic experience with its personal naming model.

- **Chapter 8** describes in detail the relationship of UIA to other prior work.

- **Chapter 9** finally concludes the thesis, discussing limitations and potential barriers to adoption, and avenues for future work.

Chapter 2

Naming

As network-enabled mobile devices such as laptops, smart phones, media players, personal digital assistants, gaming consoles, and digital cameras become ubiquitous in the lives of ordinary people, the proliferation of these devices makes it increasingly important for users to have a convenient way to *name* and connect them over the Internet, while preserving users' security and privacy. This chapter details UIA's naming layer, which provides a naming infrastructure enabling users to connect and communicate conveniently among mobile personal devices. We first explore the motivation and goals of UIA's naming system in Section 2.1, then examine UIA naming from the (nontechnical) user's perspective in Section 2.2, and finally delve into the technical details of the naming system in Section 2.3.

2.1 Motivation and Purpose of UIA Naming

The Internet's current naming infrastructure has evolved to be well-suited to *client/server* communication—e.g., a user connecting to an online service like Google or Amazon.com. Networked personal devices also require *peer-to-peer* communication, however, in order to be maximally useful. While on a trip, for example, a user in a cyber cafe may wish to copy photos from his WiFi-enabled camera to his PC at home for storage and backup. Two users who meet in a park or other off-Internet location may wish to connect their WiFi devices to exchange photos or other information, and later re-establish a connection between the same devices over the Internet after returning to their homes. A Voice-over-IP user would like his WiMax phone to be easily reachable by his friends wherever he and they are located, but not to be reachable by unknown telemarketers.

2.1.1 Global Names and Their Limitations

Convenient global communication over the Internet, however, currently requires the target device to have a *global name* in the Internet's Domain Name System (DNS) [161, 162]. Users must register with central naming authorities to obtain these global names, and since DNS names are global in scope, users can only obtain names that are not already taken by someone else. Ever since the

Internet's commercialization and "`.com` boom," short, convenient, memorable global names have become ever more difficult and more expensive to obtain.

Furthermore, for a DNS name to work, the named host must have a stable IP address [120], and the user must understand how to obtain and assign that IP address, or at least know it, in order to enter it into the DNS record for the named host. Protocols such as Dynamic DNS [256], Mobile IP [183], and Virtual Private Networks [95] can provide ways of naming hosts with no stable IP address, but the additional configuration effort and technical expertise they require makes them deployable in practice only by organizations with dedicated network administration staff. User interface refinements alone cannot overcome this deployment roadblock, because the protocols depend on centralized resources—global domain names and static, public "home" IP addresses—that are not part of most consumer-oriented Internet service packages. Ordinary users require a solution that "just works."

2.1.2 An Alternative Model: "Virtual USB Cables"

In contrast with the operation of the global name system, there is one implicit "naming system" we can take inspiration from that is both straightforward and trivially secure: plugging two devices together via a USB cable. The cable itself physically indicates which devices should participate in the transfer (the *naming* aspect), and the isolated physical medium guarantees that no other devices can eavesdrop or interfere with the communication (the *security* aspect). As personal devices begin to support wireless networking and Internet connectivity, we would like to extend the simplicity and security of a USB cable to device connectivity on a global scale. We would like to develop a naming model in which Alice can connect her WiFi-enabled iPod to her home PC via a "virtual USB cable," so that she can browse photos or play music stored there from a WiFi-enabled coffee shop or friend's house. Setting up this "virtual cable" should not require technical knowledge or special configuration on Alice's part, it should not require Alice to obtain any form of scarce global resources from any central organization, and it should continue working even when the devices it connects are behind firewalls or NATs.

Extending the "virtual USB cable" model to inter-user communication, if Alice meets Bob in a coffee shop, she should easily be able to share with him information or services located on any of her personal devices simply by "plugging" their two devices together. Bob should be able to connect to Alice's devices even after he leaves the coffee shop, until she chooses to sever their relationship—i.e., the virtual USB cable should be able to "stretch" invisibly across arbitrary distances. No one else should be able to impersonate Bob, however, in order to gain access to Alice's shared resources.

2.1.3 UIA Personal Names and Personal Groups

The Unmanaged Internet Architecture uses the above "virtual USB cable" analogy to build a peer-to-peer connectivity architecture that gives nontechnical users a simple and intuitive way to connect their mobile personal devices, via convenient *personal names* organized into *personal groups*. A *personal group* represents an administrative cluster of devices all owned or controlled by the same user—but all devices need not be in the same location. A personal name is a name for a device, user, or other object that is *locally scoped* to a particular personal group: thus, a personal name only needs to be unique within the context of a given personal group (administrative domain), and not within a global namespace like DNS that is potentially shared by everyone in the world.

The way a user defines and builds a personal group in UIA is a generalization of the "virtual cable" analogy. By *introducing* devices to each other "in person," on a local-area network for example [65, 231]—a process analogous to "plugging in" a cable between the two devices—a user can *merge* UIA devices to form a personal group. Unlike the ephemeral names used in rendezvous schemes such as Apple Bonjour [11], UIA devices once introduced work together to offer secure remote access to any device in the group from any other, even if some or all the devices move to other network attachment points. The devices forming the group present the user with a shared personal namespace, which they optimistically replicate [105, 133, 243] to ensure constant availability on each device whether on or off the Internet. The devices gossip namespace changes as connectivity permits [63], and can propagate updates via mobile devices carried by the user [176].

Since UIA interprets personal names relative to personal groups, users can assign concise, meaningful names like `ipod` instead of long globally unique names like `ipod.alicesm5186.myisp.com`. In this way UIA conforms to the intuitive model with which users already manage their cell phones' address books. Users normally create personal names by introducing devices locally on a common WiFi network, for example, but they can also introduce devices remotely by "bootstrapping" from DNS or another global name system, if at least one device to be introduced has a global name. Since UIA names persist and remain securely bound to their targets as devices migrate, once Alice introduces her iPod to her home PC, her iPod can continue accessing her PC by the same name from anywhere she finds Internet access.

Personal names are intended to supplement and not replace global DNS names: users can refer to personal names like `phone` alongside global names like `usenix.org` in the same applications. Global names are the right naming abstraction to represent global "brand names" like `amazon.com`, for which the basic idea is that the owner of the global name should be able to advertise it on billboards, on TV, etc., and expect people to be able to remember the global name and type it into any web browser in the world to reach the named business or organization. So personal names and global names serve different and complementary roles: in the long term the Internet needs both.

2.1.4 Cryptographically Secure Naming

Once created, UIA personal names remain persistently bound to their targets as devices move, via self-certifying cryptographic identities [157, 163, 203] that are fully self-configuring and hidden from the user. Besides ensuring that a given personal name always securely refers to the same device, these cryptographic identities permit applications and services running on the user's personal devices to authenticate other users wishing to access shared information. For example, Alice can instruct a desktop application to allow remote access to her home PC from any device in her personal group, but not from other devices, effectively placing a "virtual firewall" around her personal group that corresponds to her *administrative domain* rather than to any physical network topology.

In contrast with SDSI [203] and PKI-based secure naming [14, 115], UIA does not use public keys to represent a user's identity or require the user to know about and manage his keys directly across the multiple personal devices he may own. Each UIA device instead maintains its own per-device keys automatically,

2.1.5 Social Networking via Personal Names

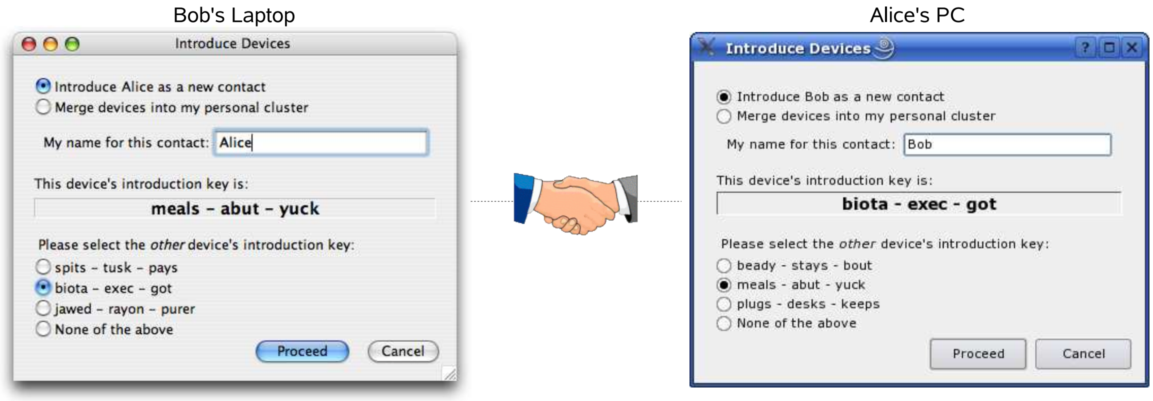
Different users can introduce their devices to name other users and link their respective personal groups. Bob can refer to his friend Alice as `Alice`, and if Alice calls her VoIP phone `phone` then Bob can call Alice's phone using the name `phone.Alice`. In this way, UIA adapts peer-to-peer social networking ideas previously explored for other purposes [60, 145, 153, 184, 188] to form a user-friendly peer-to-peer naming infrastructure. UIA replicates a user's personal namespace across all devices in the group to ensure constant availability on each device regardless of connectivity, implementing a relaxed consistency model reminiscent of Bayou [243], but without dependencies on centralized resources such as Bayou's primary commit servers. Users can also create and collect names into ad hoc *shared groups* to reflect common interests or informal organizations.

The UIA naming system makes the following primary contributions, expanding on previously proposed ideas [85]. First, UIA introduces a simple and intuitive model for connecting mobile devices into *personal groups*, providing ad hoc user identities, personal names, and secure remote access, without requiring the user to manage keys or certificates explicitly. Second, UIA presents a novel gossip and replication protocol to manage the naming and group state required by this user model, adapting optimistic replication principles previously developed for file systems and databases.

2.1.6 Goals of UIA Naming

The purpose of UIA naming is to provide users with a convenient and intuitive way to communicate among their own and their friends' personal devices. To this end, we can briefly summarize the primary goals of UIA's naming system as follows:

- Names must be *user-relative*, so as not to require global uniqueness. If Alice owns only one laptop and has only one friend named Bob, she should be able to refer to them simply as `laptop` and `Bob`, despite the millions of other laptops and people named Bob in the world.
- Names must have *strong bindings* to the identities of the objects named, independent of their current physical location or network attachment point. When Alice refers to her name `laptop`, the name should always resolve to *her* laptop or fail to resolve; no other device should be able to impersonate it.



Bob's Laptop:

● Introduce Alice as a new contact
○ Merge devices into my personal cluster

My name for this contact: Alice

This device's introduction key is:

meals – abut – yuck

Please select the *other* device's introduction key:

○ spits – tusk – pays
● biota – exec – got
○ jawed – rayon – purer
○ None of the above

Alice's PC:

● Introduce Bob as a new contact
○ Merge devices into my personal cluster

My name for this contact: Bob

This device's introduction key is:

biota - exec - got

Please select the *other* device's introduction key:

○ beady - stays - bout
● meals - abut - yuck
○ plugs - desks - keeps
○ None of the above

Figure 2-1: Bob and Alice introduce their devices

- Assigning names must be a *simple and intuitive* process. If Alice meets Bob at a conference and their laptops share a common WiFi network, assigning a name to Bob should be as simple as looking him up in a local network browser and clicking "Introduce".

- A user should only have to manage *one namespace*. If Alice already owns several devices, she should only have to name a newly purchased device once, and should not have to re-enter or manually copy the new name onto each of her existing devices.

- A user's personal namespace should be constantly *available*, remaining accessible on any of the user's devices even while that device is disconnected from the others and/or from the Internet.

- Users should be able to *authenticate* each other and *selectively share* information and device services by referring to each others' personal namespaces. If Alice gives Bob permission to access some files on her desktop PC, he should have access to them via a name as simple as `PC.Alice`.

- Finally, UIA personal names should *coexist cleanly* with DNS, allowing users to refer to personal names like `laptop` alongside global names like `amazon.com` seamlessly within the same application.

2.2 UIA Naming from the User's Perspective

This section describes UIA's operating principles from the perspective of a non-technical user; later sections detail how the system provides this user experience.

2.2.1 Introducing Devices

A UIA device ideally ships from its manufacturer pre-configured with a name for itself such as `laptop` or `phone`, which the user can keep or change as desired. The device learns additional names as its user *introduces* it to other devices owned by the same user or different users. The introduction process assigns persistent names by which the device can securely refer to other devices. UIA's introduction process builds on previously explored ideas for secure associations, such as the Resurrecting Duckling security model [231] and SPKI/SDSI introduction [65], but UIA is unique in providing the ability to build symmetric, self-managing, distributed personal groups solely out of pairwise introductions.

In a typical introduction, the owner(s) of two devices bring the devices together physically and connect them to a common local-area network. Each user then invokes a local-area rendezvous tool similar to Bonjour's [11] on his device, finds the other device on the network, and selects "Introduce." Each device displays an *introduction key* consisting of three words chosen randomly from a dictionary, as shown in Figure 2-1. Each user then picks the other device's introduction key from a list of three random keys. If one of the devices has unintentionally connected to the wrong endpoint, such as an impersonator on the same network, then the matching key is unlikely to appear on the list, so the user picks "None of the above" and the introduction procedure aborts. Unlike other analogous procedures [65], UIA uses short, user-friendly "one-time" keys that only need to withstand online and not offline attacks, and its multiple-choice design prevents users from just clicking "OK" without actually comparing the keys.

Users can follow the same procedure to introduce UIA devices remotely across the Internet, as long as one device has a global DNS name or IP address and the users have a trustworthy channel through which to exchange introduction keys: e.g., a phone conversation or an authenticated chat session. We also envision alternative introduction mechanisms adapted to specific rendezvous channels such as E-mail, web sites, SMS messages, or short-range wireless links; the details of particular introduction mechanisms are not crucial to the UIA architecture.

A user can introduce UIA devices either to *merge* his own devices into a *personal group* sharing a common namespace, or to create named *links* from his own group to other users' personal groups. The following sections describe these two forms of introduction, and other important group management actions, with the help of an example scenario illustrated in Figure 2-2.

2.2.2 Device Names and Personal Groups

At Time 1 in the scenario, Bob purchases a new laptop and Internet phone, which come preconfigured with the default names `laptop` and `phone`, respectively. At Time 2, Bob uses UIA's local rendezvous tool on each device to find the other device on his home WiFi network and selects "Introduce devices" on each. Bob chooses the "Merge devices" option in the introduction dialogs (see Figure 2-1) to merge the devices into a personal group.

The devices in Bob's group gossip both existing names and subsequent changes to the group's namespace as physical network connectivity permits. Each device attempts to preserve connectivity to other named devices as they leave the network and reappear at other locations, without user intervention whenever possible. Bob now sees his two personal names `phone` and `laptop` on both devices, and can use these names for local and remote access. Working on his laptop at home, he

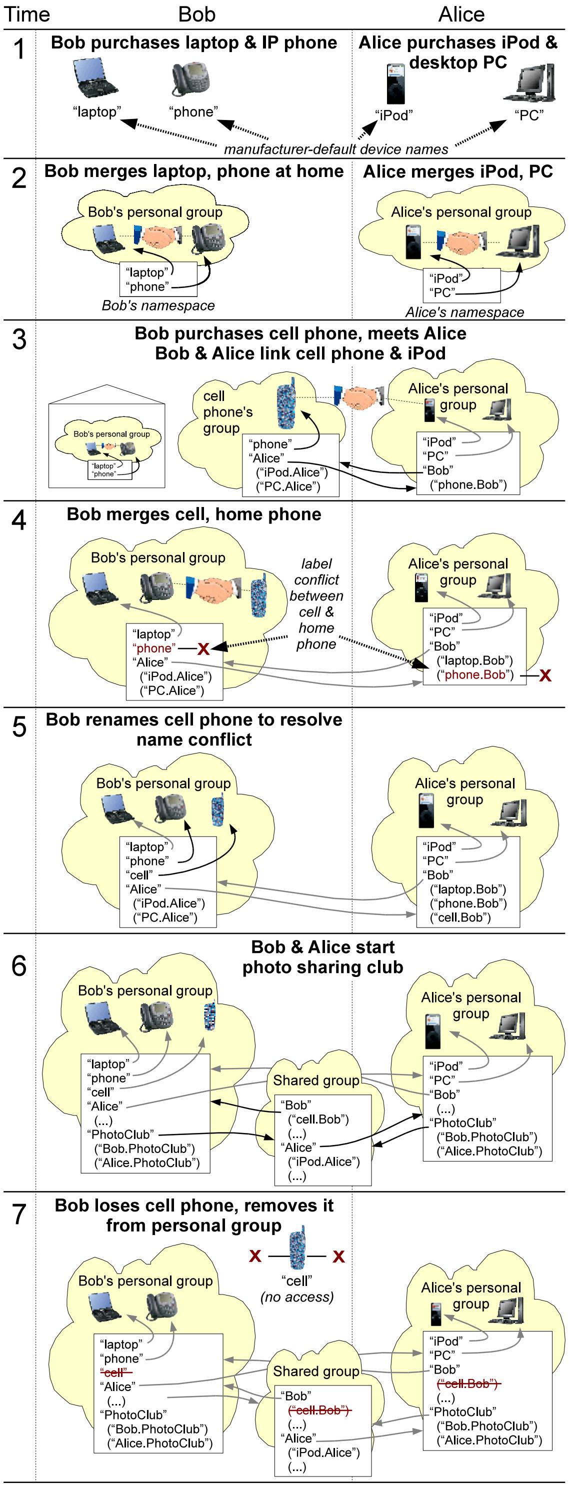


Figure 2-2: Example personal device scenario

uses his personal name phone to reach the phone via his home WiFi LAN. When Bob subsequently takes his laptop on a trip, he can remotely access his home phone from his laptop over the Internet (e.g., to check his voice messages), still using the name phone. UIA uses cryptography to guarantee that an adversary cannot impersonate the device Bob calls phone, and cannot eavesdrop on his communication.

2.2.3 User Names and Social Networking

With the second form of introduction, users link their personal groups together and assign *user names* to each other, but retain exclusive control over their respective personal groups. In the example scenario, Bob purchases a new WiFi-enabled cell phone at Time 3 and meets Alice at a cafe before he has merged his cell phone with his other devices. Bob finds Alice's iPod using his cell phone's local rendezvous tool and selects "Introduce as a new contact" (see Figure 2-1), and Alice does likewise. Bob's phone suggests Alice's self-chosen user name Alice, but Bob can override this default (e.g., to Alice-Smith or Alice-from-OSDI) if he already knows another Alice.

Bob and Alice can now refer to each others' devices by combining device names with user names in DNS-like dotted notation. If Alice runs a web server on her home PC, named PC in Alice's personal namespace, then Bob can connect to Alice's server by typing PC.Alice into his laptop's web browser, exactly as he would use a global DNS name like usenix.org.

If Alice's personal web server is UIA-aware, she can use her name Bob in the server's access control lists so that only Bob's personal devices may browse certain private areas. UIA authenticates clients so that no one else can impersonate Bob's devices to gain access to these areas.

2.2.4 Transitive Merging and Gossip

Bob now returns home and merges his cell phone with his home phone, as shown at Time 4 in Figure 2-2. Bob's home phone in turn gossips the cell phone's group membership to Bob's laptop, so the laptop and cell phone can name each other without him having to merge them explicitly. Alice's devices similarly gossip her new link named Bob and learn about Bob's three devices, after which she can, for example, refer to Bob's laptop as laptop.Bob.

Users can access or edit their personal groups from any of their devices while other devices are unreachable. If Bob and Alice are on a bus together and disconnected from the Internet, Alice can still reach Bob's laptop from her iPod via her name laptop.Bob, even if they have left all their other devices at home. Bob and Alice can continue adding names for contacts they meet on the bus, and their other devices learn the new names via gossip later when they re-connect.

2.2.5 Resolving Conflicts

Unfortunately, both of Bob's phones happened to have identical default names of phone, resulting in their names conflicting in his newly merged namespace. UIA notifies Bob of the conflict, and he can continue using the non-conflicting name laptop, but must resolve the conflict before the name `phone` will work again. Bob resolves the conflict on his cell phone at Time 5, by renaming it `cell` while leaving the home phone with the name `phone`. Bob's other devices learn the resolved name bindings via gossip, as do Alice's devices, so Alice now sees Bob's phones as `phone.Bob` and `cell.Bob`.

If Bob makes conflicting namespace changes on two of his devices while they are partitioned from each other, UIA detects the conflict once the devices reconnect. Bob can continue using other non-conflicting names in the same group while conflicts exist, and he can resolve such conflicts at leisure on any of his devices.

2.2.6 Shared Groups

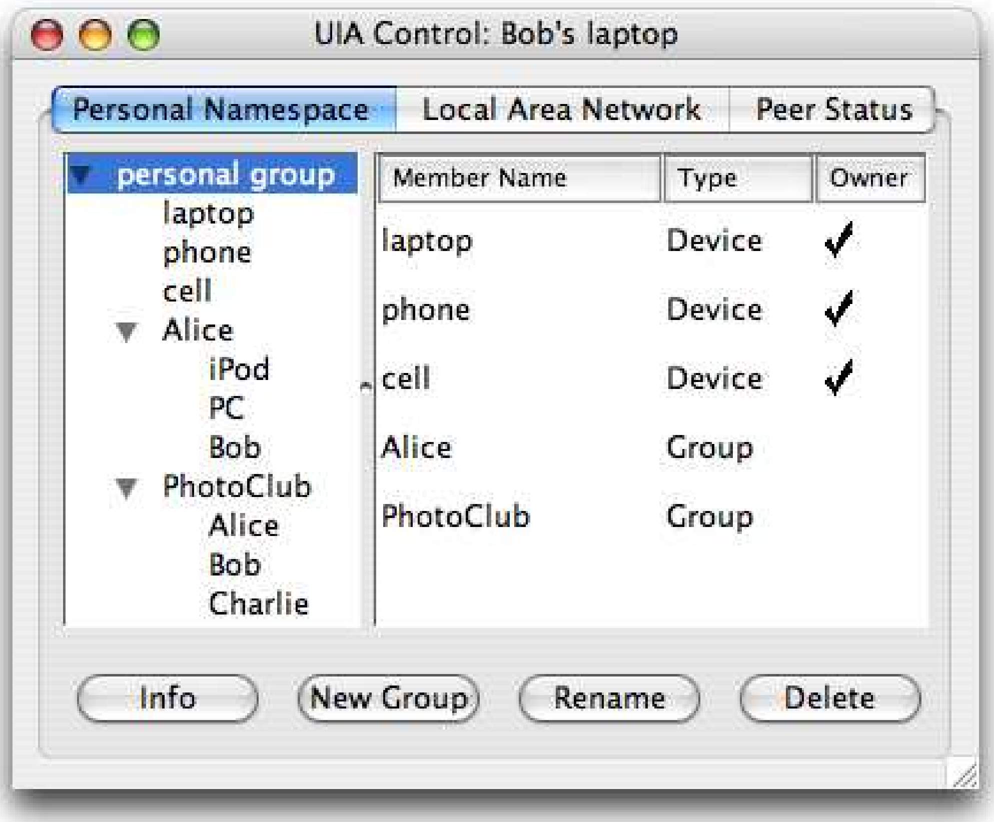
In addition to personal groups, users can create *shared groups* to help organize and share their personal names. Bob and Alice discover at Time 6 that they share an interest in photography, and decide to start a photo club for themselves and other friends sharing this interest. To enable members of the club to find each other easily and share photos among their personal devices, Bob uses his laptop to create a shared group named `PhotoClub` in his personal namespace. On creation, the shared group's only member is Bob himself. To add Alice to the group, Bob drags the name `Alice` from his personal group into `PhotoClub`, copying his name binding for Alice into the shared group and making her the second member. Bob can similarly add other friends to `PhotoClub`, and these names automatically appear in Alice's view of the group as the devices gossip the changes.

Although Alice can now refer to the new group as `PhotoClub.Bob`, she might like this group to appear directly in her own personal group instead of naming it relative to Bob. Alice drags the `PhotoClub` name from Bob's personal group into her own, giving herself a copy of the name leading to the same shared group. She can now refer to group members using the same names that Bob uses, such as `Charlie.PhotoClub`.

2.2.7 Group Ownership

One or more members of a UIA group may be designated as *owners*, or members allowed to modify the group. As Figure 2-3 illustrates, Bob's devices `laptop`, `phone`, and `cell` are owners of his personal group by default, allowing Bob to edit his personal group using any of his devices. The names `Alice` and `PhotoClub` are not owners, so Alice and members of `PhotoClub` can only browse and resolve names in Bob's namespace.

Groups can own other groups. When Bob creates his shared `PhotoClub` group, UIA automatically includes a name `Bob` in the new group that gives Bob's personal group ownership of the new group. After adding Alice to the group, Bob can give her co-ownership by clicking the owner flag by her name in the group listing, enabling her to add or remove other members herself. Ownership is transitive: Bob can modify `PhotoClub` using his laptop because Bob's laptop is an owner of Bob's personal group and Bob's personal group is an owner of `PhotoClub`.



Member Name	Type	Owner
laptop	Device	✓
phone	Device	✓
cell	Device	✓
Alice	Group	
PhotoClub	Group	

Figure 2-3: Groups and ownership

2.2.8 Security and Ownership Revocation

Returning to the scenario in Figure 2-2, Bob loses his cell phone at Time 7, and he is not sure whether it was stolen or just temporarily misplaced. If the cell phone was stolen and has no local user authentication such as a password or fingerprint reader, the thief might obtain not only Bob's data on the cell phone itself, but also remote access to services authorized to his personal group via UIA names. UIA devices capable of accessing sensitive information remotely should therefore provide strong local user authentication, and should encrypt personal data (including UIA state) stored on the device, as Apple's FileVault does for example [12]. The details of local user authentication and encryption are orthogonal to UIA, however.

To minimize potential damage if a thief does break into Bob's user account on his cell phone, Bob can revoke the cell phone's ownership of his personal group. If the cell phone re-appears and Bob realizes that he just misplaced it, then he can "undo" the revocation and return the phone to its normal status. If the cell phone remains missing, however, UIA ensures that no one can remotely access personal information or services on Bob's other devices via the lost phone once the revocation announcement has propagated to those devices. Similarly, the cell phone loses its access to the files Alice shared with Bob as soon as Alice's PC, on which the files reside, learns of the revocation from any of Bob's remaining devices.

2.2.9 Ownership Disputes

Revocation cuts both ways: a thief might try to "hijack" Bob's personal group, using the stolen cell phone to revoke the ownership of Bob's other devices before Bob finds that the phone is missing. In UIA's current ownership scheme in which all owners have full and equal authority over a group, Bob's devices cannot distinguish the "real" Bob from an impostor once a stolen device's local access control is broken. UIA therefore allows any device to *dispute* another device's revocation of its ownership.

In the example scenario, when Bob next uses his laptop, UIA informs him that his laptop's ownership of his personal group has been revoked by the cell phone, which Bob realizes was stolen. In response, Bob issues a revocation of the cell phone's ownership from his laptop. The two mutual revocations effectively split Bob's original personal group into two new, independent groups: one containing only the cell phone, the other containing Bob's remaining devices. All existing UIA names referring to Bob's old personal group, and any access authorizations based on those names, become unusable and must be manually updated to point to the appropriate new group. Alice's name `Bob` for example is now marked "disputed" in Alice's namespace, and Alice's PC rejects attempts by any of Bob's devices to access the files she shared with Bob earlier using that UIA name. To update her name for Bob and safely renew his access, Alice can re-introduce her devices directly to Bob's the next time they meet, or obtain a fresh link to Bob's new personal group from a trusted mutual friend who already has one.

Group ownership disputes need not be permanent. Suppose two people who co-own a shared group get into an argument, and split the group by issuing mutual revocations. If the original co-owners later settle their differences, they can undo their conflicting revocations or simply merge their respective "splinter" groups back together via UIA's normal merge mechanism. Links to the original group become unusable during the dispute, but function again normally after the dispute is resolved.

2.3 Personal Naming System Design

This section describes the design of the UIA personal naming system, beginning with the system's basic architecture, followed by the details of how UIA devices manage and synchronize the namespace state comprising their users' personal and shared groups.

2.3.1 Basic Architecture

The design of the UIA personal naming system is based on three crucial elements: a mechanism for *identifying* personal devices uniquely, securely, and persistently; a process for *resolving* names within personal groups; and a storage management and gossip mechanism for optimistically *replicating* namespace state across devices. This section briefly introduces these three architectural components, for which subsequent sections provide further details.

Personal Endpoint Identities

UIA devices identify each other using cryptographically unique *endpoint identifiers* or EIDs. Whereas DNS maps a name to an IP address, UIA maps a personal device name such as Bob's `laptop` to an EID. Unlike IP addresses, EIDs are *stable* and do not change when devices re-connect or move. UIA's routing layer, described in Chapter 3, tracks mobile hosts by their EIDs as they change IP addresses, and can forward traffic by EID when IP-level communication fails due to NAT or other Internet routing discontinuities. Thus, EIDs represent the "point of rendezvous" between UIA's naming and routing systems.

A UIA device creates each EID it needs automatically by generating a fresh public/private key pair and then computing a cryptographic hash of the public key. As in SFS [157], EIDs are cryptographically unique, self-configuring, and self-certifying, but not human-readable. As in HIP [163], UIA-aware network transports and applications use EIDs in place of IP addresses to identify communication endpoints. (UIA can also disguise EIDs as "actual" IP addresses for compatibility with unmodified legacy applications, as described later in Section 6.1.)

An EID corresponds to a particular user's presence on a particular device. A user who owns or has access to several devices has a separate EID for each. A device accessed by only one user needs only one EID, but a device shared among multiple users via some form of login mechanism creates a separate EID for each user account. Unlike cryptographic host identifiers in SFS and HIP, therefore, EIDs are not only stable but *personal*.

Personal EIDs allow multiple users of a shared UIA host to run independent network services on the device. Since each user's services bind to the user's EID rather than to a host-wide IP address, UIA-aware network applications can run exclusively in the context of the user and rely on UIA to provide user-granularity authentication and access control. When Bob connects his laptop to the HTTP port at the EID to which `PC.Alice` resolves, he knows he is connecting to *Alice's* personal web server and not that of another user with an account on the same PC. Alice's web server similarly knows that the connection is coming from Bob and not from someone else using his laptop, because her name `laptop.Bob` resolves to an EID specific to Bob's account on his laptop.

Resolving Names in Personal Groups

Each UIA device acts as an ad hoc name server to support name lookups and synchronize namespace state across devices. UIA names follow the same formatting rules as DNS names, consisting of a series of *labels* separated by dots, and devices resolve UIA names one label at a time from right to left. To resolve the name `PC.Alice`, for example, Bob's laptop first resolves the rightmost component `Alice` to find Alice's personal group, and from there resolves the second component `PC` to find the EID for Alice's PC as named in Alice's personal group.

Whereas DNS resolution traverses a strictly hierarchical tree of "zones" starting from a centrally-managed global root zone, each UIA device has a unique root for resolving UIA names, and users can link UIA groups to form arbitrary graphs. After Bob meets Alice at Time 3 in Figure 2-2, for example, Bob's "root" group for UIA name resolution, corresponding to his personal group, appears to Alice as a "sub-group" named `Bob`. Conversely, Alice's "root" group appears to Bob as a "sub-group" named `Alice`. Since Bob's and Alice's naming relationship forms a cycle in the graph of UIA groups, Bob could for example refer to his own phone via the redundant name `phone.Bob.Alice`.

UIA groups may at times contain *label conflicts*, or bindings of a single name to multiple distinct targets. When Bob at Time 4 merges his new cell phone with its default name `phone` into his personal group, which already contains another device named `phone`, the two `phone` bindings result in a label conflict. Label conflicts also arise if an ownership dispute splits the *target* that a group name refers to, as described in Section 2.2.9. Name resolution fails if it encounters a label conflict, preventing the user from following ambiguous links before resolving the conflict. A conflict on one label does not affect the usability of other labels in the same group, however.

State Management

UIA uses optimistic replication [105, 133, 243] to maintain a user's personal UIA namespace across multiple devices, guarding namespace state against device loss or failure and keeping the namespace available on all devices during periods of disconnection or network partitions. Each device stores in an append-only log all persistent naming state for its user's personal group and any other groups of interest to the user, and uses an epidemic protocol [63] to distribute updates of each

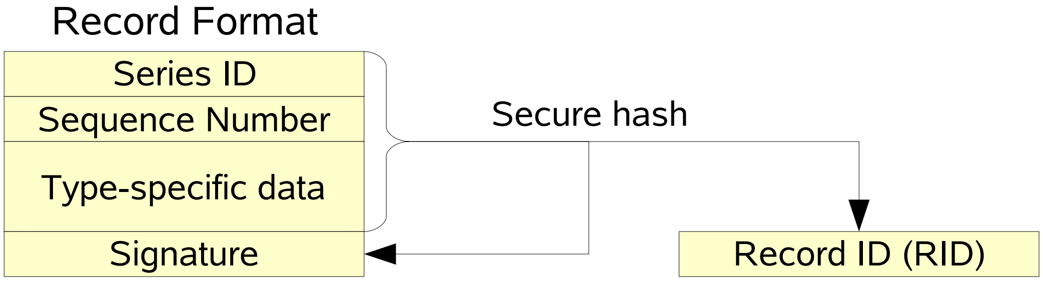


Figure 2-4: Basic log record format

group's state among the devices interested in that group. Devices use rumor mongering to propagate newly-written or newly-learned records aggressively through the network, and they perform periodic anti-entropy exchanges to ensure that updates reliably reach all interested devices as network connectivity permits.

2.3.2 Device Log Structure

Each UIA device keeps an append-only log holding all persistent UIA naming-related state. The records comprising the log collectively hold all of the persistent state for the user's personal group and any other UIA groups of interest to the user. A device's log typically holds both records generated by the device itself and gossiped replicas of relevant records generated by other devices. Devices sharing a namespace propagate updates via gossip, copying any relevant new records from other devices' logs to their own as they appear and as network connectivity permits. Storing all persistent namespace state in log form facilitates automatic synchronization of namespace changes across devices, and also provides the ability to undo accidental name deletions or malicious changes from a compromised device.

UIA organizes the records comprising a device's log into *series*, each series representing the sequence of changes a particular device writes to a particular group. The state defining a group consists of one or more series, one for each device that has written to the group. All devices participating in a group gossip and replicate all records in each of the group's series, preserving the order of records in a given series, but do not enforce any order between records in different series. Since UIA separates the naming state for each group by series, devices can limit gossip to the records relating to groups they're interested in, instead of scanning their neighbors' entire device logs.

As shown in Figure 2-4, each log record contains a series ID, a sequence number, data specific to the record type, and a signature. The series ID (SID) uniquely identifies the series to which the record belongs. The sequence number orders records within a series. The device that owns a series signs each record in that series with its private key, so that other devices can authenticate copies of records they receive indirectly. A cryptographic hash of the record yields a *Record ID*, which uniquely identifies the record for various purposes described later.

UIA currently defines four record types, listed in Figure 2-5 and summarized briefly below:

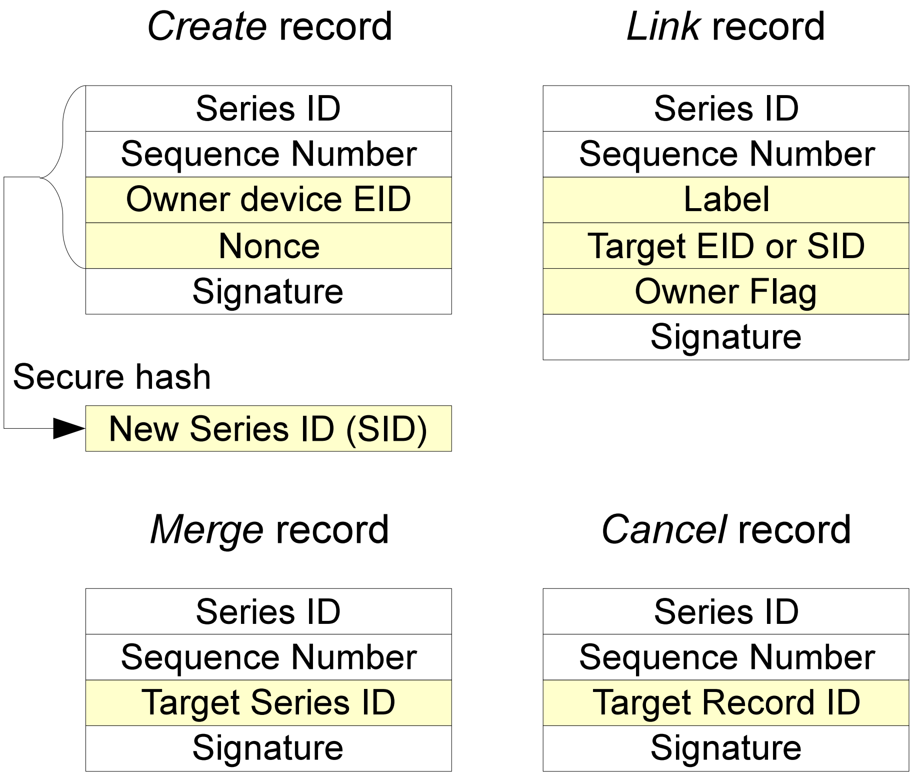


Figure 2-5: Specific log record types

- **Create:** A *create* record initiates a new series owned by the device writing the record, as identified in the record's owner field. The owner EID fixes the public/private key pair other devices use to authenticate records in the new series. The record ID of the create record becomes the new series ID; a random nonce ensures the new SID's cryptographic uniqueness. The create record itself is not part of the new series: its own series ID field is usually empty to indicate that it is not part of any series, but it can be non-empty for revocation purposes as described later.

- **Link:** A *link* record binds a human-readable label such as `Alice` to an endpoint ID or series ID denoting the link's target. Links to devices, such as Bob's names `laptop` and `phone`, contain the EID of the target device. Links to groups, such as `Alice` and `PhotoClub`, contain the SID of some series in the target group. A link record has an owner flag indicating whether the link grants ownership to the link's target, allowing the target to write changes to the group containing the link record. We refer to a link with its owner flag set as a *link-owner record*.

- **Merge:** A *merge* record joins two series to form a single UIA group. The union of all link and cancel records in all merged series determines the set of names that appear in the group, thereby forming a common distributed namespace. A merge takes effect only if the device that wrote the merge record also owns the target group, or if there is a corresponding merge record in the target group pointing back to the first group.

- **Cancel:** A *cancel* record nullifies the effect of a specific previous record, specified by the target's record ID. With certain restrictions described below, link records can be canceled to delete or rename group members. Create, merge, and cancel records cannot be canceled.

2.3.3 Namespace Operations

This section describes how UIA devices implement the important user-visible namespace control operations, in terms of the specific records the devices write to their logs at the events in the example scenario from Figure 2-2. The following section will then explain how devices evaluate the contents of their logs to determine the effective state of each group at any point in time.

Device Initialization: When Bob and Alice install or first start UIA on a device at Time 1, the device first writes a create record to its log, forming a new series to represent the user's personal "root" group on that device. The device then writes a link record to the new series, giving itself a suitable default name such as `laptop`. The device sets the owner flag in this link record to make itself the sole initial owner of the group.

Merging Device Groups: When Bob introduces and merges his devices at Time 2 to form a personal group, each device writes to its own root series a merge record pointing to the other device's root series. These cross-referencing merge records result in a *merge relationship* between the two devices, which begin to gossip the records comprising both series so that each device eventually holds a complete copy of each. This merging process does not actually create any new link records, but causes each device to obtain copies of the other device's existing link records (the laptop's link record for its default name `laptop` and the phone's record for its name `phone`) and incorporate those names into its own root group.

Aside from merging devices' root series via introduction, a user can use a single device to merge two arbitrary groups, provided the same device already has ownership of both groups. If Bob creates two shared sub-groups and later decides they should be combined, for example, he can merge them on any of his devices. The device writes cross-referencing merge records to the relevant series, exactly as in the introduction scenario.

If a user accidentally merges the wrong groups, the device that wrote a merge record can "undo" the merge via a cancel record written to *the same series*. Once either merge record is canceled this way, the groups return to their pre-merge state, retaining their original identities as defined by the disjoint sets of original series IDs. The restriction that a merge can only be undone on the same device results from the membership and ownership evaluation algorithm described later in Section 2.3.4, but appears a reasonable constraint for undo purposes. Cross-device "un-merge" can be approximated via revocation, as described later, but revocation cannot restore the distinctness of the original group identities: two links referring to the two separate groups before the merge refer to the same group after the revocation.

Meeting Other Users: When Bob and Alice introduce their devices to each other at Time 3, the devices exchange the series IDs of their respective root series, and each device writes a link record to its own root series referring to the other device's root series. Bob's new link record named `Alice` gives Alice a name in his personal group, and Alice's new link record named `Bob` likewise gives Bob a name in her group. The devices do not set the owner flags in these new link records, giving Alice and Bob only read-only access to each others' namespaces.

Transitive Merge: Individual merge relationships in UIA are always pairwise, between exactly two series, but merge relationships combine transitively to determine effective group membership. When Bob introduces his cell phone to his home phone at Time 4, the two devices form a merge relationship between their respective root series. Since Bob's home phone and laptop already have a merge relationship, Bob's laptop and cell phone transitively learn about each other via gossiped records they receive from the home phone, and the union of the records in the three root series determine the contents of the resulting group. Since the merged group has two link records named `phone` with different target EIDs, the devices flag a label conflict on `phone` and refuse to resolve this name.

Renaming Labels and Resolving Conflicts: When Bob renames his cell phone to `cell` at Time 5 to resolve the conflict, his device writes to its root series a cancel record containing the record ID of the link record defining the cell phone's previous name, then writes a new link named `cell` that is otherwise identical to the original link. Since one of the two conflicting link records is now canceled, the label conflict disappears, and the names `phone` and `cell` become usable on all of Bob's devices once they receive the new records via gossip. Bob can resolve the conflict on any of his devices, because any group owner can cancel a link written by another device.

The user can also delete a name from a group outright, in which case the device writes a cancel record without a new link. The ownership granted by a link-owner record, however, can only be nullified by the revocation process described later in Section 2.3.4.

Because UIA implements renames non-atomically with a cancel record coupled with a new link record, if Bob renames `Alice` to `Alice1` on his laptop and renames `Alice` to `Alice2` on his phone while the two devices are temporarily partitioned, on reconnection he will have two names `Alice1` and `Alice2` with no conflict detected. This corner-case behavior, while perhaps slightly surprising, seems acceptable since it "loses" no information and at worst requires Bob to delete one of the resulting redundant names. At the cost of added complexity, a non-idempotent "Rename" record type could be added to enable UIA to detect conflicting renames of a single original link record.

Creating Groups: Bob uses his laptop at Time 6 to create his shared `PhotoClub` group. To create the group, the laptop first writes a create record to generate a fresh series ID. The laptop then writes two link records: first, a link named `PhotoClub` in its root series pointing to the new series, and second, a link named `Bob` in the new series pointing back to the root series. The laptop sets the owner flag in only the latter link record, giving Bob's personal group ownership of the new group, *without* giving `PhotoGroup` ownership of Bob's personal group.

Suppose that Bob now uses a different device, his cell phone for example, to add Alice to `PhotoClub`. Bob's cell phone is already an indirect owner of `PhotoClub`, because the cell phone is an owner of Bob's personal group and Bob's personal group owns `PhotoClub`. The cell phone does not yet have a series in `PhotoClub`, however, to which it can write records: initially only the laptop, which created the new group, has a series in the group, and only it can sign records into that series. The cell phone therefore creates its own `PhotoClub` series, by writing a create record to form a new series owned by itself, and then writing a merge record to this new series pointing to the laptop's `PhotoClub` series. Although no corresponding merge record in the laptop's `PhotoClub` series points back to the cell phone's new series (in fact the laptop may be offline and unable to sign such a record), the cell phone's merge record takes effect "unilaterally" by virtue of the cell phone's indirect ownership of `PhotoClub`. The cell phone then writes a copy of Bob's link to Alice into its new `PhotoClub` series, and other devices learn of the new series and the new name as they gossip records for `PhotoClub`.

Revoking Ownership: When Bob learns at Time 7 that his cell phone is missing, he uses his laptop to revoke the cell phone's ownership of his personal group, either by deleting the name `cell` from his personal group or by clearing its owner flag. To implement this revocation, however, Bob's laptop cannot merely write a cancel record pointing to the link record for `cell`: the cell phone would still own a series in Bob's personal group and thus retain "hidden" control over the group.

To revoke the cell phone's ownership, therefore, Bob's laptop creates a new personal group for Bob and copies the original group's name content into it. To create the new group, the laptop writes a create record whose series ID field is not empty as usual, but instead contains the SID of the laptop's original root series. The laptop then writes link records to the new series corresponding to all the active links in the old series, omitting links or ownership flags to be revoked. The create record written into the old root series indicates to all interested devices that the new series forms a group that is intended to replace or act as a *successor* to the original group.

As long as only one such "create successor" record exists in Bob's old personal group, all devices treat links to any series in the old group as if they linked to the successor group instead. Upon receiving via gossip the records describing Bob's new group, for example, Alice's devices subsequently resolve her name `Bob` to the new group, and use it to calculate which devices should be given access to resources she has authorized Bob to access, effectively revoking the cell phone's access.

If the cell phone writes a conflicting "create successor" record to *its* series in Bob's original group, however, then the original group becomes *disputed*, and other devices refuse to resolve links to any series in the original group as soon as they learn about the dispute. Alice's devices thus refuse to resolve her name `Bob` and deny access to any resources she authorized using that name. Once Alice updates her broken link to refer to the correct successor group, either by re-introducing with Bob or by copying a fresh link from a mutual friend, her device writes a new link referring to a series in Bob's new group, the old group becomes irrelevant and Bob can again access Alice's resources via the devices in his new personal group.

If link or cancel records exist on Bob's other devices that his laptop has not yet received at the time of revocation, the laptop cannot copy these change records into the new group and they become *orphaned*. Bob's devices continue to monitor and gossip records in the old group after the revocation, however, to detect both orphans and ownership disputes. If a device with ownership of the new group detects an orphaned record written by itself or another device with ownership of the new group (not a revokee), it automatically "forwards" the change by writing a corresponding record to the new group.

Key Compromise and Retirement: Besides handling the issue of retired, lost, or stolen devices, UIA's revocation mechanism may also be useful if a device's public/private key pair is compromised and the user wishes to re-key the device, or if the public-key cryptography or hash algorithms the device's EID is built from become obsolete. In such a situation, the device can simply generate a new, fresh public/private key pair and EID for itself, merge this new EID into the existing group with the same device name as that associated with the old EID, and then perform a "self-revocation" of the device's old EID using the normal revocation mechanism to ensure that only the new EID and not the old one is considered a member of the user's personal group.

In the case of gradual cryptographic key or algorithm retirement (as opposed to a key compromise requiring immediate revocation), it may be useful to allow both the device's old and new EIDs to be active at once during some transition period. Allowing a single device name to be assigned more than one EID at once without this situation being interpreted as a conflict would require a minor extension to UIA's current group management mechanism.

2.3.4 Group State Evaluation

This section describes the algorithms UIA devices use to determine the current state of a given group from the set of log records they have on hand. Devices evaluate group state in three stages: (1) membership and ownership, (2) group successorship, and (3) name content.

Membership and Ownership

In the first stage, a UIA device collects the series IDs referred to by all records in its log, and clusters them into sets based on merge relationships to form UIA groups. At the same time, the device computes the set of device EIDs to be considered owners of each group, either directly or

```
global $M$: membership table: SID → SID set
global $O$: ownership table: SID set → EID set
function eval_membership_ownership():
  for each known series $sid$:
    $M[sid] \leftarrow \{sid\}$
    $O[\{sid\}] \leftarrow$ EID of device that owns series $sid$
  do:
    for each link-owner record in each series $sid$:
      if link target is a device $teid$:
        $O[M[sid]] \leftarrow O[M[sid]] \cup teid$
      else if target is a series $tsid$:
        $O[M[sid]] \leftarrow O[M[sid]] \cup O[M[tsid]]$
    for each merge record in each series $sid$:
      $tsid \leftarrow$ target series ID of merge record
      $O[M[sid]] \leftarrow O[M[sid]] \cup O[M[tsid]]$
      if owner EID of series $sid \in O[M[tsid]]$:
        $O[M[sid] \cup M[tsid]] \leftarrow O[M[sid]] \cup O[M[tsid]]$
        for each series ID $msid \in M[sid] \cup M[tsid]$:
          $M[msid] \leftarrow M[sid] \cup M[tsid]$
  until $M$ and $O$ stop changing
```

Figure 2-6: Membership and ownership evaluation pseudocode

transitively. Group membership and ownership must be computed at the same time because they are mutually dependent: group membership expansion via merge can introduce additional owners, and owner set expansion can place additional merge records under consideration.

Figure 2-6 shows pseudocode for membership and ownership evaluation. The algorithm uses a membership table M mapping each known series ID to a set of series IDs sharing a group, and an ownership table O mapping each group (represented by a set of series IDs) to a set of owner device EIDs. The algorithm first initializes the entry in M for each series to a singleton set containing only that series, and initializes the owner set entry in O for each such singleton group to the EID of the device that owns that series. The algorithm then repeatedly merges groups and expands ownership sets until it reaches a fixed point. The algorithm terminates because member and owner sets only grow, and each device knows of a finite number of series IDs at a given time.

