A Session-Based Architecture for Internet Mobility

by

Mark Alexander Connell Snoeren

S.M. Computer Science, Georgia Institute of Technology (1997)
B.S. Applied Mathematics, Georgia Institute of Technology (1997)
B.S. Computer Science, Georgia Institute of Technology (1996)

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

February 2003

Author . . . .
Department of Electrical Engineering and Computer Science
December 12, 2002

Certified by . . . .
Hari Balakrishnan
Associate Professor of Computer Science and Engineering
Thesis Supervisor

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

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

A Session-Based Architecture for Internet Mobility

by
Mark Alexander Connell Snoeren

Submitted to the Department of Electrical Engineering and Computer Science
on December 12, 2002, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

The proliferation of mobile computing devices and wireless networking products over the past decade has led to an increasingly nomadic computing lifestyle. A computer is no longer an immobile, gargantuan machine that remains in one place for the lifetime of its operation. Today's personal computing devices are portable, and Internet access is becoming ubiquitous. A well-traveled laptop user might use half a dozen different networks throughout the course of a day: a cable modem from home, wide-area wireless on the commute, wired Ethernet at the office, a Bluetooth network in the car, and a wireless, local-area network at the airport or the neighborhood coffee shop.

Mobile hosts are prone to frequent, unexpected disconnections that vary greatly in duration. Despite the prevalence of these multi-homed mobile devices, today's operating systems on both mobile hosts and fixed Internet servers lack fine-grained support for network applications on intermittently connected hosts. We argue that network communication is well-modeled by a session abstraction, and present *Migrate*, an architecture based on system support for a flexible session primitive. *Migrate* works with application-selected naming services to enable seamless, mobile "suspend/resume" operation of legacy applications and provide enhanced functionality for mobile-aware, session-based network applications, enabling adaptive operation of mobile clients and allowing Internet servers to support large numbers of intermittently connected sessions.

We describe our UNIX-based implementation of *Migrate* and show that sessions are a flexible, robust, and efficient way to manage mobile end points, even for legacy applications. In addition, we demonstrate two popular Internet servers that have been extended to leverage our novel notion of *session continuations* to enable support for large numbers of suspended clients with only minimal resource impact. Experimental results show that *Migrate* introduces only minor throughput degradation (less than 2% for moderate block sizes) when used over popular access link technologies, gracefully detects and suspends disconnected sessions, rapidly resumes from suspension, and integrates well with existing applications.

Thesis Supervisor: Hari Balakrishnan
Title: Associate Professor of Computer Science and Engineering

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

To my dad. I read the book.

It takes a village to raise a child.

- African proverb

Acknowledgments

I cannot hope to enumerate, let alone repay, all those to whom I am indebted for assistance not only in preparing this manuscript, but in helping me to survive and prosper during my years at MIT. Despite the unavoidable omissions, I still insist on mentioning a few in print.

First, I thank my advisors, Hari Balakrishnan and Frans Kaashoek, for providing an incredibly exciting and energizing work environment. I could not have asked for more supportive and engaging mentors. Each sports a razor-sharp mind and wit, and an amazing ability to apply both in just the right amount with impeccable timing. I remain astounded by their intuition, thoughtfulness, and capacities for comprehension, synthesis, and presentation. Despite the dozens of other students they advise, I have no doubt they understand the context and ramifications of the material presented here better than I.

John Guttag, the final member of my thesis committee, was always ready with interested and enthusiastic counsel. While I fell victim to the all-too-common temptation to not solicit his advice on this dissertation until far later than I should have, his career guidance and succinct comments on countless practice talks throughout my graduate career have been invaluable.

The implementation described in this dissertation was built on top of TESLA, a toolkit built by Jon Salz while working on his M.Eng. with me. I've undoubtedly pestered him with more bug reports and feature requests than he could possibly have anticipated. I owe him for his unflagging commitment to address each of them. Thanks also to Ken Steele and Jason Waterman at MIT, and Dan Aguayo and Dimitris Kalofonos at Nokia for their willingness to try out not-quite-ready-for-prime-time versions of *Migrate* and share their experiences with me. Much of the inspiration for this dissertation came out of discussions with Brian Noble. Some of the material included in Chapter 7 was co-authored with David Andersen, and Kyle Jamieson and Ken Steele assisted with measurements of power consumption. My mother, in her least-significant role in my life as chief copy editor, lovingly revised each and every chapter.

This dissertation reports on only a portion of my work at LCS. Several other faculty and staff assisted me with the all-too-frequent distractions that made my stay enjoyable and rewarding. Thanks to John Wroclawski, Karen Sollins, and David Clark for taking me in when I first arrived at MIT without an advisor, office, research project, or direction. Victor Zue, Dorothy Curtis, and Ty Sealy were instrumental in my early Hummer exploits. David Gifford's boundless generosity, sympathetic ear, and insightful guidance set me on a course to graduation. David Karger's open door and uncanny algorithmic insights were invaluable.

Day-to-day graduate student life would be quite monotonous were it not for the constant interactions with my office mates, without whom I no doubt would have gone insane. Our spontaneous discussions on topics technical and otherwise provided needed inspiration, analysis, and comic relief. Thanks especially to Alex Hartemink, Dina Katabi, Jo Kulik, Allen Miu, and Stan Rost. Dave Andersen deserves special note; despite our constant technical arguments, political rants, and occasional personal outbursts—in fact, precisely because of them—he was in every way the perfect office mate. I'll miss his big yellow ball. Jeremy De Bonet, my lifting partner and constant companion for the few years we overlapped at MIT, provided a sounding board for crazy ideas, implausible theories, and general sophistry.