The algorithm considers only *live* merge and link-owner records. A record is live if it is not targeted by any cancel record *in the same series*. The same-series restriction prevents cancel records from decreasing the sets O and M across iterations and violating the algorithm's monotonicity assumptions, hence the restrictions mentioned earlier on undo of merge and link-owner records.

In each iteration, the algorithm first follows link-owner records, expanding the ownership set of the group containing a link-owner record according to the target device EID or the current ownership set of the target group, as applicable. Across iterations, this step handles transitive propagation of ownership across multiple groups, such as Bob's laptop's ownership of `PhotoClub` via the laptop's ownership of Bob's personal group.

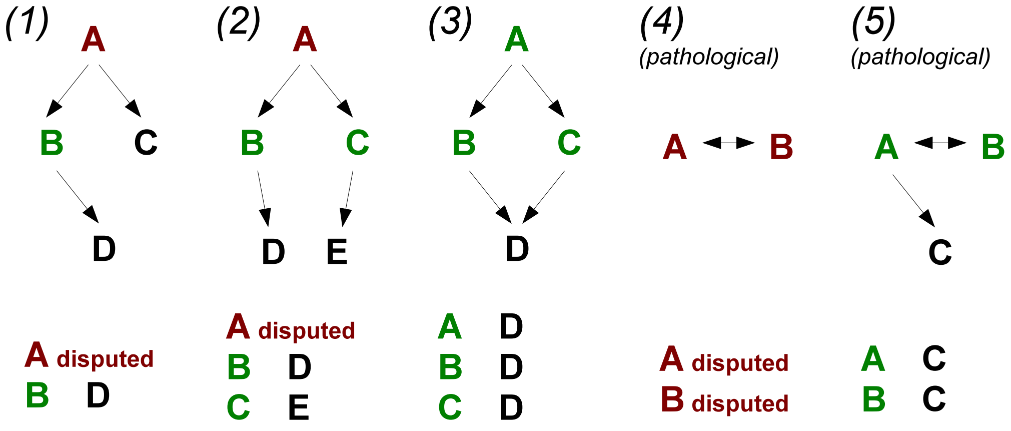


Figure 2-7: Example group successorship scenarios

Second, for each merge record, the algorithm expands the ownership set of the group containing the merge record to include the ownership set of the target group, then checks whether the device that wrote the merge record is *authorized* by virtue of having ownership of the target group. The authorization check prevents a device from merging a series into an arbitrary group without permission. In the symmetric case where two merge records refer to each others' series IDs, each merge is authorized by the fact that the other merge grants ownership of its own series to its target. Once a merge is authorized, the algorithm combines the SID sets of the respective groups to form one group containing all the merged SIDs, and similarly combines the respective owner sets.

Group Successorship

In the second stage, a device computes the *successorship* status of each group resulting from the first stage, in order to handle revocations and ownership disputes. The device first forms a directed graph reflecting immediate successor relationships: a create record in series A yielding a new series B makes the group containing B a successor to the group containing A. Next, the device takes the transitive closure of this graph to form a transitive successorship relation: if B succeeds A and C succeeds B, then C transitively succeeds A.

The device now assigns to every group G one of three states as follows. If G has no successors, it is a *head* group: no revocations have been performed in the group, and links to series IDs in the group resolve normally. On the other hand, if there is a second group G' that is a transitive successor to G and is also a transitive successor to all other transitive successors to G, then G' is the *undisputed successor* to G. In this case, links to series IDs in group G resolve to group G' instead. Finally, if G has successors but no undisputed successor, then group G is *disputed*, and links to series IDs in G do not resolve at all.

Figure 2-7 illustrates several group successorship scenarios and the corresponding results of this algorithm. In scenario (1), two conflicting revocations have placed group A under dispute; A's successor B also has a successor due a second revocation in B but B is not under dispute. Scenario

(2) is like (1) except a revocation has also been performed in group C, forming a new head group E. Scenario (3) shows the result after the warring owners in (2) settle their differences and merge their head groups D and E, resolving the original dispute over group A. Scenario (4) shows a pathological situation that should never arise under normal use but could be obtained artificially by merging A with D and B with C in (1); in this case the two groups in the cycle both become disputed. Scenario (5) shows the pathological dispute resolved by a revocation in group A. (Alternatively, the cyclic groups could simply be merged together.)

Name Content

In the third and final stage, for each head group to be used for name resolution, a device computes the group's namespace state as follows. Given the set of all link records in every series in the group, the device removes all link records targeted by a cancel record in any series of the group to form the set of *active* links. Any device that owns a group can cancel a link written by another device, but a cancel cannot revoke ownership.

The set of *active labels* in a group, shown in a namespace browser for example, is the set of labels appearing in any active link record in the group. To be *usable*, all active links for a given label must have the same permissions, and must target the same device EID or SIDs in the same group. Otherwise the label is *in conflict*, as Bob's home and cell phone are at Time 4 in the example. If Bob creates identical links on different devices independently, such as by separately introducing both his cell phone and his laptop to Alice to yield duplicate `Alice` links, this action does not create a label conflict when Bob merges his home and cell phone together because the redundant links have the same target and permissions.

2.3.5 Naming State Gossip and Replication

The UIA devices participating in a given personal group gossip and proactively replicate new records pertaining to the group among their respective logs. A UIA device by default considers itself a "participant" in a group if it owns a series in that group—i.e., if it has ownership permission and has written any records to the group.

UIA's epidemic protocol uses a classic two-phase "push/pull" algorithm [63]. In the "push" phase, when a device creates a new log record or obtains a previously unknown one from another device, it repeatedly pushes the new record to a randomly-chosen peer until it contacts a peer that already has the record. This *rumor mongering* technique works well when few devices have the record, propagating the "rumor" aggressively until it is no longer "hot." In the "pull" phase, each device periodically contacts a randomly-chosen peer to obtain any records it is missing. These *anti-entropy* exchanges work best when most devices already have a record, complementing the rumor mongering phase and ensuring that every device reliably obtains all available records.

To initiate a gossip transaction, each device periodically selects a peer and initiates a query to that device. The initiator sends a vector of *(SID, timestamp)* tuples indicating the newest records the initiator holds for each series, and the responder replies with any log records in those series that the initiator does not yet have. The responder may also include additional records relevant to the group, such as records from another series newly merged into the group.

When a device writes new records to a given series, it may opt to notify other interested devices that are reachable at that moment with the new records; otherwise, new records propagate lazily. In the current design, a device picks a random peer and gossips with it once per minute.

Devices gossip the records comprising a particular series in order, guaranteeing that the set of records each holds for a given series is complete up to some particular time in the series owner's timeline. UIA imposes no gossip ordering constraints among different series, however, so changes Bob made on his home phone and on his laptop might arrive at his cell phone out of the original order in which Bob made them. Since UIA must operate during network partitions and periods of disconnection, the potential for such misordering is unavoidable.

2.3.6 Remote Name Resolution

As outlined earlier in Section 2.3.1, a device resolves a multi-component UIA name by starting with its own device-specific root group, and resolving the labels comprising the name in right-to-left order as in DNS. When the name resolution process only traverses groups in which the resolving device participates directly, and thus for which the device keeps replicas of all the records relevant to the group in its own log, the group state calculations described above provide all the information necessary to resolve the name locally.

Resolving a name may however require traversing groups to which the resolving device only has read access, and whose records the resolving device does not proactively replicate. If Bob refers to the name `PC.Alice` on his laptop, for example, Bob's laptop may have to contact one of Alice's devices to obtain the link record required to resolve the label `PC` with respect to Alice's personal group. UIA's *remote resolution* protocol serves this purpose, and it operates similarly to traditional DNS.

UIA's remote resolution protocol does not use DNS's timeout-based cache consistency model, however, since we do not assume that users managing UIA groups understand cache consistency and know how to configure appropriate timeouts for their groups. UIA instead uses a simple lease-based protocol. When a device attempting to resolve a UIA name (the "requestor") contacts a remote device (the "responder") to resolve a particular label, the responder can return along with the lookup result a *lease* on the lookup result, or a promise to notify the requestor proactively if the result subsequently changes within a particular time period [97]. The requestor may then safely cache the result for any subsequent lookups it makes during the lease period, but flushes the result from the cache if it receives a change notification from the responder in the meantime.

Chapter 3

Routing

Regardless of whether names are global as in traditional DNS or locally scoped as in UIA's personal name system, devices must have a way to use those names to locate (*resolve*) and connect to (*route to*) their targets in order to make them useful for communication. This chapter first introduces the motivation and goals for the UIA routing layer in Section 3.1, then explores three specific approaches to routing in the remaining sections.

3.1 Motivation and Goals of UIA Routing

The Internet traditionally assumes that DNS names resolve to IP addresses, and that the IP routing mechanism alone provides the means for devices to communicate with each other via these names and IP addresses. This assumption means in practice that the target of a DNS host name must have a static, public IP address, and static IP addresses are both increasingly rare and expensive on today's commoditized Internet, and are management-intensive to assign and use even once obtained.

By default, today's mobile personal devices usually have dynamic IP addresses assigned by the Dynamic Host Configuration Protocol (DHCP) [68], and these devices are often located behind firewalls or network address translators [112], where their private IP addresses are not reachable or even globally unique outside of their private networks. Forwarding protocols like Mobile IP [183] can give a mobile device the illusion of having a static IP address independent of its actual attachment point, but this solution does not eliminate the necessity or attendant cost and difficulty of obtaining the static "home" IP address for the mobile host, and increases the cost and latency of all communication when the mobile host is away from its home location, since all traffic must be forwarded through the home address.

In UIA, whenever physically possible, we would like personal devices to provide *fully automatic connectivity* between each other whenever the user requires—especially between devices that have *social affinity* by virtue of being in the same user's personal group or nearby in the user's social network. When the user refers to a personal name, UIA should automatically find a way to communicate without requiring the user to understand or set up IP addresses or other protocol technicalities. Communication should function smoothly in a variety of scenarios such as across the

Internet, between Internet-connected private LANs, and within ad hoc networks disconnected from the Internet (e.g., among passengers in a train).

To provide this automatic connectivity, UIA devices cooperate in an *overlay routing protocol* to provide robust location-independent connectivity in the face of changing IP addresses, Internet routing failures, network address translators, or isolation from central network infrastructure. The rest of this chapter explores three specific approaches to designing this overlay routing protocol: *social routing* in Section 3.2, *identity hash routing* in Section 3.3, and *compact routing* in Section 3.4. The first approach is simple, works well in scenarios we expect to be common, and is implemented and working in the deployed UIA prototype. The other two approaches are more general and ambitious, but have as yet been validated only under simulation and will require further development before widespread deployment. All three approaches have strengths and limitations; it is not yet clear which—or what combination of ideas from each—will eventually yield the best long-term solution.

3.2 Social Routing

Because of UIA's goal of providing names that *persistently* refer to a particular device regardless of how they move or where they are attached to the Internet, UIA's endpoint identifiers (EIDs), to which UIA personal names resolve (see Section 2.3.1, cannot and do not contain any embedded information about the current *location* of the device, because that information would have to change whenever the device moves. This design contrasts with the hierarchical CIDR [199] structure of IP addresses on the Internet, in which varying-length prefixes of an IP address indicate the node's attachment point at different administrative levels, such as edge network, service provider, and network provider. The CIDR structure makes Internet routing simple and efficient, but is also the source of many of the difficulties UIA is trying to fix. Therefore, the UIA overlay routing layer must address the more difficult problem of routing over *topology-independent* or *flat* identifiers, also known as *name-independent routing* in the theory literature [2, 15].

Since efficient, scalable routing with location-independent node identities is inherently challenging in its most general form [92], we would like to find an appropriate set of simplifying assumptions that will yield a robust, efficient solution for the scenarios we primarily care about. The primary purpose of UIA's personal name system, as described in Chapter 2, is to provide connectivity with devices that are *administratively related*, by virtue of having been merged into the same personal group, or that are *socially related*, by virtue of being located in the personal groups of users who are "friends" or otherwise have personal naming relationships between their devices, and thus are likely to wish to communicate.

Given this primary purpose, the first overlay routing approach we explore—*social routing*—leverages the "social network" provided by the naming layer to reduce the scope of the routing problem, from routing between *arbitrary* devices, to routing "between friends." The total number of interconnected devices may ultimately be very large, but the number of devices in *a particular user's* personal group and the personal groups of his immediate friends should generally be much smaller, which immediately reduces the scalability challenges to a more reasonable degree. Also, since a user's own devices and those of his friends are likely to be both more trustworthy and more willing to spend resources forwarding the user's packets if needed, the naming layer's social network provides a *trust framework* that the routing layer can take advantage of.

The social routing protocol we develop here is optimized for connecting to devices in the user's immediate *social neighborhood*: primarily the user's own devices and those of friends named in the user's personal group, and occasionally "friends of friends," but rarely more indirect contacts. In practice we expect users to create (or copy from other users) names in their own personal groups for others with whom they wish to interact regularly, justifying our assumed usage model.

In brief, a UIA device builds and maintains an overlay network between itself and other devices nearby in its social neighborhood. To locate a remote device by its EID, a device floods a *location* request through the overlay to discover the EIDs, IP addresses, and ports of devices forming a path through the overlay to the target. The originating device then connects directly to the target's discovered IP address and port, or if the target is not directly reachable (e.g., because of an intervening NAT), *forwards* traffic to it by source-routing data via existing connections in the discovered path.

3.2.1 Overlay Construction and Maintenance

Each UIA device maintains an open TCP connection with up to a configurable number of overlay *peers*. A device chooses its peers from the larger set of devices in its social network based on a number of criteria. Ideally, a device's peers should be on the public Internet, so that a device behind a NAT can receive messages from devices outside via its active peering connections. A device should choose other devices when none on the public Internet are reachable, however, so that the overlay remains useful in ad hoc environments. Furthermore, the devices of friends should be close to each other in the overlay, so that location or forwarding paths between them are short.

To meet these goals, a device first prefers as peers devices that are *stable*, and secondarily prefers those that are closest to it in *friendship distance*. A device is considered *stable* if it does not have a private IP address [196] and has met a threshold level of availability in the recent past. A peer's *friendship distance* is roughly the number of labels in the local device's shortest name for that peer. The rest of this section explains how a device discovers stable peers and calculates friendship distances.

Each device maintains a *potential peer set* that contains potential peers' EIDs and the times, IP addresses, and ports at which the device has connected to those peers in the past. Initially, a device populates this set with the devices to which the user has directly introduced the device. To discover new potential peers, a device periodically exchanges its potential peer set with those of other devices within a configurable maximum friendship distance. A device adds to the set only those devices to which it is able to establish a TCP connection when it discovers them.

A device classifies a potential peer as *stable* if it meets an availability threshold (e.g., 90%) at the same public IP address and port in the recent past (e.g., the last week). To monitor availability, a device periodically chooses a random potential peer and attempts a connection to its last known location. A device need not have a static IP address to be classified as stable: a device with a dynamic non-private IP address that changes infrequently, such as a home PC left on and connected via a DSL or cable modem, will also typically be classified as stable.

A device computes the *friendship distance* of each of its potential peers by assigning a distance of 1 to its *direct peers*: those the naming layer identifies as devices in the user's personal group and in groups to which the user has linked (the user's immediate friends). The device then assigns distances to indirect peers transitively, giving the direct peer of a direct peer a distance of 2, for example.

To improve robustness, a device manufacturer can seed the potential peer sets of its products with a set of *default peers*, which devices treat as having an infinite friendship distance. Two newly-purchased mobile devices, after being introduced and exchanging potential peer sets, thus have at least one stable peer in common at the outset to help them re-connect after a move. Once the mobile devices discover other stable peers at smaller friendship distances, however, they prefer the new devices over the default peers, mitigating the manufacturer's cost in providing this robustness-enhancing service.

3.2.2 Token-limited Flooding

To communicate with a remote device, a device first attempts a direct TCP connection to the IP address and port at which it last connected to the target, if any. If this connection fails or the originator has no address information for the target device, it floods a location request through the overlay to locate the target by its EID.

UIA uses a *token count*, in place of the traditional hop count [42], to limit the scope of location request floods. The token count bounds the total number of *devices* to which a request may be forwarded, rather than the number of times each request may be re-broadcast. This distinction is important for two reasons. First, although devices seek to connect with a fixed number of peers, the number of devices that choose a given device depends on the target's stability and popularity, so the overlay's degree is highly non-uniform. Hop count is thus a poor predictor of the number of devices a request will reach. Second, the overlay network is highly redundant: two friends' devices are likely to share many common peers, for example, so searching *all* devices within some distance of a request's source is often unnecessary.

Location requests contain the EIDs, IP addresses, and ports of devices they have traversed; devices forward responses back through the overlay along the same path.

A device with an open TCP connection to a request's target immediately responds with the target's IP address and port. Otherwise, it subtracts one token for itself, divides the other tokens among its peers not already in the path, distributing any remainder randomly, and forwards the request to those peers that receive a non-zero count. The device retains the request's target EID and return path for a short period, waiting for the forwarded requests to complete, and replying to the original request when *any* of the forwarded ones succeed or when *all* of them have failed. A request also fails if the source has not received a successful response within a timeout. If a device receives a duplicate request for the same EID as an outstanding request (e.g., along a different path), it forwards the new request anyway according to its token count, giving peers for which there were not enough tokens in previous instances another chance to receive the request.

As we find in Section 7.2, most location requests succeed within the near vicinity of the source in the overlay network. To limit the cost of the search, a device thus initially sends each request with a limited number of tokens and retries after each failure with a multiplicatively increased number, up to some maximum.

3.2.3 Source-Routed Forwarding

To communicate with the target device after receiving a successful location response, the originator tries to open a direct connection to each device in the response path, starting with the target itself and proceeding backwards along the path until a connection succeeds. In the best case, the first connection attempt in this sequence—the one directly to the target device—succeeds, and no forwarding is necessary; in this case the UIA overlay routing layer merely functions as a "resolver," mapping location-independent EIDs to location-dependent IP addresses. If the first, direct connection attempt fails, however, the originator source-routes messages to the target along the tail of the path starting with the device to which it successfully connected directly.

Consider for example two devices a and b behind different NATs, both of which peer with a common stable device s. When a performs a location request for b's EID, it discovers the path $a \to s \to b$. Device a then tries to open a direct connection to b, but b's NAT blocks that connection, so a forwards traffic to b through s instead. Device s itself initiates no location requests, but merely forwards traffic along the path specified by a.

3.3 Identity Hash Routing

The social routing protocol described in the last section achieves scalability by assuming that users mostly wish to communicate with their friends, an assumption that is likely to hold in many scenarios but not all. If the user wishes to run applications on top of UIA that depend on large-scale self-organizing protocols, such as swarm downloading [55], such protocols frequently require communication between arbitrary nodes that are administratively and socially unrelated. Thus, we would like to develop a more general overlay routing protocol for UIA that efficiently provides scalable routing between *any* indirectly connected pair of UIA nodes, not just between friends.

This section introduces Identity Hash Routing (IHR), a routing scheme inspired by Distributed Hash Table (DHT) algorithms [146,156,194,209,234,271]. DHTs normally provide only *resolution* service—the ability to look up a value in a self-organizing distributed structure given its key or content hash—and assume that the network in which they are operating is fully-connected: i.e., that every node can communicate directly with every other node on demand. But many nodes on today's Internet cannot be reached directly except via forwarding or NAT traversal through other nodes, and the rules determining which nodes can and can't directly reach which others are complex, subtle, and dynamic results of the interplay between interdomain and intradomain routing, firewall policies, NATs, and many other factors. In a traditional DHT, when many member are persistently reachable by some DHT members but not others, this condition causes widespread "disagreement" in the DHT about which nodes are alive and which aren't, which can prevent convergence and make lookups unreliable [89].

With UIA routing, in contrast, we want each node to be able to find and connect to any other *even if many pairs of nodes are connected only indirectly*, requiring explicit forwarding through intermediate nodes (or NAT traversal) to facilitate that communication. Nevertheless, we find in this section that it should be feasible to adapt DHT lookup algorithms into scalable overlay routing schemes. Unlike traditional DHTs, IHR does not assume that underlying protocols provide connectivity between any two nodes. When the underlying network fails to provide direct connectivity for any reason, such as intermittent glitches, network address translators, or incompatible address-based routing technologies, we want IHR to route around these discontinuities by forwarding traffic through other IHR nodes.

Key Properties of IHR

The crucial scalability property IHR provides is that it efficiently allows each UIA node to find a route to *any* of a large number N of total nodes in the connected network, while directly storing information about (and routes to) only about $O(log\ N)$ other nodes. Like the DHT algorithms it builds on, IHR achieves this scalability by distributing routing information throughout the network in a self-organizing structure: in particular, IHR uses a structure adapted from the Kademlia DHT [156].

The cost of distributing routing information throughout the network for scalability is that individual IHR nodes rarely have enough information to determine the shortest or "best" possible route to another node. In effect, IHR does not implement a distributed "all-pairs shortest paths" algorithm like conventional protocols for flat namespaces do [126]. In fact, it is known to be impossible to achieve shortest-path routing with $o(N)$ state per node [92]. Instead, IHR attempts the more moderate goal of *efficiently* finding *some* path whenever one exists, and usually finding "reasonably short" paths. This goal is appropriate for the UIA routing layer because its purpose is to provide *some* usable communication path in the unfortunate situations when IP cannot find any (e.g., due to the devices being in different IP address domains).

In general we cannot expect IHR to be as efficient as routing protocols that take advantage of the locality and aggregation properties of structured addresses. IHR is not intended to replace address-based routing protocols, but to complement them. By using address-based protocols such as IP to move data efficiently across the many "short" hops comprising the core Internet infrastructure and other large managed networks, IHR only needs to route data across across a few "long" hops, resolving the discontinuities between address domains and bridging managed core networks to ad hoc edge networks. For this reason, it is less important for IHR to find the best possible route all the time, and more important for the algorithm to be scalable, robust, and fully self-managing.

We explore two specific IHR forwarding mechanisms based on the same routing protocol. One mechanism guarantees that nodes can operate in $O(log\, N)$ space per node on any network topology. The other forwarding mechanism allows IHR to find somewhat better routes and still uses $O(log\, N)$ space on typical networks, but may require $O(N)$ space on worst-case network topologies. With either forwarding mechanism, simulation results presented later in Chapter 7 indicate that IHR consistently finds paths that are on average within $2\times$ the length of the best possible path. IHR occasionally chooses paths that are much longer than the best possible path, but these bad paths are rare.

3.3.1 Routing Protocol Design

This section describes the distributed lookup and routing structure that enables IHR nodes to locate and communicate with each other by their topology-independent identities.

Neighbors and Links

Each node in a IHR network maintains a *neighbor table*, in which the node records information about all the other IHR nodes with which it is actively communicating at a given point in time, or with which it has recently communicated. The nodes listed in the neighbor table of a node A are termed A's *neighbors*. A neighbor of A is not necessarily "near" to A in either geographic, topological, or node identifier space; the presence of a neighbor relationship merely reflects ongoing or recent pairwise communication.

Some neighbor relationships are mandated by the design of the IHR protocol itself as described below, while other neighbor relationships are initiated by the actions of upper-level protocols. For example, a request by an upper-level protocol on node A to send a packet to some other node B effectively initiates a new IHR neighbor relationship between A and B. These neighbor relationships may turn out to be either ephemeral or long-term. A IHR node's neighbor table is analogous to the table an IPv4 or IPv6 host must maintain in order to keep track of the current path maximum transmission unit (MTU) and other vital information about other endpoints currently or recently of interest to upper-level protocols.

As a part of each entry in a node's neighbor table, the node's IHR implementation maintains whatever information it needs to send packets to that particular neighbor. This information describes a *link* between the node and its neighbor. A link between two nodes A and B may be either physical or virtual. A *physical link* is a link for which connectivity is provided directly by the underlying protocol (IP). For example, if A and B are both well-connected nodes on the Internet that can successfully communicate via their public IP addresses, then AB is a physical link from the perspective of the IHR layer, even though this communication path may in reality involve many hops at the IP layer and even more hops at the link layer. If a physical link is available between A and B, then A and B are termed *physical neighbors*, and each node stores the other's IP address or other address information for underlying protocols in the appropriate entry of its neighbor table.

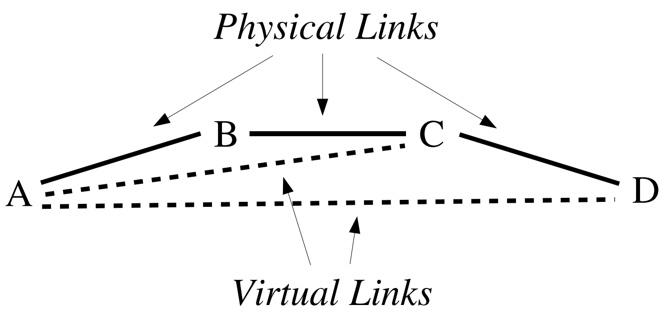


Figure 3-1: Forwarding via virtual links

A *virtual link*, in contrast, is a link between two nodes that can only communicate by forwarding packets through one or more intermediaries at the IHR level. We describe such nodes as *virtual neighbors*. The mechanism for IHR-layer packet forwarding and the contents of the neighbor table entries for a node's virtual neighbors will be described later in Section 3.3.2. For now, however, we will simply assume that the following general principle holds. Given any two existing physical or virtual links AB and BC with endpoint B in common, nodes A and C can construct a new virtual link AC between them by establishing an IHR-level forwarding path through B. That is, IHR nodes can construct new virtual links recursively from existing physical and virtual links.

In Figure 3-1, for example, virtual link AC builds on physical links AB and BC, and virtual link AD in turn builds on virtual link AC and physical link CD. Once these virtual links are set up, node A has nodes B, C, and D in its neighbor table, the last two being *virtual neighbors*. Node D only has nodes C and A as its neighbors; D does not necessarily need to know about B in order to use virtual link AC.

Constructing Virtual Links

IHR nodes construct new virtual links with a single basic mechanism, represented by the **build_link** procedure shown in Figure 3-2. A node n can only build a virtual link to some other node n_t if n already has some "waypoint" node n_w in its neighbor table, and n_w already has n_t in its neighbor table respectively. Node n can then use the **build_link** procedure to construct a link from n to n_t.

In the **build_link** procedure, n first attempts to initiate a direct connection to n_t via the underlying routing protocol, using any network- or link-layer address(es) for n_t that n may have learned from n_w. For example, if n_t is a node with several network interfaces each in different address domains, then n_t might publish both the IP addresses and the IEEE MAC addresses of all of its network interfaces, so that other IHR nodes in any of these domains can initiate direct connections with n_t even if they don't know exactly which domain they are in. If at least one of these direct connection attempts succeeds, then n now has n_t as a physical neighbor, and a virtual link is not necessary.

If all direct connection attempts fail (or do not succeed quickly enough), however, then n constructs a virtual link to n_t using n_w as a forwarding waypoint. In this way, the **build-link** procedure

```
// build a link from node $n$ to target node $n_t$,
// using node $n_w$ as a waypoint if necessary
$n$.build_link($n_w, n_t$) {

    assert ($n$ and $n_w$ are neighbors)
    assert ($n_w$ and $n_t$ are neighbors)

    try to contact $n_t$ by its IP address, MAC address, etc.
    if direct contact attempt succeeds {
        build physical link from $n$ to $n_t$
    } else {
        build virtual link from $n$ to $n_t$ via $n_w$
    }

    assert ($n$ and $n_t$ are neighbors)
}
```

Figure 3-2: Pseudocode to build a physical or virtual link

takes advantage of underlying connectivity for efficiency whenever possible, but succeeds even when only indirect connectivity is available.

IHR Network Structure

While virtual links provide a basic forwarding mechanism, IHR nodes must have an algorithm to determine *which* virtual links to create in order to form a communication path between any two nodes. For this purpose, all connected IHR nodes in a network self-organize into a distributed structure that allows any node to locate and build a communication path to any other by resolving the target node's identifier one bit at a time from left to right. The IHR network structuring algorithm is closely related to peer-to-peer distributed hash table (DHT) algorithms such as Pastry [209] and Kademlia [156]. Unlike DHTs, however, IHR uses this self-organizing structure not only to look up information such as the IP or MAC address(es) of a node from its IHR identifier, but also as a basis for *constructing* IHR-level forwarding paths between nodes for which underlying protocols provide no direct connectivity.

For simplicity of exposition we will assume that each node has only one identifier, each node's identifier is unique, and all identifiers are generated by the same l-bit hash function. UIA's endpoint identifiers (EIDs) already meet these requirements. We will treat IHR node identifiers as opaque l-bit binary bit strings. The *longest common prefix* (LCP) of two nodes n_1 and n_2, written $lcp(n_1, n_2)$, is the longest bit string prefix common to their respective IHR identifiers. The *proximity* of two nodes $prox(n_1, n_2)$ is the length of $lcp(n_1, n_2)$: the number of contiguous bits their identifiers have in common starting from the left. For example, nodes 1011 and 1001 have an LCP of 10 and a proximity of two, while nodes 1011 and 0011 have an empty LCP and hence a proximity of zero.

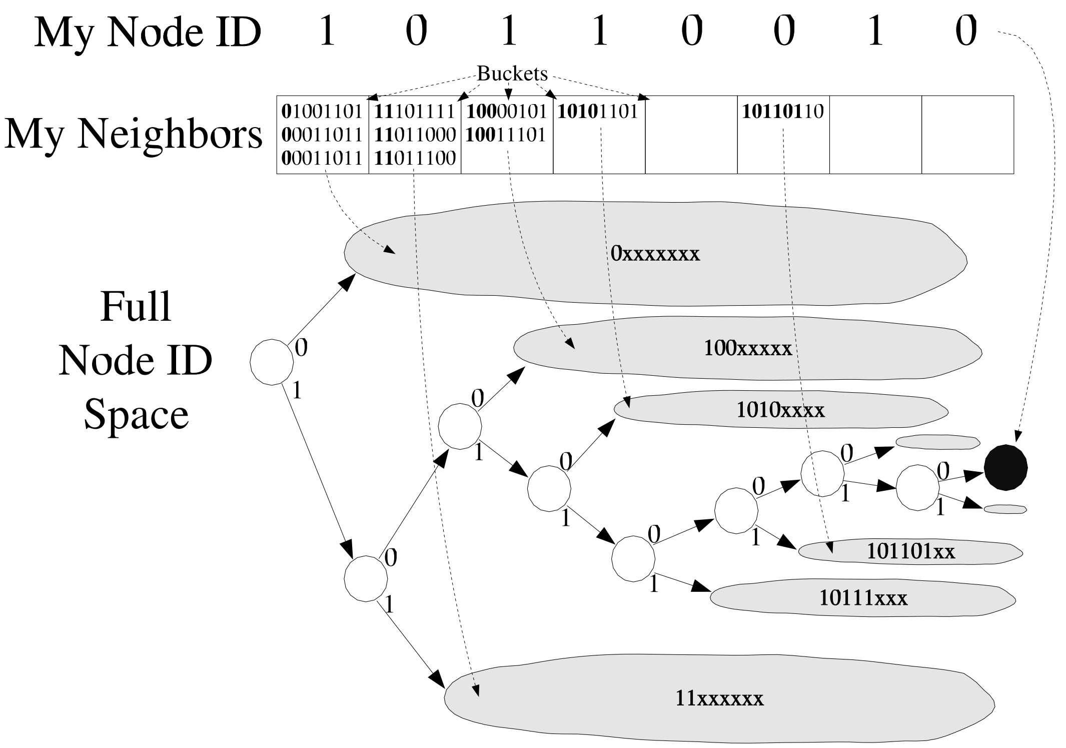


Figure 3-3: Neighbor tables, buckets, and node ID space

Nodes that are “closer” in identifier space have a higher proximity. Since node identifiers are unique, $0 \leq prox(n_1, n_2) < l$ if $n_1 \neq n_2$, and $prox(n, n) = l$.

Each node n divides its neighbor table into l buckets, as illustrated in Figure 3-3, and places each of its neighbors n_i into bucket $b_i = prox(n, n_i)$ corresponding to that neighbor’s proximity to n. This distance metric, also known as the XOR metric [156], has the important symmetry property that if node n_2 falls into bucket b of node n_1’s neighbor table, then n_1 falls into bucket b of n_2’s neighbor table. This symmetry facilitates the establishment of pairwise relationships between nodes, and allows both nodes in such a relationship to benefit from requests flowing between them in either direction.

In order for a IHR network to be fully functional, the network must satisfy the following *connectivity invariant*. Each node n perpetually maintains an active connection with at least one neighbor in every bucket b, as long a reachable node exists anywhere in the network that could fit into bucket b. In practice each node attempts to maintain at least k active neighbors in each bucket at all times, for some redundancy factor k.

Building Communication Paths

If the connectivity invariant is maintained throughout an IHR network, then any node n can communicate with any target node n_t by the following procedure, outlined in pseudocode in Figure 3-4.

```
// build a communication path from node $n$
// to target node $n_t$
$n$.**build_path**$(n_t)$ {
   $i = 1$
   $b_1 = prox(n, n_t)$
   $n_1 = n$.neighbor_table$[b_1]$
   while $(n_i \neq n_t)$ {
      $b_{i+1} = prox(n_i, n_t)$
      assert $(b_{i+1} > b_i)$

      $n_{i+1} = n_i \rightarrow$ find_neighbor_in_bucket $(b_{i+1})$
      if find_neighbor_in_bucket request failed {
         return *failure*: node $n_t$ does not exist or is not reachable.
      }

      $n$.build_link$(n_i, n_{i+1})$
      assert ($n_{i+1}$ is now $n$'s neighbor)

      $i = i + 1$
   }
   return *success:* we now have a working link to $n_t$.
}
```

Figure 3-4: Pseudocode to build a path to any node

Node n first looks in bucket $b_1 = prox(n, n_t)$ of its own neighbor table. If this bucket is empty, then n_t does not exist or is not reachable, and the search fails. If the bucket contains n_t itself, then the target node is already an active neighbor and the search succeeds. Otherwise, n picks any neighbor n_1 from bucket b_1. Since n_1's and n_t's proximity to n are both b_1, the first b_1 bits of n_1 and n_t match those of n's identifier, while their immediately following bits are both opposite that of n. The proximity of n_1 to n_t is therefore at least $b_1 + 1$.

Node n now sends a message to n_1 requesting n_1's nearest neighbor to n_t. Node n_1 looks in bucket $b_2 = p(n_1, n_t)$ in *its* neighbor table, and returns information about at least one such node, n_2, if any are found. The information returned includes the IHR identifier of the nodes found along with any known IP addresses, IEEE MAC addresses, or other underlying protocol addresses for those nodes. Node n then uses the **build_link** procedure in Figure 3-2 to establish a connection to n_2, via a direct physical link if possible, or a virtual link through n_1 otherwise.

Now n_2 is also an active neighbor of n, falling into the same bucket of n's neighbor table as n_1 but closer in proximity to n_t. The original node n continues the search iteratively from n_2, resolving at least one bit per step and building additional recursive virtual links as needed, until it finds the desired node or the search fails. If the search eventually succeeds, then n will have n_t as an active (physical or virtual) neighbor and communication can proceed.

In practice, nodes can improve the robustness and responsiveness of the **build_path** procedure by selecting a set of up to k neighbor nodes at each iteration and making **find_neighbor** requests to all of them in parallel, in much the same way that Kademlia parallelizes its DHT lookups. Parallelizing the construction of IHR communication paths has the added benefit that the originating node is likely to end up having discovered several alternate paths to the same node. The originating node can evaluate these alternative paths using some suitable criteria, choose the best of them for subsequent communication, and keep information about the others stored away for use if the primary path fails. The two endpoint nodes can even balance their traffic load across these paths if they can find reason to believe that the paths are sufficiently independent for load-balancing to be effective in improving overall performance.

The Merge Procedure

The above **build_path** procedure is much like the lookup procedure used in the Kademlia DHT, modified to support construction of indirect forwarding paths between nodes that cannot communicate directly via underlying protocols. For network construction and maintenance, however, IHR requires a much more robust algorithm than those used in Kademlia and other DHTs. DHTs generally assume not only that underlying protocols provide full any-to-any connectivity between nodes, but also that nodes join or leave the network at a limited rate and relatively independently of each other. In the discontinuous network topologies on which IHR is intended to run, however, a single broken link can split the network at arbitrary points, causing the nodes in either partition to perceive that all the nodes in the other partition have disappeared *en masse*. If the network split persists for some time, the nodes on either side will re-form into two separate networks, which must somehow be merged again once the networks are re-connected.

IHR assumes that underlying protocols provide some means by which topologically near IHR nodes can discover each other and establish physical neighbor relationships. For example, IHR nodes might use Ethernet broadcasts IPv4 subnet broadcasts, or IPv6 neighbor discovery to detect nearby neighbors automatically. The current UIA prototype depends on Bonjour [11] to provide this local-area discovery function.

Nodes might also contain “hard-coded” IP addresses of some well-known IHR nodes on the Internet, so that nodes with working Internet connections can quickly merge into the public Internet-wide IHR network. Finally, the user might in some cases explicitly provide the address information necessary to establish contact with other relevant IHR nodes. Whenever a new physical link is established by any of the above means, the node on each end of the link performs the **merge** procedure outlined in Figure 3-5, to merge itself into the network reachable from the other node.

The merge process works as follows. Suppose that node n has node n_1 as a neighbor, falling in bucket $b_1 = p(n, n_1)$ in its neighbor table. If $b_1 > 0$, then n and n_1 have one or more initial identifier bits in common, and any neighbors of n_1 in buckets 0 through $b_1 - 1$ are also suitable for the corresponding buckets in n's neighbor table. Node n therefore requests information from n_1 about about at least one of n_1's neighbors in each of these buckets, and builds a physical or virtual (via n_1) link to that node. Assuming n_1's neighbor table satisfied the connectivity invariant, n's neighbor table now does as well for buckets 0 through $b_1 - 1$.

```
// merge node $n$ into the portion of a network
// reachable from neighbor $n_1$
$n$.**merge**$(n_1)$ {
  $i = 1$
  $b_1 = prox(n, n_1)$
  while $(b_i < l)$ {

    for $j = 0$ thru $(b_i - 1)$ {
      if $n$.neighbor_table$[j]$ not already full {
        $n_j = n_i \rightarrow$ find_neighbor_in_bucket $(j)$
        if find_neighbor_in_bucket request succeeded {
          $n$.build_link$(n_i, n_j)$
        }
      }
    }

    $n_{i+1} = n_i \rightarrow$ find_neighbor_in_bucket $(b_i)$
    if find_neighbor_in_bucket request failed
      break
    $b_{i+1} = prox(n, n_{i+1})$
    assert $(b_{i+1} > b_i)$

    $n$.build_link$(n_i, n_{i+1})$
    $i = i + 1$
  }
}
```

Figure 3-5: Pseudocode to merge a node into a network

Node n now asks n_1 for any neighbor from n_1's bucket b_1 other than n itself, as if n was searching for its own identifier in n_1's network. If such a node n_2 is found, then its proximity $b_2 = p(n, n_2)$ must be at least $b_1 + 1$. Node n builds a link to n_2 via n_1, fills any empty buckets $0 < b_i < b_2$ from n_2's neighbor table as above, and then continues the process from n_2 for neighbors with proximity greater than b_2. Eventually n reaches some node n_i with proximity b_i, whose bucket b_i contains no neighbors other than n itself. This means that there are no other nodes in n_1's network with greater proximity to n than p_i, and so n has satisfied the connectivity invariant in its own neighbor table, at least with respect to the portion of the network reachable from n_1.

Merge Notifications

After a node n merges into another node n_1's network via the **merge** procedure above, however, there may be other nodes in n_1's network besides the ones that n contacted directly that also need to learn about n before *their* neighbor tables will satisfy the connectivity invariant for the new, larger network. In addition, n may not be just a "lone" node joining n_1's network, but may instead be a member of a larger existing network (reachable from n's neighbor table) that previously split from or evolved independently from n_1's network. In this case, many nodes in n's network may need to learn about nodes in n_1's network, and vice versa, before the connectivity invariant will be re-established globally.

To cause other nodes to update their neighbor tables appropriately, IHR uses a simple notification mechanism. Whenever a node n makes contact for any reason with a new physical or virtual neighbor n_n, and bucket $b_n = prox(n, n_n)$ of n's neighbor table *was not full* before the addition of n_n, n sends a message to all of its existing neighbors notifying them of the new node n_n. In response to this notification message, each of n's existing neighbors n_i contacts n_n via n_i.build_link(n, n_n), and then likewise merges into n_n's network via n_i.merge(n_n). If this process helps n_i to fill any of its previously underfull neighbor table buckets, then n_i subsequently sends notifications to *its* neighbors, and so on. The chain reaction stops when all of the affected nodes cease finding new nodes that fit into underfull buckets in their neighbor tables.

To understand this process, consider two initially separate IHR networks: a "red" network consisting of i nodes $r_1 \dots r_i$, and a "green" network consisting of j nodes $g_1 \dots g_j$. We say that any given node n satisfies the *red connectivity invariant* if each bucket in n's neighbor table contains at least one red node if any red node exists that could fit into that bucket. Similarly, we say that a node n satisfies the *green connectivity invariant* if each of n's buckets contains at least one green node if any green node exists that could fit into that bucket. We assume that all green nodes initially satisfy the green connectivity invariant, but no green nodes satisfy the red connectivity invariant because there are initially no connections between the red and green networks. Similarly, all red nodes satisfy the red connectivity invariant but no red nodes satisfy the green connectivity invariant.

Now suppose that a physical link is somehow established between nodes r_1 and g_1, connecting the two networks. In response, r_1 performs a **merge**(g_1), filling any underfull buckets in its neighbor table that can be filled from green nodes reachable from g_1, and g_1 likewise performs a **merge**(r_1) to fill its buckets from nodes in the red network. Node r_1 effectively locates and builds links with its nearest (highest-proximity) neighbors in the green network, and g_1 likewise locates and builds links with its nearest neighbors in the red network. As a result, after the merge process r_1 satisfies the green connectivity invariant and g_1 satisfies the red connectivity invariant. Since r_1 and g_1 already satisfied the red and green invariants, respectively, and adding new neighbors to a node's neighbor table cannot "un-satisfy" a previously satisfied connectivity invariant, both r_1 and g_1 now satisfy the *global* connectivity invariant covering both red and green nodes.

Assuming node identifiers are reasonably uniformly distributed, with high probability one or both of r_1 and g_1 will find one or more new nodes in the opposite network that fit into previously underfull buckets. Before the merge, bucket $b = prox(r_1, g_1)$ in both r_1 and g_1 may already have been full, which is likely if r_1 and g_1 are far apart in identifier space. There may even be *no* nodes in the green network that fall into underfull buckets in r_1, but this event is unlikely unless the green network is much smaller than the red network. Similarly, there may be no nodes in the red network that fall into underfull buckets in g_1, but only if the red network is much smaller than the green network. If the two networks are similar in size, then *both* r_1 and g_1 will almost certainly find new neighbors that fit into underfull buckets.

At any rate, the discovery of new neighbors falling in these underfull buckets causes r_1 and/or g_1 to send merge notifications to their existing neighbors in the red and green networks, respectively, supplying a link to the opposite node as a "hint" from which other nodes in each network can start their merge processes. Each node in either network that is notified in this way initiates its own **merge** process to fill its neighbor table from nodes in the other network, in the process triggering the merge process in its other neighbors, eventually leaving all nodes satisfying the global connectivity invariant.

In practice it is important to ensure that the inevitable flurry of merge notifications does not swamp the whole network, especially when two relatively large networks merge. Standard protocol engineering solutions apply to this problem, however, such as rate-limiting the acceptance or spread of notifications, propagating merge notifications periodically in batches, and keeping a cache in each node of recently-seen merge notifications to avoid performing the same merge many times in response to equivalent merge notifications received from different neighbors.

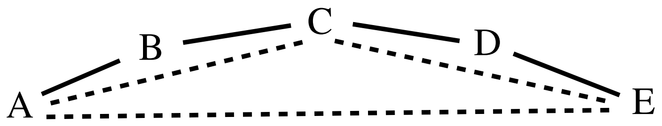
3.3.2 Packet Forwarding

The previous section described how IHR nodes form a self-organizing structure in which any node can build a communication path to any other node by recursively constructing virtual links on top of other links, but did not specify exactly how virtual links operate. In this section we explore the construction and maintenance of virtual links in more detail. We will explore in particular two alternative methods for implementing virtual links: one based on source routing, the other based on recursive tunneling. Source routing potentially enables nodes to find more efficient routes and keeps the basic forwarding mechanism as simple as possible, while the recursive tunneling approach minimizes the amount of state each node must maintain in its neighbor table.

Source Routing

With source routing, each entry in a node's neighbor table that represents a virtual neighbor contains a complete *source route* to the target node. The source route lists the IHR identifiers of a sequence of nodes, starting with the origin node and ending with the target node, such that each adjacent pair in the sequence has (or recently had) a working *physical* link between them. Of course, since these links need only be "physical" from the perspective of the IHR layer, each link in a IHR source route may represent many hops at the IP routing or link layers.

Consider for example Figure 3-6, in which the five nodes A, B, C, D, E are connected by a chain of physical links. Nodes A and C have established a virtual link AC by building a two-hop source route via their mutual neighbor B, and nodes C and E have similarly established a virtual link CE via D. Suppose node A subsequently learns about E from C and desires to create a virtual link AE via C. Node A contacts C requesting C's source route to E, and then appends C's



Source_Routing

A's Neighbor Table:

B	Phys	123.45.67.89
C	Virt	A→B→C
E	Virt	A→B→C→D→E

"→" Refers to
physical links only

Tunneling

A's Neighbor Table:

B	Phys	123.45.67.89
C	Virt	A⇨B⇨C
E	Virt	A⇨C⇨E

"⇨" Refers to
physical or virtual links

Figure 3-6: Source routing versus recursive tunneling

source route for CE (A, B, C) to A's existing source route for AC (C, D, E), yielding the complete physical route A, B, C, D, E.

To send a packet to E, node A includes in the packet's IHR header the complete source route for the virtual link AE stored in its neighbor table entry for E. Each IHR node along the path examines the header to find the packet's current position along its path, and bumps this position indicator to the next position before forwarding the packet to the next IHR node in the path. Forwarding by source routing in IHR is thus essentially equivalent to source routing in IP [120].

In theory each node may have to store up to $l \times k$ entries in its neighbor table, where l is the node identifier size and hence the number of buckets in the neighbor table, and k is the redundancy factor within each bucket. In practice only the top $log_2\ N$ buckets will be non-empty, where N is the total number of nodes in the network. With source route forwarding, neighbor table entries may have to hold source routes for paths up to $N - 1$ hops in length, in the worst-case network topology of N nodes connected together in one long chain. In this case each node may require $O(N\ log\ N)$ storage. In practical networks these source routes will of course be much shorter, so this large worst-case storage requirement may not be a problem.

Recursive Tunneling

In contrast with source routing, where each entry in a node's neighbor table for a virtual neighbor contains a complete, explicit route that depends only on physical links, recursive tunneling preserves the abstraction properties of neighbor relationships by allowing the forwarding path describing a virtual link to refer to both physical and (other) virtual links. As a result, each neighbor table entry representing a virtual link needs to hold only two IHR identifiers: the identifier of the target node, and the identifier of the "waypoint" through which the virtual link was constructed. Recursive

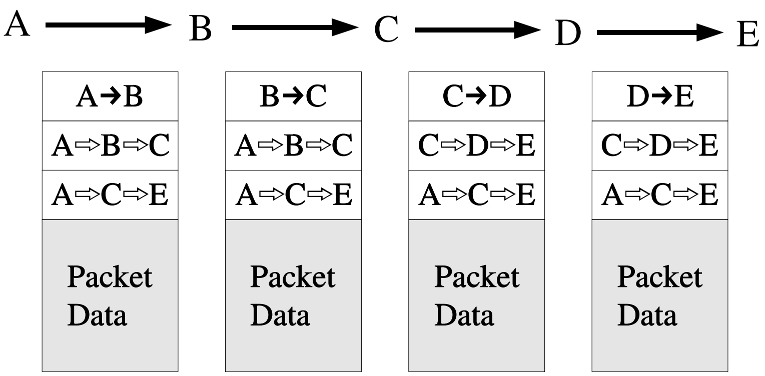


Figure 3-7: Forwarding by Recursive Tunneling

tunneling therefore guarantees that each node requires at most $O(log\ N)$ storage, since neighbor table entries have constant size.

In the example in Figure 3-6, node A has constructed virtual link AC via B, and node C has constructed virtual link CE via D, and as before, A learns about E from C and wants to construct a virtual link AE via C. With recursive tunneling, A does not need to duplicate its route C or ask C for information about its route to E in order to construct its new virtual link to E. Instead, A merely depends on the knowledge that it already knows how to get to C, and that C knows how to get to E, and constructs a neighbor table entry for E describing the "high-level" two-hop forwarding path A, C, E.

Recursive tunneling has several beneficial properties. First, since each neighbor table entry for a virtual neighbor needs to store only two IHR identifiers, the size of each neighbor table entry can be limited to a constant, and the size of a node's entire neighbor table depends only on the size of IHR identifiers (and hence the number of buckets), and the number of entries in each bucket. Second, if "low-level routes" in the network change, all "higher-level routes" that are built on them will immediately use the correct, updated information with no information propagation delays. For example, if node D above goes down making the path C, D, E unavailable, but C finds an alternate route to E, then the virtual link AE will automatically use this new route without A even having to be aware that something in C's neighbor table changed.

The actual packet forwarding mechanism for recursive tunneling is of course slightly more involved than for source routing. As illustrated in Figure 3-7, to send a packet to E, node A wraps the packet data in three successive headers. First, it prepends a IHR tunneling header describing the "second-level" virtual path from A to E via C. Only nodes C and E will examine this header. Second, A prepends a second IHR tunneling header describing the "first-level" virtual path from A to C via B. Finally, A prepends the appropriate lower-layer protocol's header, such as an IP or Ethernet header, necessary to transmit the packet via the physical link from A to B.

When the packet reaches node B, B strips off the lower-layer protocol header, and looks in the first-level (outer) IHR tunneling header to find the IHR identifier of the next hop. B then looks up this identifier in its neighbor table, prepends the appropriate (new) lower-layer protocol header, and transmits the packet to C.

When the packet reaches node C, C strips off *both* the lower-layer protocol header and the first-level IHR tunneling header (since C was the destination according to that header), and examines the second-level tunneling header to find the final destination, E. C now looks up E in its neighbor table and, finding that E is a first-level virtual neighbor, C prepends a new first-level tunneling header describing the route from C to E via D. Finally, C prepends the lower-layer protocol header for the physical link from C to D and forwards the message to D. D subsequently forwards the message to E, which finally strips off the lower-layer protocol header and both of the tunneling headers before interpreting the packet data.

Path Optimization

When an upper-layer protocol on one node attempts to contact some other node via IHR, the **build_path** procedure described in Section 3.3.1 searches the network structure for the requested node identifier, and in the process may build one or more virtual links using the **build_link** procedure of Section 3.3.1. The search process through which these virtual links are constructed is essentially driven by the distance relationships in IHR identifier space, which have nothing to do with distance relationships in the underlying physical topology.

Each IHR node has complete flexibility, however, in the way it chooses the k nodes to fill a particular bucket in its neighbor table whenever there are more than k nodes in the network that could fit into that bucket. If the network contains N nodes with uniformly distributed identifiers, then we expect nodes to have some flexibility in their choice of neighbors throughout the first $log_2\ N - log_2\ k$ buckets. Further, we naturally expect nodes to select the "best" k nodes they find for each such bucket: either the closest in terms of physical topology (IHR hop count), or the best according to some other pragmatic measure involving latency, bandwidth, and/or reliability for example.

In general, therefore, we expect the first few iterations of the **build_path** process to stay within the node's immediate topological vicinity, with subsequent hops covering larger topological distances as the remaining distance in identifier space is progressively narrowed. While the first few **build_path** hops will depend only on physical or inexpensive "low-order" virtual links, the last few hops might each depend on an expensive "high-order" virtual link, eventually resulting in a communication path that criscrosses throughout the network in a highly non-optimal fashion. It is therefore important that we find a way to optimize the routes produced using this process.

The most basic path optimization is inherent in the **build_link** procedure. If a node A locates target node B via the **build_path** process, but A subsequently finds that it can contact B directly using underlying protocols such as IP using address information it discovers during the process, then **build_link** will "short-circuit" the path from A to B with a physical link requiring no IHR-level forwarding.

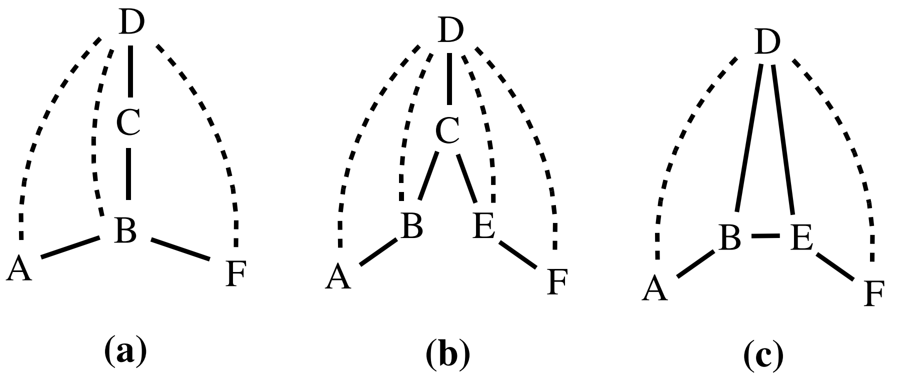


Figure 3-8: Path optimization opportunities on different topologies, when A builds a virtual link to F via D.

A second important path optimization is for nodes to check for obvious redundancies in the routes produced as higher-order virtual links are built from lower-order virtual links. Source routing makes this type of path optimization easier, since each node has information about the complete physical route to each neighbor in its neighbor table, but we will explore a more limited form of path optimization as well that works with recursive tunneling. Other more sophisticated forms of path optimization are certainly possible and desirable, such as optimizations relying on a deeper analysis of the relationships between known neighbors, or based on additional information exchanged between neighbors beyond the minimal information required to maintain the network and build virtual links. We leave more advanced path optimizations for future work, however, and focus for now on the effects of simple optimizations that rely on strictly local information.

Source Route Optimization

In IHR forwarding by source routing, we optimize source routes when combining two shorter paths into a longer one simply by checking for nodes that appear in both shorter paths. For example, in Figure 3-8(a), suppose node A has established a virtual link AD via B with path A, B, C, D, by building on virtual link BD with path B, C, D. A virtual link also exists between D and F. A now learns about F through D and attempts to create a virtual link AF via D. Without path optimization, the resulting path will be A, B, C, D, C, B, F. The path can be trivially shortened to the optimal A, B, F, however, simply by noting that B appears twice and eliminating the redundant hops between them.

The same optimization shortens the path from A to F in Figure 3-8(b) from A, B, C, D, C, E, F to the optimal A, B, C, E, F. This path optimization does not help in the case of Figure 3-8(c), however, since A does not necessarily know that B and E are direct neighbors.

Recursive Tunnel Optimization

Path optimization is not as easy in forwarding by recursive tunnels, because the information needed to perform the optimization is more spread out through the network. For example, in Figure 3-8(a), node A knows that the first hop along virtual link AD is the physical link AB, but A does not necessarily know what type of link BD is and may not even know that node C exists.

In general, for any virtual link from n to n_1 via n_2, node n also contains in its neighbor table a virtual or physical link representing the first hop from n to n_2. If the lower-order link from n to n_2 is a virtual link via some node n_3, then n also contains in its neighbor table a physical or virtual link from n to n_3, and so on. We call this chain of intermediate nodes along the path from n to n_1 that n inherently knows about n's *first hop chain* for n_1. For example, A's first hop chain for D in Figure 3-8(a) is A, B, D, whereas D's first hop chain for A is D, C, B, A.

To implement path optimization for recursive tunnels, we extend the **build_link** procedure of Section 3.3.1 so that when a node n attempts to build a new virtual link to n_t via waypoint node n_w, n contacts its existing neighbor n_w requesting n_w's first hop chain for n_t. Node n then compares the information returned against its own first hop chain for n_w, and short-circuits any redundant path elements.

For example, in Figure 3-8(a), node A is building a virtual link to F via D, so A requests D's first hop chain to F, which is D, C, B, F. A compares this chain with its first hop chain for D, which is A, B, D, discovering redundant node B and shortening the path to A, B, F.

This form of path optimization does not help in Figure 3-8(b), however, where the redundant path component between C and D is hidden from A because C is not in A's first hop chain. Similarly, this optimization does not handle Figure 3-8(c) for the same reason that the source routing optimization above fails.

3.4 Compact Routing

Scaling pressures resulting from the Internet's growth have created a strong interest in *compact routing*, which promises *provably* short routes and small forwarding tables [2,15,56,139,140,245]. Compact routing would in theory be very useful in the context of UIA, because it would ensure that (as with Identity Hash Routing) a large number of nodes may participate in a self-organizing routing structure while requiring each node to maintain only a small (sublinear in the network size) amount of state, and (unlike Identity Hash Routing) the routing algorithm would also guarantee a small (typically constant or $O(log\ N)$) bound on *stretch* or inefficiency of paths chosen. Existing compact routing schemes have so far failed to bridge the gap from theory to practice, however, in part because they were formally developed and described as centralized *algorithms* that take a monolithic network map (assumed to be static), and compute a set of routing tables for all nodes at once. Practical routing protocols such as UIA requires, in contrast, must perform their computations in a decentralized and incremental fashion, and must be robust against dynamic topology changes and failures.

This section explores one possible design for a practical compact routing protocol for UIA, based on the theoretical work of Thorup and Zwick (TZ) [245]. This decentralized protocol computes routing tables with $O(\log N)$ entries and bounds path stretch by $O(\log N)$. When more space and bandwidth are available, routes shorten smoothly and automatically. Previous compact routing work did not consider congestion caused by the choice of routes: we find that the TZ protocol is unfortunately subject to hot spots, and modify our protocol to improve congestion on a variety of simulated topologies.

3.4.1 Introduction to Compact Routing

Scaling pressures caused by the Internet's ongoing explosion in size and diversity have fueled an increasing interest in new approaches to Internet routing [37,47,80,139,235]. Recent theoretical advances in *compact routing* algorithms [2, 15, 56, 245] suggest the possibility of much more scalable routing techniques with formally provable performance properties, but these techniques have yet to cross the chasm from formal, centralized "algorithm" to practical, distributed "protocol." Some practical issues that still must be solved before any of these algorithms will become usable include: (a) creating protocols to compute compact routing tables incrementally using locally available information, (b) analyzing the performance of the resulting routes on diverse network topologies, (c) ensuring that routing table maintenance does not cause too much control traffic overhead [140], (d) providing the administrative controls over routing policy demanded by network operators, (e) providing resiliency against topology changes and temporary failures, and (f) designing a realistic, evolutionary deployment strategy. We do not solve all of these problems here, but merely attempt to take a step forward in understanding and addressing them, and to evaluate the suitability of compact routing as a method of providing overlay routing in UIA.

This section explores one approach to developing a practical compact routing protocol, based on Thorup/Zwick's variable (k-level) compact routing scheme [245], which we refer to as TZk. TZk uses a selection of *landmarks* distributed throughout the graph as the roots of multiple overlapping routing trees; two nodes wishing to communicate choose the "best" routing tree of which they are both members and route within that tree. TZk arranges these landmarks in a configurable number (k) of *levels*; routing trees built around the more numerous low-level landmarks have smaller membership, ensuring globally that every node participates in only a few trees. With one level ($k = 1$), TZk behaves similarly to classic distance-vector routing, finding optimal routes using $O(n)$-size routing tables. Two levels ($k = 2$) yields worst-case stretch of 3 and routing tables of size $O(\sqrt{n})$. With $log\ n$ levels, TZk has $O(log\ n)$ worst-case stretch and routing tables of size $O(log\ n)$. We choose TZk as a starting point because its variable-level design makes it theoretically much more flexible and scalable than fixed, two-level compact routing schemes [56], whose requirements of $O(\sqrt{N})$ state per node may be reasonable for large networks of large devices (e.g., core routers) or small networks of small devices (e.g., personal devices on an ad hoc network of limited size), but are probably not reasonable for an Internet-size network containing of billions of small personal devices participating in one overlay routing cloud.

Analysis of compact routing algorithms in prior work has focused primarily on routing table size and stretch, but other factors such as path congestion, resilience to failures, route computation/convergence delays, and control message costs are equally important in practice. The primary contributions of the present work are to build a simple distributed routing protocol based on TZk in order to evaluate the scheme's usefulness for UIA overlay routing, and to perform a large-scale simulation-based analysis of the protocol on a variety of graph types, including both synthetic graphs constructed from different theoretical models and realistic graphs of up to 190,000 nodes derived from actual Internet topology. We will note differences between our protocol and TZk as they become relevant.

We find later in Chapter 7 that TZk creates substantial congestion of global routing paths near high-level landmarks, and we introduce a means to control the severity of this congestion via an adjustable tradeoff against per-node routing table size. Congestion probably cannot be eliminated entirely, however, leading to the practical observation that landmarks may need to be chosen not randomly as in "pure" TZk but deliberately to be nodes adequately provisioned to handle this congestion. This need for careful landmark selection may unfortunately limit TZk's potential usefulness for UIA overlay routing, since we prefer UIA nodes to be fully "self-managing," but it may be possible to limit the negative effects of this congestion in the situations most normal users care about, for example by combining the techniques explored here with some of the techniques explored in the previous sections on Social Routing and Identity Hash Routing.

The compact routing protocol presented here also refines TZk by enabling nodes to use multiple independent routing trees for resilience against temporary network failures. This thesis does not develop or analyze the protocol in a fully dynamic setting in the context of long-term topology changes, however.

3.4.2 Routing Protocol Design

We assume that the network is represented by a weighted, undirected graph whose edge weights represent some suitable distance or cost metric for each link: round-trip delay for example. We assume that all network nodes are routers (which is typically the case in UIA), and we leave consideration of link capacity issues for later.

The key idea of TZk and our protocol is to replace the single routing tree of the Kleinrock/Kamoun model [134] with many overlapping routing trees of varying sizes. All nodes share membership in a few large routing trees, which they use to route between distant nodes, but many smaller routing trees provide efficient local routes. This shift from one routing tree to many enables TZk to achieve low stretch on arbitrary networks, including on highly-connected graphs where the Kleinrock/Kamoun model fails [140]. Figure 3-9 illustrates three routing trees in an example network—those centered at nodes A, B, and C—and the distinct but overlapping "clouds" of nodes participating in those trees.

There are four main processes comprising our protocol: *landmark selection*, *routing tree construction*, *global addressing*, and *route selection*. The first three processes compute the tables that

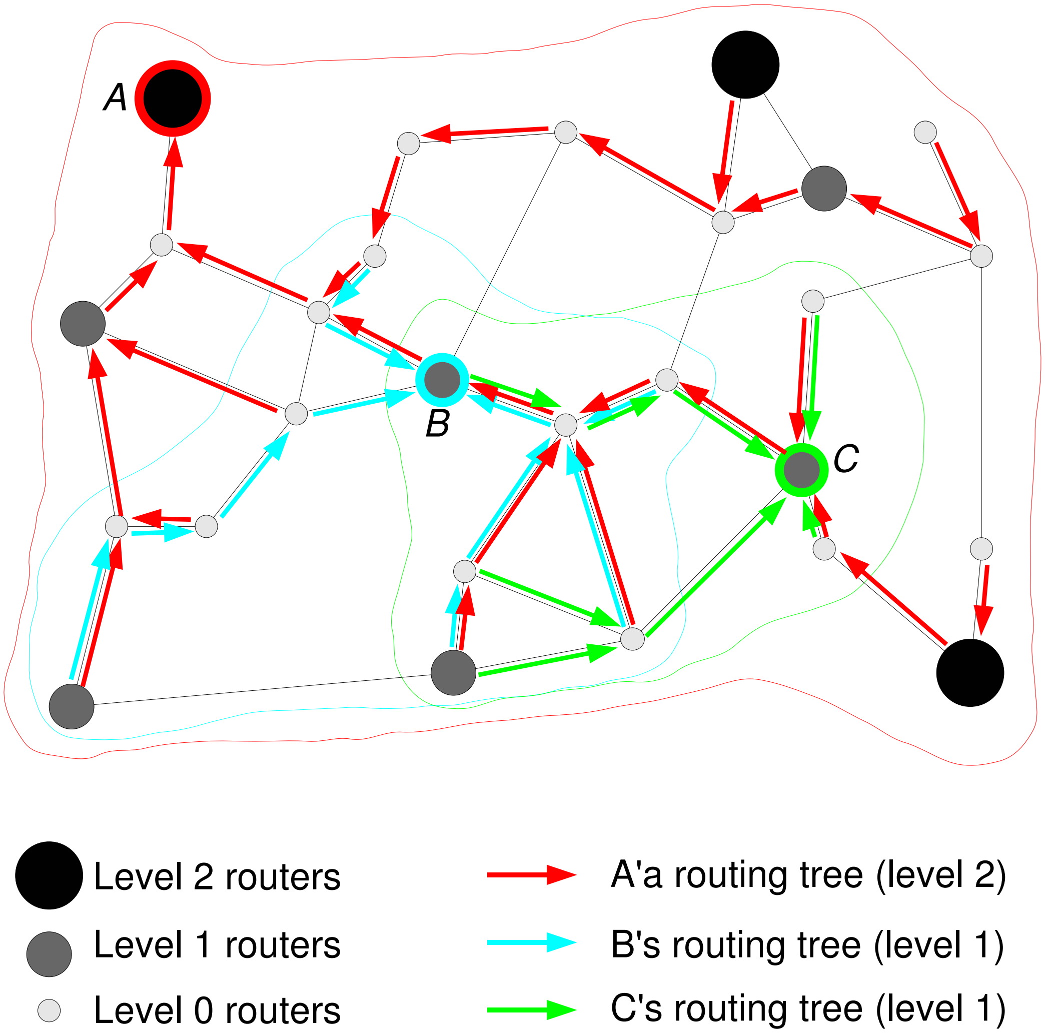
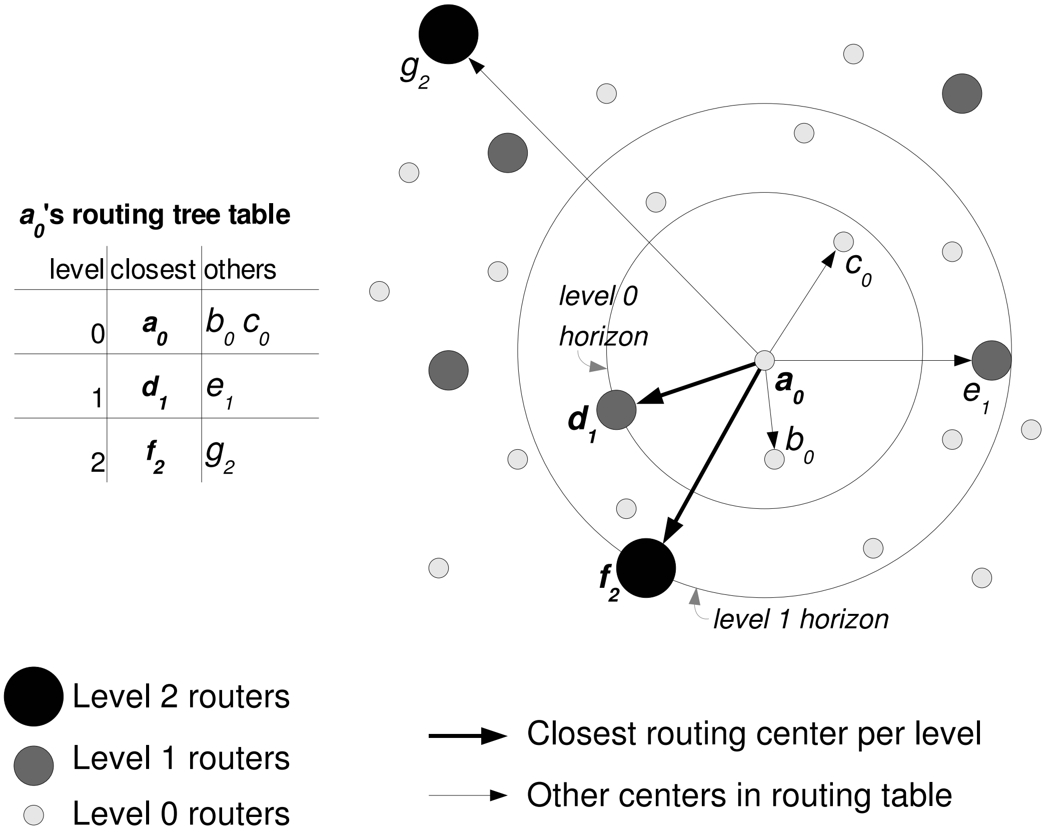

A
B
C
Level 2 routers
Level 1 routers
Level 0 routers
A'a routing tree (level 2)
B's routing tree (level 1)
C's routing tree (level 1)

Figure 3-9: Landmarks in Thorup/Zwick (TZk) scheme

nodes require to route messages; this computation could in principle be done either centrally or in distributed fashion, and either offline or online, although for UIA overlay routing we naturally want to support distributed online computation. The final route selection process occurs dynamically when a given pair of nodes wish to communicate.

In the *landmark selection* process, each node receives—or assigns itself—an integer *landmark level*. The total number of landmark levels, k, is a configurable parameter, and the specific choices of landmarks and levels may be either automatically or administratively controlled; for UIA routing we of course prefer automatic control. The basic goal is for each higher level $i + 1$ to contain approximately a constant, evenly distributed fraction of the landmarks from level i. In Figure 3-9, all nodes are landmarks, and dots of different sizes indicate different landmark levels.

Routing tree construction is largely automatic but can also be tuned administratively if necessary. In this process, every node in the network searches outward from itself for nearby landmarks, and joins multiple shortest-path routing trees rooted at a subset of those landmarks. To maintain



a_0's routing tree table

level	closest	others
0	a_0	b_0 c_0
1	d_1	e_1
2	f_2	g_2

Figure 3-10: Routing tree selection

TZk's worst-case stretch guarantee of $2k - 1$, each node u joins the routing tree of every landmark at each level i that is closer to u than u's closest level $i + 1$ landmark. With a suitable distribution of landmarks, each node is required to join only a few routing trees at each level, keeping routing tables compact. Nodes may join the routing trees of additional, more distant landmarks beyond this minimum requirement as their storage permits, however, improving both stretch and congestion. In a dynamic network, nodes may join the routing trees of new landmarks that appear within their "range" for that landmark level, and may leave the routing trees of landmarks that fail or go outside of the node's range for that level.

Figure 3-9 shows three overlapping routing trees: two at level 1, and a level 2 tree covering the whole network. Figure 3-10 illustrates how a particular node collects nearby landmarks, using its closest level $i + 1$ landmark as a "horizon" at which to stop collecting level i landmarks.

Finally, each node forms a *global address* for itself consisting of the identities of its closest r landmarks at each level, and a locator on each of those landmarks' routing trees, where r is a redundancy factor for robustness against failures. A node u wishing to send a message to another node v searches v's global address for the best landmark also known to u and routes the message over that landmark's routing tree. The two nodes may optionally further improve their route for ongoing communication by exchanging information about additional landmarks that may be present in their routing tables but not "published" in their global addresses.

The following sections describe each phase in detail.

3.4.3 Landmark Selection

There are many possible ways to choose landmarks. The specific property we wish this assignment to satisfy is that if any network node performs a best-first search outward from itself and places all the landmarks it encounters in order of distance from itself, it should encounter a small number of landmarks at each level i before encountering the first landmark at level $i + 1$. This is the property that determines the compactness of a given node's routing table.

Randomized Landmark Selection

The original TZk algorithm makes *every* node a landmark, and assigns landmark levels by a random process. Every node begins as a level 0 landmark. If k total landmark levels are desired in a network of n nodes, then TZk selects the landmarks for level $i + 1$ by including each level i landmark independently with probability $n^{-1/k}$. With this distribution of landmarks, every network node encounters about $n^{1/k}$ landmarks at any level i as it searches outward from itself before finding the first level $i + 1$ landmark.

If $k = 2$, for example, then TZk randomly chooses about $\sqrt{n}$ of the n total nodes to be level 1 landmarks, leaving all remaining nodes as level 0 landmarks. If $k = log_b\, n$ for some constant $b > 0$, then the total number of landmarks decreases by a factor of b at each level. In this latter case, randomized landmark selection is easy to perform in a fully decentralized fashion: each node chooses a random identifier expressed in base b, and its landmark level i is simply the number of leading zero digits in its identifier.

Explicit Landmark Selection

Although TZk's randomized landmark selection process provides a simple means of guaranteeing compact routing tables, in a practical large-scale network, *some* landmarks may have to be chosen deliberately for reasons of bandwidth and reliability. As we will see later in Section 7.4.3, global traffic routes tend to converge on and around high-level landmarks in a TZk network. This route convergence may cause network overload unless these landmarks are deliberately chosen (or *designed*) to be nodes that are adequately provisioned to handle the load incurred by this route convergence. In addition, a failure at or near a high-level landmark is likely to affect a disproportionately large portion of the network, so the highest-level landmarks should be chosen to be highly reliable nodes.

For these reasons, unlike TZk, our protocol assumes that some or all landmarks may be chosen administratively. Although for UIA overlay routing purposes we would prefer if *all* participating devices were fully self-managing, it may be reasonable to assume that a small percentage of devices may be administratively assigned to the highest landmark levels: e.g., high-capacity UIA routing "hubs" deployed by companies wishing to ensure that the UIA-based personal devices they sell perform well.

If a network operator is not sufficiently careful about landmark selection, however, some network nodes may end up with larger routing tables than others. If there are many level 1 but not enough level 2 landmarks in a given area of the network, for example, then nodes in that area may have overly full routing tables at level 1 because they must search outward through many level 1 landmarks before encountering the closest level 2 landmark. Such situations are easy both to detect and to rectify, however. A node whose level 1 routing table grows too large might trigger an alarm, for example. In response to this alarm, a network operator could simply pick a level 1 landmark from the middle of the node's routing table and promote it to level 2, thereby "chopping off" the node's level 1 routing table at that point. Once too many landmarks in the area have been promoted to level 2, the network operator may have to promote some of those to level 3, and so on. Our protocol thus allows routing table compactness to be ensured either automatically, through TZk's random selection process, or administratively as the network evolves, simply by "oiling the wheel that begins to squeek."

Landmark Pruning

The original TZk algorithm makes *every* node a landmark, each serving as the root of a distinct shortest-path routing tree. In our protocol, however, we assume that the set of landmarks may be a subset of all nodes; a non-landmark node participates on the routing trees of nearby landmarks but does not have a routing tree of its own. Although omitting landmarks may in general increase stretch, it is easy to prove that certain landmarks may be omitted or *pruned* without affecting the routes chosen by the protocol.

Consider for example any pair of adjacent nodes u and v such that when the edge between them is removed, the graph component containing u forms a tree. All nodes in this "edge tree" including u may be pruned from the set of landmarks, leaving v as their nearest landmark. Since the pruned nodes can only reach the rest of the network through v, the shortest-path routing trees their landmarks would form beyond v would use exactly the same routes as v's routing tree itself, and there is only one loop-free way to route within the edge tree itself.

Similar pruning optimizations may be possible even within graph cycles, but this simple edge tree optimization may be particularly useful for Internet-like topologies, which tend to be highly connected at the core but treelike at the fringes [147]. Indeed, in the large Skitter [38] graph we used as one of our test topologies, about 26% of the nodes are located in edge trees and thus need not be landmarks.

3.4.4 Routing Tree Construction

Once the landmarks are selected and assigned levels, our protocol computes a routing tree rooted at each landmark permitting any two nodes residing on the same tree to communicate. There are essentially three parts to this problem: determining the structure and extent of each routing tree, assigning *addresses* or *locators* to each node on a tree, and using those addresses to route messages over the tree. Only the first part of the problem is highly specific to our protocol or TZk; for assigning locators and routing on a particular tree many well-known techniques may be used.

Forming Routing Trees

The size and shape of the routing tree around each landmark is determined implicitly by the set of nearby nodes that "choose" that landmark for their routing tables, which in turn depends on the topology of this neighborhood. As explained above, every node u joins at least the routing trees of all level i landmarks closer than any level $i + 1$ landmark, and may join the trees of additional more distant landmarks if desired to optimize stretch or congestion at the expense of routing table size.

Since there are no landmarks at level k to serve as a "horizon" for level $k - 1$ landmarks, *every* node in the network participates in the routing trees rooted at each of the level $k-1$ landmarks; these form the network's *global routing trees*. The membership of the smaller routing tree surrounding a lower-level landmark l is termed the landmark's *cluster* by Thorup and Zwick, and is suggestive of though not the same as a cell in a Voronoi diagram.

Each landmark's routing tree should be a *shortest-path tree*: that is, the path taken along the tree from any member node u to the landmark l serving as the tree's root should be a shortest path from u to l. This shortest path property is crucial to the TZk algorithm's formal stretch guarantee.

Distributed Routing Tree Construction

In principle, routing tree computation may be performed centrally on a complete graph of the network and the resulting routing tables shipped to the nodes for use, or the nodes themselves could compute their own routing tables via a distributed protocol. The original TZk algorithm was presented in terms of a centralized computation, but to permit more detailed analysis we explore a simple distributed message-based protocol based on the algorithm. Although the current message-based protocol is rudimentary, it demonstrates that computation of TZk routing tables has deep similarities to classic distance-vector routing, so many techniques used in existing distance-vector protocols may be reapplicable to compact routing.

In our protocol, each node learns about nearby landmarks and builds its routing tables incrementally by gossiping with its neighbors. Each node's routing table stores the set of landmarks the node has discovered so far for each level, and for each such landmark, the length of the shortest known path to that landmark and the identity of the "next hop" along that path. In this respect our protocol works much like a classic Bellman-Ford distance-vector protocol, except that in our protocol each node sets a "distance horizon" on the set of landmarks it learns at a given level i, based on the closest known level $i + 1$ landmark it has learned (if any). Whenever a node learns from a neighbor about a new landmark at some level i—or a better route to an already-known level i landmark—that is closer than the closest currently-known level $i + 1$ landmark, the node updates its routing table and then broadcasts the route to all of its neighbors. If the node learns a route to a level $i + 1$ landmark that is closer than one or more known routes to level i or lower landmarks, then the protocol can either throw away the "excess" lower-level routes to keep each node's routing table to the minimum size, or retain a limited number of extra routes in order to reduce stretch and congestion, as described later.

Addressing and Routing on Individual Trees

The simple distributed protocol above effectively forms a shortest-path tree around each landmark containing the appropriate set of member nodes as per the requirements of the TZk algorithm. However, the simple distance-vector protocol only provides the information necessary to route packets *inward* on each tree toward the landmark; we require the ability to route both inward and outward on the tree, from any node on the tree to any other node. For this purpose a variety of well-studied routing schemes are suitable; we touch on only a few particularly relevant ones here.

The simplest solution is to extend the distance-vector protocol above so that nodes gossip and store complete source routes to their nearest landmarks instead of just the "next hop." A node u's "address" on a landmark l's routing tree is simply u's source route to l. If u wishes to reach another node v over l's tree, u simply reverses v's source route to l, appends the result to u's own source route to l yielding a source route from u to v through l, and trims any cycle from the resulting path. The danger of this approach is that source routes may become long, $O(n)$ on arbitrary graphs, although this should not be a major problem on Internet-like scale-free graphs, which have low diameter.

As an alternative to source routing, interval routing [211] can route over trees using addresses that are always of size $O(log\ n)$ regardless of network topology. Classic interval routing requires each node to have storage proportional to its degree, but Thorup and Zwick propose a scheme that combines interval and source routing so as to use tree-addresses of size $O(\log^2\ n)$ and constant storage per node regardless of topology [245]. Most realistic network nodes use storage proportional to their degree anyway, however, just to keep track of the status of each link for example, so classic interval routing techniques may be adequate from a practical compactness perspective.

The primary practical disadvantage of interval routing is that it requires ongoing maintenance of the address space and perhaps occasional renumbering as the network topology evolves. A straightforward way to address this challenge is for each landmark to divide time into *epochs*, and compute a new set of routing tree addresses for each epoch, tagging each tree-address with the epoch counter so that more than one epoch's addresses may be used at once and nodes in the landmark's routing tree may transition gracefully from one epoch to the next. This approach is reminiscent of the Destination Sequenced Distance Vector (DSDV) routing algorithm [182] for mobile ad hoc networks, although the epochs used for compact routing may need to be longer since our protocol must assign addresses and build a routing structure usable for routing "outward" from landmarks to nodes participating in its tree, and not just "inward" toward destinations as in DSDV.

3.4.5 Global Addresses

Once the above routing trees are constructed and all nodes have located their nearest landmarks at each level, each node chooses a *global address* for itself. In our protocol, the global address of any node u contains the identities of the r landmarks closest to u (i.e., the first r landmarks at each level of u's routing table), u's distance from each of those landmarks, and u's tree-address on the routing trees of each of those landmarks. Using source routes as tree addresses, for example, each node's global address consists of its closest r landmarks at each level and its distances and source routes to those landmarks.

Our protocol's *redundancy factor* $r \geq 1$ is an adjustable parameter that controls a tradeoff between global address size and robustness against failures, as we will explore in detail later. A value of $r = 1$ corresponds to the original TZk algorithm, which guarantees all nodes global reachability and bounded worst-case stretch, but only in the absence of failures. If tree-addresses are of size $O(log\ n)$, as with interval routing or source routing on a scale-free graph, then each node's global address is of size $O(rk\ log\ n)$.

The fact that a node's global address has a structure that depends on network topology creates two challenges, however. First, this means that TZk provides name-dependent rather than name-independent routing, so to use it in an architecture like UIA that relies on nodes having flat, location-independent endpoint identities (EIDs), nodes still require a way to *resolve* an EID into a global address that is directly usable for routing. Fortunately, this distributed lookup/resolution is precisely what DHTs were designed to accomplish, so a standard DHT protocol [146,156,194,209,234,271] built on top of TZk but below UIA's naming layer provides an obvious approach to addressing this challenge. Other potentially useful, "locality-aware" techniques for resolving location-independent names have been developed for ad hoc networking [43] and in the formal literature on name-independent compact routing [2,15].

Second, since network topology changes may result in changes to the node's set of closest landmarks, as well as to the node's position and tree-address on those landmarks' routing trees, the node's global address must change over time and the EID-to-address resolution mechanism must adapt to those global address changes, ideally while maintaining continuous connectivity during these changes whenever possible. DHT protocols generally support dynamic changes to key/value pairs they index and store, so in theory these dynamic updates are not a problem, but in practice there may be efficiency issues if many nodes' global addresses change too rapidly. We leave most of these EID resolution and dynamic adaptation challenges for future work.

3.4.6 Route Selection

There are two situations to consider when two nodes u and v wish to communicate: one in which u must "unilaterally" choose a route to v with no information other than u's routing table and v's global address, and one in which u and v are already in contact but wish to choose the best route for ongoing traffic. We consider each of these cases in turn.

Unilateral Route Selection:

If a node u wishes to communicate with an arbitrary target node v in the network, u must first obtain v's global address—by looking it up in a directory such as DNS or a DHT as suggested above, for example. Given the target v's global address, u then compares v's published list of landmarks with those in u's own routing table, picks the "best" landmark l known to both nodes, extracts v's tree-address on l's routing tree from v's global address, and uses that tree-address to send a message to v over l's routing tree. In the original TZk algorithm, in which each node's global address contains only the single closest landmark at each level (corresponding to $r = 1$ in our protocol), u simply picks the lowest-level landmark and tree-address from v's global address. When u picks a route to v "unilaterally" this way without cooperation from v, TZk guarantees a worst-case stretch of $4k - 3$, somewhat higher than the theoretical optimum of $2k - 1$ [245]. To obtain $2k - 1$ worst-case stretch TZk requires a *handshaking* phase, described below.

In our protocol, however, v's global address contains r entries per level, and the best route between u and v could be via the routing tree of some landmark (slightly) farther from v than v's closest landmark at that level but (much) closer to u. Since our protocol includes v's distance to each of its landmarks in v's global address, u can estimate the total distance over a given landmark's routing tree by adding u's distance to the landmark and v's distance to the same landmark. This result is a conservative estimate, since a message will not actually reach the landmark if the source and target nodes are within the same "branch" of the tree.

Cooperative Route Selection:

Once u and v are in communication over *some* route, they may be able to find a better route through handshaking. The original TZk algorithm requires such a handshaking step in order to achieve the optimal stretch bound of $2k - 1$. Once u first contacts v over its unilaterally-chosen path, v uses the same unilateral route selection procedure to find a route back to u; in doing so, v may find a better (lower-level) landmark known to both u and v. v then responds to u with an indication of the best landmark, and both nodes use the chosen landmark for subsequent communication. TZk's optimal worst-case stretch depends on the two nodes routing on the lowest-level landmark that either node can discover unilaterally, hence the handshaking requirement.

An interesting and potentially useful property of TZk and our protocol is that its worst-case stretch bound for the path between *a particular pair of nodes*, u and v, becomes smaller if u and v are close together. In particular, if two nodes u and v agree through handshaking to use a level i tree, the worst-case stretch for their subsequent communication is $2i + 1$.[1] Thus, routing between nearby nodes on a level 0 tree is always optimal (stretch 1), routing on a level 1 tree has a worst-case stretch of 3 regardless of the total number of levels k in the network, and so on.

In our generalization to TZk with the redundancy parameter r, u can unilaterally estimate the cost of the (at least r) available redundant paths to v as described above, but u and v may find still

[1] Thorup and Zwick do not point out this property explicitly but it may be read easily from their proofs.

better paths by dynamically exchanging additional routing table information not included in their respective global addresses. For a precise selection of the best path, the two nodes could simply try several paths and dynamically select the best (e.g., the one with the shortest delay), or ideally even use multiple paths simultaneously if supported by upper-layer protocols [121].

In the limit, if we set $r = \infty$ in our protocol, then each node publishes in its global address *all* of the landmarks it knows at each level—i.e., its full routing table. If $k = log\ n$ and each tree-address is $O(log\ n)$ size, then each node's global address is of size $O(kn^{1/k} log\ n)$, or $O(log^3\ n)$ in the case of $k = log\ n.$ This additional $log\ n$ factor in global address size not only purchases redundancy against failures, but also guarantees the asymptotically optimal $2k - 1$ worst-case stretch *without* handshaking, since u already has all the information v might provide in TZk's handshaking step.

If u finds v's global address in a directory or DHT, u need not necessarily retrieve v's entire global address; u could instead send the directory service a Bloom filter [34] summarizing the identities of the landmarks in u's routing table, and the directory service merely returns those entries from v's global address for landmarks u may know about.

Chapter 4

Transport

Once personal devices are conveniently nameable via the UIA personal naming system, and can efficiently locate and connect with each other via UIA's overlay routing system, devices still need to be able to *utilize* their connection effectively in order to support the wide variety of applications that current and future users expect to run on their personal devices. In contrast with the terminal- and batch-oriented applications that the Internet was originally designed around, modern applications are often highly interactive, allow multiple communication activities to occur at once in parallel, and can be very sensitive to network transmission or reordering delays.

This chapter rounds out UIA by introducing *Structured Stream Transport* (SST), a new transport protocol developed specifically to support the highly interactive and concurrent applications that are becoming common on modern personal devices. We first explore the motivation and goals for the new transport in Section 4.1. Section 4.2 then introduces the *structured stream* abstraction that the new transport is built on, and Section 4.3 explores how this new transport abstraction may be leveraged in a variety of different applications. Finally, Section 4.4 details the design of the new transport protocol.

4.1 Motivation and Goals

Current Internet transports offer applications a choice between two abstractions: reliable byte streams as in TCP [242] or SCTP [232], or best-effort datagrams as in UDP [185] or DCCP [136]. Streams are convenient for large or long-running activities that must preserve data content and order, such as terminal sessions or file transfers, but datagrams more efficiently support numerous small transactions that need to proceed independently as quickly as possible, such as DNS requests or frames of a media stream.

The classic stream and datagram abstractions are conceptually simple and worked reasonably well for the types of applications that dominated the early Internet. Long-lived TCP streams are well-suited to character-stream-oriented terminal protocols like Telnet [186], Rlogin [128], and Secure Shell [264–267]; batch text messaging systems such as E-mail [58, 135, 168, 201] and USENET [77, 114, 175]; and bulk data transfer protocols such as FTP [113, 187]. Small, ephemeral

UDP datagrams are suitable for simple transactional protocols with small requests and replies, such as DNS queries [256] and NFS block I/O [217].

4.1.1 The Mismatch of Traditional Transports with Modern Applications

Modern Internet applications, especially interactive applications designed for today's mobile personal devices, are placing increasing strain on the traditional stream and datagram abstractions, however. Modern interactive applications such as web browsers, for example, often use a *mixture* of small and large transactions: such an application faces awkward tradeoffs, because it could benefit from the efficiency of UDP for small transactions but needs TCP to handle the occasional large ones. Using one TCP stream per transaction as in HTTP/1.0 [29] makes poor use of network resources and is unfair to other applications when a client uses many concurrent streams; serializing transactions onto persistent streams increases transaction latency [171]; and implementing pipelining correctly in web servers has proven challenging enough that seven years after the standardization of HTTP/1.1 [78], popular browsers still leave pipelining disabled for compatibility [141, 165].

Similarly, multimedia streaming applications are becoming increasingly important on modern personal devices. RTP [256] over UDP may be adequate for fixed-rate voice communication in which all media frames are small, but modern high-definition, variable-bit-rate video encoding schemes can produce occasional large frames. These large frames are especially likely to be dropped, because UDP drops an entire datagram if *any* of its composite fragments are lost in transit due to congestion or transmission errors, and cannot retransmit lost fragments. At the same time, these large frames are often semantically the most critical ones *not* to drop, since they typically represent "key" frames with respect to which other nearby frames are represented via differential encoding. Multimedia streaming applications can and sometimes do use TCP instead of UDP because of its reliable delivery, but this approach does not work well if the multimedia stream is delay-sensitive (e.g., a real-time audio/video conferencing stream), because one dropped packet in a TCP stream delays delivery of *all* data behind it in the stream until the dropped packet is successfully retransmitted.

Modern applications for personal devices face further tradeoffs because neither streams nor datagrams offer a means to associate related transport instances. Applications such as FTP [187] and SIP [206] associate transport instances manually by passing IP addresses and port numbers in messages, causing protocol fragility and problems traversing NATs [112]. Other applications multiplex many activities onto one TCP stream, as with SSH tunneling [264], but performance suffers from TCP's total ordering of the multiplexed stream, where one lost packet blocks delivery of all data queued behind it in the stream.

4.1.2 A Transport Abstraction for Personal Device Applications

To address the above problems, the UIA architecture introduces a new transport protocol called *Structured Stream Transport*, or SST, which is specifically designed around the needs of modern interactive applications running on today's personal devices.

SST addresses the problems of transaction size and instance association by augmenting traditional streams with an explicit hereditary structure. A *structured stream* provides not only reliable data delivery as in TCP, but also permits the creation of additional lightweight child streams or *substreams*. When an application spawns a child from an existing stream, SST conveys this parent/child relationship to the remote application, ensuring that only the intended recipient can accept the new stream and preserving its structural context. Each stream delivers data reliably and in order within that stream, with its own receiver-directed flow control and optional record marking, while other streams may proceed independently with no head-of-line blocking. SST shares sequencing, congestion control, and security state among all streams between a pair of hosts, minimizing per-stream state overhead and allowing the application to use as many concurrent streams as convenient to match its logical structure. The application may also create and destroy streams rapidly to match the natural longevity of its transactions, because SST eliminates TCP's 3-way handshake delay on all streams after the first, and also eliminates the 4-minute TIME-WAIT period on close that can cause TCP state overload [75]. The application can transmit datagrams on an SST stream with best-effort delivery, but SST treats these datagrams as *ephemeral substreams* that are semantically indistinguishable to the receiving application from ordinary substreams, allowing SST to fall back to stream-oriented delivery for "datagrams" too large to transmit as such without unacceptable probability of loss. Finally, the application can prioritize SST streams relative to each other, giving preference to interactive or multimedia streams over background activities, and applications can use substreams for out-of-band signaling to change priorities mid-stream, e.g., expediting the loading of images in the currently visible area of a web page.

Although SST is primarily intended for use in new or evolving applications, analyzing the requirements of well-established protocols can serve to illustrate the general utility of structured streams. Applications like FTP and SIP that modularize their operation across several transport instances, for example, can use a top-level SST stream for control signaling and open private substreams for data or media transfers, instead of separate transport connections. Allowing SST to manage and associate the application's transport instances makes the application protocol simpler, more robust, and friendlier to other applications due to SST's shared congestion control. Transaction-oriented protocols like HTTP can adopt a simple one-transaction-per-stream model as in HTTP/1.0 [29], relying on SST to multiplex many transactions efficiently onto one persistent connection, and avoiding both the serialization-induced latencies of persistent streams [171] and the application-level complexity and fragility of pipelining [165]. Applications based on RPC [225] can use SST either as a datagram-oriented transport that combines the efficiency and statelessness of UDP for small messages with support for messages of arbitrary size, as a transaction-oriented transport using lightweight streams to associate requests with responses, or as a stream-oriented transport that eliminates the need for RPC's record marking layer. Character-oriented streaming applications such as Telnet [186] and Rlogin [128] can use structured streams in place of TCP's legacy Urgent mechanism to handle asynchronous attention signals such as CTRL-C, improving their promptness by eliminating TCP's need to deliver all data already in the pipeline reliably before it can begin delivering the actual urgent data.

4.1.3 Design and Implementation Features

In addition to the new structured stream abstraction, several other novel aspects of SST's design may suggest useful transport design principles:

- SST builds its structured streams on top of an intermediate *channel* protocol: a connection-oriented sequenced datagram service reminiscent of DCCP [136] but semantically closer to IPsec's packet sequencing [131, 132]. The channel protocol's monotonic sequence numbers and replay protection logic, in particular, enable SST's lightweight streams to avoid 3-way handshakes or TIME-WAIT periods.

- The SST channel protocol selectively acknowledges packets via *acknowledgment ranges*, which provide more information than TCP's SACK [154] and D-SACK [79] extensions, facilitate forward acknowledgment [155] and reordering tolerance [32], and offer redundancy against lost acknowledgments, without the complexity or overhead of variable-length SACK headers.

- SST separates the *multiplexing* and *rendezvous* functions that port numbers serve in traditional transports, using small, temporary *local stream identifiers* for multiplexing and more friendly *service* and *protocol names* for rendezvous during connection initiation.

- SST can *attach* a stream to multiple underlying datagram channels successively or at once, to insulate application streams from temporary failures, IP address changes, and channel lifetime limits.

- SST jumpstarts a child stream's flow control state by *borrowing* from its parent's receive window, allowing the initiator to start sending data on the child stream without waiting for the receiver's initial window update.

- SST demonstrates how layering and functional reuse enables substantially more functionality than TCP with *no additional per-packet overhead* in comparable scenarios.

A user-space prototype implementation of SST, running on real and simulated networks, attests to SST's practicality and utility. The prototype achieves within 10% of the bandwidth of mature native TCP implementations on "consumer-grade" DSL and WiFi connections, and its congestion control is TCP-fair to within ±2%. Under a simulated web browsing workload, using one HTTP/1.0-style transaction per stream with SST achieves the performance of HTTP/1.1 pipelining, with lower application complexity, and more flexibility as demonstrated by a simulation of dynamic Web download prioritization in response to user actions. Although the SST prototype is not yet integrated with the deployed UIA prototype, it should serve well both as a secure point-to-point connection and authentication protocol for the UIA routing layer to use in place of SSL over TCP, and as a new end-to-end transport for UIA naming layer communication and for new UIA-aware applications at higher levels.

The primary contributions of SST are: (a) the structured stream abstraction, (b) a novel protocol design that implements this abstraction by splitting the transport into separate *stream* and *channel* layers, and (c) a user-space prototype for easy experimentation and deployment.

4.2 Structured Streams

This section describes the structured stream abstraction as viewed by an application using SST, leaving technical details of the SST protocol itself to Section 4.4.

4.2.1 Data Transfer Semantics

Like a TCP stream [242], an SST stream is a reliable, byte-oriented conversation between applications. Apart from the ability to create substreams, SST streams are semantically almost identical to TCP streams, so porting application protocols from TCP to SST is straightforward. Each participant sends and receives segments of bytes, and the transport guarantees reliable in-order delivery of those segments in the absence of connection failure, but may fragment, combine, and buffer segments. Each endpoint may close its end of the stream independently, so a client may for example close its end to signal the completion of a request before awaiting the server's reply on the same stream.

SST data transfer semantics differs from TCP's in only two respects. First, SST dispenses with TCP's Urgent feature, but this feature may be emulated using substreams as described later in Section 4.3.4. Second, senders may set explicit record marks in an SST stream, which SST reliably delivers to the receiver, simplifying many applications and making their buffer management more efficient at little cost in the transport layer.

4.2.2 Creating and Accepting Substreams

An application can use an SST stream not only to transfer data but also to "fork off" child streams or *substreams*. The terms "substream" and "child stream" refer only to hereditary relationships between streams: once created, there is no semantic difference between a "top-level" stream and a substream. SST extends the traditional sockets API with three new operations for creating substreams:

- `create_substream`$(stream) \rightarrow new\_stream$:
  creates a new child stream from an existing stream.

- `listen_substream`($stream$):
  indicates willingness to accept substreams on a stream.

- `accept_substream`($stream$) → new_stream:
  accepts a substream initiated by the other endpoint.

An application calls `listen_substream` to indicate willingness to accept new streams as children of an existing stream, much as the application would use a traditional listen socket to receive incoming top-level streams. The peer application on the existing stream's other endpoint may then initiate a substream by calling `create_substream` on the existing stream. SST notifies the listening application of the incoming substream, and the listener accepts it via `accept_substream`. Once created and accepted, child streams are independent of and may outlive their parents.

4.2.3 Inter-Stream Semantics

Since SST conveys parent/child relationships on substream creation, the application can use contextual state negotiated in a parent stream to determine communication behavior on child streams. A videoconferencing application may for example initiate a call in a "root" stream, then initiate one child substream for video and a second for audio, negotiate codec parameters for each media stream on its respective SST substream, and finally transmit media frames themselves on "grandchild" substreams. SST's inter-stream structure enables applications to modularize and associate their related communication activities, without having to multiplex all activities onto one stream or manually track logical associations between separate transport instances. SST endeavors to make streams "lightweight enough" for the application to use a new stream for each logical transaction regardless of size, enabling the application to "frame" its protocol data units according to its needs [51].