While we never shared an office, I learned a lot from the PDOS crew, particularly Eddie Kohler, David Mazières, Chuck Blake, John Jannotti, Benjie Chen, Doug De Couto, Kevin Fu, and Emit Witchell. Similarly, my fellow NMS students were always ready and willing with helpful feedback. After hours, I appreciated being welcome at the various activities of the AI Lab, including GSL, GSB, and Tuesday-night hockey. My erstwhile roommates, Chris Conklin, Matt Lau, Sam Pearlman, and Bradley Weill provided much-needed escapes from my MIT world.

Graduate study is fraught with feelings of helplessness and self-doubt. Several students that went before me provided guiding lights, reminding me it was possible to make it out the other end. They likely never knew it, but David Wetherall, Dawson Engler, Danny Lewin and Charles Isbell were each role models I strove to emulate. While I have doubtless fallen short, their examples instilled confidence when it seemed I'd never finish.

I am indebted to my colleagues at BBN, especially Craig Partridge and Tim Strayer, for providing me with an avenue of technical exploration outside the confines of MIT and exposing me to the commercial realities of networking research in the private sector. I hope they found our years together as rewarding as I. I also thank my future colleagues at the University of California, San Diego, for their understanding in allowing me to take a leave of absence to finish this dissertation.

Finally, I cannot begin to properly thank Christine Alvarado, who was a continual source of joy in my life as I suffered through the final stages of graduate school. Instead, I offer the pronouns in this dissertation in deference to the women in computer science, of whom she is my favorite example.

Contents

List of Figures

List of Tables

Dimidium facti, qui coepit, habet: sapere aude.
(To have begun is half the job: dare to be wise.)

- Horace

Chapter 1

Introduction

The proliferation of mobile computing devices and wireless networking products over the past decade has led to an increasingly nomadic computing lifestyle. A computer is no longer an immobile, gargantuan machine that remains in one place for the lifetime of its operation. Today's personal computing devices are portable, and Internet access is becoming ubiquitous. A well-traveled laptop user might use half a dozen different networks throughout the course of a day: a cable modem from home, wide-area wireless on the commute, wired Ethernet at the office, a Bluetooth network in the car, and a wireless, local-area network at the airport or the neighborhood coffee shop.

Armed with her portable computing device and readily-available Internet access, today's user expects seamless operation for network applications. Yet her sporadic movement and occasional disconnection due to lack of network connectivity or device power-down place a considerable burden on applications that communicate across the network. These network applications receive little assistance from today's operating systems or Internet protocols in managing host movement or periods of disconnection. While a small number number of modern, mobile-aware applications have been designed to handle these adverse conditions in an *ad-hoc* fashion, the vast majority of applications have not.

This dissertation recognizes the importance of moving end points and the inevitability of periods of disconnection, and defines a set of system primitives to assist applications in dealing with these two challenges. We propose a solution based on the *session* abstraction. A session is a durable, long-term relationship between application end points that may span multiple network connections and application transactions; today's network connections, on the other hand, are ephemeral relationships between network attachment points. Everyday examples of sessions include interactive logins by users of remote hosts, sets of Web transactions between browsers and servers, multimedia conferences between remote peers, etc. These applications, along with many others, have found sessions a useful construct for managing complex interactions between remote network end points. In particular, sessions afford the opportunity to amortize certain expensive operations such as authorization, initialization, and synchronization across multiple, individual network connections. Unfortunately, current applications cannot describe their networking needs to the operating system in terms of sessions; instead, applications must describe each network connection individually and manage each one independently.

We propose elevating the session from common application construct to first-class system abstraction: an operating system-supported building block for mobile network applications. We present

Migrate, an end-to-end mobility architecture that supports *session-based* mobility, and enables sophisticated disconnection management through the use of *session continuations*, a mechanism that allows mobile-aware applications to efficiently suspend operation during periods of disconnection yet adapt to changed network conditions upon resumption.

The rest of this chapter is organized as follows. In the first section, we describe the challenges posed by portable Internet devices. The second section demonstrates, through an example, the shortcomings of today's network infrastructure when attempting to support Internet communication on portable devices. The following section then gives a brief overview of our approach, and how it ameliorates these issues. We conclude this chapter by summarizing the goals, contributions, and organization of this dissertation.

1.1 The challenges

One of the critical realizations leading to this dissertation was the observation that portable Internet hosts introduce two distinct, but inter-related challenges: first, end points move between network locations during communication, and, second, end points disconnect from the network without prior notice. Preserving communication between two moving end points on the Internet is difficult because one end point may not know to where the other end point has moved nor be able to describe or properly authenticate it if its location is discovered. In particular, remote end points may not know how to address packets destined for a moving end point. Furthermore, because Internet end points are commonly referred to by their locations in the network, a mobile end point that moves to a new location needs some way to identify itself as the node formerly at its previous location. Unexpected disconnection is particularly problematic for session-based applications (*e.g.*, streaming media [114], file transfer applications [94], X windows [112], SSH (Secure SHell) [144], etc.) that maintain state and consume resources on behalf of remote end points. Deciding how long to wait for a remote end point to reconnect is not obvious; giving up too early results in a poor user experience (*e.g.*, aborted downloads, canceled transactions, etc.), waiting too long wastes precious system resources such as power, memory, and bandwidth.