SST ensures reliability and data ordering within a stream but not between streams, so the loss and retransmission of a data segment on one stream does not delay communication on other streams. If a client issues many requests to a server at once on separate streams, for example, the server may accept several requests concurrently and respond to them in any order, completing "easy" requests quickly without blocking them behind long-running requests submitted earlier. Each stream provides independent flow control, so the receiving application may accept data at different rates on different streams, or accept data continuously on one stream while temporarily blocking the sender on another. SST's flow control applies to new child streams as well as to stream data, allowing a server to prevent its clients from overloading it with too many concurrent requests.

4.2.4 Datagrams as Ephemeral Substreams

TCP delivers data reliably if the participants wait for all necessary retransmissions and acknowledgments, as the TCP standard dictates [242], but most TCP stacks offer an option to limit the time a host buffers unacknowledged data after stream close (e.g., `SO_LINGER` in BSD). By imposing a fixed deadline, this option can have the side-effect of deviating from TCP's usual reliable semantics and turning it into a "best-effort" delivery service. Suppose an application opens a stream, sends some data, then closes the stream with a short linger timeout without ever receiving on the stream. The sending TCP transmits the data and closing FIN, but does not hold unacknowledged segments beyond the linger timeout. The receiver obtains the data and a graceful close indication if every segment including the FIN arrives intact, or if retransmissions complete before the linger timeout; otherwise the receiver sees a connection reset. If the linger timeout expires within one round-trip time of the first data transmission, then no retransmissions are possible, making this usage of TCP semantically equivalent to a best-effort, "fire and forget" datagram delivery service—if less efficient.

SST has a `send_datagram` operation providing best-effort delivery, but SST treats this operation as equivalent to creating a child stream, sending data on it, and closing the child with a short linger timeout to avoid buffering the sent data. SST's `receive_datagram` operation is similarly a shorthand for accepting a substream, reading data, and closing the substream. Since an SST "datagram" is semantically just an *ephemeral substream*, the receiver can accept the substream with `accept_substream` instead of `receive_datagram`, and can then read the substream's content as it would with a normal substream. The receiver gets a connection reset if it ever tries to send on this ephemeral substream. The receiving application thus cannot tell whether the sender invoked `send_datagram` or the equivalent sequence of operations.

SST can use an optimization described later to deliver best-effort datagrams efficiently and statelessly as in datagram transports such as UDP or DCCP. Because the receiving application cannot tell whether this optimization was used, however, the sending SST is free *not* to use it when network conditions may not be suitable. If for example a datagram to be sent is large enough that at least one fragment is likely to be lost, given the current loss rate computed from congestion control statistics, SST forgoes the datagram delivery optimization and sends the datagram instead on a "real" substream, transparently to the application. SST thus solves the "large datagram" problem of traditional best-effort services, where datagram loss rates quickly become unacceptable as datagram size increases.

4.2.5 Other Application-Visible Features

SST may be deployed either at system level as a "native transport" alongside TCP and UDP, or at application level atop UDP. The latter usage allows applications to ship with a library implementation of SST without requiring special privileges or extensions to existing operating systems, and they can use it as they would SSL/TLS [64] or DTLS [200]. Deploying SST atop UDP also allows it to traverse existing NATs that only natively support TCP and UDP.

Since communication security is now essential for most applications on today's Internet, but IP-layer security [132] is still not widely deployed other than for VPNs, SST provides optional transport-layer security built on standard practices. SST's security is mostly transparent to applications, except for configuration of security parameters and secure identification of communication peers.

Since a large percentage of hosts on the Internet today are connected behind NATs or firewalls, many applications require traversal of these barriers, especially applications with peer-to-peer communication patterns such as Voice-over-IP. For this reason, SST supports hole-punching [82] to traverse BEHAVE-compliant NATs [16], and provides an optional host registration protocol to assist applications in establishing peer-to-peer connectivity between communication partners behind NATs.

4.3 Using Structured Streams

To examine the practical benefits of structured streams, this section briefly explores requirements and challenges faced by several classic application protocols. We use these protocols for illustrative purposes only, without implying that specific applications should necessarily be migrated to SST.

4.3.1 Application Protocol Modularity

FTP [187] modularizes its operation across multiple transport instances: one TCP stream for control signaling and a separate stream for each file transfer. With SST, such an application can use a top-level stream for control and simply open substreams for data communication. Since SST conveys the parent/child relationship on substream creation, application protocols need not communicate IP addresses or port numbers explicitly to associate the related transport instances, simplifying the application and avoiding difficulties traversing NATs [112].

Internet telephony similarly uses a TCP stream for call setup via SIP [206], and separate UDP streams for media delivery via RTP and RTCP [214]. The disconnect between these streams at the transport layer, and the resulting need for SIP messages to carry IP addresses and UDP port numbers, causes even worse NAT traversal problems than in FTP, resulting in complex extensions [205] as well as competing proposals for ground-up redesign using application-level multiplexing [224]. With SST, a telephony application could run SIP in a top-level stream, open substreams for RTP and RTCP during a call, and send individual media datagrams as ephemeral substreams of those respective substreams, using SST to multiplex the streams while preserving their association.

4.3.2 Transactional Protocols

The need to support transaction-oriented applications efficiently has long been recognized [31, 35, 46]. In the absence of a general transport satisfying this demand, HTTP/1.0 [29] used a separate TCP connection for each request, each requiring a 3-way handshake and TIME-WAIT period after close, leading to unacceptable overhead as the Web became riddled with small images. Since TCP congestion control distributes available bandwidth per stream, opening many concurrent TCP connections is also unfair to other users [18]. HTTP/1.1 [78] allows reuse of a TCP connection for successive requests, reducing the total number of connections, with the disadvantage of serializing requests and often making pages take longer to load [171]. In theory requests may be pipelined to improve latency, but seven years after the publication of HTTP/1.1, today's common browsers still avoid pipelining by default because many servers implement it incorrectly [141, 165]. Pipelining also fixes response order to request order, blocking the server from satisfying simple static content requests promptly until it finishes processing dynamic content requests submitted earlier for example.

HTTP could benefit from UDP's efficiency—particularly for the many conditional GET requests browsers use to test the freshness of cached objects, whose responses are usually small—but HTTP cannot use UDP because responses *may* be arbitrarily large, and the client has no way of knowing this when it submits the request. SIP supports operation over UDP, but gets into trouble for precisely this reason [102].

With SST, applications can use one stream per transaction, with minimal startup latency on all streams after the first, and without serialization of independent requests or long-term state retention after close. Alternatively, applications can use SST as a best-effort datagram transport, sending requests and replies in separate ephemeral streams, achieving UDP's statelessness for small messages without UDP's message size limits. In either case, SST ensures fairness relative to TCP applications that use only one stream, since all of an application's SST streams between the same pair of hosts share a single congestion control context.

4.3.3 RPC

ONC RPC [225] can run over either TCP or UDP, but the choice between the two transports presents application designers with an awkward tradeoff. Many applications would like to use UDP for its efficient handling of small requests, lack of unnecessary serialization or head-of-line blocking, and minimal server state requirements. Because of the "large datagram" problem discussed in Section 4.2.4, however, an RPC application can only use UDP if it guarantees that every possible request and reply message fits in a "reasonable" size for datagram-oriented delivery—typically around 8KB by current conventions.

By running RPC in "datagram mode" over SST, the application can obtain the same efficient delivery for small requests via the optimization described in Section 4.4.3, while still being able to rely on the transport to convey occasional larger requests reliably. Alternatively, RPC could run over SST using one transaction per stream as discussed above for HTTP, using SST to associate requests with replies and ensuring reliable delivery of all messages, at minimal performance cost since substreams impose no round-trip startup delays or serialization. If serialization is actually desired, running RPC in "stream mode" over one SST stream eliminates the need for RPC's record marking layer since SST provides reliable record marking.

4.3.4 Out-of-Band Signaling

Many application protocols require some form of out-of-band signaling, and traditionally must use ad hoc methods to implement it due to the lack of transport layer support. Internet telephony sets

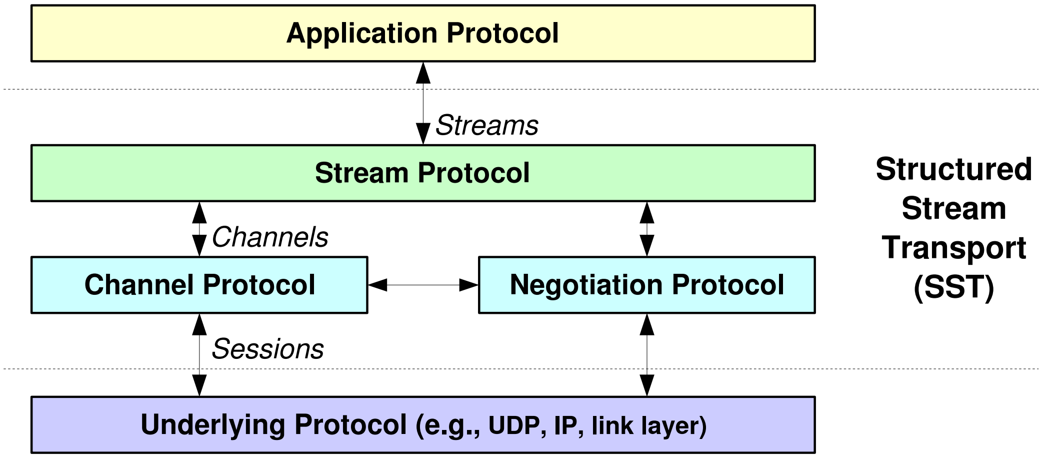


Figure 4-1: SST protocol architecture

up two separate UDP media sessions, for example: an RTP stream for data and a parallel RTCP stream for related out-of-band signaling [214]. With SST, an application need not set up out-of-band control channels in advance: the application can create a new substream at any time in which to convey an out-of-band message relating to the parent stream, without interfering with the normal operation of the parent. Section 7.5 explores one experimental use of out-of-band signaling in SST.

TCP's Urgent feature is a limited out-of-band signaling mechanism, used primarily in terminal protocols such as Telnet [186] and Rlogin [128] to clear the communication pipeline when the user invokes an "attention" signal such as CTRL-C. Although SST does not support the Urgent feature, an SST application can emulate it by opening an ephemeral substream and transmitting an explicit urgent pointer, expressed as a byte offset in the parent stream for example.

In the specific case of terminal applications, while TCP reliably delivers all data up to the Urgent pointer, terminal applications typically just discard this intervening data—a semantic mismatch resulting in unnecessary network transmission and delays. Instead of just emulating the Urgent facility, a terminal application designed for SST could achieve better responsiveness by using its main stream only for control and opening two substreams for character data, one for each direction. To clear its character transmit pipeline, a host resets its character send substream and opens a new one, prepending a generation counter to each new stream so the receiver can distinguish the most recent. SST immediately stops sending obsolete data in the old stream and promptly begins delivering the new stream, as desired for the terminal application's semantics. (Adapting a particular existing protocol such as Telnet or Rlogin this way would involve additional subtleties such as avoiding the loss of control signals embedded in the character stream; this example is intended only to suggest a general approach and not a precise recipe.)

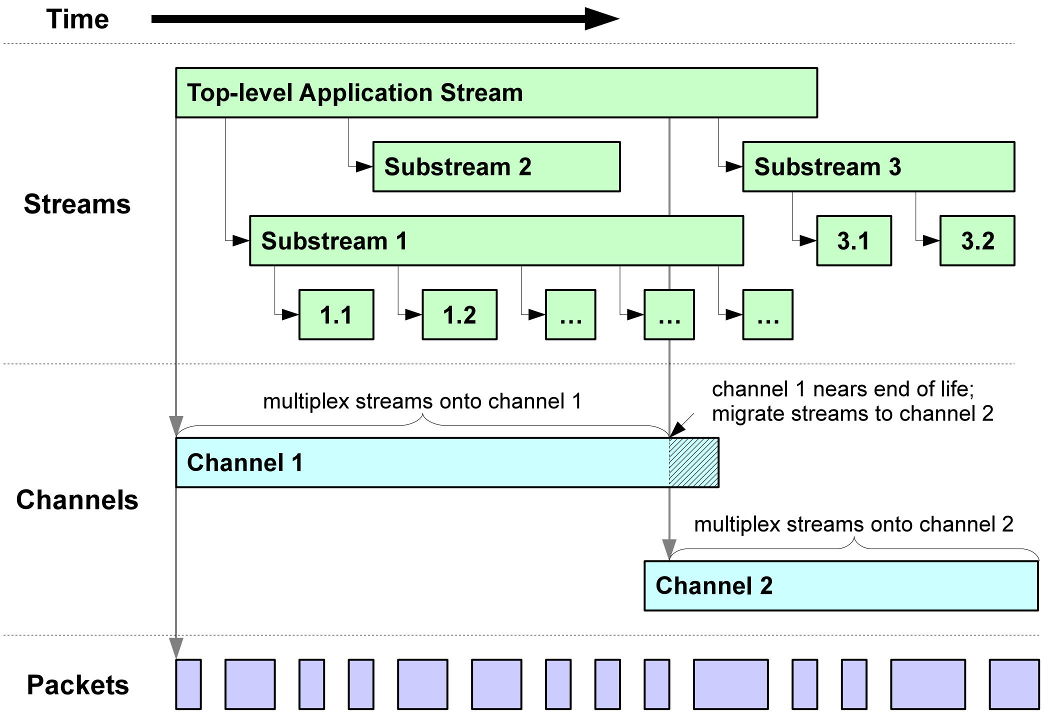

Time
Streams
Top-level Application Stream
Substream 2
Substream 3
Substream 1
3.1
3.2
1.1
1.2
...
...
...
Channels
multiplex streams onto channel 1
channel 1 nears end of life;
migrate streams to channel 2
Channel 1
multiplex streams onto channel 2
Channel 2
Packets

Figure 4-2: SST communication abstractions

4.4 Protocol Design

SST consists of three related protocols, organized as shown in Figure 4-1. The *channel protocol* is a connection-oriented best-effort delivery service that provides packet sequencing, integrity and privacy protection, selective acknowledgment, and congestion control. The *negotiation protocol* sets up the channel protocol's state, negotiating shared security keys and optional features. Finally, the *stream protocol* builds on the channel and negotiation protocols to implement the reliable structured stream abstraction SST presents to the application. This thesis focuses on the channel and stream protocols, which are of primary importance for implementing the structured stream abstraction.

4.4.1 Packets, Channels, and Streams

Figure 4-2 illustrates three key communication abstractions that form the interfaces between SST's layers—packets, channels, and streams—and how instances of these communication abstractions relate to each other over time.

At the lowest level, SST assumes that the underlying protocol (e.g., IP or UDP) provides only an unreliable, connectionless, unicast packet delivery service, in which packets are independent of each other and have ephemeral lifetimes. Each packet carries some number of bytes, up to the network path's current maximum transfer unit (MTU), which may change at any time. All communication within a channel or stream ultimately reduces to a series of packets at this level.

On top of this connectionless service SST's channel protocol builds a connection-oriented packet delivery service. Semantically, the channel protocol's most important functions are to associate each transmitted packet with a particular connection instance or *channel*, and to assign monotonically increasing *packet sequence numbers* to all packets transmitted within a particular channel. The channel protocol also attaches acknowledgment information to packets to determine when packets have arrived successfully, and uses this acknowledgment information internally to implement congestion control at channel granularity. Finally, the channel protocol protects each packet with a message authenticator and an optional encryption wrapper to provide end-to-end security for all packets transmitted over the channel.

The stream protocol builds on this intermediate channel abstraction to provide TCP-like reliable byte streams to the application. The stream protocol handles segmentation and reassembly, retransmission of lost segments, receiver-directed flow control, and substream creation and management. At any point in time, the stream protocol normally multiplexes all streams between the same pair of hosts onto a single channel. All streams multiplexed onto one channel therefore share that underlying channel's packet sequence number space, and the channel's congestion control, acknowledgment, and security state. Each stream has its own *byte sequence number* space, however, which the stream protocol uses to preserve the order of data segments within a stream in much the same way that TCP does.

The stream protocol leverages the channel protocol's association of packets with channel instances, and the channel protocol's monotonic packet sequence numbers, to manage per-stream state efficiently. The channel protocol's services in particular enable the stream protocol to avoid TCP's 3-way handshake on stream startup, and TCP's TIME-WAIT state after close, enabling lightweight stream creation and shutdown. The fact that a channel's sequence number space does not wrap facilitates efficient stream creation and termination, but this property also implies that the stream protocol must *migrate* long-lived streams from one channel to another to give streams unlimited lifetimes as in TCP.

Figure 4-3 shows the layout of a typical SST packet, not including any lower-layer headers for the underlying packet delivery service. SST can be layered directly atop most any best-effort packet delivery service, such as directly atop IP, atop UDP for compatibility with NATs and firewalls on today's Internet, or atop UIA's overlay routing layer (see Chapter 3). The channel header is always eight bytes in the current version of SST, and includes fields for channel identification, packet sequencing, and acknowledgment. The stream header is typically four or eight bytes depending on packet type, and contains fields for stream identification, receive window control, and sequencing bytes within a stream. Following the application payload, the channel protocol's trailing message authenticator is typically four bytes for lightweight TCP-grade security, and 12 or more bytes for cryptographic security.

The next section details SST's channel protocol, and Section 4.4.3 describes the stream protocol. Finally, Section 4.4.4 briefly outlines the negotiation protocol.

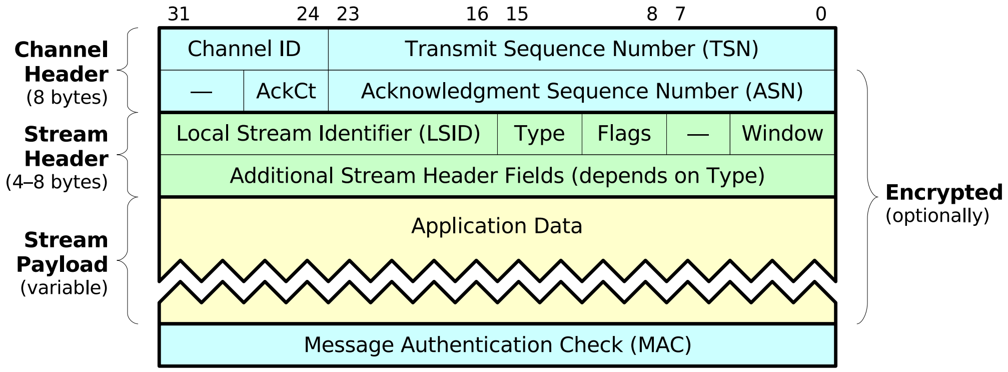

31
24
23
16
15
8
7
0
Channel Header (8 bytes)
Channel ID
Transmit Sequence Number (TSN)
—
AckCt
Acknowledgment Sequence Number (ASN)
Stream Header (4–8 bytes)
Local Stream Identifier (LSID)
Type
Flags
—
Window
Additional Stream Header Fields (depends on Type)
Stream Payload (variable)
Application Data
Message Authentication Check (MAC)
Encrypted (optionally)

Figure 4-3: SST packet layout

4.4.2 Channel Protocol

The channel protocol's purpose is to provide transport services that are independent of how the application "frames" its communication into protocol data units [51], or streams in SST. The communication state required to provide these services is thus shareable across many application streams. The channel protocol's services consist of associating transmitted packets with connection state instances or *channels*, assigning monotonic sequence numbers to packets within a channel, providing integrity and optional privacy protection for each packet, acknowledging packets and passing this acknowledgment information to upper layers, and providing congestion control.

SST borrows its connection state, sequencing, and packet security model from IPsec [132] instead of TCP. SST channels are inspired by IPsec security associations, and SST packet sequence numbers work like those IPsec uses for replay protection in the AH and ESP [131]. While IPsec "hides" its sequencing semantics from upper layers in the interest of operating invisibly to existing transports, one of SST's design insights is that this sequencing model provides a useful building block for new, richer transport services.

The following sections detail the channel protocol's identification, sequencing, security, acknowledgment, and congestion control services. While one packet submitted to the channel protocol currently translates to one packet in the underlying protocol (e.g., IP or UDP), the channel protocol could be extended to provide services such as chunk bundling [232] or quality-of-service enhancement [238].

Channel Identification

SST's negotiation protocol sets up a channel's initial state when an application first connects to a target host: each run of the negotiation protocol yields a new channel. As part of channel negotiation, each host assigns a *channel ID* for each direction of flow. Each transmitted packet includes the receiver's channel ID, enabling the receiver to find the associated channel instance state; channel IDs thus play the role of IPsec's security parameters index (SPI). A channel's ID may be reassigned immediately once the channel's lifetime ends: the packet authenticator described below rejects old or replayed packets from prior channel instances. As SST's channel protocol is designed for use by the stream protocol, which needs at most a few active channels at once, the channel ID field may be relatively small (currently 8 bits). Channel zero is reserved for the negotiation protocol.

Packet Sequencing and Replay Protection

TCP uses its byte sequence numbers for three different purposes: to distinguish and order data segments within a connection instance, to distinguish segments belonging to entirely different connection instances [239,246], and to protect against packet forgery [26]. SST uses its packet sequence numbers for only the first purpose, leaving the other two functions to an explicit packet authenticator keyed with a secret unique to the channel instance, as described below.

Each channel has a 64-bit packet sequence number space for each direction of flow, from which the channel protocol assigns sequence numbers consecutively to each packet transmitted. As in IPsec or DCCP [136], every packet sent gets a new sequence number, including acknowledgments and retransmissions. The sequence number space does not wrap, so if a host sends 2^{64} packets on one channel, it negotiates and switches to a new channel with a fresh space. A channel's lifetime may also be limited in other ways, such as by key expiration or a change in a host's IP address. If a host foresees the imminent expiration of a channel, it can begin negotiating a new one while still using the old one for a seamless handover.

Like IPsec's 64-bit sequence number extension [131], SST authenticates packets using full sequence numbers but transmits only the low bits explicitly in each packet. While the size of the sequence number space limits a channel's total lifetime, the size of the sequence number field in the packet header limits the *window* of packets a host may transmit within one round-trip. SST's current 24-bit sequence number field requires the sender to stay less than about 2^{22} packets ahead of the receiver to ensure that the receiver extrapolates sequence numbers correctly. To achieve maximum performance on fast links with long delay, SST may need an optional header extension like DCCP's long header format [136], or a timestamp extension as in PAWS [122].

Also like IPsec, the channel protocol checks the sequence number of each received packet against a bit mask of sequence numbers recently received, and discards without further processing both duplicates and packets too old to be covered by the bit mask. The width of this bit mask therefore limits how far out of order within a channel's sequence space a packet may be delivered to upper-layer protocols; we call this value the channel's *mis-ordering limit* (MOL). While IPsec hides its replay protection behavior from higher-level protocols, SST exposes this behavior, simplifying the stream layer's state management as described later in Section 4.4.3.

As one way to protect upper layers from out-of-order packets at critical synchronization points, the sequencing layer can provide *barriers* analogous to the memory barriers used in weakly consistent processor architectures [221]. When an upper layer requests a barrier, the sequencing layer adjusts its replay protection state to reject any packet that subsequently arrives with a sequence number lower than the highest it has seen so far. Barriers are easy for the sequencing layer to implement, and they enable the stream layer to avoid TIME-WAIT state retention. Setting barriers too frequently can reduce communication efficiency by limiting the channel layer's ability to accept packets received out of order, however, so this facility should be used sparingly by upper layers.

Authentication and Encryption

SST's packet security follows the design of IPsec's ESP [131], including its use of "Encrypt-then-MAC" composition as per accepted practice [25]. The negotiation protocol determines on channel setup the cryptographic algorithms and shared secrets used to protect the channel. A channel's keys and security parameters remain fixed throughout its lifetime: a host re-keys a connection by negotiating a new channel and migrating existing application streams to the new channel. Since SST uses a 64-bit packet sequence number at most once per channel, these sequence numbers can feed counter-based encryption modes [69], which are easily parallelized and require no plaintext expansion. SST's packet security therefore need not add any per-packet overhead other than the MAC.

If strong security is not needed, the application can disable encryption and request a lightweight 32-bit authenticator in place of a cryptographic MAC. The lightweight authenticator protects the channel against stale or replayed packets sent in prior channel instances, and against attackers who can inject forged packets but cannot monitor the communication stream, providing security comparable to other unsecured Internet transports like TCP. SST computes this authenticator by calculating an ordinary 32-bit checksum over the packet and a pseudo-header containing the full 64-bit sequence number, then adding two values to the resulting checksum: first, the same 4-microsecond timer that TCP uses to choose its initial sequence numbers [59, 246]; and second, a hash of the connection identifiers and a random internal secret, as in Bellovin's method of "keying" TCP ISNs [26]. SST's keyed checksum offers slightly better protection than TCP's keyed sequence numbers, because an SST attacker must blindly guess exactly the correct checksum key, not just any sequence number within some validity window.

Acknowledgments and Congestion Control

The channel protocol attaches acknowledgment information to each transmitted packet, and uses this information to implement congestion control. Each transmitted packet acknowledges a contiguous *range* of received sequence numbers, indicated in the packet header's Ack Sequence Number (ASN) and Ack Count (AckCt) fields. A host sends an acknowledgment for every one or two upper-level data segments received as in TCP [6], but sets the AckCt field as large as possible reflecting the most recent contiguous run of packets received. Successive acknowledgments therefore usually cover overlapping sequence number ranges, minimizing the effects of a lost acknowledgment. Figure 4-4 for example illustrates the behavior of a host that acknowledges every packet, upon receiving sequence numbers 1 through 7 in order except for a lost packet with sequence number 4.

Sending an acknowledgment range in each packet gives SST the benefits of selective acknowledgment without the overhead or complexity of variable-length SACK headers [154]. Because SST assigns each packet a fresh sequence number, including retransmissions of previously-transmitted

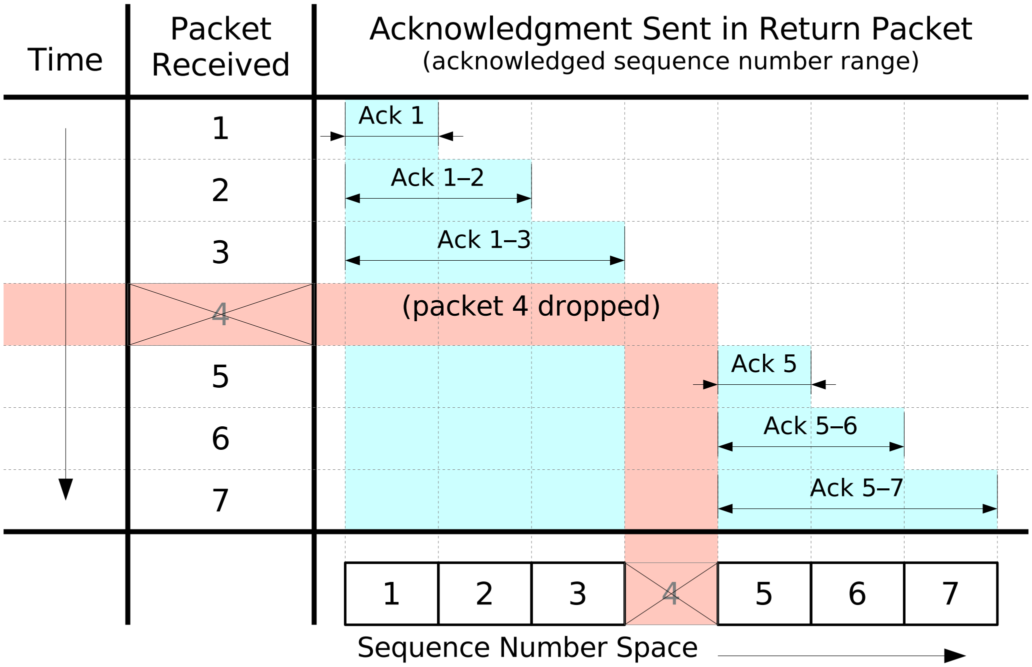


Figure 4-4: Packet acknowledgment example

data, its acknowledgments provide more information than even TCP with D-SACK [79]: the sender can tell exactly which copy of a retransmitted segment arrived and detect false retransmits before the receiver has seen duplication. SST can thus implement improvements to TCP congestion control [6] such as forward acknowledgment [155], reordering tolerance [32], and delay-sensitive schemes like TFRC [107]. As SST implements congestion control at channel granularity, applications may use many concurrent streams without behaving "antisocially" as with redundant TCP connections [78]. SST can negotiate at channel setup any protocol extensions necessary to implement congestion control schemes requiring cooperation with the receiver.

4.4.3 Stream Protocol

The stream layer multiplexes streams onto channels by dividing application data into *segments* according to the current maximum transfer unit (MTU) of the network path, and transmitting each segment as a separate packet on the current channel. The receiving stream layer accepts these potentially out-of-order segments and delivers them in order to the application. The sender uses the channel layer's packet-oriented acknowledgments to determine when a segment has been successfully received, so the stream layer requires no byte-oriented acknowledgments as in TCP.

SST's stream layer can migrate an active stream from one channel to another transparently to the application, giving streams potentially unlimited lifetimes independent of the limited lifetimes of the underlying channels. The stream layer can attach a stream to a new channel before detaching it from the old one, allowing the application to continue using the stream without interruption while transparently migrating the stream to the new channel. SST does not treat channel failure due to loss of connectivity as a "hard failure" like a TCP timeout. At the application's option, SST can retain stream state indefinitely until connectivity resumes and the negotiation protocol creates a new channel. At this point SST migrates the application's streams to the new channel and the application resumes where it left off. Stream migration also supports end-to-end host mobility [222, 223]: if a host's IP address changes, SST channels bound to that address fail, but if either endpoint can find the other's new address (e.g., one host is non-mobile or has a dynamic DNS name), SST can create a fresh channel between the new address pair and transparently migrate the application's streams to the new channel. Of course, if one endpoint host reboots, then its stream state is normally lost and the application must create new streams.

Because the stream protocol relies on the channel protocol for loss detection and congestion control, and these mechanisms operate within a particular channel or network path, the latency differences and resulting packet reordering across multiple paths should not confuse these mechanisms as they do for TCP.

Stream Identification and Attachment

When the stream layer creates a new stream on behalf of an application, it assigns the stream a *Unique Stream Identifier* or USID. A stream's USID is at least 128 bits, remains fixed through the stream's lifetime, and identifies the stream as it migrates from one channel to another. The stream layer must then *attach* the stream to a particular channel before it can transmit application data. In the process of attaching the stream to a channel, the stream layer assigns a shorter, temporary 16-bit *Local Stream Identifier* (LSID) to identify the stream efficiently within the scope of that particular channel. The stream layer uses this LSID in place of the stream's full USID in data segments it transmits on this channel. The stream keeps its USID but gets a new LSID each time SST attaches it to a different channel. The application is not normally aware of either USIDs or LSIDs.

While each stream has only one USID, LSIDs are specific not only to a channel but to each direction of flow within that channel. Each endpoint has its own LSID space for each channel, in which it assigns LSIDs to streams independently of its peer. Each host tracks both its own and its peer's LSID space, using its own LSID space to identify the stream for a data segment it transmits to its peer, and using its peer's LSID space to identify the stream for a data segment it receives. For bidirectional use, a stream must have two LSIDs, one assigned by each host for each direction of flow. A stream may in fact have more than one LSID at once for each flow direction: two in each direction, and all potentially on different channels, allowing seamless stream migration from one channel to another.

Root and Top-Level Application Streams

When the stream protocol initiates negotiation of a new channel, it creates and implicitly attaches a *root stream* to the channel. SST uses this root stream only for internal signaling: the application is unaware of its existence. When an application opens a "top-level" application stream via `connect`, SST first opens a channel to the desired host if none already exists, then creates a child of the channel's invisible root stream for the application's use. SST can reuse the same channel and root stream to create many top-level streams to the same target host, avoiding 3-way handshakes for successive top-level streams in the same way that it does for substreams the application creates via `create_substream`.

TCP uses its port numbers for two purposes: to distinguish among transport instances between the same pair of hosts, and to name application *rendezvous points* via well-known port numbers. SST splits these two functions, using LSIDs exclusively for stream multiplexing, and using explicit *service negotiation* for rendezvous. When the stream layer creates a new top-level stream, before handing this stream to the application, it first sends a message on the new stream to the responding host's stream layer, indicating the rendezvous point the initiating application wishes to connect to. The responding stream layer intercepts this message and replies indicating whether an application is listening at the given rendezvous point. On success, both stream layers then hand this top-level stream to the respective applications. An SST rendezvous point is currently a pair of strings: a *service name* (e.g., "`www`") and a *protocol name* (e.g., "`http`").

Creating Streams

To create a new child of a stream already attached to a channel, the stream layer sends an Init packet, shown in Figure 4-5. The initiator specifies the LSID it assigned the new stream and the LSID of the existing parent, both in the initiator's LSID space. The packet may also carry application data for the new stream, as described below.

Although Init packets carry no USIDs, the hosts must *agree* on a USID for the new stream so they can attach the stream to other channels. Each host uses a convention for assigning LSIDs that allows an Init packet's receiver to extrapolate a USID from the new stream's 16-bit LSID. Each host assigns LSIDs in its LSID space for a channel using the low 16 bits of a 64-bit counter, and its peer tracks this counter and extrapolates the full 64-bit value from a received LSID the same way the channel layer extrapolates packet sequence numbers. The hosts use these 64-bit counters, plus a pseudorandom nonce unique to the channel and flow direction, to agree on the new stream's USID implicitly. The initiator may skip counter values corresponding to LSIDs still in use by other streams, provided it doesn't get too far ahead of its peer and lose counter synchronization.

Upon receiving an Init packet with an unknown LSID, the responder records the new LSID, then sends a Reply packet to assign its own "return-path" LSID to the new stream for data transfer in the opposite direction. The Reply packet has the same format as an Init packet, except it contains the initiator's just-assigned LSID for the new stream in place of the parent stream's LSID (see Figure 4-5). The responder assigns LSIDs using its 64-bit counter as above to maintain counter synchronization, but does not use the counter to derive a USID since each stream requires only one USID.

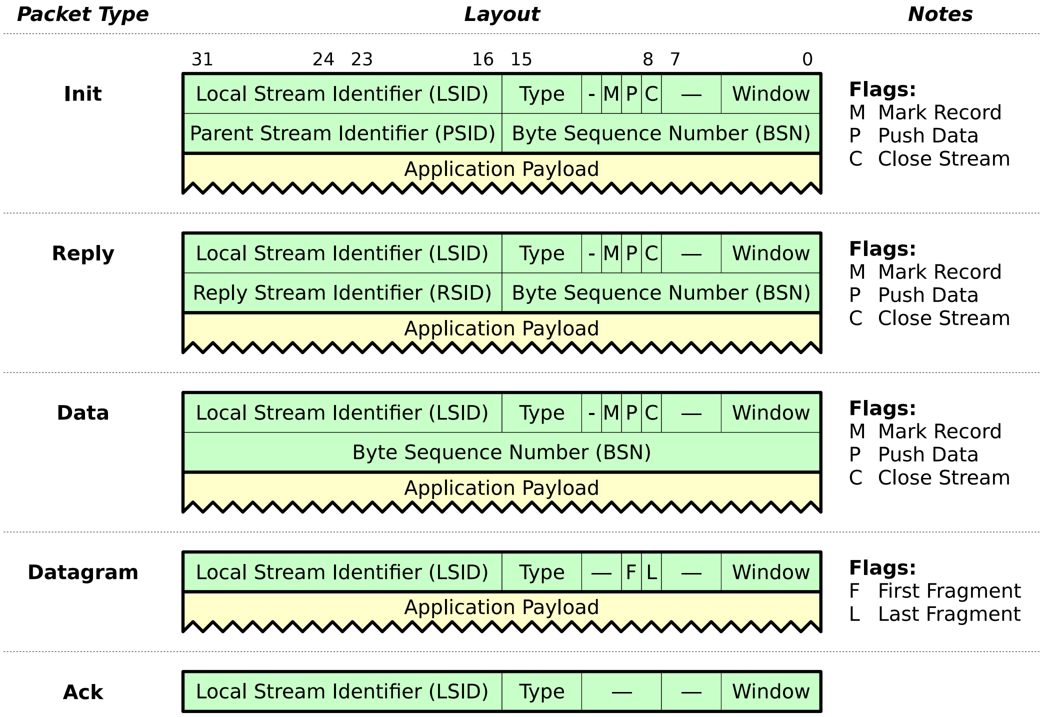


Figure 4-5: Stream data transfer packets

Data Transfer and Acknowledgment

Both Init and Reply packets may contain application data and stream control flags. The initiator uses Init packets to start sending data on the new stream immediately without waiting for the receiver's acknowledgment, eliminating TCP's 3-way handshake delay on new streams once a channel has been opened to the desired host. The responder may similarly start sending response data immediately via Reply packets. The Init and Reply packets contain a 16-bit Byte Sequence Number (BSN) field indicating the data segment's logical offset in the new stream, so a host can send up to $2^{16} - 1 + MTU$ bytes of data this way before it must start using using ordinary Data packets, which it can do only after receiving an acknowledgment indicating that the peer is aware of the newly assigned LSID.

Data packets have the same structure as Init and Reply packets, except that they have a 32-bit BSN and no PSID or RSID (see Figure 4-5). The stream layer uses this BSN to reassemble data segments in the correct order at the receiver, using wraparound arithmetic as in TCP, making the longevity of a stream effectively unlimited.

A host buffers each data segment it sends until it receives an acknowledgment for that segment, periodically retransmitting unacknowledged segments. Since the channel layer's acknowledgments refer to packet sequence numbers and not to byte sequence numbers, the sender records the packet sequence numbers the channel protocol assigns each data segment during recent transmission attempts, in order to lookup and free the segment when a corresponding acknowledgment arrives. Since the channel layer's packet sequence numbers are independent of the stream layer's LSIDs, a packet in one stream may effectively acknowledge segments in other streams. The stream layer uses a separate Ack packet type to send acknowledgments that cannot be piggybacked onto data flowing in the opposite direction.

Since the channel layer's acknowledgments are definitive, the receiver must only acknowledge a data segment once it has fully processed and locally buffered the segment. If the receiver cannot process a segment due to a temporary resource shortage, it may drop the segment without acknowledgment, but using flow control to avoid the need to drop segments is preferred since dropped segments trigger the sender's congestion control and affect the entire channel.

If a data segment already transmitted must be retransmitted with a smaller MTU, the sender "re-fragments" the segment into smaller segments for retransmission, adjusting the BSN fields in the new segments accordingly. A host may also repackage an Init or Reply packet's data into an ordinary Data packet for retransmission, if an acknowledgment for some other Init or Reply packet arrives in the meantime.

Init, Reply, and Data packets contain several flags indicating special stream events. The Push (P) and Close (C) flags work like TCP's PSH and FIN flags, whereas the Mark (M) flag is new:

- **Push (P)** indicates that the stream's data up through this segment should be pushed to the receiving application without waiting for more segments to arrive, exactly as in TCP.

- **Mark (M)** indicates the end of a message or record, directing the stream layer to deliver the segment as soon as possible, and *never* to combine it with subsequent data segments for delivery to the application, even if subsequent data is already available due to out-of-order delivery for example. The stream layer indicates this record boundary to the receiving application, so that the application does not have to embed its own markers in the stream.

- **Close (C)** indicates the last segment in the stream in one direction. After sending its final Close segment, the sender may continue receiving data as in TCP; the stream terminates after each participant closes its end or one issues a Reset. Section 4.4.3 below describes how SST garbage collects stream state after close.

Datagram Delivery

When the application submits a datagram to be sent as an "ephemeral substream" with best-effort semantics as described in Section 4.2.4, the stream layer checks that the datagram is small enough to ensure a reasonable chance of successful delivery, and if so transmits it using a sequence of Datagram packets shown in Figure 4-5. The first packet in the sequence has the First (F) flag set, the last packet has the Last (L) flag set, and the packets have consecutive sequence numbers, allowing the receiver to determine when a received datagram is complete. Each packet's LSID refers to the parent stream; the sender never assigns an LSID or maintains any state for the ephemeral child substream.

If the sending stream layer judges the ephemeral substream to be too large for delivery as a datagram, it sends the substream instead in standard reliable fashion using Init and Data packets, retransmitting individual segments as necessary, and closes the substream when finished. The receiving application obtains no indication of the actual delivery method by which the ephemeral substream arrived.

Flow Control

While congestion control operates at channel granularity, SST provides flow control for each stream individually, allowing the receiving application to accept data at different rates on each stream. Every packet the stream layer sends contains a receive window update, indicated in the header's 5-bit Window field. This field uses an exponential encoding: a value n indicates a window of *at least* $2^n - 1$ bytes. When the window is large, the sender does not need to know its size precisely since it will take a while to fill anyway, but the receiver's updates become more precise as its buffers fill and the window shrinks. To avoid a variant of silly window syndrome [53], the sender never fragments data segments just to make a partial segment fit into the receive window: instead it waits until the window can accommodate a full-size segment, or a short segment containing a Push marker.

TCP uses its cumulative acknowledgment position as a "base" from which to calculate the window horizon, but SST has no cumulative acknowledgments from which to calculate such a horizon. SST's window credit instead represents the total number of unacknowledged bytes the sender may have in flight. The receiver deducts from its advertised window the size of each segment it receives and acknowledges, and cancels this deduction once it delivers the segment to the application. The sender similarly deducts each segment it sends from its window credit and cancels this deduction when the segment is acknowledged, even if the segment was received and acknowledged out of order. The sender tracks the packet sequence numbers of window updates and always uses only the most recently-sent update.

For flow control, SST treats an Init packet's data as belonging to the *parent* stream—the stream specified in the packet's PSID field—although the data is semantically part of the new child stream. In effect, when a host sends data on a new stream without waiting for an initial window size from the responder, the sender "borrows" from the parent's receive window to send this initial data. This borrowing maintains proper flow control and avoids receive buffer overrun while allowing stream creation with no round-trip delay.

Detaching and Migrating Streams

The stream layer sends an Attach packet, shown in Figure 4-6, to attach an existing stream to a new channel. The Attach packet contains the LSID assigned by the sender and the permanent USID of the stream to be attached. The sender can attach a stream to a limited number of channels at once (currently two), and indicates via an *attachment slot number* which of these potential attachments it is using. The receiver looks up the stream by the specified USID, associates the specified attachment slot in that stream with the specified LSID in the channel on which the Attach packet arrived, and acknowledges the packet via the channel layer.

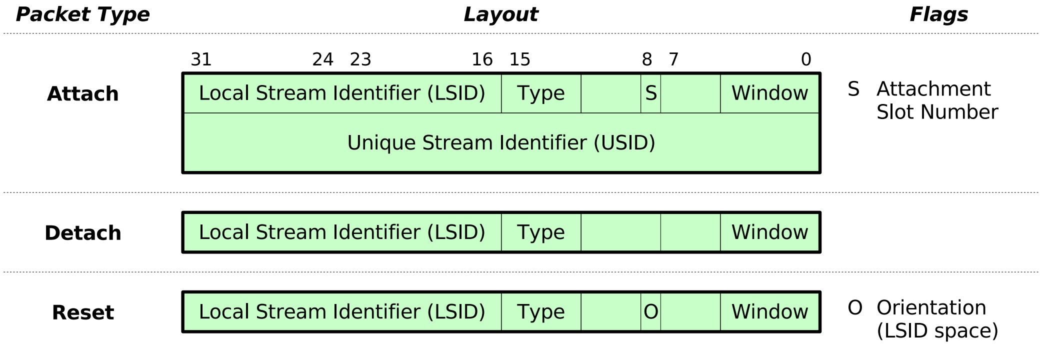

Packet Type
Layout
Flags
31
24
23
16
15
8
7
0
Attach
Local Stream Identifier (LSID)
Type
S
Window
Unique Stream Identifier (USID)
S Attachment Slot Number
Detach
Local Stream Identifier (LSID)
Type
Window
Reset
Local Stream Identifier (LSID)
Type
O
Window
O Orientation (LSID space)

Figure 4-6: Stream control packets

A host may detach a stream from a channel, freeing the stream's LSID in that channel for use by other streams, by sending a Detach packet (Figure 4-6). By detaching idle streams the application has not used for some time and treating its LSID space as a cache, SST can manage an arbitrary number of streams. Host API issues may impose limits on the number of open streams, such as Unix's file descriptor limit—but in the current user space SST prototype, which does not use file descriptors for streams, the number of open streams is limited only by available memory. Section 4.4.3 later describes how SST avoids confusing packets referring to different uses of the same LSID.

Forceful Reset

As in TCP, either host may unilaterally terminate an SST stream in both directions and discard any buffered data. A host resets a stream by sending a Reset packet (Figure 4-6) containing an LSID in either the sender's or receiver's LSID space, and an O (Orientation) flag indicating in which space the LSID is to be interpreted. When a host uses a Reset packet to terminate a stream it believes to be active, it uses its own LSID referring to the stream, and resends the Reset packet as necessary until it obtains an acknowledgment.

A host also sends a Reset in response to a packet it receives referring to an unknown LSID or USID. This situation may occur if the host has closed and garbage collected its state for a stream but one of its acknowledgments to its peer's data segments is lost in transit, causing its peer to retransmit those segments. The stateless Reset response indicates to the peer that it can garbage collect its stream state as well. Stateless Reset responses always refer to the peer's LSID space, since by definition the host itself does not have an LSID assigned to the unknown stream.

Garbage Collecting and Reusing LSIDs

An SST application that uses one stream per transaction may create and destroy streams rapidly: in the worst case, a host can create a stream, assign it an LSID, transmit up to an MTU of data, and close its end of the stream, all with one Init packet. The responder may similarly acknowledge the Init packet, send up to one MTU of response data, and close the stream, with one Reply packet. SST may therefore reuse 16-bit LSIDs for many successive streams within a channel's lifetime, leading to the risk of confusing packets referring to different uses of the same LSID. This is the same problem that at a lower level motivates TCP's ISN selection [239, 246] and the channel protocol's keyed authenticators (Section 4.4.2). SST's stream protocol, however, benefits from the channel protocol's monotonic packet sequence numbering and its mis-ordering limit described in Section 4.4.2.

To avoid confusing old and new uses of an LSID, after detaching an LSID the stream layer imposes a "quiet period" before creating or attaching another stream with the same LSID. This quiet period corresponds to TCP's TIME-WAIT, but SST counts the duration of this quiet period in packet sequence numbers instead of wall-clock time, relying on the channel layer's mis-ordering limit (MOL). With a 32 packet MOL, for example, after detachment a host waits for both its and its peer's sequence numbers to advance 32 packets beyond a point when both hosts know about the detachment. The channel's replay logic drops packets that arrive so late that they might confuse new stream attachments using this LSID. Because the MOL is a (typically small) constant, and one packet can attach and/or detach only one stream, the number of LSIDs that may be stuck in this quiet period is similarly small, regardless of packet rate. The mis-ordering limit thus avoids the need for time-bounded TIME-WAITs and eliminate the risk of state overload under heavy use [75].

The channel protocol's barrier mechanism, described earlier in Section 4.4.2, provides an alternative method of handling LSID reuse that can eliminate the "quiet period" completely. If the stream layer on a host sets a barrier once it has closed a stream or detached its LSID and received acknowledgment of that close or detach from the other host, the channel layer guarantees to the stream layer that no old packets sent before this point that might refer to the obsolete LSID will subsequently be delivered, even if they arrive out of order. Thus, setting an explicit barrier has the effect of temporarily setting the MOL to zero and allowing it to "grow" gradually back to its normal value as further (in-order) packets arrive. While barriers enable stream state to be garbage collected even more quickly than with a quiet time defined by the MOL, they have the potential disadvantage of unnecessarily preventing the delivery of other packets that might happen to arrive out of order, including packets associated with completely unrelated streams. For this reason, imposing a MOL-based quiet period should probably be considered the preferred approach.

4.4.4 The Negotiation Protocol

The negotiation protocol is responsible for setting up new channels with either weak or strong security. Negotiation with weak security sets up the unpredictable checksum key described in Section 4.4.2. The initiator may piggyback application data onto the first negotiation protocol packet, allowing channel setup with no effective round-trip overhead, but the responder may ignore this initial data and return a cookie challenge if it is loaded or under DoS attack.

The strong security mode uses Just Fast Keying [4] to establish shared cryptographic secrets and verify host identities using a simple, fixed four-message (two round-trip) exchange. The last two messages may carry piggybacked application data, for an effective minimum channel setup overhead of one round trip, identical to TCP's.

SST is designed to work with UIA ad hoc naming [84] and UIP routing [80] to support seamless communication among both fixed and mobile personal devices. UIA, UIP, and SST use cryptographic host identifiers analogous to those of HIP [163] to identify endpoints securely, so when a host's IP address changes, SST merely reruns the negotiation protocol to establish a channel between the new addresses, and migrates existing streams to the new channel.

Chapter 5

NAT Traversal

One of the primary goals of UIA is to support convenient peer-to-peer communication among personal mobile devices. Unfortunately, now-ubiquitous *middleboxes* on the Internet such as firewalls and Network Address Translators (NAT) [70, 250] cause well-known difficulties for peer-to-peer (P2P) communication, since the peers involved may not have a publicly reachable or even globally unique IP address [112, 226, 228]. While the UIA routing layer's ability to forward traffic through publicly reachable intermediate nodes provides a general-purpose method of traversing middleboxes, this explicit forwarding can increase end-to-end latency and unnecessarily burden these intermediate nodes. In some cases indirect overlay forwarding paths may be "optimized" into direct IP-level paths between peers located behind middleboxes, via *NAT traversal* techniques.

Several NAT traversal techniques are known, but their documentation is slim, and data about their robustness or relative merits is slimmer. This chapter presents the first thorough development and analysis of a simple but robust and practical NAT traversal technique, known as *hole punching*. While Dan Kegel publicly documented the basics of UDP hole punching in 1999 [129], the present work examines previously unaddressed scenarios such as the increasingly common situation of hosts behind multiple levels of NAT: e.g., behind both a home NAT router and a large ISP-deployed NAT. We also explore a clean, symmetric approach to TCP hole punching, which works over a substantial percentage of deployed NATs without relying on delicate, timing-sensitive techniques such as NAT port number prediction or address spoofing, as other NAT traversal proposals for TCP do [30, 72, 100]. To evaluate the effectiveness of the proposed hole punching techniques, Chapter 7 will later present the first published empirical study on the compatibility of a wide variety of NATs deployed "in the wild" with TCP and UDP hole punching techniques.

Section 5.1 first introduces NAT traversal background, terminology, and basic concepts, and Section 5.2 summarizes hole punching techniques for UDP, which are already moderately well-understood. Section 5.3 then explores how hole punching can be reliably used to set up peer-to-peer TCP streams as well. Finally, Section 5.4 summarizes middlebox properties these techniques depend on. Later in Chapter 7 we will evaluate the reliability of this technique on a wide variety of deployed NATs, finding that about 82% of the NATs tested support hole punching for UDP, and about 64% support hole punching for TCP streams. As NAT vendors become increasingly conscious

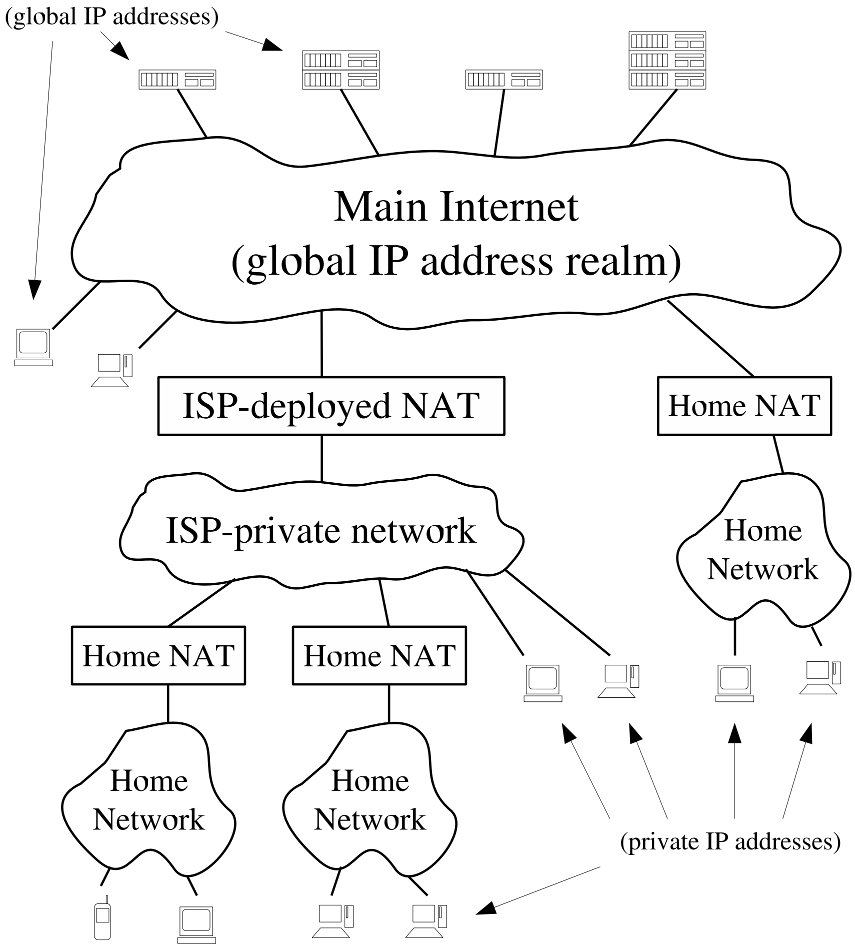


Figure 5-1: Public and private IP address domains

of the needs of important P2P applications such as Voice over IP and online gaming protocols, support for hole punching is likely to increase in the future, making this optimization ever more important for UIA and other networking architectures supporting peer-to-peer communication.

5.1 Background and NAT Traversal Concepts

The combined pressures of tremendous growth and massive security challenges have forced the Internet to evolve in ways that make life difficult for many applications. The Internet's original uniform address architecture, in which every node has a globally unique IP address and can communicate directly with every other node, has been replaced with a new *de facto* Internet address architecture, consisting of a global address realm and many private address realms interconnected by Network Address Translators (NAT). In this new address architecture, illustrated in Figure 5-1, only nodes in the "main," global address realm can be easily contacted from anywhere in the network, because only they have unique, globally routable IP addresses. Nodes on private networks can connect to other nodes on the same private network, and they can usually open TCP or UDP connections to “well-known” nodes in the global address realm. NATs on the path allocate temporary public endpoints for outgoing connections, and translate the addresses and port numbers in packets comprising those sessions, while generally blocking all incoming traffic unless otherwise specifically configured.

The Internet’s new *de facto* address architecture is suitable for client/server communication in the typical case when the client is on a private network and the server is in the global address realm. The architecture makes it difficult for two nodes on *different* private networks to contact each other directly, however, which is often important to the “peer-to-peer” communication protocols used in applications such as teleconferencing, online gaming, and sharing information among personal devices as UIA is intended to support. We clearly need a way to make such protocols function smoothly in the presence of NAT.

One of the most effective methods of establishing peer-to-peer communication between hosts on different private networks is known as “hole punching.” This technique is already widely used in UDP-based applications, but essentially the same technique also works for TCP. Contrary to what its name may suggest, hole punching does not compromise the security of a private network. Instead, hole punching enables applications to function *within* the the default security policy of most NATs, effectively signaling to NATs on the path that peer-to-peer communication sessions are “solicited” and thus should be accepted. This chapter documents hole punching for both UDP and TCP, and details the crucial aspects of both application and NAT behavior that make hole punching work.

Unfortunately, no traversal technique works with all existing NATs, because NAT behavior is not standardized. Chapter 7 presents some experimental results evaluating hole punching support in current NATs. This data is derived from results submitted by users throughout the Internet by running “NAT Check,” a tool specially designed for this experiment, over a wide variety of NATs by different vendors. While the data points were gathered from a “self-selecting” user community and may not be representative of the true distribution of NAT implementations deployed on the Internet, the results are nevertheless generally encouraging.

While evaluating basic hole punching, we also point out variations that can make hole punching work on a wider variety of existing NATs at the cost of greater complexity. Our primary focus, however, is on developing the *simplest* hole punching technique that works cleanly and robustly in the presence of “well-behaved” NATs in any reasonable network topology. We deliberately avoid excessively clever tricks that may increase compatibility with some existing “broken” NATs in the short term, but which only work some of the time and may cause additional unpredictability and network brittleness in the long term.

Although the larger address space of IPv6 [62] may eventually reduce the need for NAT, in the short term IPv6 is *increasing* the demand for NAT, because NAT itself provides the easiest way to achieve interoperability between IPv4 and IPv6 address domains [249]. Further, the anonymity and inaccessibility of hosts on private networks has widely perceived security and privacy benefits. Firewalls are unlikely to go away even when there are enough IP addresses: IPv6 firewalls will still commonly block unsolicited incoming traffic by default, making hole punching useful even to IPv6 applications.

5.1.1 Terminology

We adopt the NAT terminology and taxonomy defined in RFC 2663 [228], as well as additional terms defined more recently in RFC 3489 [207] and RFC 4787 [16]. Readers are directed to the above RFCs for detailed information about terminology, but the most basic terms and concepts are summarized below.

- **Session:** Of particular importance is the notion of session. A *session endpoint* for TCP or UDP is an (IP address, port number) pair, and a particular *session* is uniquely identified by its two session endpoints. From the perspective of one of the hosts involved, a session is effectively identified by the 4-tuple (local IP, local port, remote IP, remote port). The *direction* of a session is normally the flow direction of the packet that initiates the session: the initial SYN packet for TCP, or the first user datagram for UDP.

- **Traditional NAT:** Of the various flavors of NAT [228], the most common type is *traditional* or *outbound* NAT [226], which provides an asymmetric bridge between a private network and a public network. Outbound NAT by default allows only outbound sessions to traverse the NAT: incoming packets are dropped unless the NAT identifies them as being part of an existing session initiated from within the private network. Outbound NAT conflicts with peer-to-peer protocols because when both peers desiring to communicate are “behind” (on the private network side of) two different NATs, whichever peer tries to initiate a session, the other peer’s NAT rejects it. NAT traversal entails making P2P sessions look like “outbound” sessions to *both* NATs.

- **Basic NAT versus NAPT:** Outbound NAT has two sub-varieties: *Basic NAT*, which only translates IP addresses, and *Network Address/Port Translation* (NAPT), which translates entire session endpoints. NAPT, the more general variety, has also become the most common because it enables many hosts on a private network to share the use of a *single* public IP address. Throughout this chapter we assume NAPT, though the principles and techniques we discuss apply equally well (if sometimes trivially) to Basic NAT.

- **Loopback/Hairpin Translation:** When a host in the private domain of a NAT device attempts to connect with another host behind the same NAT device using the public address of the host, the NAT device performs the equivalent of a “Twice-NAT” translation on the packet as follows. The originating host’s private endpoint is translated into its assigned public endpoint, and the target host’s public endpoint is translated into its private endpoint, before the packet is forwarded to the target host. We refer to the above translation performed by a NAT device as *loopback translation*; it may also be referred to as *hairpin translation*.

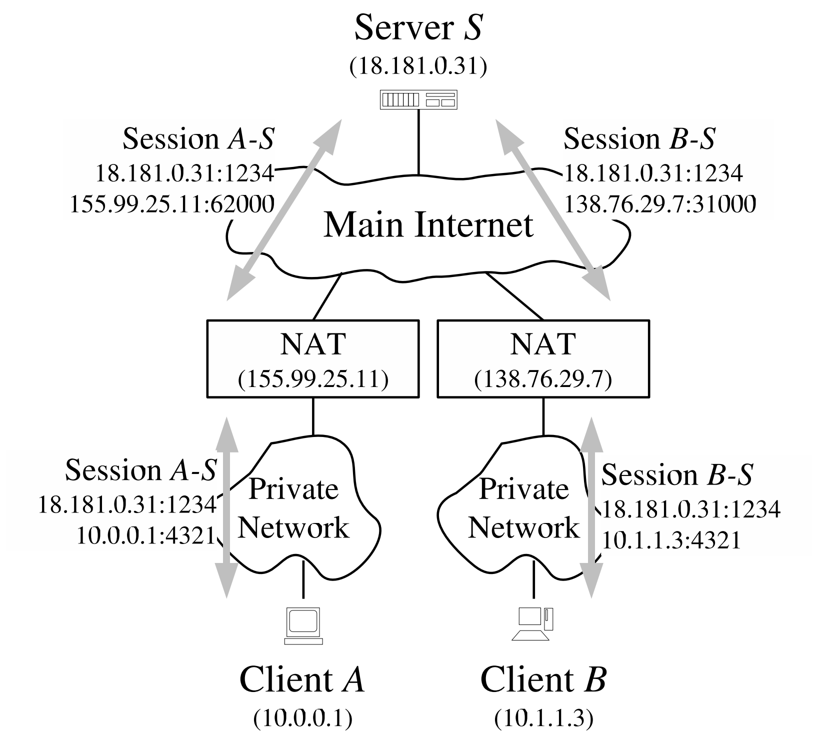


Figure 5-2: NAT Traversal by Relaying

5.1.2 Relaying

The most reliable but least efficient method of P2P communication across NAT is simply to make the communication look to the network like standard client/server communication, through relaying. Suppose two client hosts A and B have each initiated TCP or UDP connections to a well-known server S, at S's global IP address 18.181.0.31 and port number 1234. As shown in Figure 5-2, the clients reside on separate private networks, and their respective NATs prevent either client from directly initiating a connection to the other. Instead of attempting a direct connection, the two clients simply use the server S to relay messages between them. For example, to send a message to client B, client A sends the message to server S along its already-established client/server connection, and server S forwards the message on to client B using its existing client/server connection with B.

Relaying always works as long as both clients can connect to the server. Its disadvantages are that it consumes the server's processing power and network bandwidth for every data packet forwarded this way, and communication latency between the peering clients is likely increased even if the server is well-connected. Nevertheless, since there is no more efficient technique that works reliably on *all* existing NATs, relaying is a useful fall-back strategy if maximum robustness is desired. The TURN protocol [208] defines a method of implementing relaying in a relatively secure fashion.

UIA's overlay routing layer effectively provides a generalized relaying service, which can create indirect paths between UIA nodes even when such a path must traverse several intermediate

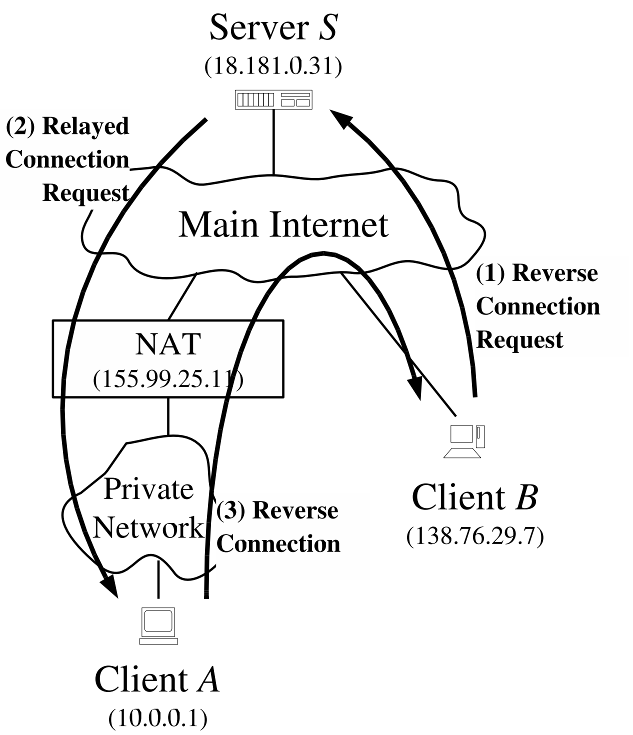


Figure 5-3: NAT Traversal by Connection Reversal

nodes or must be relayed for purposes other than NAT traversal (e.g., to cross between different network technologies, such as IP and Bluetooth). In the UIA worldview, the NAT traversal solutions developed in the following sections represents optimizations that permit the dynamic creation of "shortcuts" that bypass these intermediate relay hops at the IP level, once the P2P connection is established with the help of the intermediary. Since UIA treats NAT traversal only as an optimization, it is acceptable that NAT traversal does not work on 100% of existing NATs, as we will see in Chapter 7: when hole punching does not work, UIA can always fall back on less efficient, explicit relaying.

5.1.3 Connection Reversal

Some P2P applications use a straightforward but limited technique, known as *connection reversal*, to enable communication when both hosts have connections to a well-known rendezvous server S and only one of the peers is behind a NAT, as shown in Figure 5-3. If A wants to initiate a connection to B, then a direct connection attempt works automatically, because B is not behind a NAT and A's NAT interprets the connection as an outgoing session. If B wants to initiate a connection to A, however, any direct connection attempt to A is blocked by A's NAT. B can instead relay a connection request to A through a well-known server S, asking A to attempt a "reverse" connection back to B. Despite the obvious limitations of this technique, the central idea of using a well-known rendezvous server as an intermediary to help set up direct peer-to-peer connections is fundamental to the more general hole punching techniques described next.

5.2 UDP Hole Punching

UDP hole punching enables two clients to set up a direct peer-to-peer UDP session with the help of a well-known rendezvous server, even if the clients are both behind NATs. This technique was mentioned in section 5.1 of RFC 3027 [112], documented more thoroughly elsewhere on the Web [129], and used in recent experimental Internet protocols [116, 205]. Various proprietary protocols, such as those for on-line gaming, also use UDP hole punching.

5.2.1 The Rendezvous Server

Hole punching assumes that the two clients, A and B, already have active UDP sessions with a rendezvous server S. When a client registers with S, the server records *two* endpoints for that client: the (IP address, UDP port) pair that the client *believes* itself to be using to talk with S, and the (IP address, UDP port) pair that the server *observes* the client to be using to talk with it. We refer to the first pair as the client's *private* endpoint and the second as the client's *public* endpoint. The server might obtain the client's private endpoint from the client itself in a field in the body of the client's registration message, and obtain the client's public endpoint from the source IP address and source UDP port fields in the IP and UDP headers of that registration message. If the client is *not* behind a NAT, then its private and public endpoints should be identical.

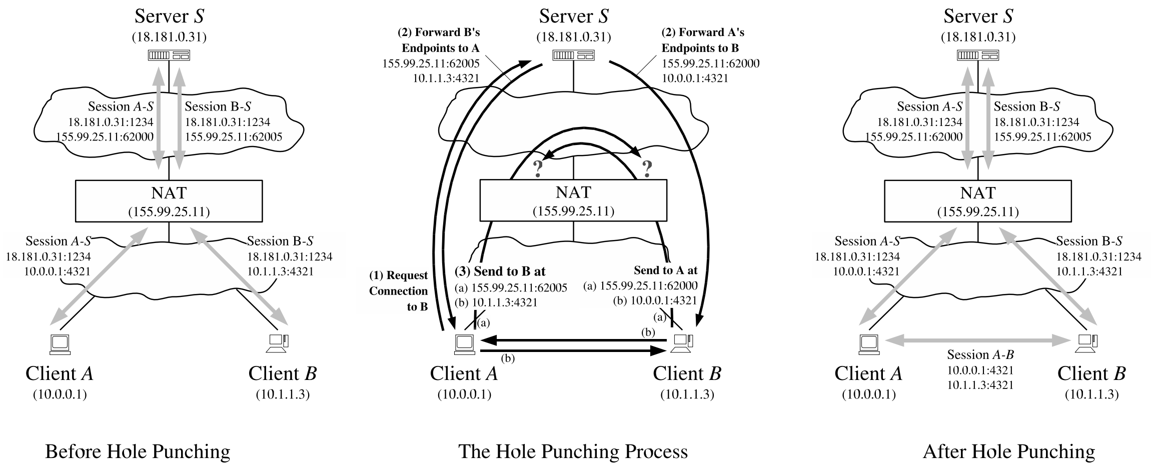
A few poorly behaved NATs are known to scan the body of UDP datagrams for 4-byte fields that look like IP addresses, and translate them as they would the IP address fields in the IP header. To be robust against such behavior, applications may wish to obfuscate IP addresses in message bodies slightly, for example by transmitting the one's complement of the IP address instead of the IP address itself. Of course, if the application is encrypting its messages, then this behavior is not likely to be a problem.

5.2.2 Establishing Peer-to-Peer Sessions

Suppose client A wants to establish a UDP session directly with client B. Hole punching proceeds as follows:

1. A initially does not know how to reach B, so A asks S for help establishing a UDP session with B.

2. S replies to A with a message containing B's public *and* private endpoints. At the same time, S uses its UDP session with B to send B a connection request message containing A's public and private endpoints. Once these messages are received, A and B know each other's public and private endpoints.



Before Hole Punching

The Hole Punching Process

After Hole Punching

Figure 5-4: UDP hole punching, peers behind a common NAT

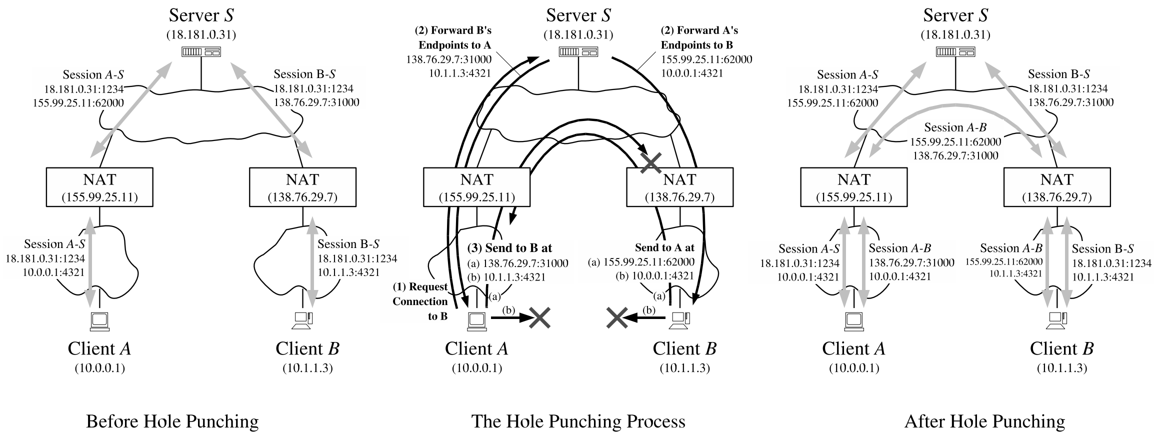
3. When A receives B's public and private endpoints from S, A starts sending UDP packets to *both* of these endpoints, and subsequently "locks in" whichever endpoint first elicits a valid response from B. Similarly, when B receives A's public and private endpoints in the forwarded connection request, B starts sending UDP packets to A at each of A's known endpoints, locking in the first endpoint that works. The order and timing of these messages are not critical as long as they are asynchronous.

We now consider how UDP hole punching handles each of three specific network scenarios. In the first situation, representing the "easy" case, the two clients actually reside behind the same NAT, on one private network. In the second, most common case, the clients reside behind different NATs. In the third scenario, the clients each reside behind *two* levels of NAT: a common "first-level" NAT deployed by an ISP for example, and distinct "second-level" NATs such as consumer NAT routers for home networks.

It is in general difficult or impossible for the application itself to determine the exact physical layout of the network, and thus which of these scenarios (or the many other possible ones) actually applies at a given time. Protocols such as STUN [207] can provide some information about the NATs present on a communication path, but this information may not always be complete or reliable, especially when multiple levels of NAT are involved. Nevertheless, hole punching works automatically in all of these scenarios *without* the application having to know the specific network organization, as long as the NATs involved behave in a reasonable fashion. ("Reasonable" behavior for NATs will be described later in Section 5.4.)

5.2.3 Peers Behind a Common NAT

First consider the simple scenario in which the two clients (probably unknowingly) happen to reside behind the same NAT, and are therefore located in the same private IP address realm, as shown in Figure 5-4. Client A has established a UDP session with server S, to which the common NAT has



Before Hole Punching

The Hole Punching Process

After Hole Punching

Figure 5-5: UDP hole punching, peers behind different NATs

assigned its own public port number 62000. Client B has similarly established a session with S, to which the NAT has assigned public port number 62005.

Suppose that client A uses the hole punching technique outlined above to establish a UDP session with B, using server S as an introducer. Client A sends S a message requesting a connection to B. S responds to A with B's public and private endpoints, and also forwards A's public and private endpoints to B. Both clients then attempt to send UDP datagrams to each other directly at each of these endpoints. The messages directed to the public endpoints may or may not reach their destination, depending on whether or not the NAT supports hairpin translation as described below in Section 5.2.5. The messages directed at the private endpoints *do* reach their destinations, however, and since this direct route through the private network is likely to be faster than an indirect route through the NAT anyway, the clients are most likely to select the private endpoints for subsequent regular communication.

By assuming that NATs support hairpin translation, the application might dispense with the complexity of trying private as well as public endpoints, at the cost of making local communication behind a common NAT unnecessarily pass through the NAT. As our results in Section 7.6 show, however, hairpin translation is still much less common among existing NATs than are other "P2P-friendly" NAT behaviors. For now, therefore, applications may benefit substantially by using both public and private endpoints.

5.2.4 Peers behind different NATs

Suppose clients A and B have private IP addresses behind different NATs, as shown in Figure 5-5. A and B have each initiated UDP communication sessions from their local port 4321 to port 1234 on server S. In handling these outbound sessions, NAT A has assigned port 62000 at its own public IP address, 155.99.25.11, for the use of A's session with S, and NAT B has assigned port 31000 at its IP address, 138.76.29.7, to B's session with S.

In A's registration message to S, A reports its private endpoint to S as 10.0.0.1:4321, where 10.0.0.1 is A's IP address on its own private network. S records A's reported private endpoint, along with A's public endpoint as observed by S itself. A's public endpoint in this case is 155.99.25.11: 62000, the temporary endpoint assigned to the session by the NAT. Similarly, when client B registers, S records B's private endpoint as 10.1.1.3:4321 and B's public endpoint as 138.76.29.7:31000.

Now client A follows the hole punching procedure described above to establish a UDP communication session directly with B. First, A sends a request message to S asking for help connecting with B. In response, S sends B's public and private endpoints to A, and sends A's public and private endpoints to B. A and B each start trying to send UDP datagrams directly to each of these endpoints.

Since A and B are on different private networks and their respective private IP addresses are not globally routable, the messages sent to these endpoints will reach either the wrong host or no host at all. Because many NATs also act as DHCP servers, handing out IP addresses in a fairly deterministic way from a private address pool usually determined by the NAT vendor by default, it is quite likely in practice that A's messages directed at B's private endpoint will reach *some* (incorrect) host on A's private network that happens to have the same private IP address as B does. Applications must therefore authenticate all messages in some way to filter out such stray traffic robustly. The messages might include application-specific names or cryptographic tokens, for example, or at least a random nonce pre-arranged through S.

Now consider A's first message sent to B's public endpoint, as shown in Figure 5-5. As this outbound message passes through A's NAT, this NAT notices that this is the first UDP packet in a new outgoing session. The new session's source endpoint (10.0.0.1:4321) is the same as that of the existing session between A and S, but its destination endpoint is different. If NAT A is well-behaved, it preserves the identity of A's private endpoint, consistently translating *all* outbound sessions from private source endpoint 10.0.0.1:4321 to the corresponding public source endpoint 155.99.25.11:62000. A's first outgoing message to B's public endpoint thus, in effect, "punches a hole" in A's NAT for a new UDP session identified by the endpoints (10.0.0.1:4321, 138.76.29.7:31000) on A's private network, and by the endpoints (155.99.25.11:62000, 138.76.29.7: 31000) on the main Internet.

If A's message to B's public endpoint reaches B's NAT before B's first message to A has crossed B's own NAT, then B's NAT may interpret A's inbound message as unsolicited incoming traffic and drop it. B's first message to A's public address, however, similarly opens a hole in B's NAT, for a new UDP session identified by the endpoints (10.1.1.3:4321, 155.99.25.11:62000) on B's private network, and by the endpoints (138.76.29.7:31000, 155.99.25.11:62000) on the Internet. Once the first messages from A and B have crossed their respective NATs, holes are open in each direction and UDP communication can proceed normally. Once the clients have verified that the public endpoints work, they can stop sending messages to the alternative private endpoints.

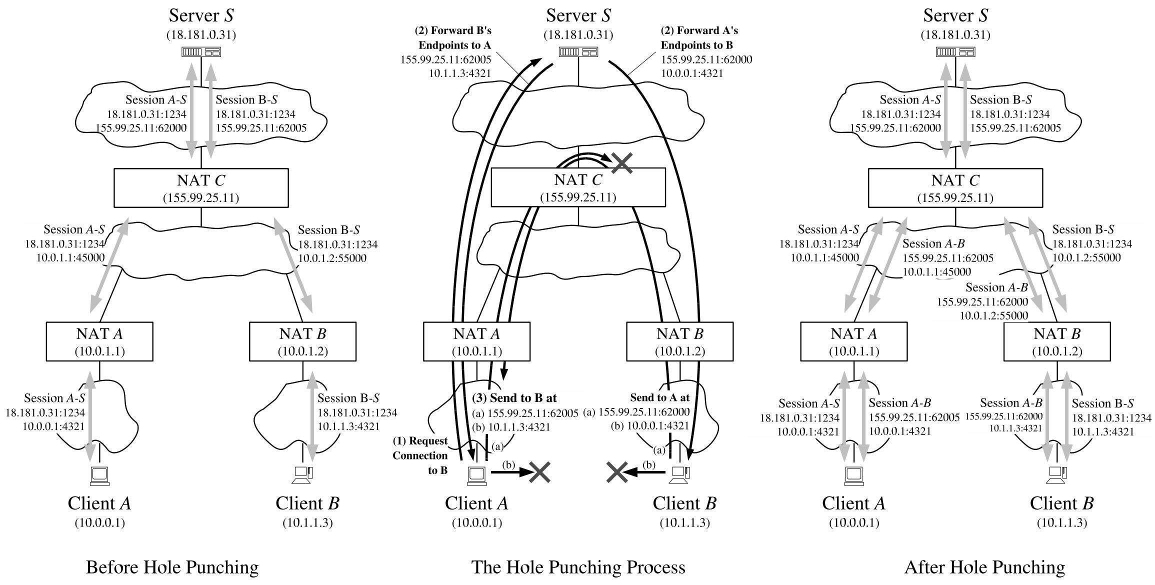


Figure 5-6: UDP hole punching, peers behind multiple levels of NAT

5.2.5 Peers Behind Multiple Levels of NAT

In some topologies involving multiple NAT devices, two clients cannot establish an "optimal" P2P route between them without specific knowledge of the topology. Consider a final scenario, depicted in Figure 5-6. Suppose NAT C is a large industrial NAT deployed by an internet service provider (ISP) to multiplex many customers onto a few public IP addresses, and NATs A and B are small consumer NAT routers deployed independently by two of the ISP's customers to multiplex their private home networks onto their respective ISP-provided IP addresses. Only server S and NAT C have globally routable IP addresses; the "public" IP addresses used by NAT A and NAT B are actually private to the ISP's address realm, while client A's and B's addresses in turn are private to the addressing realms of NAT A and NAT B, respectively. Each client initiates an outgoing connection to server S as before, causing NATs A and B each to create a single public/private translation, and causing NAT C to establish a public/private translation for each session.

Now suppose A and B attempt to establish a direct peer-to-peer UDP connection via hole punching. The optimal routing strategy would be for client A to send messages to client B's "semi-public" endpoint at NAT B, 10.0.1.2:55000 in the ISP's addressing realm, and for client B to send messages to A's "semi-public" endpoint at NAT B, namely 10.0.1.1:45000. Unfortunately, A and B have no way to learn these addresses, because server S only sees the truly global public endpoints of the clients, 155.99.25.11:62000 and 155.99.25.11:62005 respectively. Even if A and B had some way to learn these addresses, there is still no guarantee that they would be usable, because the address assignments in the ISP's private address realm might conflict with unrelated address assignments in the clients' private realms. (NAT A's IP address in NAT C's realm might just as easily have been 10.1.1.3, for example, the same as client B's private address in NAT B's realm.)

The clients therefore have no choice but to use their global public addresses as seen by S for their P2P communication, and rely on NAT C providing *hairpin* or *loopback* translation. When A sends a UDP datagram to B's global endpoint, 155.99.25.11:62005, NAT A first translates the datagram's source endpoint from 10.0.0.1:4321 to 10.0.1.1:45000. The datagram now reaches NAT C, which recognizes that the datagram's destination address is one of NAT C's own translated *public* endpoints. If NAT C is well-behaved, it then translates *both* the source and destination addresses in the datagram and "loops" the datagram back onto the private network, now with a source endpoint of 155.99.25.11:62000 and a destination endpoint of 10.0.1.2:55000. NAT B finally translates the datagram's destination address as the datagram enters B's private network, and the datagram reaches B. The path back to A works similarly. Many NATs do not yet support hairpin translation, but it is becoming more common as NAT vendors become aware of this issue.

5.2.6 UDP Idle Timeouts

Since the UDP transport protocol provides NATs with no reliable, application-independent way to determine the lifetime of a session crossing the NAT, most NATs simply associate an idle timer with UDP translations, closing the hole if no traffic has used it for some time period. There is unfortunately no standard value for this timer: some NATs have timeouts as short as 20 seconds. If the application needs to keep an idle UDP session active after establishing the session via hole punching, the application must send periodic keep-alive packets to ensure that the relevant translation state in the NATs does not disappear.

Unfortunately, many NATs associate UDP idle timers with individual UDP sessions defined by a particular pair of endpoints, so sending keep-alives on one session will not keep other sessions active even if all the sessions originate from the same private endpoint. Instead of sending keep-alives on many different P2P sessions, applications can avoid excessive keep-alive traffic by detecting when a UDP session no longer works, and re-running the original hole punching procedure again "on demand."

5.3 TCP Hole Punching

Establishing peer-to-peer TCP connections between hosts behind NATs is slightly more complex than for UDP, but TCP hole punching is remarkably similar at the protocol level. Since it is not as well-understood, it is currently supported by fewer existing NATs. When the NATs involved *do* support it, however, TCP hole punching is just as fast and reliable as UDP hole punching. Peer-to-peer TCP communication across well-behaved NATs may in fact be *more* robust than UDP communication, because unlike UDP, the TCP protocol's state machine gives NATs on the path a standard way to determine the precise lifetime of a particular TCP session.

5.3.1 Sockets and TCP Port Reuse

The main practical challenge to applications wishing to implement TCP hole punching is not a protocol issue but an application programming interface (API) issue. Because the standard Berkeley sockets API was designed around the client/server paradigm, the API allows a TCP stream socket to be used to initiate an outgoing connection via `connect()`, or to listen for incoming connections via `listen()` and `accept()`, *but not both.* Further, TCP sockets usually have a one-to-one correspondence to TCP port numbers on the local host: after the application binds one socket to a particular local TCP port, attempts to bind a second socket to the same TCP port fail.

For TCP hole punching to work, however, we need to use a single local TCP port to listen for incoming TCP connections and to initiate multiple outgoing TCP connections concurrently. Fortunately, all major operating systems support a special TCP socket option, commonly named `SO_REUSEADDR`, which allows the application to bind multiple sockets to the same local endpoint as long as this option is set on all of the sockets involved. BSD systems have introduced a `SO_REUSEPORT` option that controls port reuse separately from address reuse; on such systems *both* of these options must be set.

5.3.2 Opening Peer-to-Peer TCP Streams

Suppose that client A wishes to set up a TCP connection with client B. We assume as usual that both A and B already have active TCP connections with a well-known rendezvous server S. The server records each registered client's public and private endpoints, just as for UDP. At the protocol level, TCP hole punching works almost exactly as for UDP:

1. Client A uses its active TCP session with S to ask S for help connecting to B.

2. S replies to A with B's public and private TCP endpoints, and at the same time sends A's public and private endpoints to B.

3. From *the same local TCP ports* that A and B used to register with S, A and B each asynchronously make outgoing connection attempts to the other's public and private endpoints as reported by S, while simultaneously listening for incoming connections on their respective local TCP ports.

4. A and B wait for outgoing connection attempts to succeed, and/or for incoming connections to appear. If one of the outgoing connection attempts fails due to a network error such as "connection reset" or "host unreachable," the host simply re-tries that connection attempt after a short delay (e.g., one second), up to an application-defined maximum timeout period.

5. When a TCP connection is made, the hosts authenticate each other to verify that they connected to the intended host. If authentication fails, the clients close that connection and continue waiting for others to succeed. The clients use the first successfully authenticated TCP stream resulting from this process.

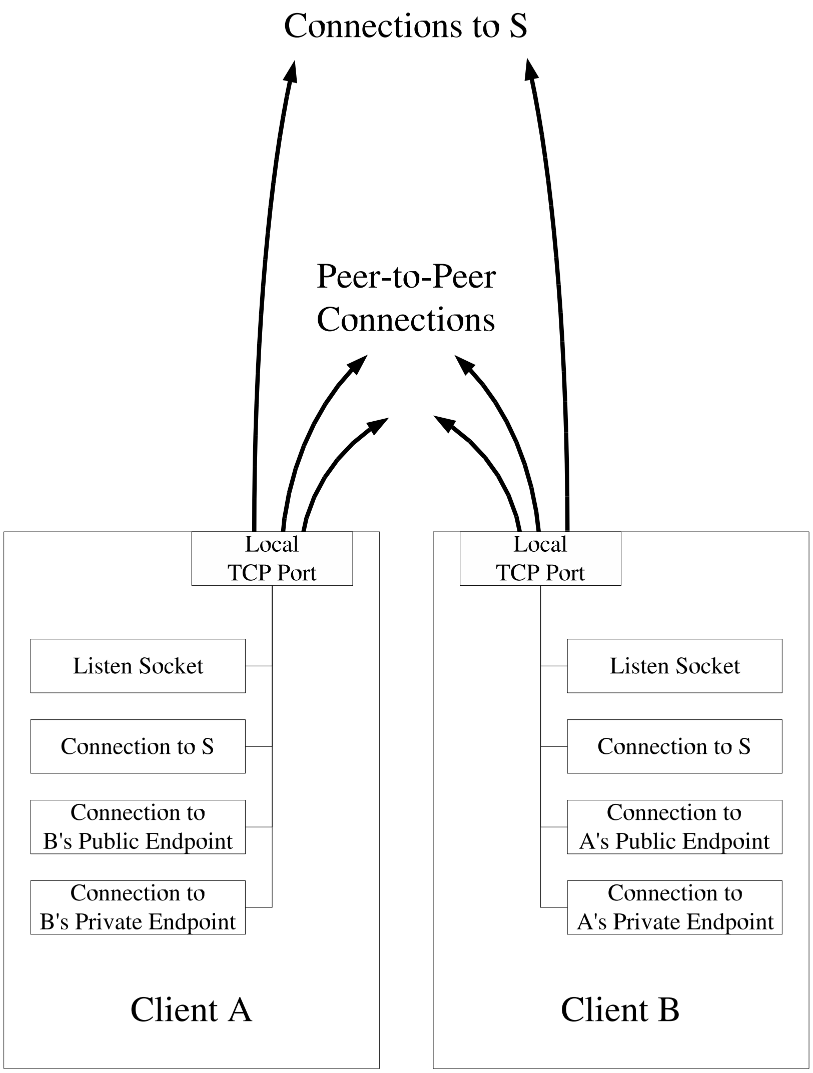


Figure 5-7: Sockets versus ports for TCP hole punching

Unlike with UDP, where each client only needs one socket to communicate with both S and any number of peers simultaneously, with TCP each client application must manage several sockets bound to a single local TCP port on that client node, as shown in Figure 5-7. Each client needs a stream socket representing its connection to S, a listen socket on which to accept incoming connections from peers, and at least two additional stream sockets with which to initiate outgoing connections to the other peer's public and private TCP endpoints.

Consider the common-case scenario in which the clients A and B are behind different NATs, as shown in Figure 5-5, and assume that the port numbers shown in the figure are now for TCP rather than UDP ports. The outgoing connection attempts A and B make to each other's private endpoints either fail or connect to the wrong host. As with UDP, it is important that TCP applications authenticate their peer-to-peer sessions, due of the likelihood of mistakenly connecting to a random host on the local network that happens to have the same private IP address as the desired host on a remote private network.

The clients' outgoing connection attempts to each other's *public* endpoints, however, cause the respective NATs to open up new "holes" enabling direct TCP communication between A and B. If the NATs are well-behaved, then a new peer-to-peer TCP stream automatically forms between them. If A's first SYN packet to B reaches B's NAT before B's first SYN packet to A reaches B's NAT, for example, then B's NAT may interpret A's SYN as an unsolicited incoming connection attempt and drop it. B's first SYN packet to A should subsequently get through, however, because A's NAT sees this SYN as being part of the outbound session to B that A's first SYN had already initiated.

5.3.3 Behavior Observed by the Application

What the client applications observe to happen with their sockets during TCP hole punching depends on the timing and the TCP implementations involved. Suppose that A's first outbound SYN packet to B's public endpoint is dropped by NAT B, but B's first subsequent SYN packet to A's public endpoint gets through to A before A's TCP retransmits its SYN. Depending on the operating system involved, one of two things may happen:

- A's TCP implementation notices that the session endpoints for the incoming SYN match those of an outbound session A was attempting to initiate. A's TCP stack therefore associates this new session with the socket that the local application on A was using to `connect()` to B's public endpoint. The application's asynchronous `connect()` call succeeds, and nothing happens with the application's listen socket.

  Since the received SYN packet did not include an ACK for A's previous outbound SYN, A's TCP replies to B's public endpoint with a SYN-ACK packet, the SYN part being merely a replay of A's original outbound SYN, using the same sequence number. Once B's TCP receives A's SYN-ACK, it responds with its own ACK for A's SYN, and the TCP session enters the connected state on both ends.

- Alternatively, A's TCP implementation might instead notice that A has an active listen socket on that port waiting for incoming connection attempts. Since B's SYN looks like an incoming connection attempt, A's TCP creates a *new* stream socket with which to associate the new TCP session, and hands this new socket to the application via the application's next `accept()` call on its listen socket. A's TCP then responds to B with a SYN-ACK as above, and TCP connection setup proceeds as usual for client/server-style connections.

  Since A's prior outbound `connect()` attempt to B used a combination of source and destination endpoints that is now in use by another socket, namely the one just returned to the application via `accept()`, A's asynchronous `connect()` attempt must fail at some point, typically with an "address in use" error. The application nevertheless has the working peer-to-peer stream socket it needs to communicate with B, so it ignores this failure.

The first behavior above appears to be usual for BSD-based operating systems, whereas the second behavior appears more common under Linux and Windows.

5.3.4 Simultaneous TCP Open

Suppose that the timing of the various connection attempts during the hole punching process works out so that the initial outgoing SYN packets from *both* clients traverse their respective local NATs, opening new outbound TCP sessions in each NAT, before reaching the remote NAT. In this "lucky" case, the NATs do not reject either of the initial SYN packets, and the SYNs cross on the wire between the two NATs. In this case, the clients observe an event known as a *simultaneous TCP open*: each peer's TCP receives a "raw" SYN while waiting for a SYN-ACK. Each peer's TCP responds with a SYN-ACK, whose SYN part essentially "replays" the peer's previous outgoing SYN, and whose ACK part acknowledges the SYN received from the other peer.

What the respective applications observe in this case again depends on the behavior of the TCP implementations involved, as described in the previous section. If *both* clients implement the second behavior above, it may be that *all* of the asynchronous `connect()` calls made by the application ultimately fail, but the applications running on the clients nevertheless each receive a new, working peer-to-peer TCP stream socket via `accept()`—as if this TCP stream had magically "created itself" on the wire and was merely passively accepted by the two endpoints! As long as the application does not care whether it ultimately receives its peer-to-peer TCP sockets via `connect()` or `accept()`, the process results in a working stream on any TCP implementation that properly implements the standard TCP state machine specified in RFC 793 [242].

Each of the alternative network organization scenarios discussed in Section 5.2 for UDP works in exactly the same way for TCP. For example, TCP hole punching works in multi-level NAT scenarios such as the one in Figure 5-6 as long as the NATs involved are well-behaved.

5.3.5 Sequential Hole Punching

In a variant of the above TCP hole punching procedure implemented by the NatTrav library [72], the clients attempt connections to each other sequentially rather than in parallel. For example: (1) A informs B via S of its desire to communicate, *without* simultaneously listening on its local port; (2) B makes a `connect()` attempt to A, which opens a hole in B's NAT but then fails due to a timeout or RST from A's NAT or a RST from A itself; (3) B closes its connection to S and does a `listen()` on its local port; (4) S in turn closes its connection with A, signaling A to attempt a `connect()` directly to B.

This sequential procedure may be particularly useful on Windows hosts prior to XP Service Pack 2, which did not correctly implement simultaneous TCP open, or on sockets APIs that do not support the `SO_REUSEADDR` functionality. The sequential procedure is more timing-dependent, however, and may be slower in the common case and less robust in unusual situations. In step (2), for example, B must allow its "doomed-to-fail" `connect()` attempt enough time to ensure that at least one SYN packet traverses all NATs on its side of the network. Too little delay risks a lost SYN derailing the process, whereas too much delay increases the total time required for hole punching. The sequential hole punching procedure also effectively "consumes" both clients' connections to the server S, requiring the clients to open fresh connections to S for each new P2P connection to be forged. The parallel hole punching procedure, in contrast, typically completes as soon as both clients make their outgoing `connect()` attempts, and allows each client to retain and re-use a single connection to S indefinitely.

5.4 Properties of P2P-Friendly NATs

This section describes the key behavioral properties NATs must have in order for the hole punching techniques described above to work properly. Not all current NAT implementations satisfy these properties, but many do, and NATs are gradually becoming more "P2P-friendly" as NAT vendors recognize the demand for peer-to-peer protocols such as voice over IP and on-line gaming.

This section is not meant to be a complete or definitive specification for how NATs "should" behave; we provide it merely for information about the most commonly observed behaviors that enable or break P2P hole punching. The IETF started the BEHAVE working group to define official "best current practices" for NAT behavior [118]. As of this writing, the BEHAVE group has standardized NAT behavior recommendations for UDP [16] and IP multicast [261], and is finalizing similar recommendations for TCP [101], which include the considerations outlined in this section and others. NAT vendors should of course follow the IETF working group directly as official behavioral standards are formulated.

5.4.1 Consistent Endpoint Translation

The hole punching techniques described here only work automatically if the NAT consistently maps a given TCP or UDP source endpoint on the private network to a *single* corresponding public endpoint controlled by the NAT. A NAT that behaves in this way is referred to as a *cone NAT* in RFC 3489 [207] and elsewhere, because the NAT "focuses" all sessions originating from a single private endpoint through the same public endpoint on the NAT.

Consider again the scenario in Figure 5-5, for example. When client A initially contacted the well-known server S, NAT A chose to use port 62000 at its own public IP address, 155.99.25.11, as a temporary public endpoint to representing A's private endpoint 10.0.0.1:4321. When A later attempts to establish a peer-to-peer session with B by sending a message from the same local private endpoint to B's public endpoint, A depends on NAT A preserving the identity of this private endpoint, and re-using the existing public endpoint of 155.99.25.11:62000, because that is the public endpoint for A to which B will be sending its corresponding messages.

A NAT that is only designed to support client/server protocols will not necessarily preserve the identities of private endpoints in this way. Such a NAT is a *symmetric NAT* in RFC 3489 terminology. For example, after the NAT assigns the public endpoint 155.99.25.11:62000 to client A's session with server S, the NAT might assign a different public endpoint, such as 155.99.25.11:62001, to the P2P session that A tries to initiate with B. In this case, the hole punching process fails to provide connectivity, because the subsequent incoming messages from B reach NAT A at the wrong port number.

Many symmetric NATs allocate port numbers for successive sessions in a fairly predictable way. Exploiting this fact, variants of hole punching algorithms [30, 100] can be made to work "much of the time" even over symmetric NATs by first probing the NAT's behavior using a protocol such as STUN [207], and using the resulting information to "predict" the public port number the NAT will assign to a new session. Such prediction techniques amount to chasing a moving target, however, and many things can go wrong along the way. The predicted port number might already be in use causing the NAT to jump to another port number, for example, or another client behind the same NAT might initiate an unrelated session at the wrong time so as to allocate the predicted port number. While port number prediction can be a useful trick for achieving maximum compatibility with badly-behaved existing NATs, it does not represent a robust long-term solution. Since symmetric NAT provides no greater security than a cone NAT with per-session traffic filtering, symmetric NAT is becoming less common as NAT vendors adapt their algorithms to support P2P protocols.

5.4.2 Handling Unsolicited TCP Connections

When a NAT receives a SYN packet on its public side for what appears to be an unsolicited incoming connection attempt, it is important that the NAT just silently drop the SYN packet. Some NATs instead actively reject such incoming connections by sending back a TCP RST packet or even an ICMP error report, which interferes with the TCP hole punching process. Such behavior is not necessarily fatal, as long as the applications re-try outgoing connection attempts as specified in step 4 of the process described in Section 5.3.2, but the resulting transient errors can make hole punching take longer.

5.4.3 Leaving Payloads Alone

A few existing NATs are known to scan "blindly" through packet payloads for 4-byte values that look like IP addresses, and translate them as they would the IP address in the packet header, without knowing anything about the application protocol in use. This bad behavior fortunately appears to be uncommon, and applications can easily protect themselves against it by obfuscating IP addresses they send in messages, for example by sending the bitwise complement of the desired IP address.

5.4.4 Hairpin Translation

Some multi-level NAT situations require hairpin translation support in order for either TCP or UDP hole punching to work, as described in Section 5.2.5. The scenario shown in Figure 5-6, for example, depends on NAT C providing hairpin translation. Support for hairpin translation is unfortunately rare in current NATs, but fortunately so are the network scenarios that require it. Multi-level

NAT is becoming more common as IPv4 address space depletion continues, however, so support for hairpin translation is important in future NAT implementations.

Chapter 6

UIA Implementation

This chapter describes UIA's current implementation status. All of the UIA components described in prior chapters have been prototyped, although these prototypes are not yet feature-complete or fully integrated into a single easily installable and usable system. UIA represents an ambitious new networking architecture, and fully implementing and refining it into a robust system ready for production use will take substantial additional work. Section 6.1 describes the current implementation of the personal name system described in Chapter 2, Section 6.2 describes the prototypes of the three approaches to UIA routing explored in Chapter 3, and Section 6.3 describes the Structured Stream Transport (SST) prototype.

6.1 Personal Naming Implementation

A UIA prototype implementing the personal naming (Chapter 2) and social routing (Section 3.2) protocols currently runs on Linux and Mac OS X, and is partially ported to FreeBSD. As illustrated in Figure 6-1, the prototype consists of two user-level daemons implementing UIA's naming and routing layers, respectively, and a graphical application for browsing and controlling devices and groups. The control application and other UIA-aware applications on the device interface directly to the naming and routing daemons via Sun RPC. Through these interfaces, UIA-aware applications can resolve UIA names to EIDs, explore and modify groups on behalf of the user, send packets to EIDs, receive packets on the device's EID, and discover peers on the local-area network.