1.1.1 Moving end points

A requirement of Internet mobility support is the need to allow network applications to continue to function as hosts change *attachment point*—the location in the network where hosts send and receive packets. Applications on such hosts should be able to continue communication from where they left off at previous attachment points. More generally, this problem affects not only portable hosts but all classes of mobile *end points*, whether they be the hosts themselves, individual applications, services, processes, or even users. Traditionally, researchers have categorized movement based on the end-point being considered:

- *Host* or *terminal* mobility refers to the common case where an entire host, such as a laptop or handheld, changes its network attachment point [49, 89].
- *Personal* mobility ignores the computing device(s), instead, focusing on the user as she moves between Internet hosts [65, 113]. For example, a user may start reading her email on a PDA, but wish to continue reading from her desktop PC when she arrives at her office.
- *Session* mobility tracks communication sessions as they move, either coincident with one of the above forms of mobility or not. For example, a Web server farm may wish to move a client's session to a different server to balance load across the available servers.

The third class, session-based mobility, is the most general, as the first two classes can be cast as specific instances of the third. A host movement can be viewed as the simultaneous movement of all sessions terminating at that host but not *vice versa*. For example, a user may be both reading email and browsing the Web on her PDA but wish to move only one of the sessions (her Web browsing, say) to her office PC. Similarly, a Web session may be moved to a new server for failover or load-balancing concerns. Hence, this dissertation focuses on session-based mobility, but many of the ideas apply equally well to the other two cases. For the most part, we will restrict our examples to the case of host mobility, but the session-based approach is designed to handle migration across hosts. Such migration, however, requires significant additional support and will be discussed only briefly in Chapter 8.

For the purposes of this dissertation, we are concerned with movements that are evident to other hosts on the Internet, namely, changes to the network attachment point. Such movements may or may not be coincident with an actual physical movement in any of the above models. In fact, there are many common reasons why Internet end points may appear to move without any physical change whatsoever. Two common reasons are *readdressing* and *multi-homing*. Internet attachment points may receive new addresses from time to time due to configuration changes in the network (*e.g.*, DHCP [31] lease expiration or NAT [122] reconfiguration). Multi-homed hosts have multiple, distinct network attachment points and may communicate using different attachment points at various instances or even multiple points concurrently.

The process of managing moving end points entails two issues: before communication can begin, a moving end point must be *located*; then, once an initial location is determined, the end point must be *tracked* as it moves. Both issues depend on how applications describe end points. Applications must somehow name the remote end point they wish to locate and track. Depending on the mechanism used to describe the end point, however, this tracking procedure may or may not rely on the initial location mechanism.

1.1.2 Unexpected disconnection

As mobile hosts change network attachment points there are often accompanying periods of disconnectivity. Despite improvements in technology and the increasingly widespread deployment of so-called 3G wireless technologies [33], we expect devices will continue to operate under non-negligible periods of disconnection due to resource constraints. Wireless communication consumes power, a resource that is in limited supply in untethered mobile devices and shows no signs of dramatically improving anytime soon. Similarly, commercial wireless access costs money—a constrained resource for most users.

Furthermore, disconnection is frequently unexpected, from the points of view of both the user (*e.g.*, the wireless interface moves out of range of a basestation) and the system (*e.g.*, a network cable comes unplugged). Disconnection also occurs with surprising frequency in many wire-line networks due to routing failures and other network instabilities [4]. Additionally, because it is often unanticipated, the duration of disconnection—regardless of cause—is often unknown, highly variable (across several orders of magnitude), and frequently long (*e.g.*, several hours).

To manage intermittent connectivity, we adopt a “suspend/resume” model of interaction for network applications because this model is already prevalent among laptop users. Mobile laptop users have grown accustomed to suspending their activities at arbitrary points and being able to resume the interaction from the points at which they were suspended, despite arbitrary periods of inactivity during which the laptops enter a resource conservation mode. Unfortunately, as demonstrated in the

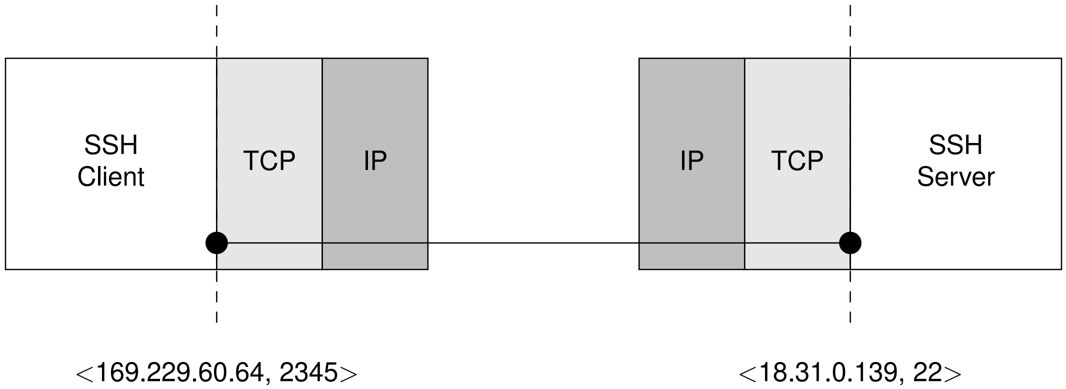

Figure 1-1: SSH establishes a TCP connection between the client and server application end points. The system instantiates this connection by binding the application end points to their current network attachment points as specified by an ⟨*IP address, port*⟩ pair.

example in the next section, today's Internet hosts lack support for seamless operation of session-based network applications across periods of disconnectivity. Disconnection, when discovered by today's Internet protocols, is considered a permanent failure and communication is aborted. Hence, contemporary operating systems do not provide "suspend/resume" support for network applications. Instead, disconnection events are either concealed inside the network or exposed as communication failures to the application, which is then forced to abandon open communication sessions and begin new ones.