6.1.1 Prototype Status

The UIA prototype's name daemon is written in Python and implements most of the design described in Section 2.3, providing group creation, merging, named links between groups, naming state gossip, state evaluation, multi-component name resolution, and ownership revocation. The prototype name daemon does not yet implement certain aspects of the name system design, however, such as multiple personal EIDs per device (Section 2.3.1), undo of merge operations (Section 2.3.3), copying orphaned change records across revocations (Section 2.3.3), and the caching

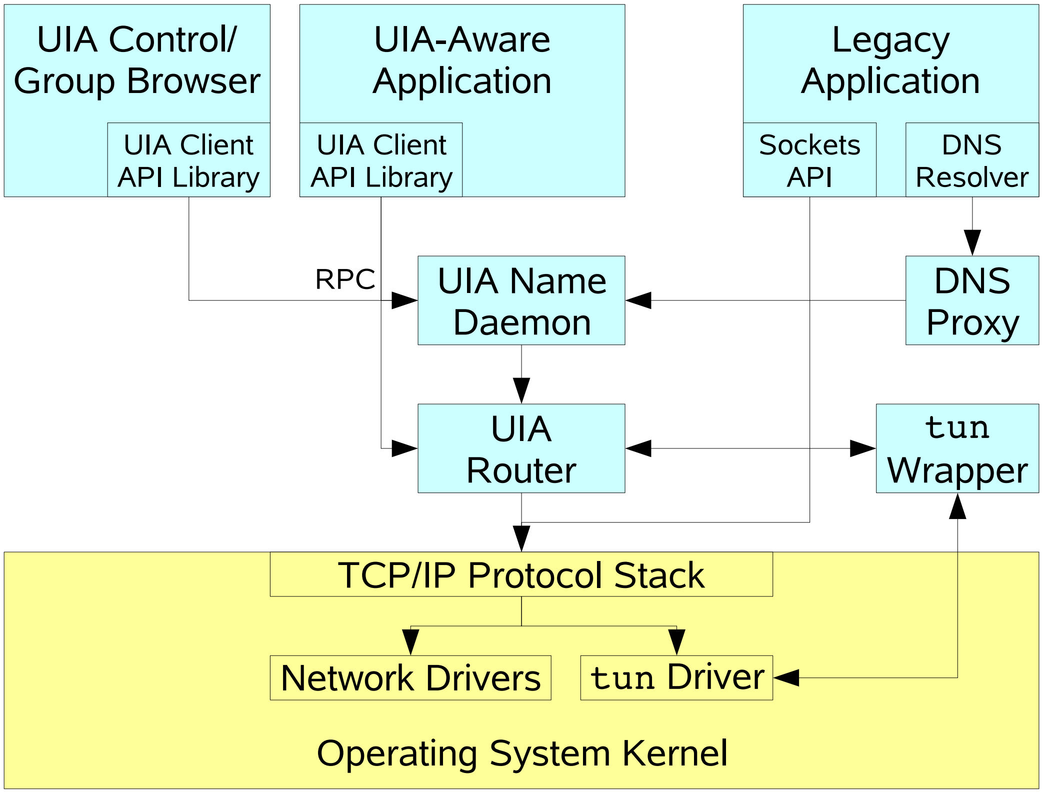


Figure 6-1: Structure of UIA prototype implementation

remote name resolution protocol (Section 2.3.6). The prototype routing daemon will be described later in Section 6.2.

The UIA control application allows the user to browse the UIA namespace and create and modify groups, as illustrated earlier in Figure 2-3, and supervises the device introduction process as illustrated in Figure 2-1. The control application is still unpolished and does not yet fully support shared groups or revocation, however.

The source code for the UIA prototype is available on the Web at:

```
http://pdos.csail.mit.edu/uia/
```

6.1.2 Support for Smaller Devices

The UIA prototype has been ported to the Nokia 770 Internet Tablet, a Linux-based Internet appliance with an ARM processor. The naming and routing layers have the full functionality of the regular Linux/Mac version of UIA, but the port of the GUI control application is not yet complete. In general, the routing and naming modules should port easily among smaller devices, while the GUI component requires more modifications because of the more specialized and restrictive user interface frameworks available on each class of mobile device. UIA does not rely on extensive data entry or other forms of user interaction that are fundamentally difficult to achieve on small devices, however.

6.1.3 Using UIA Names in Legacy Applications

The UIA prototype supports legacy applications through a `tun` wrapper and DNS proxy. The `tun` wrapper disguises EIDs as device-local IP addresses and uses the kernel's `tun` device to forward applications' TCP and UDP packets for these special IP addresses over UIA's routing layer. The DNS proxy similarly intercepts name lookups made by local applications and resolves UIA names to device-local IP addresses for the corresponding EIDs. We have run Apache, Firefox, OpenSSH, and Apple's Personal File Sharing over UIA using this legacy interface without modification or recompilation.

UIA's legacy application support layer makes the user's personal group appear to applications like a global virtual private network, by intercepting network-local broadcast packets that applications send to UIA's special IP addresses and forwarding them securely to each of the user's personal devices. Because of this feature, many broadcast-based "local-area" service discovery protocols such as Bonjour [11] automatically work across all the devices in the user's personal group, even when some of the devices are in fact remote. We have used Apple's Bonjour-based Personal File Sharing, for example, to locate and share files remotely between devices in a UIA personal group as if they were present on the same LAN.

6.2 Overlay Routing Implementation

Of the three approaches to UIA overlay routing explored in Chapter 3, only the first (Social Routing) is fully implemented and deployed in an immediately usable form; the other two currently have only minimal prototypes suitable for simulation and analysis purposes. We briefly examine each of these implementations in turn.

6.2.1 Social Routing

The social routing protocol described in Section 3.2 is implemented and in regular use in the UIA prototype. The social routing overlay protocol is implemented in C++ as a standalone UIA routing daemon, which interacts with other UIA components via RPC, as illustrated in Figure 6-1.

Local Discovery

The social routing daemon uses Apple's Bonjour [11] to discover and connect with other devices that happen to be located on the same local area network. UIA devices use this local-area discovery mechanism for two purposes. First, the UIA naming layer depends on local-area discovery to implement its secure local introduction process, described in Section 2.2.1. Second, once devices are already introduced at the UIA naming layer, the routing layer takes advantage of local-area discovery to find and re-connect with devices that may have moved or changed IP addresses since last contact, but are now located on the same local-area network. Since the local-area discovery mechanism works even if the devices in that LAN are not actually connected to the Internet, through this local-area discovery mechanism the routing protocol can successfully connect and route to (or *through*) devices in small "ad hoc" edge networks not connected to the Internet, such as the ad hoc network a small group of people might create at a business lunch or on a bus or plane.

Locating and Routing to Remote Devices

To connect with devices that are not directly reachable at their last known IP address or via the local-area discovery mechanism, the social routing daemon uses the flooding mechanism described earlier in attempt to find the desired EID within the social network formed by the UIA devices that *are* currently reachable. The prototype currently uses SSL [173] over TCP connections to implement cryptographic EID authentication and provide secure communication between UIA devices.

Routing Legacy Application Traffic over UIA

As mentioned earlier, the UIA prototype supports legacy applications through a `tun` wrapper and DNS proxy. The `tun` wrapper disguises EIDs as device-local IP addresses and uses the kernel's `tun` device to forward applications' TCP and UDP packets for these special IP addresses over UIA's routing layer. In this way, applications can bind a socket to the local device's EID or connect to a remote device by EID.

The current overlay routing layer represents EIDs as "fake" IP version 6 (IPv6) addresses [62], because the 128-bit size of IPv6 addresses makes it simple to generate these fake addresses uniquely (with high probability), simply but truncating the cryptographic EID and prepending a prefix denoting these addresses as *unique local addresses* [110]. No IPv6 infrastructure is required for the UIA routing daemon to work, however, and all messages forwarded over network links are tunneled through IPv4 sockets. In any case, the current prototype's use of IPv6 is purely for convenience; it could just as well represent EIDs as fake IPv4 addresses, providing full compatibility with applications that only support IPv4, at a small cost in the complexity of managing the space of fake IPv4 addresses within each host's address space.

UIA's legacy application support layer effectively makes the user's personal group appear to applications like a global virtual private network, by intercepting network-local broadcast packets that applications send to UIA's special IP addresses and forwarding them securely to each of the user's personal devices. Because of this feature, many broadcast-based "local-area" service discovery protocols such as Bonjour automatically work across all the devices in the user's personal group, even when some of the devices are in fact remote. We have used Apple's Bonjour-based Personal File Sharing, for example, to locate and share files remotely between devices in a UIA personal group as if they were present on the same LAN.

6.2.2 Identity Hash Routing

The Identity Hash Routing algorithm described in Section 3.2 is implemented for simulation and experimental evaluation purposes in about 2300 lines of C++, including a custom simulation testbed. The prototype implements most of the IHR algorithm as described including the merge procedure, and supports both source routing and virtual link forwarding. The prototype does not yet simulate a dynamically evolving network topology in general, but does support simulation of "net split" events of various magnitudes for purposes of testing the merge procedure.

6.2.3 Compact Routing

The TZk-based compact routing algorithm described in Section 3.4 is implemented for simulation and experimental evaluation purposes in two versions: a mostly-Python version consisting of about 2000 lines of Python [189] and 630 lines of C to optimize performance-critical calculations, and a 4500-line all-C++ version. The two implementations are mostly equivalent. The prototypes implement randomized landmark selection and distributed routing tree construction based on simulated messages, but they do not yet implement address assignment for interval routing or attempt to simulate dynamically changing networks.

6.3 Structured Stream Transport Implementation

A prototype implementation of the Structured Stream Transport (SST) protocol described in Chapter 4 is implemented and usable in both simulation and real-world network environments, although not yet complete or integrated with the main UIA prototype. The SST prototype takes the form of a user-space library written in C++, which runs on Linux, BSD, Mac OS X, and Windows. The library builds on the Qt portability framework [248], relying on the Core and Network components of Qt (but not the large graphical user interface component).

The prototype implements the SST protocol atop UDP, so its use requires no special privileges or OS extensions, and the library can be statically linked into or distributed with applications to minimize deployment burden on users. Although running transport protocols atop UDP is not the traditionally accepted and sanctioned approach, this approach has a number of advantages in addition to application-level deployment, such as compatibility with existing NATs and firewalls [86]. The prototype implements most of the SST protocol design, including classic TCP congestion control [6], but a few features such as flow control and MTU discovery are still incomplete.

The prototype also allows the application to assign priority levels to streams, for explicit control of data transmission within the scope of a congestion controlled channel. For controlled testing and simulation, the library allows client applications to run multiple instances of SST simultaneously in one process, and to virtualize SST's use of the host's timing and networking facilities. The prototype currently totals about 13,000 source lines, or 4,400 semicolons, and is available on the Web at:

`http://pdos.csail.mit.edu/uia/sst/`

Chapter 7

Evaluation

This chapter evaluates the various components and algorithms comprising UIA in terms of real-world experience using the currently-deployed parts of UIA, experimental tests, and simulation studies.

7.1 Experience Using UIA Personal Names

The primary goal of UIA's naming system is convenience and usability by non-technical users, a goal that can only be evaluated effectively once UIA has been deployed longer and more widely in the field. We currently run the UIA prototype on a number of desktop and laptop machines in our lab, however, and regularly run existing applications such as SSH over UIA to reach our mobile devices via their short personal device names, and our personal experience so far confirms the convenience and simplicity of the basic UIA personal naming model.

Many practical challenges remain before UIA personal naming can be widely deployed, however. Our decision to implement UIA initially using user-space daemons facilitated UIA's portability to multiple operating systems, but its level of integration with any one system is limited: a production-quality implementation of UIA naming will probably have to be somewhat more OS-specific in order to integrate cleanly and smoothly. For example, most mature operating systems provide a clean, OS-sanctioned mechanism to hook into the system's name resolution mechanism, whose use would eliminate our prototype's need to redirect all DNS lookups to a local DNS proxy and avoid the associated potential confusion if the user manually edits the system's DNS resolver configuration. The OS-sanctioned mechanisms to hook into name resolution tend to change completely from one OS to another, however, so we avoided using them for portability reasons.

7.1.1 Interaction with Global Names and Legacy Applications

One minor pragmatic difficulty we encountered in using UIA personal names in legacy applications is that certain applications, particularly web browsers like Firefox, assume that any valid DNS host name *must* contain at least one period ('`.`'); otherwise the application interprets the name as some other word such as a search term to be submitted to a Web search engine. This "at least one dot" assumption is valid in the traditional DNS world in which there are only a few top-level domains (TLDs), none of which represent host names, but becomes invalid in the context of UIA, where single-component host names are commonplace and represent hosts within the user's own personal group. Fortunately, until such applications can be modified to remove this assumption or be made specifically aware of UIA host names, there is an easy workaround: simply append a period to the end of the name when typing it: e.g., '`homepc.`' instead of just '`homepc`'. The recent decision of ICANN to open up the allocation of new top-level domains [45, 117] may eventually force the reevaluation of this traditional assumption even in the case of global DNS names.

One danger inherent in UIA's current behavior of resolving personal names as if they were global names is the potential confusion that might be caused if the user creates a personal name that conflicts with a DNS top-level domain. If the user names one of her personal devices '`com`', for example, she will no longer be able to refer to any global DNS name ending in '`.com`'. This danger could be ameliorated by detecting and warning the user if she attempts to create a personal name that conflicts with a known TLD, and could be avoided entirely at some cost in the conciseness and convenience of personal names by placing all personal names under some particular "virtual" TLD such as '`.pers`'. This latter approach would also conveniently avoid the problem mentioned above of applications that assume every DNS name contains at least one dot.

7.2 Social Routing Evaluation

This section uses real-world experience and simulations to evaluate the Social Routing protocol implemented in the currently deployed UIA prototype, in order to verify that the proposed design is capable of providing the desired connectivity on realistic networks.

7.2.1 Experience with Social Routing

We currently run the UIA prototype on a number of desktop and laptop machines in our lab, and regularly run existing applications such as SSH over UIA to reach our mobile devices via their short personal device names. The overlay routing protocol automatically accounts for IP address changes and traverses NATs as necessary; SSH connections open when we take a laptop home need not be restarted. Although these uses are already possible via alternate protocols such as Mobile IP, the complexity of configuring these alternatives has generally deterred even those of us with the necessary technical knowledge from deploying them. We feel that UIA's zero-configuration personal naming and connectivity provides a crucial missing element in making mobile devices usable.

One of the key requirements for the social routing protocol to work reliably is for every user to have at least one stable, publicly reachable node somewhere in his own or one of his friends' personal groups. While we found this requirement easy to satisfy since most machines at MIT and many other universities have their own public IP address, this requirement could prove more of a problem for users whose only "stable" desktop machines are located behind home and/or corporate firewalls and NATs. Although we implemented the *default peer* mechanism described in Section 3.2.1 to address this difficulty, UIA has not yet seen sufficient deployment among such users to evaluate this issue thoroughly.

7.2.2 Experimental Setup

We use as our simulated network a crawl of the social networking site Orkut gathered by Li and Dabek [145]. This graph is merely suggestive; until UIA is more widely deployed, it will not be clear how accurately the Orkut graph characterizes UIA's likely usage model. The graph has 2,363 users, which we take to represent devices, as if each user owned one device. Friend relationships are bidirectional, and the number of friends per user is highly skewed: the median is only 7, but the most popular user has over 1,000.

Our simulator takes as input a *percent stable* parameter and randomly chooses that percent of the devices to be *stable* and publicly accessible. All other devices the simulator considers to be *mobile* and inaccessible except through some stable device, as if behind a NAT. We assume that all devices agree as to which devices are stable.

Each device chooses 16 peers and allows at most 64 devices to choose it, to limit each device's overlay maintenance cost in a real network. Devices choose peers in order of friendship distance. A device can only choose a given target as a peer if the target does not already have 64 peers, or if the new device is closer than one of the target's existing peers, in which case the target discards a random peer at higher friendship distance. Since we do not yet have traces with which to simulate the network's evolution, the simulated devices choose peers in random order, iterating until the network reaches a fixed point.

The simulator then performs token-limited location requests on the resulting overlay between 10,000 random pairs at friendship distances 1, 2, and 3. Each lookup starts with 16 tokens, and doubles after each failure, up to a maximum of 256 tokens. The simulator records the percentage of requests that succeed after a given number of rounds and the total number of messages sent.

7.2.3 Location Success Rate

An important performance metric for the location algorithm is the number of tokens needed to locate a target device successfully. Using more tokens increases the chance of success (assuming that the overlay is in fact connected), but also increases the cost of concluding that an unreachable device is offline. Figure 7-1 shows the success rate measured in the simulation for locating devices at friendship distance 1. Using 256 tokens, the location algorithm achieves greater than a 99.5% success rate for 10% or more stable devices. Using 64 tokens the algorithm achieves 97.5% success for 10% or more stable devices. The vast majority of requests—80% of requests at 10% or more stable devices—succeed within the first inexpensive round of 16 tokens.

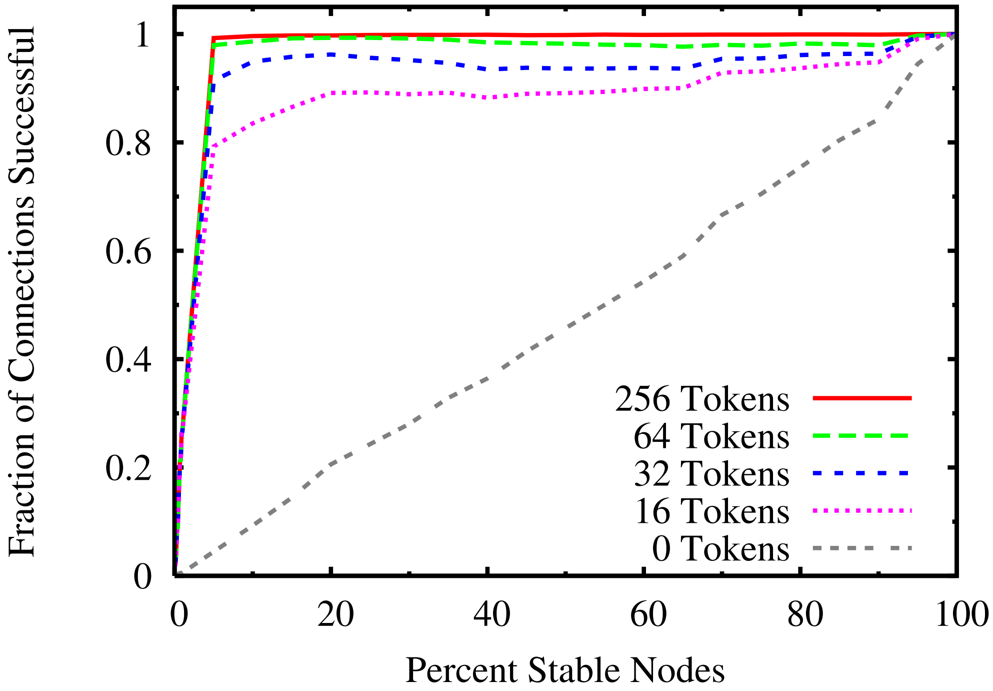


Figure 7-1: Location request success rate

At the far left of the graph where few stable devices are available, the success rate drops off because each stable device can only support 64 peers, and there are not enough stable devices for each mobile device to choose a full 16 peers, or in some cases any. As the percentage of stable devices increases, a linearly increasing number of location requests are to stable devices that can be contacted directly without flooding, thus requiring no tokens.

We also measured the success rates for locating devices at friendship distances of 2 and 3, though we omit the corresponding graphs. The results for distance 2 are almost as good as for distance 1, presumably because two devices at friendship distance 2 are likely to peer with some common stable device at distance 1 from each of them. Success rate drops considerably at distance 3, however, achieving only 50% success with 256 tokens in networks of 40% or more stable devices, for example.

7.2.4 Messages Sent

The lower line in Figure 7-2 shows the total number of messages sent during successful token-limited lookup requests for devices at friendship distance 1. A request's message count is ultimately bounded by its token count, but is often much less because successful lookups usually do not require all available tokens.

At the left edge of the graph, there are not enough stable devices for every mobile device to have a peer, so few requests succeed. Those that do succeed do so cheaply, however, because all of the connected mobile devices have clustered around the same few stable devices. The message

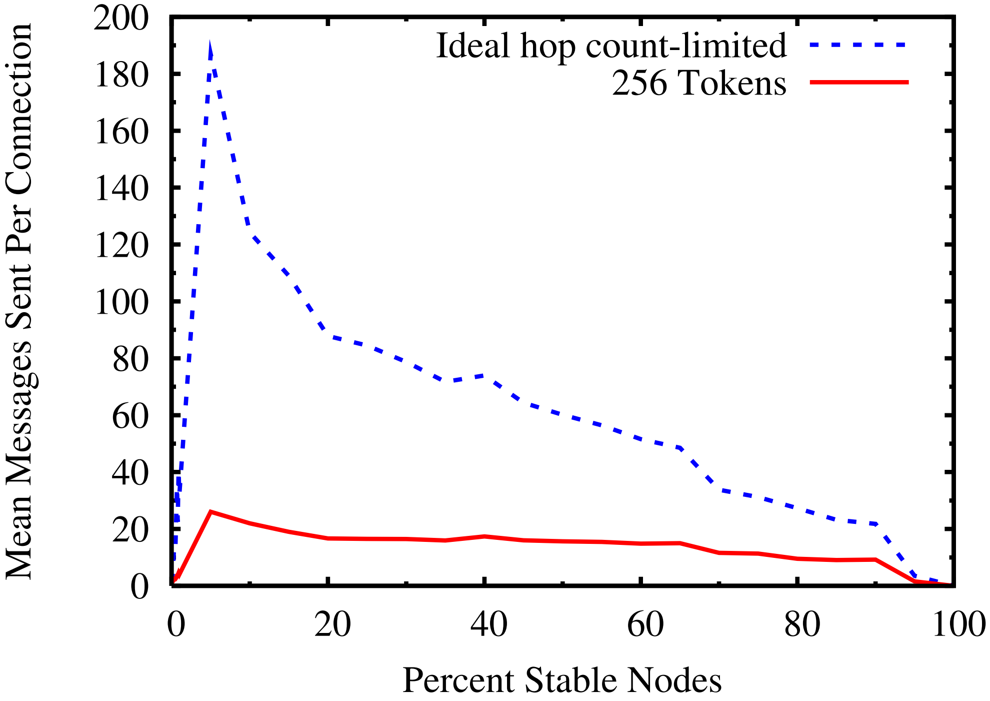


Figure 7-2: Mean messages sent per location request

count peaks near the point where the number of stable devices becomes barely sufficient to serve all of the mobile devices, so the requests usually succeed but only after contacting many devices. As the number of stable devices increases further, more requests complete without flooding at all, since stable targets are reachable directly at their last known IP addresses.

To contrast UIA's token-limited scheme with flooding limited by hop count [42], the upper line in the figure shows the total number of messages sent for successful hop count-controlled location requests in which the originating device knows via an oracle the exact hop count required for the search to succeed in one round. As the graph shows, the token-based scheme requires far fewer messages than even this "ideal" hop count-limited scheme. The inefficiency of the hop count scheme results from the skewed popularity distribution and redundancy of the friendship graph, as discussed in Section 3.2.2.

7.3 Identity Hash Routing Evaluation

Although Identity Hash Routing (IHR) is not yet fully implemented as a usable distributed protocol or integrated into the UIA prototype, in this section we use simulation results to evaluate the potential utility of IHR's routing and forwarding algorithms for a future version of UIA. Until a full implementation has been deployed and a substantial critical mass of users has developed, simulations provide the only realistic option for tuning the protocol and predicting how it will behave on the large networks it is intended to support.

7.3.1 Performance Metrics

In order to asses the basic viability of the IHR routing protocol, we focus here on measuring the efficiency of the network paths the protocol finds through random network topologies. Many other important factors that will affect the performance of real-world IHR networks remain for future study. In particular, while simulations confirm that the protocol recovers from random node failures and network partitions, we do not yet have a full characterization of the dynamic behavior of a IHR network under continuous change.

In order to measure the efficiency of routing paths chosen by IHR nodes, we define the *IHR path length* between two nodes n_1 and n_2 to be the total number of physical hops in the path that n_1 constructs to n_2 using the **build_path** procedure in Figure 3-4. We define the *stretch* between n_1 and n_2 to be the ratio of the IHR path length to the length of the best possible path through the underlying topology.

We measure the stretch for a given pair of nodes by using **build_path** to construct a path from one node to the other, measuring the total number of physical hops in the path, and then eliminating all the virtual links that **build_path** constructed so that the measurement of one path does not affect the measurement of subsequent paths. On networks of 100 nodes or less we measure all possible paths between any two nodes; on larger networks we take a sample of 10,000 randomly chosen node pairs.

7.3.2 Test Network Topology

Selecting appropriate network topologies for simulations of IHR is difficult, because we have no way to predict the topologies of the networks on which a protocol like IHR will actually be deployed. Using topological maps of the existing IPv4 Internet would not make sense: the existing well-connected Internet is precisely the portion of today's global network infrastructure across which IHR will *not* have to find paths, because IP already does that well enough, and IHR simply treats these paths as direct physical links. For the function IHR is designed provide, finding paths between nodes on the Internet and nodes on the many private and ad hoc networks attached to it, no reliable topological data is available precisely because most of these adjoining networks are private.

Nevertheless, we can construct artificial topologies that approximate the most important characteristics we believe this global network infrastructure to have. First, we expect the topology on which IHR is deployed to consist of many clusters, in which each node in a given cluster can reliably address and connect with any other node in the same cluster, but nodes in one cluster have limited connectivity to nodes in other clusters. Second, because of the diversity of existing networking technologies and deployment scenarios, we expect the size of these clusters to follow a power law distribution, with larger clusters having better connectivity to neighboring clusters. Finally, we expect all of these clusters to be within at most a few hops from a single huge, central cluster, namely the public IP-based Internet.

To construct an artificial topology having these characteristics, we start with a single distinguished cluster we will call the *root cluster*, initially containing a single node. We then randomly "grow" the network one node at a time as follows. For each new node, we choose the number of attachment points the node will have based on a geometric random variable with a *multihoming probability* parameter p_m. Approximately $p_m N$ of the network's N nodes will have at least two attachment points, $p_m^2 N$ have at least three attachment points, and so on.

We choose each attachment point for a new node via a random walk from the root cluster using a *downstream probability* parameter p_d and a *new cluster probability* parameter p_n. At each step, with probability p_d we move the attachment point downstream, and with probability $1 - p_d$ we terminate the process. To move the attachment point downstream, we choose a node at random from the current cluster, then we either create a new cluster "private" to that node with probability p_n, or else we with probability $1 - p_n$ we pick at random any cluster that node is attached to (which could be the cluster we just came from). Once the random walk terminates, we add the new node to the cluster at which the walk ended.

We call the resulting random network topology a *rooted* topology, since it consists of many small clusters centered around the single large root cluster, approximating the well-connected IP-based Internet surrounded by many smaller private networks.

We choose somewhat arbitrarily the following "baseline" parameters for the experiments. We use network topologies of varying sizes constructed with a multihoming probability $p_m = 1/10$, a downstream probability $p_d = 3/4$, and new link probability $p_n = 1/2$. On these topologies we build IHR networks with a redundancy factor $k = 3$, by adding nodes to the network one at a time in random order. We will vary these parameters to explore their impact on the efficiency of the routing protocol.

7.3.3 Source Routing versus Recursive Tunneling

Figure 7-3 shows the average and maximum path stretch observed (vertical axis) between any two nodes on networks of a given size (horizontal axis), for both source routing and recursive tunneling. The error bars indicate standard deviation of the measured stretch. In the random 10,000-node rooted topology, the root cluster contains 3233 nodes (32% of the network), the average distance between any two nodes is 2.5 hops, and the maximum distance between any two nodes (total network diameter) is 8 hops.

With both source routing and recursive tunneling, we see that the IHR routing protocol consistently finds paths that are on average no more than twice as long as the best possible path. The average-case efficiency of recursive tunneling is slightly worse than for source routing, due to the more limited amount of information nodes have to optimize paths they find through the network. The routing protocol occasionally chooses bad paths—up to 6× stretch for source routing and up to 16× for recursive tunneling—but the low standard deviation indicates that these bad paths occur rarely.

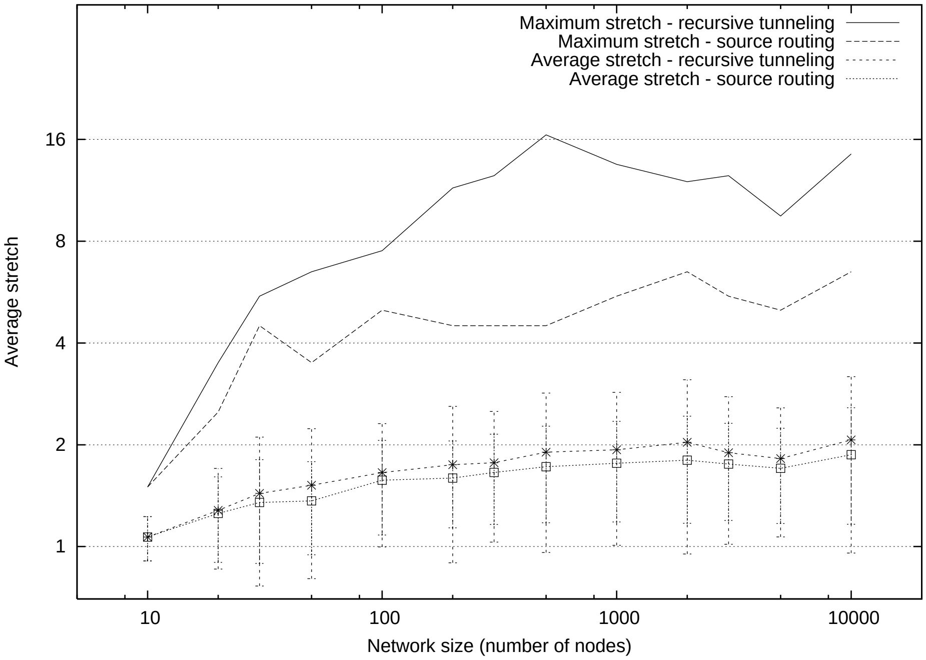


Figure 7-3: Network path stretch for source routing versus recursive tunneling

7.3.4 Rooted versus Unrooted Networks

We would next like to determine how much the IHR routing protocol benefits from the tree-like structure of rooted network topologies. Is the IHR routing protocol only viable when some underlying protocol such as IP is doing most of the work of routing within the large central cluster, or could IHR routing also be used to internetwork a number of small link-layer networks joined in ad-hoc fashion?

To explore this question, we modify the random network creation procedure of Section 7.3.2 so that the random walk to find each new attachment point for a given starts at a cluster chosen uniformly at random from all existing clusters, rather than at a well-known root cluster. The resulting *unrooted* topologies have a much more uniform and unpolarized distribution in their cluster sizes and in the connections between clusters. In the random 10,000-node unrooted topology, for example, the largest cluster contains only 11 nodes, the average distance between any two nodes is 7.7 hops, and the network diameter is 19 hops. We expect efficient routing on such a diffuse network to be more difficult than on a rooted network.

Figure 7-4 compares the path efficiency of IHR source route-based forwarding on rooted and unrooted networks of varying sizes. We find that unrooted networks indeed yield greater stretch, but not by a wide margin. This result suggests that IHR routing is not highly dependent on rooted topologies and may be useable as well on more diffuse topologies.

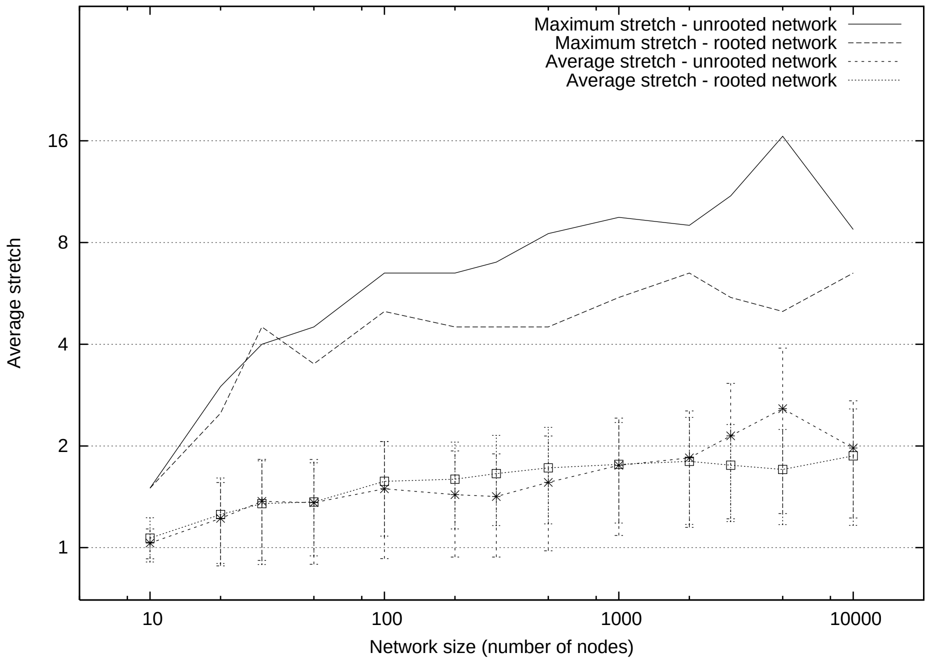


Figure 7-4: Network path stretch for rooted versus unrooted networks

7.3.5 Discussion and Future Work

The above development and evaluation of IHR suggests that the basic approach of adapting DHT lookup algorithms into overlay routing schemes may indeed be viable. Since IHR has so far only been analyzed under simulation and is not integrated into the deployed UIA prototype, however, it is likely that many practical issues will still need to be addressed in order to make it into a robust distributed protocol.

One potential issue with the scheme as described above is the danger of temporary routing loops developing between nodes due to dynamic changes in the network topology. For example, with forwarding based on recursive tunneling, if node A has a virtual link through B to C, then B's physical link to C fails, B might see A's advertised (virtual) link to C and try to use it to build a virtual link to C through B. These mutually recursive virtual links will eventually time out and be garbage collected, but the resulting loop could cause considerable network inefficiency and slow routing table convergence in the meantime. These issues are not different from the looping issues of classic Bellman-Ford routing, however, and the many solutions to this problem that have been explored in the context of traditional routing protocols are likely to be applicable to IHR as well.

A second, somewhat more serious challenge with IHR is the security of the distributed routing structure against the variety of disruption or denial-of-service attacks that malicious nodes might mount. These threats have already received considerable attention in the context of traditional DHT lookup schemes [40,67,220,230], and many of the proposed remedies may apply to IHR as well, but fully understanding the similarities and differences between DHT lookup and DHT-inspired overlay routing for security purposes will require further analysis. One approach that may be particularly useful in the context of UIA is to build on the social trust framework provided by UIA's naming layer, as the Sprout DHT does [152].

IHR lacks a formal bound on stretch—i.e., on the inefficiency of the paths it finds. Furthermore, the typical stretch measured in the above simulations may be reasonable in practice, when multi-hop UIA routing is only needed as a "backup solution" when IP routing provides no path at all, but is not what we might hope for. In the next section, therefore, we explore the use of routing schemes that can guarantee bounded stretch on any topology—though as we will see there are downsides to these schemes as well.

7.4 Compact Routing Evaluation

As with IHR, the TZk-based compact routing protocol has not yet been developed into a fully operational protocol ready for use on real-world networks, but this section uses simulations to analyze a variety of general properties of the protocol that are likely to be important to its practical utility. We first examine stretch, then congestion of network routes, and finally robustness against failures in the network. Although these are far from the only properties important for practical deployment, these first-step results appear promising and help confirm the protocol's scalability and understand the remaining challenges to be addressed.

7.4.1 Simulation Framework and Method

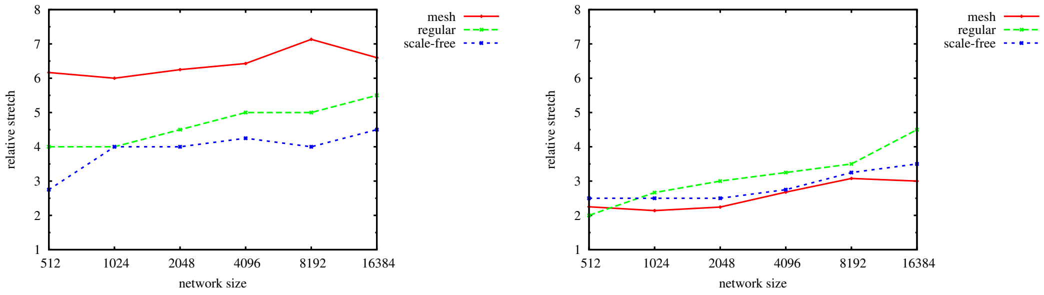
The simulated message-based compact routing protocol builds on a custom, lightweight simulation framework in C++, because we found existing simulation frameworks insufficiently scalable to handle the large graph sizes we wished to test (over 190,000 nodes in one case). The current simulation environment and routing protocol represents a second generation to an earlier Python-based simulator, which yielded comparable results on smaller graphs but did not scale sufficiently to handle the largest ones.

Due to the unavailability of realistic network topology datasets or generators that included useful edge weights—e.g., measured round-trip delays between network nodes—we did not try to model time realistically in our protocol. Instead, all simulated nodes perform protocol processing in lock step: they receive messages from neighbors, process them, and send new messages to their neighbors in one time step, and once sent, messages arrive at their destinations one time step later.

The simulated protocol selects landmarks randomly so as to reduce the expected number of landmarks by a factor of two at each level, yielding about $log_2\, n$ landmark levels for an n-node topology. Given this landmark selection, the protocol computes compact routing trees in the way described earlier in Section 3.4.4, where at each time step each node gossips all new routing table information with its neighbors. The routing tables of all nodes thus gradually converge toward shortest-path routes to their landmarks in similar fashion to classic Bellman-Ford routing protocols—with the key difference of course that each node does not end up with routing table pointers to *all* other nodes but merely a selection of nearby landmarks at each level. Our current protocol does not solve the

Name	Description
mesh	random 2D mesh on a Euclidean plane
scale-free	scale-free graph generated by preferential attachment
regular	random regular graph

Table 7.1: Synthetic topologies used for simulation



(a) Pure TZk

(b) With 25 extra table entries

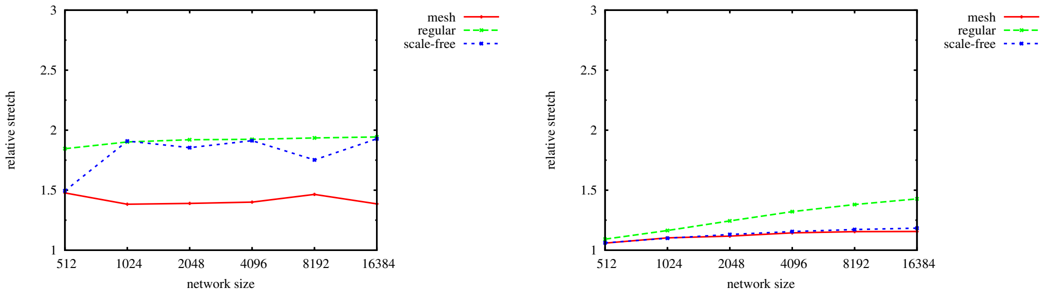
Figure 7-5: Maximum stretch relative to shortest-path routes

well-known "counting to infinity" problem of Bellman-Ford protocols, but many existing solutions to this problem such as destination sequencing [182] (or in this case "landmark sequencing") are applicable.

For comparison, we evaluate the compact routing protocol's performance on several different types of synthetic networks of varying size, as well as topologies from Skitter [38] and DIMES [244] derived from actual Internet measurements. The types of synthetic topologies used are shown in Table 7.1. The "mesh" generator places nodes randomly on a 2D plane and connects all nodes closer than a certain radius. This is an approximate model for wireless ad-hoc networks. The "scale-free" generator starts with a loosely-connected network, and connects nodes with probability proportional to their degree. The Internet's core exhibits some scale-free-like properties. The "regular" generator creates a random d-regular graph. It exercises the behavior of the protocol on highly disordered networks.

7.4.2 Stretch

After running the compact routing protocol to produce routing trees, the simulation chooses 10 random routing targets per node, measures the length of the paths chosen by the routing protocol, and compares it against the shortest path length computed via Dijkstra's single-source shortest path algorithm to compute path stretch. (It was not practical to evaluate all pairs on the larger networks because of the $O(n^2)$ time required.) Figure 7-5 shows the maximum measured stretch observed on all sampled paths, and Figure 7-6 shows average stretch. The figure presents measurements for two versions of the protocol in each case: the "Pure TZk" version conforms to the original TZk algorithm, and the "With 25 extra table entries" version permits each node to grow its routing table by as many as 25 entries beyond the minimum number required for pure TZk—a fairly arbitrary



(a) Pure TZk

(b) With 25 extra table entries

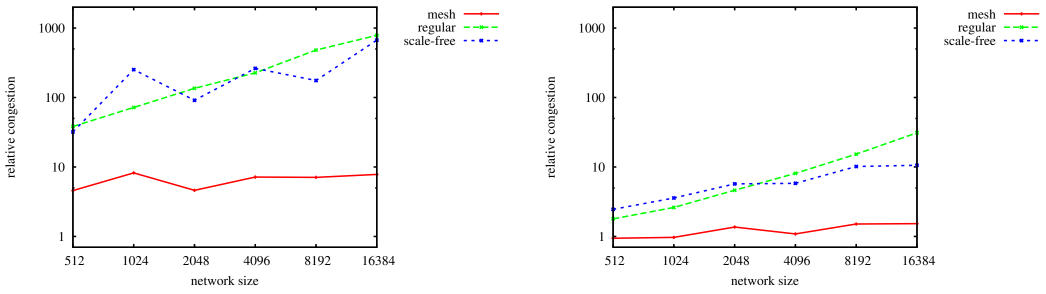
Figure 7-6: Average stretch relative to shortest-path routes

"slop factor" that we use to get an idea of the sensitivity of the routing algorithm to routing table size. These extra entries are distributed evenly over all the levels of the routing table, which gives the optimal benefit to the maximum stretch bound. The number of levels in these simulations is always $\log_2 n$, which gives small TZk routing tables and $O(\log n)$ max stretch.

Krioukov et al. [139] formally analyzed the average-case stretch of another compact routing algorithm presented by Thorup and Zwick in the same paper as TZk [245], which we call TZ2 because it uses a fixed two-level landmark hierarchy. They later compared the stretch of TZ2 and several other 2-level, stretch 3 algorithms via simulations [140]. Although TZ2 selects landmarks and constructs routing tables somewhat differently from TZk, the resulting behavior is conceptually similar for purposes of computing stretch: namely, a message from a node u to a node v follows a shortest path directly from u to v if the two nodes are close together, and otherwise takes a shortest path from u toward v's closest landmark l followed by a shortest path from l toward v—which is equivalent to routing on a shortest-path tree rooted at l. Therefore, although no formal average-case stretch analysis of TZk has been done yet, there is reason to believe that a similar analysis might yield similar results. Krioukov's specific stretch results are not directly comparable to ours despite this conceptual similarity, however, because the TZ2 algorithm they evaluated is a fixed 2-level scheme with $O(\sqrt{n})$ size routing tables, whereas the simulations done here use a variable number of landmark levels (approximately $log_2\ n$) with $O(log\ n)$ size routing tables.

7.4.3 Route Congestion

Because TZk routes all "long-distance" traffic over a few large trees rooted at the top-level landmarks, a high percentage of this long-distance traffic passes through or near these top-level landmarks, causing high path congestion in the links around them. To evaluate this congestion bottleneck, the simulation computes the routes chosen by the compact routing protocol between all random pairs, and counts the number of routes that pass through the most congested edge in the network. Figure 7-7 shows the maximum congestion measured for the routing protocol divided by the congestion measured for shortest-path distance-vector routing, for both the original TZk algorithm and the protocol with extra routing table entries.



(a) Pure TZk

(b) With 25 extra table entries

Figure 7-7: Network congestion relative to shortest-path routes

	# nodes	Stretch	Rel. Cong.
Skitter (TZ)	7,536	1.67	45.8
. . . (25 entries)	7,536	1.32	3.42
Skitter-RTR (TZ)	192,245	1.53	26.9
. . . (25 entries)	192,245	1.33	4.87
DIMES (TZ)	16,847	2.08	29.2
. . . (25 entries)	16,847	1.19	0.79

Table 7.2: Skitter and DIMES simulations, with and without optimizations.

We observe that extra table entries improve congestion considerably, although congestion around high-level landmarks still unavoidably occurs. Some level of congestion is probably acceptable in a deployment of compact routing on real networks, because real networks are typically designed to aggregate many low-capacity edge links and “bundle” them over a high-capacity core [147]. As long as the high-level landmarks used by the routing protocol are deliberately chosen to be nodes within that high-capacity core, this congestion around high-level landmarks may in effect turn out to be exactly what the network operator is seeking. If congestion does turn out to be a problem, the protocol provide a means for operators to trade off congestion for routing table size, the same as for stretch.

Measurements on AS adjacency graphs from skitter [38] and DIMES [244] are shown in Table 7.2, along with one skitter router-adjacency graph. These datasets have a small number of highly connected nodes, which tends to result in high congestion near the hubs. The pure TZk approach has high congestion relative to shortest-paths; the modified protocol does considerably better with only 25 extra table entries.

7.4.4 Reliability

Since recomputation of all compact routing trees in response to any topology change may be expensive [140], we would like to avoid this cost when possible and instead take advantage of the natural redundancy provided by the protocol’s many separate routing trees in order to “route around” temporary failures. In this way, no recomputation of routing trees is necessary as a result of such

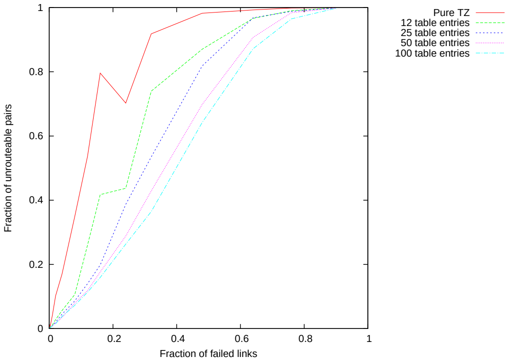


Figure 7-8: Resiliency to network failures

temporary failures; instead any end hosts trying to communicate through a broken link simply try a different routing tree instead.

To determine the effectiveness of the protocol at providing robustness in this way, we examine the failure rate of routing attempts when some percentage of network edges are randomly disabled, breaking any tree routing paths using those edges. For this test the protocol is configured for full redundancy ($r = \infty$): i.e., each node's global address contains all the nodes in its routing table. Figure 7-8 shows the routing failure rate over the random edge failure rate on the Skitter topology, with several settings for the number of extra landmarks each node collects at each level. From this graph we observe that the protocol provide substantial resilience at least against small numbers of randomly-distributed failures, especially when a considerable set of landmarks are available in which to seek a usable routing tree.

Of course, failures on real networks are often not randomly distributed; evaluating our protocol under more sophisticated failure models remains for future work.

7.5 Structured Stream Transport Evaluation

This section reports on preliminary experience implementing and using SST in real and simulated environments. We examine how SST scales across transaction sizes in comparison with TCP and UDP, how Web-style transactions on SST compare with non-persistent, persistent, and pipelined HTTP over TCP, and how applications can dynamically prioritize SST streams to improve interactive responsiveness.

7.5.1 Experience with Applications

The SST prototype is in regular use by Netsteria, an experimental peer-to-peer application supporting text-based chat, voice-over-IP calling and conferencing, and swarming file transfers. As the application is primarily a vehicle for exploring networking ideas such as SST, it focuses little attention on backward compatibility or interoperability with existing application protocols on existing transports.

Netsteria's combination of different types of network activities operating concurrently serves well to exercise SST's capabilities and drive its development. The file transfer mechanism, for example, divides files into variable-length blocks and uses a separate SST stream for each block request/reply transaction, making use of SST's scalability over transaction sizes. The voice chat mechanism uses SST's ephemeral substreams to transmit small media frames efficiently with best-effort delivery to minimize latency.

7.5.2 Performance Validation

To test SST's basic performance against the "gold standard" of TCP, we first run microbenchmarks of raw bandwidth and TCP-friendliness on three transports: the SST prototype, the host operating system's native TCP, and a user-space TCP implementation that was developed alongside the SST library for comparison purposes. Though the native TCPs are more mature, the user-space TCP can run on either a real or simulated network like the SST library. Since SST always uses selective acknowledgments, the user-space TCP implements TCP's SACK extension to ensure a fair comparison, as do the native TCP stacks on the hosts used for testing. Since TCP does not provide cryptographic security, the benchmarks run SST in its comparable checksum-based authentication mode.

Downloading a 10MB file from a PC running SuSE Linux 10.0 to a MacBook Pro running Mac OS 10.4.8 over a real 1.5Mbps DSL connection, and taking the best of three runs to factor out out possible delays caused by unrelated system daemon activity, SST was measured to be 1.0% slower than native TCP, and user-space TCP was 2.1% slower—a difference barely out of the noise, but attributable to the overhead of implementing transports outside the kernel atop UDP. Running the same benchmark over an 802.11g wireless LAN providing about 18Mbps maximum throughput, SST was 7.1% slower than native TCP, and user-space TCP was 3.6% slower. These results suggest that even the unoptimized SST prototype performs adequately on "consumer-grade" networks, although a more optimized implementation would be desired on high-speed networks. Comparing SST against the user-space TCP on simulated networks with similar parameters, the two transports exhibited identical performance to within 0.2%.

The second benchmark runs two downloads at once—one using the native TCP, the other using either SST or the user-space TCP—to verify "TCP-friendly" congestion control behavior. The user-space transports were found to be fair, and just barely less aggressive than native TCP: SST takes a 48.8% bandwidth share against native TCP's 51.2% share, and the user-space TCP takes 48.1% against native TCP's 51.9%. This result is unsurprising given that both user-space transports implement the classic TCP congestion control schemes.

7.5.3 Scalability over Transaction Size

We now compare SST against TCP and UDP when used for Web-style transactions in which the request is small but the response varies in size. Since we wish to examine how SST's performance

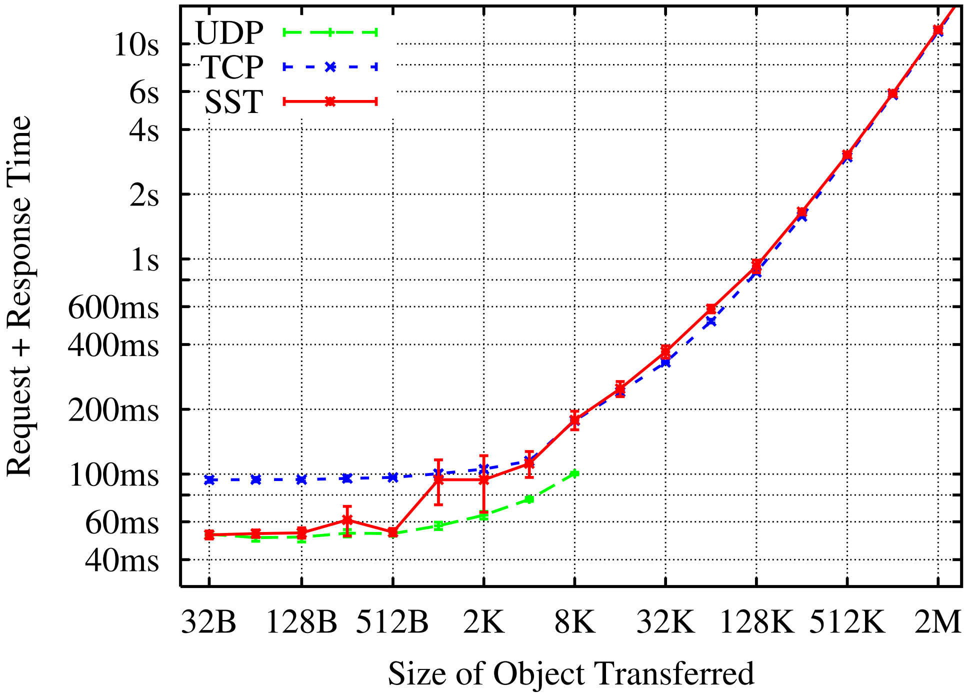


Figure 7-9: Transactional use of UDP, TCP, and SST over a $2^{16}\times$ range of transaction sizes.

scales when the application uses transport instances to match its transaction structure, this test uses one SST or TCP stream per transaction as in HTTP/1.0, leaving HTTP/1.1 persistent streams to the next section. The UDP test operates as in DNS or RPC, with each request datagram soliciting a single response datagram.

Figure 7-9 shows client-observed transaction latency on a log/log plot for responses ranging from 32 bytes to two megabytes, measured on the real 1.5Mbps DSL connection described above, which has about 50ms minimum latency. For small transactions where network latency dominates, TCP takes twice as long as UDP due to its 3-way handshake. UDP ceases functioning beyond around 8KB due to middleboxes on the test connection, and IP version 4 limits datagrams to 64KB in any case. In this test the network connection was quiescent and no UDP datagrams were lost, but in a second test whose results are not shown, on a connection loaded with two concurrent long-running TCP downloads, the effective UDP datagram loss rate often exceeded 50% even at the widely-used 8KB datagram size.

As the graph shows, SST can create new streams for small transactions with the same low latency as UDP, while scaling to support long-running transfers. The SST test runs its transactions over a “warm” communication channel already set up by the negotiation protocol, representing the common case in which a client makes multiple requests to the same server. Even without a warm channel, SST can piggyback the first application request and response data segments onto the negotiation protocol packets if cryptographic security is not required and the responder is not

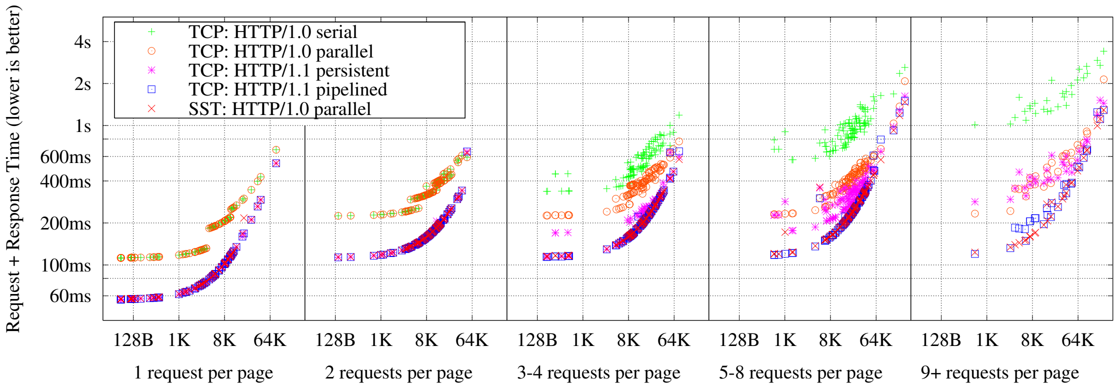


Figure 7-10: Web workload comparing single-transaction SST streams against four HTTP flavors over TCP.

heavily loaded, retaining a total latency of one round trip. Otherwise, SST adds one round trip delay for channel setup.

7.5.4 Web Traffic Workload

HTTP/1.1 addressed the inefficiency of short-lived TCP streams through persistent connections, which are now in common use, and pipelining, which is not. Since SST attempts to offer the benefits of persistent streams with the simplicity of the one-transaction-per-stream model, we now compare SST against the behavior of several flavors of HTTP over TCP, under a simulated web workload. While HTTP is a useful protocol for comparison purposes because it is well-understood and supports several connection schemes, the issues we explore here are not specific to HTTP but apply to any application protocol based on a request/response model.

For this test we simulate a series of web page loads, each page consisting of a “primary” HTTP request for the HTML, followed by a batch of “secondary” requests for embedded objects such as images. As the simulation’s workload we use a fragment of the UC Berkeley Home IP web client traces available from the Internet Traffic Archive [119]. We sort the trace by client IP address so that each user’s activities are contiguous, then we use only the order and sizes of requests to drive the simulation, ignoring time stamps. Since the traces do not indicate which requests belong to one web page, the simulation approximates this information by classifying requests by extension into “primary” (e.g., ‘`.html`’ or no extension) and “secondary” (e.g., ‘`gif`’, ‘`.jpg`’, ‘`.class`’), and then associating each contiguous run of secondary requests with the immediately preceding primary request. The simulation pessimistically assumes that the browser cannot begin requesting secondary objects until it has downloaded the primary object completely, but at this point it can in theory request all of the secondary objects in parallel. This assumption may slightly decrease the amount of potential parallelism available to any of the downloading schemes tested: since SST is designed to take maximum advantage of available parallelism, any experimental error caused by this assumption is merely likely to make SST's benefit appear less in the simulations than it would be in reality.

Figure 7-10 shows a scatter plot of the total duration of each web page load against the total size of all downloads for that page, on the simulated 1.5Mbps network used in Section 7.5.2. The plot is divided into five groups by the total number of HTTP requests per web page. The leftmost group, for pages with no secondary requests, has a best-case load time half that of other groups, because in the latter groups secondary requests do not start until the primary request completes. The points labeled "HTTP/1.0 serial" reflect the behavior of early web browsers that load pages by opening TCP connections for each request sequentially, "HTTP/1.0 parallel" represents browsers that open up to eight single-transaction TCP streams in parallel, "HTTP/1.1 persistent" represents modern browsers that use up to two concurrent persistent TCP streams as per RFC 2616 [78], and "HTTP/1.1 pipelined" uses two concurrent streams with up to four pipelined requests each. The SST case uses one transaction per stream, as in HTTP/1.0, but imposes no limit on the number of parallel streams. As the graph indicates, HTTP/1.0 over SST achieves performance comparable to pipelined HTTP/1.1 streams over TCP, both of which are much faster than other methods, including the current common case of persistent but non-pipelined TCP streams.

7.5.5 Dynamic Prioritization

In a final experiment, we consider a hypothetical SST-enabled web browser in which a user views a "photo album" page containing several large images. Traditional browsers load the images on a page from top to bottom, so if the user immediately scrolls within the page after opening it, or clicks on a link to a text anchor somewhere in the middle of the page, she must wait until the browser loads the (probably invisible) images above the visible area before the desired images begin to appear. Our SST-enabled browser instead expedites the loading of the image(s) within the currently visible scroll area—perhaps in particular the image immediately under the user's mouse pointer. In this scenario, the image to be expedited might change at any time as the user scrolls the window or moves the mouse.

With persistent or pipelined TCP connections, the browser cannot change the order of requests already in the pipeline, but with SST the browser and web server can cooperate to achieve the desired result. The client specifies an initial priority for each request it submits, and changes the priority of a request already in progress by spawning a temporary substream from the request's original stream and sending a short "change priority" message on this substream. On receipt, the server attaches this new priority level to the appropriate SST stream on its end, causing its stream layer to transmit data for high-priority streams before others. This prioritization feature required no changes to the SST protocol as described in Section 4.4, and only a minor API extension to the SST implementation for the server's use. The server could theoretically also use the client's priority information to control other processing resources on behalf of the client, such as the relative priorities of disk reads or threads performing CGI computations.

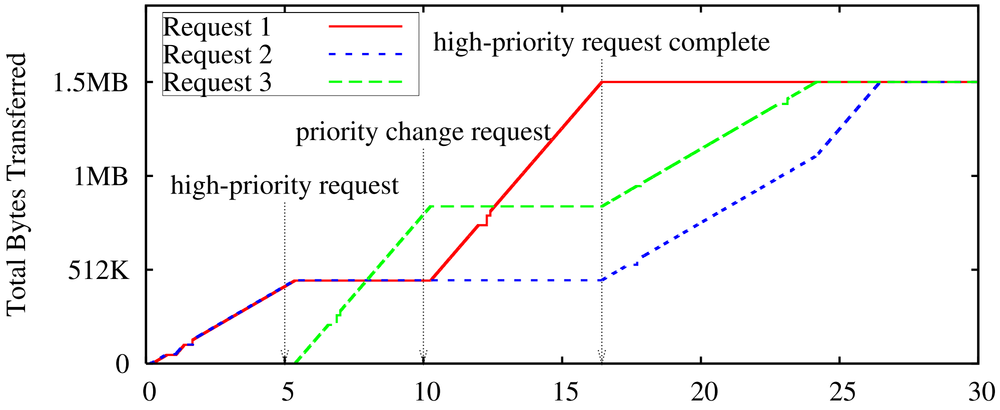


Figure 7-11: Dynamic request prioritization

	Stream Delivery			Datagram Delivery		
	SST	TCP	SCTP	SST	UDP	DCCP
Data Packet	20	20	28	16	8	12
Ack Packet	20	20	28	16	—	16
Sack Packet	20	32+	28+	16	—	20+
Data + Ack	20	20	44	16	—	16
Data + Sack	20	32+	44+	16	—	20+

Table 7.3: Transport layer header overhead

Figure 7-11 shows the behavior observed by the client in a simple scenario on the usual simulated 1.5Mbps network. At time zero the client requests two 1.5MB files at normal priority, and the server divides return bandwidth evenly between them. At five seconds the client submits a third request labeled high-priority, causing the server to commit all bandwidth to the new request, temporarily blocking the old ones. At ten seconds the client submits two priority change requests, changing Request 1 to high-priority and Request 3 to normal, and the client observes the priority changes take effect one round-trip later. When Request 1 finally completes, the remaining two requests again divide available bandwidth evenly until they complete as well.

7.5.6 Wire Efficiency

Minimizing the per-packet overhead of transport layer headers is important to many applications, especially voice applications that frequently send frames only a few bytes in size. Table 7.3 compares SST's header overhead in bytes against the minimal overhead imposed by several other transports. The numbers for SST include the 32-bit lightweight checksum that SST uses in its non-cryptographic security mode, but do not include a UDP encapsulation header since SST could be run directly atop IP like the other transports. The DCCP numbers are for DCCP's short header format, which uses 24-bit sequence numbers instead of 48-bit at the cost of weakening the protocol against packet forgery attacks. SST also transmits 24-bit sequence numbers in packets, but does not

	SST	TCP	SCTP	DCCP	RDP	UDP
Prototype	4400	540				
Linux 2.6.20		5400	8000	2900		630
FreeBSD 5.4		4400				510
4.3BSD		990			900	170

Table 7.4: Transport code size (semicolon count)

rely on them to protect against forgery, instead relying on 32-bit keyed checksums that depend on full 64-bit internal sequence numbers. SST effectively provides most of the functionality of SCTP and DCCP, along with structured stream support not available in any existing transport, with no more wire overhead than basic TCP.

7.5.7 Implementation Size

For a rough comparison of implementation complexity, Table 7.4 shows the code size of several transports measured in number of semicolons. The top line shows the user-space C++ implementations of SST and TCP used in the above experiments; the other lines show existing transports written in C. The user-space TCP is “bare-bones” and implements only the TCP features needed for the above experiments. The SST prototype will no doubt grow as it matures, but it already includes cryptographic security functionality that none of the other transports do. In comparison, `libssl` from OpenSSL 0.9.8e is about 13,000 semicolons (41,000 lines).

7.6 NAT Traversal Evaluation

To evaluate the robustness of the TCP and UDP hole punching techniques described in Chapter 5 on a variety of existing NATs, we implemented and distributed a test program called NAT Check [169], and solicited data from Internet users about their NATs.

NAT Check’s primary purpose is to test NATs for the two behavioral properties most crucial to reliable UDP and TCP hole punching: namely, consistent identity-preserving endpoint translation (Section 5.4.1), and silently dropping unsolicited incoming TCP SYNs instead of rejecting them with RSTs or ICMP errors (Section 5.4.2). In addition, NAT Check separately tests whether the NAT supports hairpin translation (Section 5.4.4), and whether the NAT filters unsolicited incoming traffic at all. This last property does not affect hole punching, but provides a useful indication the NAT’s firewall policy.

NAT Check makes no attempt to test every relevant facet of NAT behavior individually: a wide variety of subtle behavioral differences are known, some of which are difficult to test reliably [123]. Instead, NAT Check merely attempts to answer the question, “how commonly can the proposed hole punching techniques be expected to work on deployed NATs, under typical network conditions?”

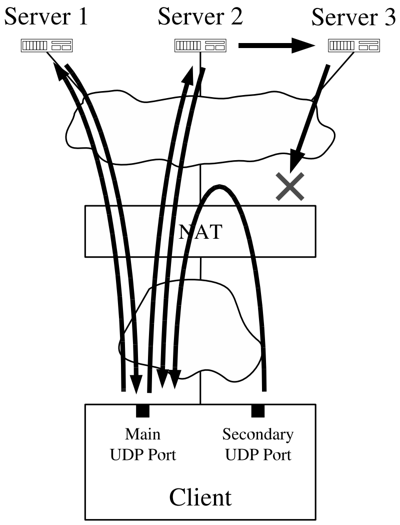


Figure 7-12: NAT check test method for UDP

7.6.1 Test Method

NAT Check consists of a client program to be run on a machine behind the NAT to be tested, and three well-known servers at different global IP addresses. The client cooperates with the three servers to check the NAT behavior relevant to both TCP and UDP hole punching. The client program is small and relatively portable, currently running on Windows, Linux, BSD, and Mac OS X. The machines hosting the well-known servers all run FreeBSD.

UDP Test: To test the NAT's behavior for UDP, the client opens a socket and binds it to a local UDP port, then successively sends "ping"-like requests to servers 1 and 2, as shown in Figure 7-12. These servers each respond to the client's pings with a reply that includes the client's public UDP endpoint: the client's own IP address and UDP port number as observed by the server. If the two servers report the same public endpoint for the client, NAT Check assumes that the NAT properly preserves the identity of the client's private endpoint, satisfying the primary precondition for reliable UDP hole punching.

When server 2 receives a UDP request from the client, besides replying directly to the client it also forwards the request to server 3, which in turn replies to the client from its own IP address. If the NAT's firewall filters "unsolicited" incoming traffic on a per-session basis, then the client never sees these replies from server 3, even though they are directed at the same public port as the replies from servers 1 and 2.

To test the NAT for hairpin translation support, the client simply opens a second UDP socket at a different local port and uses it to send messages to the *public* endpoint representing the client's first UDP socket, as reported by server 2. If these messages reach the client's first private endpoint, then the NAT supports hairpin translation.

TCP Test: The TCP test follows a similar pattern as for UDP. The client uses a single local TCP port to initiate outbound sessions to servers 1 and 2, and checks whether the public endpoints reported by servers 1 and 2 are the same, the first precondition for reliable TCP hole punching.

The NAT's response to unsolicited incoming connection attempts also impacts the speed and reliability of TCP hole punching, however, so NAT Check also tests this behavior. When server 2 receives the client's request, instead of immediately replying to the client, it forwards a request to server 3 and waits for server 3 to respond with a "go-ahead" signal. When server 3 receives this forwarded request, it attempts to initiate an inbound connection to the client's public TCP endpoint. Server 3 waits up to five seconds for this connection to succeed or fail, and if the connection attempt is still "in progress" after five seconds, server 3 responds to server 2 with the "go-ahead" signal and continues waiting for up to 20 seconds. Once the client finally receives server 2's reply (which server 2 delayed waiting for server 3's "go-ahead" signal), the client attempts an outbound connection to server 3, effectively causing a simultaneous TCP open with server 3.

What happens during this test depends on the NAT's behavior as follows. If the NAT just drops server 3's "unsolicited" incoming SYN packets, then nothing happens on the client's listen socket during the five second period before server 2 replies to the client. When the client finally initiates its own connection to server 3, opening a hole through the NAT, the attempt succeeds immediately. If on the other hand the NAT does *not* drop server 3's unsolicited incoming SYNs but allows them through (which is fine for hole punching but not ideal for security), then the client receives an incoming TCP connection on its listen socket before receiving server 2's reply. Finally, if the NAT actively rejects server 3's unsolicited incoming SYNs by sending back TCP RST packets, then server 3 gives up and the client's subsequent attempt to connect to server 3 fails.

To test hairpin translation for TCP, the client simply uses a secondary local TCP port to attempt a connection to the public endpoint corresponding to its primary TCP port, in the same way as for UDP.

7.6.2 Test Results

The NAT Check data we gathered consists of 380 reported data points covering a variety of NAT router hardware from 68 vendors, as well as the NAT functionality built into different versions of eight popular operating systems. Only 335 of the total data points include results for UDP hairpin translation, and only 286 data points include results for TCP, because we implemented these features in later versions of NAT Check after we had already started gathering results. The data is summarized by NAT vendor in Table 7.5; the table only individually lists vendors for which at least five data points were available. The variations in the test results for a given vendor can be accounted for by a variety of factors, such as different NAT devices or product lines sold by the same vendor, different software or firmware versions of the same NAT implementation, different configurations, and probably occasional NAT Check testing or reporting errors.

Out of the 380 reported data points for UDP, in 310 cases (82%) the NAT consistently translated the client's private endpoint, indicating basic compatibility with UDP hole punching. Support for

	UDP				**TCP**			
	Hole Punching		Hairpin		Hole Punching		Hairpin	
NAT Hardware								
Linksys	45/46	(98%)	5/42	(12%)	33/38	(87%)	3/38	(8%)
Netgear	31/37	(84%)	3/35	(9%)	19/30	(63%)	0/30	(0%)
D-Link	16/21	(76%)	11/21	(52%)	9/19	(47%)	2/19	(11%)
Draytek	2/17	(12%)	3/12	(25%)	2/7	(29%)	0/7	(0%)
Belkin	14/14	(100%)	1/14	(7%)	11/11	(100%)	0/11	(0%)
Cisco	12/12	(100%)	3/9	(33%)	6/7	(86%)	2/7	(29%)
SMC	12/12	(100%)	3/10	(30%)	8/9	(89%)	2/9	(22%)
ZyXEL	7/9	(78%)	1/8	(13%)	0/7	(0%)	0/7	(0%)
3Com	7/7	(100%)	1/7	(14%)	5/6	(83%)	0/6	(0%)
OS-based NAT								
Windows	31/33	(94%)	11/32	(34%)	16/31	(52%)	28/31	(90%)
Linux	26/32	(81%)	3/25	(12%)	16/24	(67%)	2/24	(8%)
FreeBSD	7/9	(78%)	3/6	(50%)	2/3	(67%)	1/1	(100%)
All Vendors	310/380	(82%)	80/335	(24%)	184/286	(64%)	37/286	(13%)

Table 7.5: User reports of NAT support for UDP and TCP hole punching

hairpin translation is much less common, however: of the 335 data points that include UDP hairpin translation results, only 80 (24%) show hairpin translation support.

Out of the 286 data points for TCP, 184 (64%) show compatibility with TCP hole punching: the NAT consistently translates the client's private TCP endpoint, and does not send back RST packets in response to unsolicited incoming connection attempts. Hairpin translation support is again much less common: only 37 (13%) of the reports showed hairpin support for TCP.

Since these reports were generated by a "self-selecting" community of volunteers, they do not constitute a random sample and thus do not necessarily represent the true distribution of the NATs in common use. The results are nevertheless encouraging: it appears that the majority of commonly-deployed NATs already support UDP and TCP hole punching at least in single-level NAT scenarios.

7.6.3 Testing Limitations

There are a few limitations in NAT Check's current testing protocol that may cause misleading results in some cases. First, we only learned recently that a few NAT implementations blindly translate IP addresses they find in unknown application payloads, and the NAT Check protocol currently does not protect itself from this behavior by obfuscating the IP addresses it transmits.

Second, NAT Check's current hairpin translation checking may yield unnecessarily pessimistic results because it does not use the full, two-way hole punching procedure for this test. NAT Check currently assumes that a NAT supporting hairpin translation does not filter "incoming" hairpin connections arriving from the private network in the way it would filter incoming connections arriving at the public side of the NAT, because such filtering is unnecessary for security. We later realized, however, that a NAT might simplistically treat *any* traffic directed at the NAT's public ports as "untrusted" regardless of its origin. We do not yet know which behavior is more common.

Finally, NAT implementations exist that consistently translate the client's private endpoint as long as *only one* client behind the NAT is using a particular private port number, but switch to symmetric NAT or even worse behaviors if two or more clients with different IP addresses on the private network try to communicate through the NAT from the same private port number. NAT Check could only detect this behavior by requiring the user to run it on two or more client hosts behind the NAT at the same time. Doing so would make NAT Check much more difficult to use, however, and impossible for users who only have one usable machine behind the NAT. Nevertheless, we plan to implement this testing functionality as an option in a future version of NAT Check.

7.6.4 Corroboration of Results

Despite testing difficulties such as those above, our results are generally corroborated by those of a large ISP, who recently found that of the top three consumer NAT router vendors, representing 86% of the NATs observed on their network, all three vendors currently produce NATs compatible with UDP hole punching [252]. Additional independent results recently obtained using the UDP-oriented STUN protocol [123], and STUNT, a TCP-enabled extension [98, 100], also appear consistent with our results. These latter studies provide more information on each NAT by testing a wider variety of behaviors individually, instead of just testing for basic hole punching compatibility as NAT Check does. Since these more extensive tests require multiple cooperating clients behind the NAT and thus are more difficult to run, however, these results are so far available on a more limited variety of NATs.

7.7 Summary

In this chapter, we first informally evaluated the currently implemented and deployed naming and routing prototype from a high-level usability perspective, then delved into each of the alternative approaches to routing in Sections 7.2–7.4, focusing on performance metrics such as success rate, routing table size, and route stretch. Section 7.5 then evaluated UIA's structured stream transport from the perspective of responsiveness—a particularly important usability metric for personal devices running highly interactive communication applications. Finally, Section 7.6 presented an experimental study on the effectiveness of hole punching as a tool for optimize the UIA routing layer or other peer-to-peer communication mechanisms.

A common conclusion that may be safely drawn from all of these sections is that the possible design space in each functional area is vast and much more work will be needed to achieve the full promise of "ubiquitous computing" using mobile personal devices, but the evaluation in this chapter suggests that the ideas proposed in this thesis represent a promising stepping stone for the promise of "ubiquitous computing."

Chapter 8

Related Work

UIA builds on a large body of related work in the areas of naming systems, location-independent identifiers, gossip and optimistic replication protocols, and social networks. This chapter explores the relationship of UIA and its components to prior work. The most important contributions of UIA with respect to all of this prior work are the new abstractions that UIA introduces for personal device connectivity, although Section 1.7 also lists other contributions in all the functional areas that don't necessarily involve new network abstractions:

- **Personal groups** built from pairwise introductions and managed without reference to central authorities or services.

- **Personal names** supporting global connectivity between users and their personal devices, without requiring global uniqueness.

- **Structured streams** for efficient, responsive communication between personal devices, designed for delay-sensitive interactive applications that manage many communication activities in parallel.

We first examine related work at the network architecture level, then work related to individual UIA components.

8.1 Network Architectures

UIA's basic usability goal of making personal device connectivity "just work," without users having to configure protocols or register with central services, is consistent with the classic ubiquitous computing idea of trying to make computers "disappear" from the user's consciousness [260]. UIA's technical design builds on and is heavily inspired by the Internet's existing architecture [54], including its foundation on best-effort packet-switched delivery and a separation of naming, routing, transport, and application concerns. In addition, UIA adopts several existing architectural extensions to the classic Internet that have been proposed before but not yet widely adopted.

IP addresses are traditionally overloaded to serve the roles of both node identification and routing, causing both confusion and technical difficulties when nodes need to move while retaining their logical identities [50, 159, 210, 219]. UIA incorporates the idea of splitting the "locator" and "identifier" roles of Internet addresses, as developed in other architectures such as Nimrod [39,191], HIP [163], FARA [52], JXTA [96], and $i3$ [235]. As in HIP, UIA borrows from SFS [157] the idea of making node identifiers self-certifying by generating them from the hash of a public key. In contrast with prior architectures, however, UIA supports not just identification of *hosts* but also the convenient identification and management of *personal groups* of administratively related hosts, so that users can identify and name each other via persistent, location-independent logical group identities whose specific device membership may change over time. UIA's design also refines previous notions of host identity by introducing cryptographic EIDs that securely identify not just a host but a particular *user* on that host. Different users of a shared UIA host can run independent personal network services without conflicting or requiring host-wide configuration, and network services can leverage UIA names and EIDs to authenticate clients at user granularity.

UIA is also influenced by the architectural principle of application level framing [51]. One of the primary goals of UIA's structured stream transport is to provide transport objects that directly mirror the structure of the "application data units" with which the application is concerned.

The Mobile People Architecture (MPA) [150] proposes adding a new "Person Layer" above the Application Layer in the current Internet architecture, which provides each individual user with a unique identifier and a Personal Proxy that acts as a central point of contact for the user regardless of which personal device(s) he is using at the moment. The user's Personal Proxy can for example forward incoming calls to the mobile personal device the user is currently carrying, to a fixed device at the user's current location, or to the user's voicemail if the user cannot be located. The Universal Inbox [190] provides similar services, but implements them in the Internet core rather than at the edges. MPA and Universal Inbox are complementary to UIA in that they address user location tracking, call redirection, and media conversion, which UIA leaves to higher layers: it should be natural to build services analogous to MPA's "Person Layer" or the Universal Inbox atop UIA's naming and routing facilities. UIA on the other hand addresses the problem of making it easy for technically unsophisticated users to build and manage groups of personal devices, a problem neither MPA nor Universal Inbox addresses. UIA also supports communication within disconnected edge networks, which fails in MPA and Universal Inbox because the user becomes unreachable by anyone whenever the user's Personal Proxy or other central intermediary services are unreachable.

UIA is a continuation of work begun with Unmanaged Internet Protocol [80, 81]. UIA extends the earlier work with its personal naming system, new overlay routing algorithms, and the structured stream transport.

8.2 Naming

UIA's personal naming model is inspired in part by SDSI/SPKI [71, 203]. Like SDSI, UIA allows users to define locally-scoped personal names bound to cryptographic targets and groups to form decentralized, composable namespaces. While SDSI associates public keys with users (principals) and expects users to know about and manage their own public keys, however, UIA simplifies key management by making each device responsible for creating and managing its own device-specific key invisibly to the user. UIA devices form *user* identities out of cooperating groups of personal devices, which the user builds through simple device introduction and merge. SDSI's model for designated certificate servers does not adapt well to disconnected mobile devices, however. UIA handles lost or stolen devices without re-keying and thus losing the user's identity.

UIA's introduction process builds on previously explored ideas for secure associations, such as the Resurrecting Duckling security model [231] and SPKI/SDSI introduction [65], but UIA is unique in providing the ability to build symmetric, self-managing, distributed personal groups solely out of pairwise introductions.

Existing Internet protocols can provide some of UIA's naming features, but they require configuration effort and technical expertise that deters even sophisticated users. Dynamic DNS [256] can name devices with dynamic IP addresses, but requires configuration on both the name server and the named device, and devices still become inaccessible when behind NAT. DNS Security [14] cryptographically authenticates DNS names, but its administration cost has hindered deployment even by the Internet's root naming authorities, let alone by ordinary users.

Uniform Communication Identifiers [74] provide a common identifier for phone, E-mail, and other forms of communication, along with a central address book shareable among communication devices. HINTS [149] uses name-history trails to map obsolete user names to current ones. These systems still rely on globally unique names with centralized registration and management, however.

Bonjour [11] allows devices to choose their own names on local-area networks, but these names are insecure and ephemeral: any device can claim any name, and its name becomes invalid as soon as it moves to a different network. UIA uses Bonjour libraries to discover new devices on the local network, but UIA names persist and remain bound to the original target device despite later migration.

DHT-based naming systems such as DDNS [57], $i3$ [235], and CoDoNS [192] provide new mechanisms for resolving global names. TRIAD [47] provides content delivery and NAT traversal by routing on global DNS names. In place of global names, UIA focuses on global connectivity via *personal* names, which users can choose without the restriction of global uniqueness. In addition, UIA's optimistic replication of naming state keeps the user's namespace available on his devices even while disconnected from the Internet and its global name services or DHTs.

Ficus [103, 105], Coda [133], and Ivy [167] develop optimistic replication algorithms for file systems, and Bayou [243] does so for databases. Rumor [104] and P-Grid [61] explore optimistic data replication on mobile devices, Roma [240] uses one mobile device to offer central management of data on other devices, and Footloose [176] uses mobile devices the user carries to propagate updates among other devices. UIA builds on all of this work to address distributed naming and ad hoc group management, confronting the additional challenge of maintaining consistency when not only the *data content* but the *set of participants* may change independently on different devices.

The design of UIA's personal name system may have applicability beyond the domain of personal devices to other peer-to-peer systems. For example, systems such as SFR [24, 258] and $i3$ [235] also identify network endpoints using cryptographically secure, but otherwise meaningless, bit strings. UIA's name system can be used to provide human-readable names to such endpoints in a secure and user-friendly manner.

UIA's support for naming and connecting to other users via personal group names effectively creates an explicit social network, comparable to those maintained by popular web sites such as friendster [90], orkut [174], and facebook [76], except in a completely decentralized fashion with no central database. UIA's social network could be useful to higher-level services and applications for various purposes, such as for sybil-resistant DHT key/value lookup [60] and other sybil-resistant services [268, 269], and for social data sharing, search, and storage systems such as Turtle [184], SPROUT [153], F2F [145], and Tribler [188].

8.3 Routing

Existing Internet protocols can provide some of UIA's connectivity features, but they require configuration effort and technical expertise that deters even sophisticated users. Mobile IP [183] gives a mobile device the illusion of a fixed IP address, but requires setting up a dedicated forwarding server at a static, public IP address. Virtual Private Networks (VPNs) [95] provide secure remote access to corporate networks, but their infrastructure and administration requirements make them unsuitable for deployment by average consumers for their personal networks.

A resilient overlay network (RON) [9] serves a function similar in spirit to the UIA routing layer, increasing the reliability of an IP network by detecting connectivity failures in the underlying network and forwarding traffic around them. RON makes no attempt at scalability beyond a few dozen nodes, however, and assumes that all participating nodes have unique IP addresses.

Several protocols have been developed to provide connectivity through firewalls and NATs, such as SOCKS [143], STUN [207], and UPnP [253]. These special-purpose protocols are tied to the characteristics and network topologies of commonly deployed NATs and firewalls, however, and do not solve the more general problem of routing between different address domains connected in arbitrary fashion.

The following sections examine related work specific to the specific routing methods explored in UIA.

8.3.1 Social Routing

In classic distance-vector routing algorithms [126] such as RIP [109], as well as variants such as MS [158], ExBF [44], WRP [166], and DSDV [182], each router continuously maintains routing information about every other addressable node or subnet. In link-state algorithms such as OSPF [164] and FSR [180], routers maintain complete network connectivity maps, achieving faster routing table convergence and avoiding the looping problems of basic distance-vector algorithms. With all of these protocols, however, each router requires at least $O(N)$ storage for a network of size N, and must regularly exchange connectivity information of size $O(N)$ with each of its neighbors. Since UIA is intended to run on personal mobile devices with limited storage and network bandwidth, but which may be interconnected over the Internet in large numbers, it is not practical for each UIA device to maintain individual records for every other UIA device in the network: thus, UIA requires a more scalable approach. UIA's social routing protocol achieves this scalability by limiting scope to routing among social neighbors, whereas the other two schemes use distributed routing structures to provide scalable routing among arbitrary nodes while requiring limited state at each node. In effect, UIA's social routing protocol does not attempt to provide connectivity between *arbitrary* graph nodes efficiently: instead it focuses on providing simple and efficient connectivity between *socially or administratively related* nodes: i.e., a user's own personal devices, and those of friends and acquaintances.

Reactive or "on demand" routing algorithms designed for ad hoc networks, such as DSR [124] and AODV [181], require routers to store information only about currently active routes, limiting maintenance traffic and storage overheads on networks with localized traffic patterns. Routing queries for distant nodes may have to be broadcast through most of the network before the desired route is found, however, limiting the scalability of these protocols on networks with global traffic patterns. The social routing protocol uses scoped flooding to locate and create a forwarding path to an arbitrary node on demand, similar to ad hoc protocols like DSR and AODV, and to unstructured peer-to-peer search mechanisms as in Gnutella [42]. UIA's social routing algorithm performs this flooding on a graph based on the user's *social network*, however, rather than on a graph based on the underlying physical topology. In this sense UIA is analogous to systems that leverage social networks for other purposes, such as Turtle [184] and Tribler [188] for peer-to-peer file sharing, SPROUT [153] and other DHTs that leverage social networks for secure key lookup [60], and F2F [145] for friend-to-friend data storage.

8.3.2 Identity Hash Routing

Distributed hash tables (DHTs) [19, 209, 234] provide scalable lookup of arbitrary flat identifiers in large distributed address spaces, but tolerate only limited asymmetry or non-transitivity in the underlying network [89]. UIA's Identity Hash Routing scheme is inspired by DHTs, but it addresses the more difficult problem of providing both resolution of flat identifiers *and* creating any necessary forwarding paths over potentially arbitrary topologies with asymmetric connectivity.

Like IHR, ROFL [37] borrows DHT techniques to build a scalable routing mechanism. While ROFL's goal is to build a new routing infrastructure for the Internet core to replace BGP [198] and other routing protocols based on structured CIDR [91] addresses, IHR's goal is to provide a scalable routing *overlay* atop the Internet's existing routing infrastructure, implemented primarily by personal edge devices instead of core routers. Since it is relatively easy to deploy operating system extensions incrementally on edge devices but difficult to modify Internet core routers and routing protocols, the overlay approach is likely to be easier to deploy in practice. ROFL and IHR share the common limitation of offering no formal bound on stretch, and no obvious way to modify the algorithms to guarantee such a bound; this limitation may be more of a problem for ROFL, since it aspires to be the "primary" routing mechanism to forward IP traffic across many link layer hops, whereas UIA's overlay routing layer merely provides a "backup" routing mechanism that forwards traffic over a few IP-level hops only when direct IP-level connectivity is unavailable.

8.3.3 Compact Routing

UIA's compact routing protocol builds on recent theoretical work, which has produced a new class of routing algorithms ensuring both small (sub-linear) routing table size on every node and bounded worst-case stretch on arbitrary graphs [2, 15, 56, 245], Unfortunately it is not self-evident how to convert any of these centralized, static *algorithms* into practical distributed *protocols*, let alone deploy them on existing networks. UIA's TZk-based protocol is intended to be one step toward making formal compact routing algorithms practical.

Prior analysis of one compact routing algorithm indicates that the algorithm should provide average-case stretch close to 1 on Internet-like graphs [139], even though its worst-case stretch is 3. This analysis was done on a different algorithm by Thorup and Zwick from the one used in this thesis, although both algorithms were presented in the same paper [245]. The two algorithms are closely related, however, and may have similar average-case stretch properties.

Some formal analysis has also been done on the communication cost of responding to dynamic topology changes. The worst-case cost is known to be proportional to network diameter, and thus $\Omega(n)$ for arbitrary graphs [3]. Korman and Peleg presented a routing scheme that efficiently adapts to changes in edge weights (but not additions or removals) in graphs with low local density [138]. Krioukov et al. noted that random scale-free graphs have high local density [140], shedding doubt on the practicality of large-scale compact routing due to the control overhead of maintaining routing tables. This cost analysis assumes however that routing tables are always recomputed on every minor topology change such as a temporary link or router failure. UIA's compact routing model assumes in contrast that routing tables need to be recomputed only occasionally as the network's *nominal* topology evolves, leaving temporary outages to be handled by redundancy built into the routing algorithm and into lower-level networking layers.

UIA builds on prior work analyzing compact routing algorithms by examining other practical aspects of routing protocol behavior not previously considered, such as route congestion and robustness to failures. UIA's protocol has also been validated via simulation on a wider variety of real and synthetic network topologies, and at much larger scales than has been done previously: e.g., one graph used is a Skitter [38] AS-adjacency graph containing about 190,000 nodes and 600,000 edges.

UIA's TZk-based protocol implements *name-dependent* compact routing, meaning that the protocol assigns each node a topology-sensitive locator that other nodes use to contact it. There are *name-independent* schemes that accept arbitrary location-independent node labels [2, 15] and have similar worst-case stretch, but their *average-case* stretch is generally higher than for name-dependent routing, because any message must be sent to some node that knows the location of the target before being forwarded to the target itself [140]. Since nodes on real networks usually exchange multiple messages in a conversation, it may be preferable for purposes of minimizing stretch to perform a single address lookup in a directory of some kind, such as the DNS or a DHT, paying this cost only once and then using more efficient routing for subsequent messages.

TZk's hierarchy of landmarks is reminiscent of Landmark [251], and related hierarchical ad hoc routing protocols such as LANMAR [93], L+ [43], and PeerNet [73], all of which dynamically arrange mobile nodes into a tree. These protocols assign each node a hierarchical address corresponding to its current location in this tree, and implement a location-independent identity-based lookup service by which the current address of any node can be found. Each non-leaf node serves as a *landmark* for all of its children, and is responsible for routing traffic to them from nodes outside its local subtree. An important technical difference between TZk and landmark routing is that TZk nodes do not fit into a strict hierarchy but instead choose their reference landmarks based on absolute proximity. In TZk, for example, a level 2 reference landmark may be different from the level 2 reference landmark of the node's level 1 reference landmark: that is, a node in TZk can choose a "grandparent" different from its "parent's parent." This choice of landmarks based on absolute proximity without imposing a strict hierarchy is a crucial property upon which TZk's formal stretch bound depends. Similar to landmark routing, however, TZk routes local traffic purely within the lowest levels of the tree, providing scalability when traffic patterns are predominantly local. Since global traffic must pass through the landmark nodes at the upper levels of the hierarchy, however, these upper-level nodes may be overloaded in a network with global traffic patterns.

8.4 Transport

The popularity of SSL [64] and SSH tunneling [264] attest to the demand for multiplexing logical streams onto a secure channel. MUX [94] and BEEP [204] similarly multiplex logical streams onto one TCP stream, layering their own flow control atop TCP's. These protocols exacerbate TCP's drawbacks, however, by totally ordering many unrelated activities so that one lost packet blocks everything behind it.

SST builds on many ideas borrowed from other transports. RDP [177, 178] provides reliable datagram delivery, in-sequence or as-available according to the application's choice at connection setup. SCTP [232] multiplexes multiple "streams of datagrams" onto one session, provides both reliable and best-effort delivery, and supports "multihomed" endpoints for rapid failover. Its streams do not have individual flow control, however, and cannot be dynamically created or destroyed, but must be negotiated *en masse* at session startup. DCCP [136] is a best-effort datagram service with congestion control, comparable to SST's channel layer, but without SST's packet security features. RDP, SCTP, and DCCP all suffer from the "large datagram" problem: datagrams with too many fragments are almost certain to be lost or require many retransmissions.

SST does not provide multihoming as SCTP does, but its ability to attach streams to more than one channel at once could be extended to support multihoming, or even to load-balance a stream across multiple channels following different network paths. Since SST's stream protocol relies on the channel protocol for loss detection and congestion control, these mechanisms automatically operate at channel—i.e., path—granularity. Performing "end-to-end" load balancing this way could thus avoid both the harmful performance side-effects caused by load balancing in lower layers [27], and the complexity of managing multi-path congestion and retransmission control in a stream transport [121].

The need for efficient transport support for transaction-oriented application protocols has long been recognized [31,35]. VMTP [46] supports lightweight RPC-style communication in a clustered environment, but provides no congestion control and limits messages to 16KB. T/TCP [36] enhances TCP to re-open recently-closed streams quickly, but this serial reuse has the same disadvantages as HTTP/1.1 persistent connections without the benefits of pipelining [171].

TCP has also been extended to share congestion control state across streams [18, 247]. The Congestion Manager [22] enables congestion control sharing across multiple transports; SST should fit well into such an architecture if available.

Another TCP extension provides end-to-end support for host mobility [222]. SST's separation into channel and stream layers, and its ability to migrate streams across channels, provides a cleaner solution reminiscent of a session layer [223].

The rich literature on prioritization in the network layer to ensure quality of service [270] is relevant to SST's use of prioritization in the transport layer to schedule an application's streams relative to each other. Hierarchical schemes [28] may be particularly well-matched to the structured stream abstraction. SST's channel layer could be enhanced with techniques developed in OverQoS [238] to provide better QoS for aggregates of logical streams on ordinary Internet paths.

8.5 NAT Traversal

UDP hole punching was first explored and publicly documented by Dan Kegel [129], and is by now well-known in peer-to-peer application communities. Important aspects of UDP hole punching have also been indirectly documented in the specifications of several experimental protocols, such as STUN [207], ICE [205], and Teredo [116]. We know of no existing published work that thoroughly analyzes hole punching, however, or that points out the hairpin translation issue for multi-level NAT (Section 5.2.5).

We also know of no prior work that develops TCP hole punching in the symmetric fashion described here. Even the existence of the crucial `SO_REUSEADDR`/`SO_REUSEPORT` options in the Berkeley sockets API appears to be little-known among P2P application developers. NatTrav [72] implements a similar but asymmetric TCP hole punching procedure outlined earlier in Section 5.3.5. NUTSS [100] and NATBLASTER [30] implement more complex TCP hole punching tricks that can work around some of the bad NAT behaviors mentioned in Section 5.4, but they require the rendezvous server to spoof source IP addresses, and they also require the client applications to have access to “raw” sockets, usually available only at root or administrator privilege levels.

Protocols such as SOCKS [143], UPnP [253], NAT-PMP [48], and MIDCOM [229] allow applications to traverse a NAT through explicit cooperation with the NAT. These protocols are not widely or consistently supported by NAT vendors or applications, however, and do not appear to address the increasingly important multi-level NAT scenarios. Explicit control of a NAT further requires the application to locate the NAT and perhaps authenticate itself, which typically involves explicit user configuration. When hole punching works, in contrast, it works with no user intervention.

Recent proposals such as Nimrod [39, 191], HIP [163], and FARA [52] extend the Internet’s basic architecture by decoupling a host’s identity from its location [50, 159, 210, 219]. IPNL [87], UIP [80, 81], and DOA [259] propose schemes for routing across NATs in such an architecture. While such extensions are probably needed in the long term, hole punching enables applications to work over the existing network infrastructure immediately with no protocol stack upgrades, and leaves the notion of “host identity” for applications to define.

Chapter 9

Conclusion

This thesis has presented the Unmanaged Internet Architecture, a connectivity architecture for interactive personal devices owned and managed by nontechnical users. The most important contributions of this architecture are three novel networking abstractions: *personal groups* built by pairwise introduction and managed in a fully distributed fashion; *personal names* that provide global connectivity despite being locally scoped to a user's personal group for convenience; and *structured streams* enabling interactive, delay-sensitive applications to manage concurrent transactions effectively and obtain maximum responsiveness from given network conditions. The architecture builds on the central theme of full decentralization, and is comprised of three components: a naming layer implementing personal groups and personal names, a routing layer to provide connectivity among those names, and a transport layer to support high-level application communication via the structured stream abstraction.

9.1 Decentralization

Avoiding unnecessary reliance on central services or authorities is a basic principle applied throughout UIA's design. This principle is motivated by three goals: preserving the user's *autonomy* to manage his own devices without having to register with some organization or get help from a technical expert; preserving *connectivity* among those devices even when they are connected only via an isolated local-area network or a temporarily disconnected Internet fragment; and preserving the user's *privacy* by avoiding an architectural requirement that some central service know the user's device names and where on the Internet they are located at all times.

While the decentralization principle applies most obviously to UIA's naming and routing components, it also factors into the motivation for UIA's structured transport component, although less directly. In a centralized client/server communication model, it may usually be assumed that the server is "well-connected"—i.e., via a high-bandwidth, low-latency connection—and thus if there is a high-latency component of the path it is probably on only one end (the client's). As peer-to-peer communication between personal devices becomes more common and universally supported, however, it will be increasingly common for a peer-to-peer transport session to traverse *two* high-latency links, one for each peer, effectively doubling the total round-trip time and increasing the user-observable responsiveness benefits of avoiding unnecessary serialization onto one TCP stream.

9.2 Naming

The UIA naming system provides a new approach for introducing, naming, and globally connecting mobile personal devices, which is complementary to the traditional global name system embodied by the Internet's Domain Name System. UIA gives users persistent personal names for conveniently finding and expressing who they want to talk to, what devices they wish to access, and who can access their own devices.

Each device starts with a generic name for itself, such as `laptop`, and a cryptographic end-system identifier to provide authentic and private communication. A user can merge devices to form personal groups, which cooperate to maintain a distributed namespace by gossiping logs of the user's changes. A user's group can name both the user's devices and other users' groups; users can form links securely either by physical device introduction or via other trusted channels. Since UIA names are local and personal, users need not register with central authorities to obtain scarce globally unique names.

9.3 Routing

UIA's overlay routing layer provides robust connectivity between personal devices, even as they migrate among different attachment points having different IP addresses, must communicate from behind firewalls and NATs, or have connectivity only to a limited set of nodes in an ad hoc edge network and not to the main Internet. This thesis has explored three approaches to overlay routing, each of which holds promise but also has limitations: *social routing*, *identity hash routing*, and *compact routing*.

- **Social Routing** is UIA's currently deployed overlay routing scheme, which uses ad hoc forwarding through social neighbors' devices to cope with a spectrum of connectivity environments while focusing on providing connectivity between nearby social neighbors. Scoped flooding ensures robustness when groups of devices form isolated islands of connectivity, and a social overlay enables devices to find a target's current IP address efficiently when they have Internet connectivity.

- **Identity Hash Routing** (IHR) builds on scalable structuring ideas borrowed from distributed hash table (DHT) technology. Whereas conventional DHTs provide only a lookup service and assume the underlying network is already fully connected, IHR builds a combined lookup *and* routing service that can both locate a node by its flat, location-independent Endpoint Identifier, and at the same time create a forwarding path to the node. IHR supports two specific forwarding mechanisms: source routing and virtual links. Simulation experiments suggest that IHR should scale well and provide reasonably efficient paths in the common case, although an important weakness of the algorithm is the absence of any formally guaranteed bound on stretch, or path length inefficiency compared to shortest-path routing.

- **Compact Routing** builds on recent theoretical advances to build an overlay routing protocol that should provide both scalability and a formal stretch bound. The protocol selects a hierarchy of landmarks and builds routing trees centered at each landmark; each node in the network joins a limited number of routing trees and thus requires limited-size (e.g., logarithmic) routing tables. Simulation experiments suggest that this protocol should indeed scale well, although it has two downsides: the routing scheme tends to create congestion around high-level landmarks, which we can address either by selecting high-capacity, well-connected nodes as high-level landmarks or by increasing the routing table sizes of individual nodes; and the routing scheme depends on structured addresses, so a lookup mechanism such as a DHT must be layered above it to make it fully usable as an overlay routing protocol for UIA.