1.2 A motivating example

One need look no further than interactive terminal applications like SSH [144] or telnet [98], members of the Internet's oldest class of applications, for a practical example of the lack of support for mobility in the Internet. A user with an open SSH session might pick up her laptop and disconnect from the network. After traveling for some period of time, she reconnects at some other network location and expects that her SSH session will continue where it left off. Sadly, the user will find the SSH application has aborted, and she is unable to resume her session. This occurs because the remote end point (the SSH server) was unable to determine the new attachment point and unsure that the session would continue. The following section deconstructs the technical reasons for these limitations.

1.2.1 Current network abstraction

Most contemporary applications and operating systems use the Berkeley Sockets API [67], which exports a *connection* abstraction. Figure 1-1 illustrates the connection between an SSH client and server. A connection defines a communication channel between two network attachment points that an application uses to transfer data packets between two remote end points. Unfortunately, since the connection end points are specified in terms of their network attachment points, the operating system has no way of naming or locating an application end point that changes attachment points. Therefore, any change in attachment point at either end point terminates the connection because the previously specified attachment points no longer correspond to the current location of the end points. Perhaps even more frustratingly, the connection abstraction, when used in conjunction with reliable transport protocols like the Internet-standard Transmission Control Protocol (TCP) [97], is unable to accommodate periods of disconnection longer than a few round-trip times. Rather than blithely consume resources while waiting for the remote end point to return, current Internet protocols expose the disconnection after some period of time by aborting the connection. Hence, unless explicit mobility and disconnection support is provided by the application, it will be unable to survive any changes in attachment points or even a brief period of disconnection.

The inability of transport protocols to handle changes in attachment points over the duration of a connection has led to attempts to conceal attachment-point changes and periods of disconnection from the end points. These efforts can be broadly grouped into two categories: those that manage changes in attachment point inside the network, and those that enable transport connections to survive periods of disconnection. While these approaches are discussed in detail in the next chapter ("Related Work"), we briefly explain why they are inadequate for the current example.

1.2.2 Network-layer techniques

Network-layer mobility techniques that handle changes in attachment point inside the network routing layer are unable to handle the scenario described above due to the associated period of disconnection. For example, even if the Internet protocols allowed end points to change their attachment points, the user would still find her SSH session aborted upon reconnection if there was any activity at all on the session during the period of disconnectivity. This failure is due to the transport protocol's inability to handle the extended period of packet loss experienced during disconnection. Furthermore, by concealing changes in attachment point from applications, network-layer techniques make it difficult for applications to adapt to dynamic network conditions. For example, the security parameters selected by SSH may have been conditioned on the user's initial attachment point. If she later moves to a more hostile network, SSH may wish to increase the strength of its security measures (*e.g.*, by increasing key length or using a stronger cipher). However, if the change in attachment point is hidden inside the network, SSH will be unaware that such a change is warranted.

1.2.3 Transport-layer techniques

A complimentary approach could equip transport connections to survive periods of disconnection. Consider, for example, if the user's TCP connection were to silently persevere across periods of disconnection. The user would then be able to resume her session, but that doesn't speak to the server's activity in her absence. Using this transparent approach, the remote application end point is oblivious to the disconnection, likely resulting in a considerable waste of resources (in the form of power, processing, memory, kernel buffers, network ports, file descriptors, etc.), decreased performance, and, perhaps, even loss of data or incorrect operation depending on the application at hand. For example, an application may invoke a default operation unless instructed otherwise by a remote end point before some period of time elapses. If the remote end point is disconnected—and, therefore, unable to inform the application of its desires in a timely fashion—but the disconnection is concealed by the transport layer, the application may incorrectly assume silence represents ascension and invoke the default operation.

As discussed at length in the next chapter, network- and transport-layer proposals to handle Internet mobility address only some of the issues, do not allow mobility-aware applications to adapt to changes in network conditions, or require application-specific point solutions.

1.3 Supporting session-based mobility with *Migrate*

In this dissertation, we propose addressing both challenges—moving end points and unexpected disconnection—in a consistent, unified fashion based on the session abstraction. We present an overview of our approach, called *Migrate*, by way of describing our solution to three separate problems:

- Applications describe network end points and their interactions through a system session abstraction.

- On-going connections between moving end points are preserved via *connection migration.*

- Applications specify desired session resumption functionality through *session continuations*, enabling resource conservation during periods of disconnection and intelligent adaptation to new network characteristics upon resumption.

While addressing each of these issues, *Migrate* attempts to remain neutral with respect to both security and policy. Because different applications and users have widely varying security goals and mechanisms, *Migrate* provides an *attack-equivalent* environment, which we define to mean an environment in which any attack that can be launched against the application can be shown to be equivalent in power to one that could have been launched in an environment without *Migrate*. Because this guarantee does not depend on the particular mechanisms employed by the application, it does not suffer from the inadequacies (or benefit from the strengths) of any particular implementation—each application is left to choose the mechanisms most appropriate for it. Similarly, *Migrate* allows applications and their users to set their own *mobility policy*. In conjunction with system-wide rules regarding resource usage, available network interfaces, and so on, a user's mobility policy specifies her preference for particular attachment points, resource consumption tolerance, network access restrictions, and so forth.

1.3.1 Describing network end points

Migrate adopts the session abstraction, defining an Application Programming Interface (API) that enables applications to describe their sessions to the operating system. The API is sufficiently flexible to allow applications to specify remote end points using whatever naming mechanism they find convenient, but we expect that users will typically use the Domain Name Service (DNS) [68]. *Migrate* separates the task of end-point location from tracking, enabling applications to specify their own naming mechanisms for location operations while leveraging system support for the straightforward tracking operation. Further, this separation often enables *Migrate* to preserve communication between moving end points *in the absence* of an available location service.

1.3.2 Migrating open connections

One of the difficulties in preserving communication between moving end points is maintaining the semantics of standard transport protocol connections. In particular, today's connection-oriented transport protocols do not support *migrating* connections—that is, changing attachment points mid-connection. We propose two distinct solutions, *virtualization* and *rebinding*, each with different advantages and disadvantages.

In the virtualization approach, an *indirection layer* presents the application with a virtual connection to the remote end point. The indirection layer dynamically maps the virtual connection to ephemeral transport connections between the current attachment points by creating and destroying transport connections as necessary to support different attachment points; the layer creates a new connection each time an end point moves to a new attachment point. The advantages of this approach are that it can be applied to any transport protocol and carried out entirely at user-level, requiring no extensions or modifications to existing operating systems; the major drawback is the virtualization overhead.

The second approach involves extending transport protocols such as TCP to support *rebinding* the connection to new attachment points. We modify TCP to support a set of TCP options that allow TCP connections to continue across changes in either attachment point. This migration is transparent to an application that expects uninterrupted reliable communication with the peer. Using this approach, there is no overhead outside of mobility events. However, since many operating systems implement TCP inside the kernel, deploying this approach will require kernel modifications to support the new TCP options.

1.3.3 Supporting disconnection

We observe that the challenge of suspending a session upon disconnection and resuming it later bears resemblance to the problems encountered by a compiler when handling returns from procedure calls. In both cases, naming scopes, environment settings, and mutable state must be saved and restored. This management can be avoided completely if procedures never return. Instead, every procedure can maintain a notion of "the rest of the computation," or *continuation*, and simply invoke (or call) the continuation upon completion. Hence, procedure calls can be replaced by jumps, and many of the complications of managing activation records and call stacks go away. Continuations provide abstractions that simplify the constructions of many compilers and have been applied in other domains [30, 36].

We propose *session continuations* as a generic mechanism for supporting the conservation of system resources during periods of disconnection and the resumption of session processing upon restoration of connectivity. A session continuation contains all the state and functionality necessary to compute the "rest of the session," including any required reconciliation between the state of the suspended session and prevailing network conditions. Upon reconnection, a disconnected session simply invokes its continuation. Since such continuations never implicitly depend on previous state the system can safely discard resources consumed by disconnected sessions. When a session resumes, its continuation generates all the state and resources necessary to complete it.

In *Migrate*, an application handles disconnection in one of two ways. In the preferred method, applications specify a session's resumption context as a session continuation that allows them to conserve resources during disconnection and adapt to possibly changed network conditions present at reconnection. When generating a session continuation is unnecessary or difficult, however, applications may choose to simply preserve the session state and system resources and rely on transparent *Migrate* services to conceal periods of disconnection. This transparent support allows unmodified, legacy applications to function in a mobile environment, but may waste resources during disconnection.

1.4 Contributions

This dissertation postulates that unexpected end-point movement and temporary periods of disconnection are unavoidable in the mobile Internet. It then explores the hypothesis that system support for a robust session abstraction can enable mobile operation of legacy applications and provide enhanced functionality for mobile-aware, session-based network applications, allowing servers to support large numbers of dormant sessions. Further, this dissertation shows end-to-end protocols are sufficient to provide adequate connectivity monitoring and graceful handling of host movement, session suspension, and their subsequent reactivation. It demonstrates that location tracking and session security can both be efficiently decoupled from baseline mobility support. Along the way, this dissertation makes the following specific contributions:

- The development of an *end-to-end* approach to Internet mobility that is based on a system-supported *session* abstraction, along with a specific architecture, *Migrate*, that implements this approach.

- The design and implementation of Migrate options that extend TCP to support the migration of TCP connections to new attachment points.

- *Session continuations*, extensions to the session abstraction that enable application-specific suspend/resume handling. By providing system support for session continuations, hosts can realize significant resource savings during periods of disconnection. Session continuations also have applications to problems outside of host mobility such as load balancing and application migration. Carefully crafted continuations can be executed in environments (hosts) other than those that created them, enabling a form of inter-host session migration.

- An application-agnostic, *attack-equivalence* security model that ensures any attacks enabled by mobility support reduce to ones already present in a non-mobile environment.

- A proposed session API that allows mobile-aware applications to specify intelligent disconnection handling through the use of *continuation-passing style*, and an understanding of how a variety of network-based applications can use the API to provide application-specific functionality.

- An evaluation of the efficiency of our prototype *Migrate* implementation. Our results show that the throughput impact of connection virtualization is small (2% or less for moderate block sizes) for sessions operating over common access link technologies. The overhead can be considerably larger, however, when virtualizing extremely-high bandwidth ($>$ 350-Mbps) connections or those using small ($<$ 200-byte) block sizes. When used in conjunction with the TCP Migrate options, *Migrate*'s overhead becomes almost negligible and is restricted to session establishment and migration events.