One take-away message from this exploration is that there are clearly multiple ways to implement a routing service to fit into the UIA architecture, and none of them may be clearly dominant, so in the long term it is probably important for UIA to remain agnostic as to which specific routing scheme is used to connect clouds of personal devices.

9.4 Transport

UIA's structured stream transport (SST) is designed to support application communication between personal devices once the naming and routing layers have successfully located and connected to a desired target device. The modern interactive applications that users wish to run on personal devices typically involve many independent or concurrent transactions, as well as multimedia streams, none of which TCP can support efficiently and responsively. Although UDP is always an option, it is inconvenient when transactions are *usually* small but *sometimes* may be large, because IP fragmentation only works well in practice up to a limited datagram size. SST subsumes both the TCP stream and UDP datagram abstractions by enabling the application to create and manage many fine-grained streams easily and efficiently, creating a new stream for each transaction and using "ephemeral streams" in liu of datagrams. The central characteristic of the structured stream abstraction is a "fork" operation to create new child streams from an existing stream.

SST is in its infancy and will require refinement, more optimized implementation, further analysis, and integration into the UIA naming and routing prototype. Structured streams nevertheless appear to be a promising enhancement to the classic reliable stream abstraction, in order to better support the needs of modern, highly interactive applications for personal devices. SST's multiplexing of independent lightweight streams onto persistent channels gives applications the flexibility to match their use of streams to their natural structure and transaction granularity, avoiding the start-up delays and serialization imposed by TCP and the datagram size limitations of UDP.

9.5 NAT Traversal

One of the UIA routing layer's functions is to provide communication between personal devices that are behind (the same or different) NATs. Although the UIA routing layer can address this requirement generically by forwarding all application traffic through a well-connected intermediary, more efficient solutions are often available to this special case in the form of NAT traversal techniques.

Hole punching is a general-purpose NAT traversal technique that does not require any explicit communication with the NATs involved. As long as the NATs involved meet certain behavioral requirements, hole punching works consistently and robustly for both TCP and UDP communication, and can be implemented by ordinary applications with no special privileges or specific network topology information. Hole punching fully preserves the transparency that is one of the most important hallmarks and attractions of NAT, and works even with multiple levels of NAT—though certain corner case situations require hairpin translation, a NAT feature not yet widely implemented. Now that many of the "P2P-friendly NAT behaviors" described in this thesis including hairpin translation are being standardized as part of the IETF's BEHAVE working group [118], we might hope that the percentage of NATs that support hole punching effectively will grow in the future.

9.6 Possible Barriers to Adoption

An important pragmatic consideration for any significantly new networking architecture such as UIA is what barriers to adoption the new architecture is likely encounter in any attempt at large-scale deployment. While we make no attempt in this thesis to explore this complex issue exhaustively, it is worth briefly enumerating some of the most likely deployment challenges. These challenges may be organized into three categories: technical, economic, and political.

9.6.1 Technical Barriers

There are of course many technical challenges to migrating existing operating systems and Internet protocols to support a new networking architecture like UIA, some of which have been discussed already throughout this thesis. In particular, the fact that UIA affects multiple architectural layers means that various parts of the network protocol stack ideally need to be changed to effect a clean and complete migration, and such changes in practice need to be incrementally deployable one device (or even one application) at a time without breaking interoperability with existing, "legacy" systems, in order to see widespread use in the real world. The current UIA prototype takes considerable effort to address these incremental deployment challenges, but additional issues remain.

Operating System Modifications: The current UIA naming and routing prototype is designed to be installed at system level and operate transparently to applications, but deploying operating system extensions at least require the user to have administrative privilege. Many users in practice do not have administrative privileges on at least some of the machines they use regularly, however, such as their work PCs. An implementation of the UIA naming and routing layers that can be linked into and distributed with particular applications, operating entirely in user space like the prototype SST transport library already does, could help address this barrier to adoption, but would bring with it additional technical challenges such as making multiple UIA-enabled applications in one system cooperate coherently with each other.

Heterogeneity of Devices and Operating Systems: In pursuing the goal of portability across various operating systems, the current UIA naming and routing prototype is not as well-integrated into any particular operating system as it needs to be for ease of installation and use by nontechnical users, but cleanly integrating UIA into each of the many popular operating systems used in modern personal devices—especially small mobile devices such as smart phones and Internet tablets, which run a wide variety of embedded operating systems—is likely to be a major engineering challenge requiring the support of the vendors of those devices.

Prevalence of Firewalls and NATs: While UIA's overlay routing layer is designed to support peer-to-peer communication between personal devices located behind firewalls or NATs, using traffic forwarding or NAT traversal techniques as appropriate, the routing layer still depends on the existence of *some* UIA-enabled devices that are not behind NATs or highly restrictive firewalls, in order to provide those forwarding or NAT traversal rendezvous services. If the prevalence of NATs and firewalls continues to increase, however, there may not be enough non-NATted UIA devices within a user's nearby social neighborhood to provide these forwarding or NAT traversal services to the other devices, forcing UIA (and all other networking architectures and applications oriented around peer-to-peer communication) into increasing reliance on centralized servers to provide these forwarding and NAT traversal functions. Such a development would not prevent the deployment of UIA—the routing layer's "default peer" mechanism (Section 3.2.1) could make use of central forwarding/rendezvous services provided by a device manufacturer or software vendor, for example, or a user's accounts in UIA-aware online services could act as "virtual UIA devices" providing forwarding and rendezvous for the user's "real" devices—but a high effective reliance on central servers due to the scarcity of "first-class" UIA devices may undermine the fault tolerance and disconnected operation benefits of UIA's fully decentralized design.

9.6.2 Economic Barriers

While surmounting the above technical barriers is probably possible, the amount of work it will require implies that such a project would eventually have to receive support and funding from a variety of market players, such as the manufacturers of personal devices, and the vendors of the operating systems and applications that run on them. On the positive side, there is clearly a lot of hardware and software vendor interest in improving the power and capabilities of the networking features in the personal devices they sell, including peer-to-peer naming and communication features, as evidenced by commercial initiatives such as Apple's MobileMe service [13] and Microsoft's Live Mesh [160].

On the negative side, however (and equally evidenced by the above examples), market incentives often tend to favor "vendor-specific walled garden" approaches, in which devices manufactured by or running software written by one vendor work well together but do not easily interoperate with those from other vendors. To be maximally useful, a personal device connectivity architecture like UIA will ultimately require support across a wide variety of devices and operating systems. Achieving such interoperability will in turn require standards development activity supported by many device and software vendors in the industry, and the willingness of vendors to engage in such standards activity may in turn hinge on the existence of a strong and vocal *demand* from users for standards-based, cross-device, vendor-neutral personal device networking.

Another possible economic roadblock to the widespread deployment of a decentralized architecture like UIA is the common preference of device and software vendors for relying on centrally managed services located in data centers and under full control of the vendor. From the vendor's perspective, such services have two important advantages over decentralized services running partly on customers' devices. First, the vendor always has full control of the data center servers running the centralized service, and can for example roll out software upgrades all at once without having to worry about maintaining backward compatibility with an arbitrary number of prior software versions that may still be running somewhere on some customer's device. Second, a centralized design gives vendors greater access to their customers' information and activities, which they often like to leverage for targeted advertising and other purposes. This latter advantage from vendors' perspective of course may create secrecy and privacy issues from the customers' perspective, so customers that are particularly concerned with privacy may prefer more a decentralized connectivity architecture like UIA.

9.6.3 Political Barriers

Finally, associated with the above "centralized versus decentralized" tension is the ongoing political debate on the privacy rights of individuals versus the desire of governments to monitor personal communications for law enforcement purposes. UIA's architecture is based on the idea that an individual should be able to use his personal devices to communicate freely with each other, and with the devices of his friends and associates, via secure peer-to-peer communication channels that are cryptographically authenticated and privacy protected to prevent eavesdropping or interference from third parties or intermediaries within the network.

Although Western democratic systems have in theory generally upheld individuals' rights to engage in private communication of this form, until a few years ago strong encryption technologies were export-controlled in the US [125] and even banned entirely in France [195]. While the laws pertaining to use of strong encryption were liberalized substantially due to their usefulness for e-commerce, and this liberalization did not specifically distinguish between "peer-to-peer" and "customer-vendor" communication, there is no fundamental reason legal regimes could *not* distinguish between these different uses. It is not clear that there is the same level of economic power backing the preservation of the right to private peer-to-peer communication as there is backing the right to private customer-vendor communication: in fact the long series of severe anti-piracy laws passed in the last decade in the US and elsewhere under the backing of large media companies, as part of their war against online file sharing, suggests that the opposite is likely to be true. Thus, although the private peer-to-peer communication capabilities provided by a connectivity architecture like UIA are theoretically neutral to any particular (legal or illegal) use, the desires of both governments and large economic powers to monitor peer-to-peer communications for their own respective purposes could ultimately represent a major political barrier to the widespread use and deployment of an architecture like UIA in the long term.

9.7 Future Work

Although the results of this thesis suggest that the Unmanaged Internet Architecture provides a promising approach to managing and globally connecting personal devices, many avenues for future work remain, some of which are briefly highlighted in this section.

9.7.1 Naming

The UIA naming system currently operates only at the granularity of personal groups and devices, but for many applications what the user cares about directly is not devices but logical *distributed applications* or *services* [20]. If Alice makes a voice call to Bob, for example, she might prefer her voice application to figure out automatically which of Bob's devices he is currently reachable at, as in the Mobile People Architecture [150] or Universal Inbox [190], instead of her having to name a specific device as the target of the call. In other words, Alice wants to invoke Bob's logical voice communication *service* rather than a particular device, leaving the service itself to manage its own distributed operation across Bob's personal group. While it should already be possible to build such distributed services atop the current UIA mechanisms as a separate layer, exploring the details of how such services would operate and how they might be most cleanly integrated with the UIA naming paradigm represents a compelling direction for future research.

UIA's personal naming system is designed to coexist with DNS and support the concurrent use of UIA's personal names and DNS's global names, but UIA and DNS currently use different name resolution mechanisms: while DNS uses a client/server resolution protocol with limited caching, UIA relies on optimistic replication of name records. It may be, however, that a single name management and resolution protocol supporting a *continuum* of state replication policies, between completely uncached and completely replicated, could best serve the requirements of both global and personal naming. Developing a clean, next-generation "unified naming protocol" of this kind that supports both personal and global names effectively, while permitting a smooth transition from the legacy DNS protocols, may require considerable additional effort. Such a naming protocol might be able to support both UIA-style personal names and DNS-style global names at once, however, which would both reduce barriers to UIA's adoption, since the change isn't "just for personal names" anymore, and provide the responsiveness benefits of UIA's gossip-based namespace synchronization model to the resolution of global names as well.

UIA currently provides no read access control for its namespaces, only write access control via group ownership. Users may wish to hide certain names, such as links to business partners, from view of the general public, or limit visibility of devices at work to business colleagues while allowing family members to see devices at home. One difficulty in providing read access control in UIA's highly decentralized environment is that devices appear to require read access to a group merely in order to determine who owns it for access control purposes, which may effectively limit the usefulness of read access control. In order for Alice's PC to authenticate a change record Bob writes on his laptop to their shared `PhotoClub` group, for example, Alice's PC (and any other device that wishes to read this group) must be able to read Bob's personal group to verify that Bob's laptop is indeed an indirect owner allowed to write to `PhotoClub`.

The UIA naming daemon currently assumes that groups are small and change infrequently, so that it is reasonable for devices always to gossip entire groups and store change records forever (or until the device is replaced or the user's account wiped). The caching-based remote name resolution protocol described in Section 2.3.6, which the current prototype does not yet implement, should complement the current prototype's gossip mechanism by allowing devices to resolve names in large or rarely accessed groups held on other devices without replicating the entire group. Using caching instead of replication for some groups comes at some cost in availability, of course, since it is no longer guaranteed that the required name records will be available if all the owners of a remote group are unreachable.

The current design also may not yet adequately address the gradual state buildup of name records over long periods of time. A potentially useful modification to the current design would be for a UIA device to keep a separate log for each group or series, and garbage collect logs of groups the device does not own and has not accessed recently. A state checkpoint mechanism might similarly enable devices to garbage collect old change records for groups they own.

UIA currently assumes that groups are owned by one person or a few people managing the group by consensus: any group owner can modify the group without restriction. Users may wish to configure groups so that changes require approval from multiple distinct owners, or to make some owners more "trusted" than others. Treating a PC locked up in a data center as more trustworthy than a laptop or cell phone could eliminate the risk of ownership disputes if the mobile device is stolen, for example. The user would have to think ahead and perform more manual configuration, however, and the consequences might be worse if the trusted PC is compromised.

As pointed out in Section 2.3.3, UIA's current naming and revocation model can handle the immediate re-keying of a device due to key compromise or algorithm obsolescence, but to support smooth, gradual key or algorithm retirement it may be useful to allow a device to have multiple EIDs at once during some "key transition period" after introducing a new EID but before revoking the old one. There may be other reasons as well to allow a device to have multiple EIDs at once:

so that the device can maintain a collection of keys with a diversity of cryptographic algorithms or key strengths, for example, allowing the user to pick an algorithm or key strength appropriate to the security requirements of a particular application or communication session. The current naming system would interpret the presence of two device name records referring to different EIDs as a name conflict, and thus would need to be modified to support multiple EIDs per device. One way to support this feature would be to allow the "target" field of a device name record to contain an *EID set* instead of just one EID. Another approach would be to re-apply UIA's personal group management algorithm at the "intra-device" level, so that a "personal device" is no longer an atomic entity but is itself treated as a "group of keys," and keys can be added or revoked according to mechanisms analogous to those UIA currently uses to add or revoke devices in the user's personal group.

As an alternative to the digital signature algorithm with which UIA normally signs namespace change records, we are experimenting with incorporating Alpaca [144], a security framework based on proof-carrying authentication [10], into the UIA personal naming model. In Alpaca, instead of a signature, a change record contains a structured *proof* that the record's meaning (e.g., "resolve name x to EID y") has been endorsed by the group owner. Proof-carrying authentication enables new types of proofs to be created and deployed without changing the verifier's code. We have used this mechanism to create a UIA group whose records are certified by MIT's central X.509 certification authority (CA), so that `alice.mit` securely maps to the person the MIT CA has endorsed as `alice@mit.edu` even though UIA contains no explicit code to check X.509 certificates. Since UIA's goal is to provide a fully decentralized network architecture, proof-carrying authentication furthers that goal by enabling decentralized development of new infrastructure features.

9.7.2 Routing

The routing schemes explored in this thesis indicate several alternative approaches to implementing the routing layer of the architecture, none of them can be considered the last word in routing for UIA, and determining the best routing protocol design and developing it into a solid, widely deployable implementation remains for future work. It is possible that the best routing layer design may result from a combination of ideas from the routing schemes explored here: for example, leveraging the naming layer's social network for security and efficiently as in the current social routing scheme, but doing so in the context of a larger scalable distributed structure like those explored in the Identity Hash Routing and Compact Routing schemes.

Social Routing Improvements

Apart from major new algorithmic work on UIA's routing algorithms, the routing layer's search algorithm could be improved incrementally in many ways, such as by using additional hints from the naming layer to improve its performance. To locate `laptop.Charlie.Bob.Alice`, for example, the router might first locate some device belonging to `Bob` and ask that device to locate `laptop.Charlie`. Also, the social routing protocol does not yet, but should, implement the NAT traversal optimizations described in Chapter 5.

Forwarding Protocol Limitations

The routing layer currently uses SSL over TCP for all UIA connectivity, including for tunneling the UDP datagrams of legacy applications. Although the prototype router implementation benefitted in the short term from the maturity and portability of SSL, the downside of using SSL over TCP is that it unnecessarily serializes all upper-level protocol traffic into one stream, creating head-of-line blocking problems in which one dropped packet delays all traffic (logically related or not) behind it in the SSL stream. This serialization does not cause significant problems for many applications such as HTTP [78] and SSH [172], but currently prevents the UIA prototype from being usable for delay- or jitter-sensitive applications such as audio/video conferencing. The best solution would be to use the new structured stream transport described in Chapter 4 as UIA's secure forwarding substrate for the overlay routing layer, and not just as a transport protocol to support applications. Another alternative would be to use DTLS [200], an adaptation of SSL for UDP; or to use IPsec [130–132] *above* the overlay routing layer.

When forwarding traffic across multiple UIA devices, the routing layer currently encrypts and authenticates traffic both hop-by-hop and end-to-end, which is inefficient but provides strong security. We intend to make hop-by-hop protection optional in the future.

Legacy Application Support

UIA's legacy application interface currently cannot provide each user of a multi-user machine with a fully separate TCP/UDP port space for its own EID, because the kernel's protocol stack offers no way to ensure that only a particular user's applications can bind a socket to the device-local IP address representing that user's EID. Thus, without enhancing the kernel's transport protocols, only UIA-aware applications can make full use of personal EIDs. Fixing this issue requires changes to kernel-level code and is thus less portable.

9.7.3 Transport

Some optional extensions to the SST wire protocol presented in this thesis may be desirable for certain purposes, which could be negotiated at channel setup time. A payload chunking similar to SCTP's could allow transmitters to bundle several small stream-level segments into one packet for more efficient use of the network, in media stream trunking scenarios for example. A payload padding option as in IPsec's ESP could help protect against traffic analysis. Some congestion control algorithms requiring cooperation between sender and receiver may similarly be negotiated at channel setup time.

The field sizes in the SST header format defined allow a host to send up to about 2^{22} packets in one round trip time, to create about 2^{14} new streams per round trip to send about 2^{16} bytes of data immediately on a new stream before receiving the responder's acknowledgment, and to send about 2^{30} bytes of data per round trip on an established stream (the same as TCP's maximum window size). While these limits should be ample enough for common purposes, extremely high-bandwidth, high-latency networks might benefit from a negotiable extended header format with wider fields to support larger windows.

SST's stream hierarchy provides an obvious potential affinity with hierarchical scheduling algorithms such as Hierarchical Fair Queuing (HFQ) [111], Hierarchical Packet Fair Queuing (H-PFQ) [28], and Lottery Scheduling [257], suggesting a simple, powerful, and flexible method of ensuring fairness among an application's concurrent communication activities. The naive approach of matching the scheduling hierarchy exactly to the hereditary stream structure may be inadequate, however, because an activity that was started for one reason might be continued for other reasons. In a tabbed web browser that concurrently downloads several web pages, for example, a low-priority web page might commence downloading a certain image that, as the web browser only later discovers, a high-priority web page also needs. If the image download stream's scheduling priority is based only on its original context as a child of the low-priority web page, but the high-priority web page needs to the image in order to complete, then a classic priority inversion scenario results. Thus, a mechanism may be needed to allow scheduling contexts to be dynamically "transferred" within the hierarchy, or better yet, to allow a stream to be "multihomed" and given several parent streams for scheduling purposes.

Finally, it is easy to envision SST's stream attachment, detachment, and migration functionality being extended to support features such as streams that can persist across long-term disconnection [223] or reboots of either endpoint, or streams whose endpoints can migrate from one endpoint to another, reminiscent of distributed capabilities [8, 66, 193, 241].

9.7.4 Other Features

While UIA focused on naming, routing, and transport, there are many other functional areas that bear reconsideration in terms of the requirements of modern personal devices, especially under the additional challenging demands of UIA's philosophy of full decentralization. Some important topics that the current UIA design currently leaves untouched include:

- How to replicate and synchronize not just namespace state but also application-level data across personal devices in a fully decentralized model like UIA's.

- How to support the naming not only of devices and groups or "users" but also *services* effectively, where a logical service might be distributed across several devices in an application-specific fashion.

- How best to allow new applications to interact with UIA for purposes of making security decisions, such as supporting access control policies using UIA names, and the potential security risks of doing so.

Inevitably, UIA can be said to have barely scratched the surface.

Bibliography

[1] ABI Research. Wi-Fi hotspots stay hot in 2008, July 2008.

[2] Ittai Abraham et al. Compact name-independent routing with minimum stretch. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2004.

[3] Yehuda Afek, Eli Gafni, and Moty Ricklin. Upper and lower bounds for routing schemes in dynamic networks. In *30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 370–375, October 1989.

[4] William Aiello et al. Just fast keying: Key agreement in a hostile Internet. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):1–32, May 2004.

[5] S. Alexander and R. Droms. DHCP options and BOOTP vendor extensions, March 1997. RFC 2132.

[6] M. Allman, V. Paxson, and W. Stevens. TCP congestion control, April 1999. RFC 2581.

[7] Mark Allman, Chris Hayes, Hans Kruse, and Shawn Ostermann. TCP performance over satellite links. In *5th International Conference on Telecommunication Systems*, March 1997.

[8] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[9] David G. Andersen et al. Resilient overlay networks. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[10] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[11] Apple Computer, Inc. Bonjour. `http://developer.apple.com/networking/bonjour/`.

[12] Apple Computer, Inc. FileVault. `http://www.apple.com/macosx/features/filevault/`.

[13] Apple Computer, Inc. MobileMe. `http://www.apple.com/mobileme/`.

[14] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements, March 2005. RFC 4033.

[15] Marta Arias et al. Compact routing with name independence. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2003.

[16] F. Audet, ed. and C. Jennings. Network address translation (NAT) behavioral requirements for unicast UDP, January 2007. RFC 4787.

[17] Brikam S. Bakshi, P. Krishna, N. H. Vaidya, and D. K. Pradhan. Improving the performance of TCP over wireless networks. In *17th International Conference on Distributed Computer Systems (ICDCS)*, May 1997.

[18] Hari Balakrishnan et al. TCP behavior of a busy Internet server: Analysis and improvements. In *IEEE INFOCOM*, March 1998.

[19] Hari Balakrishnan et al. Looking up data in P2P systems. *Communications of the ACM*, February 2003.

[20] Hari Balakrishnan, Karthik Lakshminarayanan, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Michael Walfish. A layered naming architecture for the internet. In *ACM SIGCOMM*, September 2004.

[21] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE Transactions on Networking*, 5(6), December 1997.

[22] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for Internet hosts. In *ACM SIGCOMM*, September 1999.

[23] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP performance over wireless networks. In *1st International Conference on Mobile Computing and Networking (MOBICOM)*, November 1995.

[24] Hari Balakrishnan, Scott Shenker, and Michael Walfish. Semantic-free referencing in linked distributed systems. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2003.

[25] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Lecture Notes in Computer Science*, 1976:531–545, September 2000.

[26] S. Bellovin. Defending against sequence number attacks, May 1996. RFC 1948.

[27] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. Packet reordering is not pathological network behavior. *Transactions on Networking*, 7:789–798, December 1999.

[28] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In *ACM SIGCOMM*, pages 143–156, August 1996.

[29] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol — HTTP/1.0, May 1996. RFC 1945.

[30] Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, and Adrian Perrig. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *ACM SIGCOMM Asia Workshop*, April 2005.

[31] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *Transactions on Computer Systems*, 2(1):39–59, February 1984.

[32] E. Blanton and M. Allman. On making TCP more robust to packet reordering. *Computer Communications Review*, 32(1), January 2002.

[33] E. Blanton and M. Allman. Using TCP duplicate selective acknowledgement (DSACKs) and stream control transmission protocol (SCTP) duplicate transmission sequence numbers (TSNs) to detect spurious retransmissions, February 2004. RFC 3708.

[34] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[35] R. Braden. Towards a transport service for transaction processing applications, September 1985. RFC 955.

[36] R. Braden. T/TCP – TCP extensions for transactions, July 1994. RFC 1644.

[37] Matthew Caesar et al. ROFL: Routing on flat labels. In *ACM SIGCOMM*, September 2006.

[38] CAIDA. Macroscopic topology AS adjacencies. `http://www.caida.org/tools/measurement/skitter/as_adjacencies.xml`.

[39] I. Castineyra, N. Chiappa, and M. Steenstrup. The Nimrod routing architecture, August 1996. RFC 1992.

[40] Miguel Castro et al. Secure routing for structured peer-to-peer overlay networks. In *5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[41] Rory Cellan-Jones. Broadband — are you mobile or fixed? *BBC News dot.life*, August 2008.

[42] Yatin Chawathe et al. Making Gnutella-like P2P systems scalable. In *ACM SIGCOMM*, pages 407–418, August 2003.

[43] Benjie Chen and Robert Morris. L+: Scalable landmark routing and address lookup for multi-hop wireless networks. Technical Report 837, Massachusetts Institute of Technology Laboratory for Computer Science, March 2002.

[44] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free Bellman-Ford routing protocol without bouncing effect. In *ACM SIGCOMM*, pages 224–237, September 1989.

[45] Jacqui Cheng. .confusion: ICANN opens up Pandora's Box of new TLDs. *Ars Technica*, June 2008.

[46] David R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. *Computer Communications Review*, 16(3):406–415, August 1986.

[47] David R. Cheriton and Mark Gritter. TRIAD: A new next-generation Internet architecture, July 2000.

[48] Stuart Cheshire, Marc Krochmal, and Kiren Sekar. NAT port mapping protocol, June 2005. Internet-Draft (Work in Progress).

[49] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security*. Addison-Wesley, February 2003.

[50] J. Noel Chiappa. Endpoints and endpoint names: A proposed enhancement to the internet architecture, 1999. Internet-Draft (Work in Progress).

[51] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM*, pages 200–208, 1990.

[52] David Clark, Robert Braden, Aaron Falk, and Venkata Pingali. FARA: Reorganizing the addressing architecture. In *ACM SIGCOMM FDNA Workshop*, August 2003.

[53] David D. Clark. Window and acknowledgement strategy in TCP, July 1982. RFC 813.

[54] David D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, August 1988.

[55] Bram Cohen. Incentives build robustness in BitTorrent. In *1st Workshop on Economics of Peer-to-Peer Systems (p2pecon)*, June 2003.

[56] Lenore J. Cowen. Compact routing with minimum stretch. In *10th Symposium on Discrete Algorithms (SODA)*, pages 255–260, Philadelphia, PA, 1999. Society for Industrial and Applied Mathematics.

[57] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using Chord. In *1st International Workshop on Peer-to-Peer Systems*, March 2002.

[58] M. Crispin. Internet message access protocol - version 4rev1, March 2003. RFC 3501.

[59] Yogen K. Dalal. More on selecting sequence numbers. *SIGOPS Operating Systems Review*, 9(3):25–36, July 1975.

[60] G. Danezis, C. Lesniewski-Laas, F. Kaashoek, and R. Anderson. Sybil-resistant DHT routing. In *ESORICS*, 2005.

[61] A. Datta, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *23rd International Conference on Distributed Computing Systems*, 2003.

[62] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification, December 1998. RFC 2460.

[63] Alan Demers et al. Epidemic algorithms for replicated database maintenance. In *6th ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.

[64] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. RFC 2246.

[65] Steve Dohrmann and Carl Ellison. Public-key support for collaborative groups. In *1st Annual PKI Research Workshop*, April 2002.

[66] James E. (Jed) Donnelley. Managing domains in a network operating system. In *Local Networks and Distributed Office Systems Conference*, pages 345–361, May 1981.

[67] John R. Douceur. The Sybil attack. In *1st International Workshop on Peer-to-Peer Systems*, March 2002.

[68] R. Droms. Dynamic host configuration protocol, March 1997. RFC 2131.

[69] Morris Dworkin. Recommendation for block cipher modes of operation, December 2001. NIST Special Publication 800-38A.

[70] K. Egevang and P. Francis. The IP network address translator (NAT), May 1994. RFC 1631.

[71] C. Ellison et al. SPKI Certificate Theory, 1999. RFC 2693.

[72] Jeffrey L. Eppinger. TCP connections for P2P apps: A software approach to solving the NAT problem. Technical Report CMU-ISRI-05-104, Carnegie Mellon University, January 2005.

[73] Jakob Eriksson, Michalis Faloutsos, and Srikanth Krishnamurthy. PeerNet: Pushing peer-to-peer down the stack. In *2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[74] European Telecommunications Standards Institute. User identification solutions in converging networks, April 2001.

[75] Theodore Faber, Joe Touch, and Wei Yue. The `TIME-WAIT` state in TCP and its effects on busy servers. In *IEEE INFOCOM*, volume 3, pages 1573–1583, March 1999.

[76] Facebook. `http://www.facebook.com/`.

[77] C. Feather. Network news transfer protocol (NNTP), October 2006. RFC 3977.

[78] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, June 1999. RFC 2616.

[79] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (SACK) option for TCP, July 2000. RFC 2883.

[80] Bryan Ford. Scalable Internet routing on topology-independent node identities. Technical Report MIT-LCS-TR-926, MIT Laboratory for Computer Science, October 2003.

[81] Bryan Ford. Unmanaged Internet protocol: Taming the edge network management crisis. In *2nd Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[82] Bryan Ford. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, April 2005.

[83] Bryan Ford. Structured streams: a new transport abstraction. In *ACM SIGCOMM*, August 2007.

[84] Bryan Ford et al. Persistent personal names for globally connected mobile devices. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[85] Bryan Ford et al. User-Relative Names for Globally Connected Personal Devices. In *5th International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2006.

[86] Bryan Ford and Janardhan Iyengar. Breaking up the transport logjam. In *7th Workshop on Hot Topics in Networks (HotNets-VII)*, October 2008.

[87] Paul Francis and Ramakrishna Gummadi. IPNL: A NAT-extended Internet architecture. In *ACM SIGCOMM*, August 2002.

[88] N. Freed. Behavior of and requirements for Internet firewalls, October 2000. RFC 2979.

[89] Michael J. Freedman et al. Non-transitive connectivity and DHTs. In *USENIX WORLDS 2005*, December 2005.

[90] Friendster. `http://www.friendster.com/`.

[91] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, September 1993. RFC 1519.

[92] Cyril Gavoille and Marc Gengler. Space-efficiency for routing schemes of stretch factor three. *Journal of Parallel and Distributed Computing*, 61(5):679–687, 2001.

[93] Mario Gerla, Xiaoyan Hong, and Guangyu Pei. Landmark routing for large ad hoc wireless networks. In *IEEE GLOBECOM*, November 2000.

[94] Jim Gettys. Simple MUX protocol specification, October 1996. W3C Working Draft.

[95] B. Gleeson et al. A Framework for IP Based Virtual Private Networks, February 2000. RFC 2764.

[96] Li Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, May 2001.

[97] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *12th SOSP*, 1989.

[98] Saikat Guha and Paul Francis. Simple traversal of UDP through NATs and TCP too (STUNT). `http://nutss.gforge.cis.cornell.edu/`.

[99] Saikat Guha and Paul Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *Internet Measurement Conference (IMC)*, October 2005.

[100] Saikat Guha, Yutaka Takeday, and Paul Francis. NUTSS: A SIP-based approach to UDP and TCP network connectivity. In *SIGCOMM 2004 Workshops*, August 2004.

[101] S. Guha, Ed., K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT behavioral requirements for TCP, April 2007. Internet-Draft (Work in Progress).

[102] V. Gurbani and S. Lawrence. Handling large user datagram protocol (UDP) responses in the session initiation protocol (SIP), October 2006. Internet-Draft (Work in Progress).

[103] R. G. Guy, G. J. Popek, and T. W. Page, Jr. Consistency algorithms for optimisic replication. In *First International Conference on Network Protocols*, 1993.

[104] Richard Guy et al. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.

[105] Richard G. Guy et al. Implementation of the Ficus replicated file system. In *USENIX Summer Conference*, pages 63–71, June 1990.

[106] J. Hagino and K. Yamamoto. An IPv6-to-IPv4 transport relay translator, June 2001. RFC 3142.

[107] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP friendly rate control (TFRC): Protocol specification, January 2003. RFC 3448.

[108] Chris Hare and Karanjit Siyan. *Internet Firewalls and Network Security*. New Riders, 1996.

[109] C. Hedrick. Routing information protocol, June 1988. RFC 1058.

[110] R. Hinden and B. Haberman. Unique local IPv6 unicast addresses, October 2005. RFC 4193.

[111] David Hogan. Hierarchical fair queuing. Technical Report 506, University of Sydney, May 1996.

[112] M. Holdrege and P. Srisuresh. Protocol complications with the IP network address translator, January 2001. RFC 3027.

[113] M. Horowitz and S. Lunt. FTP security extensions, October 1997. RFC 2228.

[114] M. Horton and R. Adams. Standard for interchange of USENET messages, December 1987. RFC 1036.

[115] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, April 2002. RFC 3280.

[116] C. Huitema. Teredo: Tunneling IPv6 over UDP through NATs, March 2004. Internet-Draft (Work in Progress).

[117] ICANN. Biggest expansion in gTLDs approved for implementation, June 2008. `http://www.icann.org/en/announcements/announcement-4-26jun08-en.htm`.

[118] Internet Engineering Task Force (IETF). Behavior engineering for hindrance avoidance (behave). `http://www.ietf.org/html.charters/behave-charter.html`.

[119] The Internet traffic archive. `http://ita.ee.lbl.gov/`.

[120] Internet protocol, September 1981. RFC 791.

[121] Janardhan R. Iyengar, Paul D. Amer, and Randall Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *Transactions on Networking*, 14(5):951–964, October 2006.

[122] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance, May 1992. RFC 1323.

[123] C. Jennings. NAT classification results using STUN, October 2004. Internet-Draft (Work in Progress).

[124] David B. Johnson. Routing in ad hoc networks of mobile hosts. In *IEEE Workshop on Mobile Computing Systems and Applications*, pages 158–163, December 1994.

[125] Margret Johnston. U.S. relaxes encryption export policy. *PC World*, January 2000.

[126] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton N.J., 1962.

[127] D. N. Kalofonos, Z. Antoniou, F. D. Reynolds, M. Van-Kleek, J. Strauss, and P. Wisner. MyNet: a platform for secure P2P personal and social networking services. In *6th Annual IEEE Conference on Pervasive Computing and Communications (PerCom2008)*, March 2008.

[128] B. Kantor. BSD Rlogin, September 1991. RFC 1258.

[129] Dan Kegel. NAT and peer-to-peer networking, July 1999. `http://www.alumni.caltech.edu/~dank/peer-nat.html`.

[130] S. Kent. IP authentication header, December 2005. RFC 4302.

[131] S. Kent. IP encapsulating security payload (ESP), December 2005. RFC 4303.

[132] S. Kent and K. Seo. Security architecture for the Internet protocol, December 2005. RFC 4301.

[133] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 213–225, 1991.

[134] Leonard Kleinrock and Farouk Kamoun. Hierarchical routing for large networks: Performance evaluation and optimization. *Computer Networks*, 1(3):155–174, 1977.

[135] J. Klensin, ed. Simple mail transfer protocol, April 2001. RFC 2821.

[136] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), March 2006. RFC 4340.

[137] Eddie Kohler, Mark Handley, and Sally Floyd. Designing DCCP: Congestion control without reliability. In *ACM SIGCOMM*, 2006.

[138] Amos Korman and David Peleg. Dynamic routing schemes for general graphs. In *33rd International Colloquium on Automata, Languages and Programming (ICALP)*, July 2006.

[139] Dmitri Krioukov, Kevin Fall, and Xiaowei Yang. Compact routing on internet-like graphs. In *IEEE INFOCOM*, 2004.

[140] Dmitri Krioukov, kc claffy, Kevin Fall, and Arthur Brady. On compact routing for the Internet. *SIGCOMM Computer Communications Review*, 37(3):41–52, 2007.

[141] Venkat Kudallur et al. IE7 networking improvements in content caching and decompression. *IEBlog*, October 2005.

[142] L-A. Larzon, M. Degermark, S. Pink, L-E. Jonsson, Ed., and G. Fairhurst, Ed. The lightweight user datagram protocol (UDP-Lite), July 2004. RFC 3828.

[143] M. Leech et al. SOCKS protocol, March 1996. RFC 1928.

[144] Christopher Lesniewski-Laas, Bryan Ford, Jacob Strauss, M. Frans Kaashoek, and Robert Morris. Alpaca: extensible authorization for distributed services. In *ACM Computer and Communications Security*, October 2007.

[145] Jinyang Li and Frank Dabek. F2F: Reliable storage in open networks. In *5th IPTPS*, February 2006.

[146] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, May 2005.

[147] Lun Li, David Alderson, Walter Willinger, and John Doyle. A first-principles approach to understanding the Internet's router-level topology. In *ACM SIGCOMM*, August 2004.

[148] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

[149] Petros Maniatis and Mary Baker. A historic name-trail service. In *5th Workshop on Mobile Computing Systems and Applications*, October 2003.

[150] Petros Maniatis et al. The mobile people architecture. *Mobile Computing and Communications Review*, 3(3):36–42, July 1999.

[151] Duncan Martell. Atom sales help Intel expand beyond PC market. *NewsFactor Network*, August 2008.

[152] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. DHT routing using social links. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2004.

[153] Sergio Marti, Prasanna Ganesan, and Hector Garcia-Molina. SPROUT: P2P routing with social networks. In *1st International Workshop on Peer-to-Peer Computing and Databases (P2P&DB)*, March 2004.

[154] M. Mathis, J. Mahdav, S. Floyd, and A. Romanow. TCP selective acknowledgment options, October 1996. RFC 2018.

[155] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *ACM SIGCOMM*, August 1996.

[156] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[157] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *17th Symposium on Operating System Principles*, December 1999.

[158] P. M. Merlin and A. Segall. A failsafe distributed routing protocol. *IEEE Transactions on Communications*, COM-27(9):1280–1287, September 1979.

[159] D. Meyer, Ed., L. Zhang, Ed., and K. Fall, Ed. Report from the IAB workshop on routing and addressing, September 2007. RFC 4984.

[160] Microsoft Corporation. Live Mesh tech preview. `https://www.mesh.com/`.

[161] P. Mockapetris. Domain names: concepts and facilities, November 1987. RFC 1034.

[162] P. Mockapetris. Domain names: implementation and specification, November 1987. RFC 1035.

[163] R. Moskowitz and P. Nikander. Host identity protocol (HIP) architecture, May 2006. RFC 4423.

[164] J. Moy. OSPF version 2, July 1991. RFC 1247.

[165] Mozilla.org. Firefox tips & tricks: Pipelining. http://www.mozilla.org/support/firefox/tips#oth_pipelining.

[166] Shree Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mobile Networks and Applications*, 1(2):183–197, 1996.

[167] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[168] J. Myers and M. Rose. Post office protocol — version 3, May 1996. RFC 1939.

[169] NAT check. http://midcom-p2p.sourceforge.net/.

[170] D. Newman. Benchmarking terminology for firewall performance, August 1999. RFC 2647.

[171] H. F. Nielsen et al. Network performance effects of HTTP/1.1, CSS1, and PNG, June 1997. W3C NOTE-pipelining-970624.

[172] The OpenSSH project. http://www.openssh.org/.

[173] The OpenSSL project. http://www.openssl.org/.

[174] Orkut. http://www.orkut.com/.

[175] J. Palme. Common internet message headers, February 1997. RFC 2076.

[176] J.M. Paluska et al. Footloose: A case for physical eventual consistency and selective conflict resolution. In *5th Workshop on Mobile Computing Systems and Applications*, 2003.

[177] C. Partridge and R. Hinden. Version 2 of the reliable data protocol (RDP), April 1990. RFC 1151.

[178] Craig Partridge. Implementing the reliable data protocol (RDP). In *USENIX Summer Conference*, June 1987.

[179] Craig Partridge and Timothy J. Shepard. TCP/IP performance over satellite links. *IEEE Network*, 11(5):44–49, September 1997.

[180] Guangyu Pei, Mario Gerla, and Tsu-Wei Chen. Fisheye state routing: A routing scheme for ad hoc wireless networks. In *IEEE International Conference on Communications (ICC)*, pages 70–74, June 2000.

[181] Charles E. Perkins and Elizabeth M. Belding-Royer. Ad hoc on-demand distance vector routing. In *2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[182] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[183] C. Perkins, Editor. IP mobility support for IPv4, August 2002. RFC 3344.

[184] B. Popescu, B. Crispo, and A. Tanenbaum. Safe and private data sharing with Turtle: Friends team-up and beat the system. In *12th Cambridge Workshop on Security Protocols*, 2004.

[185] J. Postel. User datagram protocol, August 1980. RFC 768.

[186] J. Postel and J. Reynolds. Telnet protocol specification, May 1983. RFC 854.

[187] J. Postel and J. Reynolds. File transfer protocol (FTP), October 1985. RFC 959.

[188] J.A. Pouwelse et al. Tribler: A social-based peer-to-peer system. In *5th International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2006.

[189] The Python programming language. `http://www.python.org/`.

[190] Bhaskaran Raman, Randy H. Katz, and Anthony D. Joseph. Universal Inbox: Providing extensible personal mobility and service mobility in an integrated communication network. In *3rd IEEE Workshop on Mobile Computing Systems and Applications*, December 2000.

[191] R. Ramanathan. Mobility support for Nimrod: Challenges and solution approaches, February 1997. RFC 2103.

[192] Venugopalan Ramasubramanian and Emin Gün Sirer. The design and implementation of a next generation name service for the Internet. In *ACM SIGCOMM*, August 2004.

[193] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *5th ACM Symposium on Operating Systems Principles (SOSP)*, December 1981.

[194] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, August 2001.

[195] Jim Reavis. Trends in government encryption policies. *NetworkWorldFusion*, August 1999.

[196] Y. Rekhter et al. Address allocation for private internets, February 1996. RFC 1918.

[197] Y. Rekhter and T. Li. Implications of various address allocation policies for Internet routing, October 1996. RFC 2008.

[198] Y. Rekhter, T. Li, and S. Hares (editors). A border gateway protocol 4 (BGP-4), January 2006. RFC 4271.

[199] Y. Rekhter and T. Li (editors). An architecture for IP address allocation with CIDR, September 1993. RFC 1518.

[200] E. Rescorla and N. Modadugu. Datagram transport layer security, April 2006. RFC 4347.

[201] P. Resnick, ed. Internet message format, April 2001. RFC 2822.

[202] Matt Richtel. Smaller PCs cause worry for industry. July 2008.

[203] R.L. Rivest and B. Lampson. SDSI: A simple distributed security infrastructure, April 1996. `http://theory.lcs.mit.edu/~cis/sdsi.html`.

[204] M. Rose. The blocks extensible exchange protocol core, March 2001. RFC 3080.

[205] J. Rosenberg. Interactive connectivity establishment (ICE), October 2007. Internet-Draft (Work in Progress).

[206] J. Rosenberg et al. SIP: session initiation protocol, June 2002. RFC 3261.

[207] J. Rosenberg et al. STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs), March 2003. RFC 3489.

[208] J. Rosenberg, C. Huitema, and R. Mahy. Traversal using relay NAT (TURN), October 2003. Internet-Draft (Work in Progress).

[209] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[210] J. Saltzer. On the naming and binding of network destinations, August 1993. RFC 1498.

[211] Nicola Santoro and Ramez Khatib. Labelling and implicit routing in networks. *Computer Journal*, 28(1):5–8, 1985.

[212] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI), April 2004. RFC 3720.

[213] Michael Scharf and Sebastian Kiesel. Head-of-line blocking in TCP and SCTP: Analysis and measurements. In *IEEE GLOBECOM*, November 2006.

[214] H. Schulzrinne et al. RTP: A transport protocol for real-time applications, July 2003. RFC 3550.

[215] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (RTSP), April 1998. RFC 2326.

[216] SecurityInfoWatch.com. Top computer security risks for healthcare. March 2008.

[217] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol, April 2003. RFC 3530.

[218] Site multihoming by IPv6 intermediation (shim6). `http://www.ietf.org/html.charters/shim6-charter.html`.

[219] John F. Shoch. Inter-network naming, addressing, and routing. In *IEEE COMPCON Fall*, pages 72–79, 1978.

[220] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[221] Richard L. Sites. Alpha AXP architecture. *Digital Technical Journal*, 4(4), 1992.

[222] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *6th ACM/IEEE International Conference on Mobile Computing and Networking*, August 2000.

[223] Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek. Reconsidering Internet mobility. In *8th Workshop on Hot Topics in Operating Systems*, May 2001.

[224] M. Spencer et al. IAX2: Inter-Asterisk eXchange version 2, October 2006. Internet-Draft (Work in Progress).

[225] R. Srinivasan. RPC: remote procedure call protocol specification version 2, August 1995. RFC 1831.

[226] P. Srisuresh and K. Egevang. Traditional IP network address translator (Traditional NAT), January 2001. RFC 3022.

[227] P. Srisuresh, B. Ford, and D. Kegel. State of peer-to-peer (P2P) communication across network address translators (NATs), March 2008. RFC 5128.

[228] P. Srisuresh and M. Holdrege. IP network address translator (NAT) terminology and considerations, August 1999. RFC 2663.

[229] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox communication architecture and framework, August 2002. RFC 3303.

[230] Mudhakar Srivatsa and Ling Liu. Vulnerabilities and security threats in structured peer-to-peer systems: A quantitative analysis. In *20th Annual Computer Security Applications Conference (ACSAC)*, December 2004.

[231] Frank Stajano and Ross Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *7th International Workshop on Security Protocols*, April 1999.

[232] R. Stewart et al. Stream control transmission protocol, October 2000. RFC 2960.

[233] R. Stewart, ed. Stream control transmission protocol, September 2007. RFC 4960.

[234] Ion Stoica et al. Chord: A scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM*, August 2001.

[235] Ion Stoica et al. Internet indirection infrastructure. In *ACM SIGCOMM*, August 2002.

[236] Brad Stone. What's in a domain name? Serious money. *International Herald Tribune*, January 2008.

[237] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, June 1997.

[238] Lakshminarayanan Subramanian et al. OverQoS: An overlay based architecture for enhancing Internet QoS. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[239] Carl A. Sunshine and Yogen K. Dalal. Connection management in transport protocols. *Computer Networks*, 2(6):454–473, December 1978.

[240] E. Swierk, E. Kıcıman, V. Laviano, and M. Baker. The Roma personal metadata service. In *3rd IEEE Workshop on Mobile Computing Systems and Applications*, December 2000.

[241] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *6th International Conference on Distributed Computing Systems (ICDCS)*, May 1986.

[242] Transmission control protocol, September 1981. RFC 793.

[243] Douglas B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *15th ACM Symposium on Operating System Principles*, 1995.

[244] The DIMES project. `http://www.netdimes.org/`.

[245] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–10, New York, NY, USA, 2001. ACM Press.

[246] Raymond S. Tomlinson. Selecting sequence numbers. *SIGOPS Operating Systems Review*, 9(3):11–23, July 1975.

[247] J. Touch. TCP control block interdependence, April 1997. RFC 2140.

[248] Trolltech. Qt cross-platform application framework. `http://trolltech.com/products/qt/`.

[249] G. Tsirtsis and P. Srisuresh. Network address translation - protocol translation (NAT-PT), February 2000. RFC 2766.

[250] Paul F. Tsuchiya and Tony Eng. Extending the IP internet through address reuse. *Computer Communications Review*, 23(1):16–33, January 1993.

[251] Paul Francis Tsuchiya. The Landmark hierarchy: A new hierarchy for routing in very large networks. In *ACM SIGCOMM*, pages 35–42, August 1988.

[252] Justin Uberti. E-mail on IETF MIDCOM mailing list, February 2004. Message-ID: ⟨`402CEB11.1060906@aol.com`⟩.

[253] UPnP Forum. Internet gateway device (IGD) standardized device control protocol, November 2001. `http://www.upnp.org/`.

[254] Robert Vamosi. Security watch: Don't get burned by viruses and hackers. September 2006.

[255] David Velten, Robert Hinden, and Jack Sax. Reliable data protocol, July 1984. RFC 908.

[256] P. Vixie, Editor, S. Thomson, Y. Rekhter, and J. Bound. Dynamic updates in the domain name system, April 1997. RFC 2136.

[257] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 1994.

[258] Michael Walfish, Hari Balakrishnan, and Scott Shenker. Untangling the Web from DNS. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[259] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *USENIX Symposium on Operating Systems Design and Implementation*, December 2004.

[260] Mark Weiser. The computer for the 21st century. *Mobile Computing and Communications Review*, 3(3), July 1999.

[261] D. Wing and T. Eckert. IP multicast requirements for a network address translator (NAT) and a network address port translator (NAPT), February 2008. RFC 5135.

[262] Ye Xia and David Tse. Analysis on packet resequencing for reliable network protocols. In *IEEE INFOCOM*, April 2003.

[263] Raj Yavatkar and Namrata Bhagawat. Improving end-to-end performance of TCP over mobile internetworks. In *Workshop on Mobile Computing Systems and Applications*, December 1994.

[264] T. Ylonen and C. Lonvick, Ed. The secure shell protocol architecture, January 2006. RFC 4251.

[265] T. Ylonen and C. Lonvick, Ed. The secure shell (SSH) authentication protocol, January 2006. RFC 4252.

[266] T. Ylonen and C. Lonvick, Ed. The secure shell (SSH) connection protocol, January 2006. RFC 4254.

[267] T. Ylonen and C. Lonvick, Ed. The secure shell (SSH) transport layer protocol, January 2006. RFC 4253.

[268] Haifeng Yu, Phillip B. Gibbons, Michael Kaminsky, and Feng Xiao. SybilLimit: A near-optimal social network defense against sybil attacks. In *IEEE Symposium on Security and Privacy*, May 2008.

[269] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. SybilGuard: Defending against sybil attacks via social networks. In *ACM SIGCOMM*, September 2006.

[270] Hui Zhang and Srinivasan Keshav. Comparison of rate-based service disciplines. In *ACM SIGCOMM*, pages 113–121, 1991.

[271] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.