- A demonstration of the effectiveness, flexibility, and ease-of-use of our session continuation abstraction. We show that servers for two popular Internet applications, SSH and FTP, require only small modifications to support session continuations, and that suspended sessions for both applications consume only a few tens of bytes of secondary storage and between one and three file descriptors.

The current version of *Migrate* has two main limitations which arise from our restrictive definition of a session. In particular, we consider a session to be a stateful relationship between two application end points. While our abstraction proves useful for a large class of applications, there are popular network applications that violate this definition in two important ways. First, some applications form sessions between three or more end points. Examples include multi-party conferencing, gaming, and multi-cast based content distribution applications. We have restricted our definition to sessions containing exactly two end points because the semantics of disconnection, suspension, and resumption are straightforward. The appropriate semantics for disconnection and end-point tracking in a group scenario are more complicated and extending our session abstraction to consider multiple remote end points remains an area for future work.

In contrast, some popular applications do not maintain any stateful relationships between application end points. Such so-called *stateless* network applications often use non-connection oriented transfer protocols (*e.g.*, UDP) and asynchronous remote procedure calls (RPC) [14]. (Transport protocol alone does not necessarily indicate whether an application is session-oriented or not, however. We present *Migrate*'s support for non-connected UDP sockets used in a session-oriented fashion in Section 4.4). These communication paradigms have little to gain from session-based mobility handling. While *Migrate* does not interfere with the operation of such applications, the additional mobility support provided by *Migrate* is generally extraneous and introduces unnecessary overhead. When used in conjunction with stateless legacy applications which do not opt-out of *Migrate* support themselves, users can disable *Migrate* through the use of the system-wide policy file (see Section 6.3).

1.5 Organization

The remainder of this dissertation presents the session-based mobility model and a prototype implementation, *Migrate*. Chapter 2 motivates our work by describing related work in the fields of network-layer Internet mobility, connection migration, disconnected operation, application checkpointing, and resource management and adaptation. We describe our session-based approach to mobility in Chapter 3, including the session abstraction, end-point naming, attack-equivalent security model, and details of the API. Chapter 4 presents two approaches to connection migration, one based on virtualization and the other based on rebinding. We present session continuations in Chapter 5, discussing their implementation in *Migrate*. Chapter 6 describes the implementation of *Migrate*, including the API, network connectivity monitors, and policy engines. We evaluate *Migrate* in Chapter 7 by benchmarking its performance and showing how several well-known applications can be extended to support session continuations. Finally, Chapter 8 presents a summary of the dissertation and concludes with a discussion of both specific contributions and general principles that can be extracted from the work.

It is what we think we know already that often prevents us from learning.
– Claude Bernhard

Originality is nothing but judicious imitation.
– Voltaire

Chapter 2

Background & Related Work

Mobility has been a fertile area of research for many years. In this chapter, we provide an introduction to basic networking concepts and discuss the issues raised by mobility. By surveying previous approaches to address these issues, we motivate the three focusing problems of this thesis: handling changes in session attachment points, migrating open connections, and supporting disconnection in session-based applications.

We begin in Section 2.1 with a brief tutorial on Internet concepts. Section 2.2 presents previous proposals to support mobile Internet hosts and argues that network-layer solutions fail to fully address the needs of mobile end points. We then explore alternative approaches, beginning in Section 2.3 with a discussion of connection migration. Section 2.4 describes the session abstraction and explains how it can be used to support mobile end points. Finally, the chapter concludes in Section 2.5 with a discussion of the impact of disconnection on Internet hosts and the need for session-based suspend/resume support.

2.1 Internet basics

A packet-switched network consists of *end points* that can send packets to each other. Depending on one's perspective, end points can be variously construed to be hosts, applications, services, processes, or even users. In most places in this dissertation, we will not distinguish among the different kinds of end points. Instead, we will consider the general case of an application process running on a particular host providing a service to a user. In some cases we will distinguish between applications and hosts.

2.1.1 End-point addressing

Each end point connects to the network at one or more *network attachment points*. End points without a current attachment point are said to be detached or *disconnected*. Network attachment points are the locations in a network where end points send and receive packets. Each packet sent by an end point must be addressed to a network attachment point. In the Internet, attachment points are identified by IP addresses (*e.g.*, 18.31.0.100). An IP address is a hierarchical name that reflects the topological location of an attachment point in the Internet and enables the Internet routing infrastructure to deliver packets destined to network attachment points.

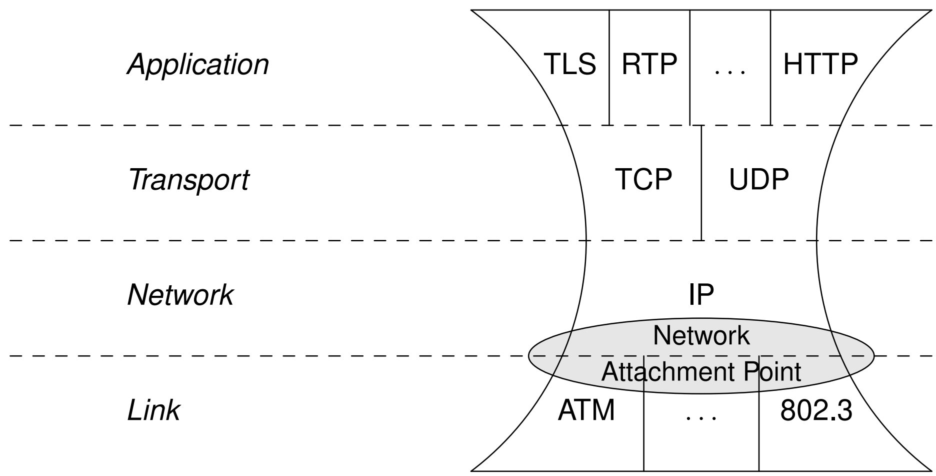

Figure 2-1: The hourglass model of the Internet protocol stack. A network attachment point is an interface between the network layer and a link layer. Anything above the network attachment point can be construed as an end point.

2.1.2 Layering

End points on the Internet communicate using a suite of composable protocols, typically referred to as the Internet *protocol stack*. Within the stack, each protocol uses the lower layers to provide a well-understood service to higher layers. Intuitively, the protocol stack is commonly pictured as an hourglass as shown in Figure 2-1. The hourglass shape emphasizes the “layered” structure of the protocol stack, where each protocol provides a well-defined interface to those above it, and the simplicity of the network layer. At the lowest level of the stack are the *link-layer* protocols employed by various networking devices like Asynchronous Transfer Mode (ATM), Ethernet (defined by IEEE standard 802.3 [47]), the Point-to-Point Protocol (PPP) [117], and many others. The link layer enables point-to-point communication between *network interfaces* connected to the same physical medium. The *network layer* abstracts away the vagaries of actual network topology and *routes* packets between any two end points regardless of whether or not they are both on the same physical network. The Internet is defined by the use of one particular network-layer protocol: the Internet Protocol (IP) [96].

2.2 Network-layer mobility

End points may, from time to time, associate with different attachment points; we call end points that move from one attachment point to another *mobile* end points. Similarly, architectures that support mobile end points are generally said to support end-point *mobility*. End points may change attachment point for a variety of reasons:

- An Internet host may physically move to a new location
- An application may be migrated from one host to another [38, 83]
- An Internet service may fail-over from one server to another server [85]
- An end point with multiple attachment points may choose to use a new one

- A network interface may be assigned a new IP address (thus becoming a different attachment point).

Regardless, the identity of a mobile end point does not change—only its attachment point to the network does. Communicating with a mobile end point is problematic, however, as *correspondent* end points—those end points currently communicating with the mobile end point—must address packets to a particular attachment point, which may or not be the current location of the mobile end point. Recognizing this difficulty, a number of researchers have proposed solutions focused on the common case of portable Internet hosts that move from place to place in the network—so called *mobile hosts*.

Many proposed approaches work by inserting a layer of indirection between an IP address and its corresponding network attachment point, allowing the same IP address to refer to varying network attachment points, depending on which attachment point is currently being used by the host [13]. We describe proposals based on six different techniques: Nimrod [18, 102], Mobile IP [49, 89], the Host Identity Payload (HIP) [72], IP-based redirection [41, 128, 132, 143], Network Address Translators (NATs) [128] and Virtual Private Networks (VPNs) [22], and multicast [43, 76].

2.2.1 Nimrod

Castiñeyra, Chiappa, and Steenstrup proposed the Nimrod architecture as a choice for the next-generation of the Internet [18]. Nimrod introduces the notion of persistent *end-point identifiers* (EIDs) that are separate from network attachment point addresses. In Nimrod, each host has its own EID which can be used to address data packets; EIDs are mapped to current network attachment points by the routing infrastructure itself. To support mobility within Nimrod, Ramanathan proposes the concept of a Dynamic Association Module (DAM), an abstract entity that manages changes in the mapping between EIDs and network attachment points [102]. The problem addressed by the DAM is not unlike the standard IP mobility problem, however, and Ramanathan's proposed implementation is actually based upon Mobile IP.

2.2.2 Mobile IP

Mobile IP [49, 89] is the current Internet Engineering Task Force (IETF) standard for supporting host mobility on the Internet. It provides transparent support for host mobility by inserting a level of indirection into the routing architecture, similar to Nimrod. Unlike Nimrod, however, Mobile IP does not require a redesign of the IP routing infrastructure. Mobile IP introduces a notion of a *home network*—the network to which a mobile host "belongs" (conversely, all other networks are known as *foreign*). The assumption is that whenever a host is connected to its home network, it will always use the same network attachment point. Hence, a host using Mobile IP has a well-defined *home address*, which is the IP address of the host's network attachment point on its home network.

Mobile IP elevates the mobile host's home address from its traditional function as a network attachment point identifier to an end-point identifier. A mobile host always uses its home address as the source address in any packets it transmits; Mobile IP ensures that packets addressed to a mobile host's home address are delivered to the host's current network attachment point, regardless of where that attachment point might be. Hence, correspondent end points see only the host's home address and have no indication that the host is mobile, or what its current network attachment point might be.

Mobile IP manages packet delivery by placing a *home agent* on the local-area network corresponding to the mobile host's home address—its *home network*—which listens for packets destined for

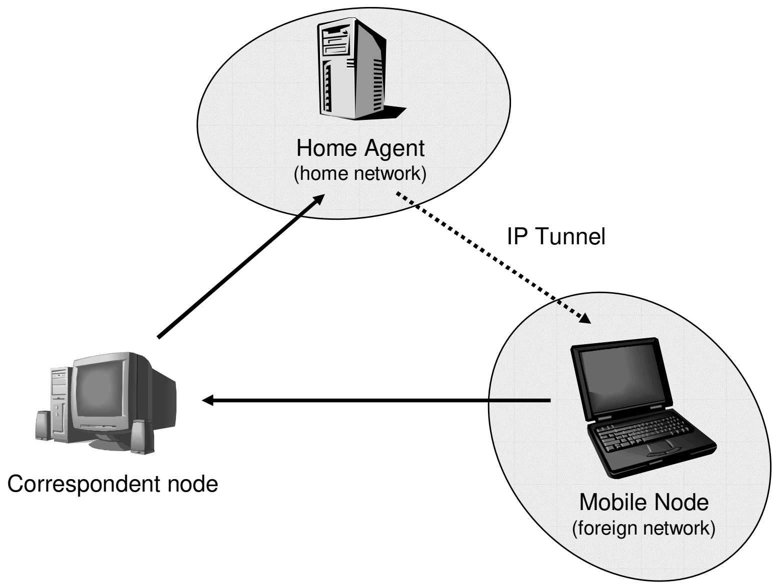

Figure 2-2: Triangle routing in Mobile IP without route optimization. Correspondent nodes send packets destined for a mobile node to its home address on its home network, where a home agent intercepts the packets and tunnels them to the mobile node at its care-of address in the foreign network. In some cases, a mobile node can send packets directly to the correspondent node, avoiding the need to tunnel outgoing packets back through the home agent.

the mobile host and forwards them on to the host when it is attached to a foreign network. Conceptually, the home agent takes over the mobile host's attachment point in its home network when the host moves to a foreign network. A mobile host in a foreign network acquires a *care-of address* (the IP address of the host's network attachment point in the foreign network), which the home agent uses to forward packets; a mobile host notifies its home agent of its new care-of address any time it changes attachment points.

To forward a packet to a mobile host's care-of address, a home agent *encapsulates* [88] the packet in an IP *tunnel*. The home agent prepends an additional IP header to the packet; the new packet is addressed from the home agent to the mobile host's care-of address. Once the encapsulated packet is received at the mobile host's foreign attachment point, it is unwrapped and delivered to the mobile host in its original form. The original packet is said to be *tunneled* since it is routed from home agent to foreign attachment point based upon the addresses included in the encapsulating packet, not those in the original packet itself. Hence, packets destined for a mobile host attached to a foreign network first travel from source to the host's home agent and then from the home agent to the mobile host itself. This often circuitous path is referred to as *triangle* or *dog-leg* routing and is depicted in Figure 2-2.

Further compounding the triangle routing problem is the widespread deployment of *ingress filters* [34], promoted by the IETF as a "Best Current Practice." Ingress filtering was developed to prevent address *spoofing*, where a network attachment point places an IP address other than its own into outgoing packets in an attempt to obscure its identity from the packets' recipient. Ingress filters prevent the forwarding of packets with source addresses that are not "appropriate" for the network from which they were received. Conceptually, an address is appropriate if it is received from a network that is on the reverse-forwarding path for packets destined for that address. In other words, from the point of view of router R, a source address S is appropriate for network N if R might route packets destined for S to network N. Unfortunately for Mobile IP, mobile hosts use their home addresses as the source IP addresses for all packets, regardless of their current network attachment points. In foreign networks with ingress filtering, the ingress filter will block the packets sent by a mobile host.

To work around this problem, Mobile IP advocates the use of *reverse tunneling*, which tunnels packets originating at a mobile host currently in a foreign network back to the host's home agent (using the host's care-of address as a source address), which then forwards them on to their destination (using the mobile host's home address as the source address) [69]. Thus, when a mobile host visits a foreign network with ingress filtering, triangle routing occurs in both directions.

Route optimization

Perkins and Johnson present a secure route optimization option for Mobile IP to avoid triangle routing [91]. Here, correspondent hosts cache the care-of addresses of mobile hosts, allowing communication to proceed directly. It requires the home agent to send an authenticated update to correspondent hosts [90], notifying them of the mobile host's current care-of address. The resulting Mobile IP scheme eliminates triangle routing in the forward direction but requires modifications to the IP layer of all end hosts (regardless of whether they are mobile or not) and the existence of some authentication mechanism to validate the care-of address updates—a task that Perkins and Johnson note is far from trivial: "One of the most difficult aspects of Route Optimization for Mobile IP in the Internet today is that of providing authentication for all messages that affect the routing of datagrams to a mobile node" [91, pp. 23].

IPv6

The IETF has standardized the next version of the IP protocol, IPv6, which provides a number of enhancements [26]. (The current version, which we refer to simply as IP, is properly known as IPv4.) Because IPv6 provides native support for multiple simultaneous host addresses, route optimization does not require further modifying IPv6 hosts—of course, IPv6 itself requires a massive overhaul of the entire unicast infrastructure; deployment has been understandably slow. If deployed, however, IPv6 extensions allow for the specification of a care-of address, which explicitly separates the role of the EID (the host's home IP address) and routing location (the care-of address). The rest of Mobile IP stays largely the same. In particular, Mobile IPv6 continues to require a home agent, resident on the mobile node's home network, that intercepts packets destined to the mobile node and tunnels them to the remote, care-of address. In addition, IPv6 does not simplify the task of securing the care-of address update in any way.

2.2.3 Host Identity Payload

Recently, Moskowitz has proposed to name each Internet host by a cryptographic Host Identity [72]. In this scheme, every packet includes a source Host Identity, which is the public half of a public/private key pair. When applied to host mobility, the Host Identity is similar to a home address in Mobile IP in that it remains constant, regardless of a host's current network attachment point. Rather than replace the source address of an IP packet with their home addresses, however, mobile hosts use their care-of addresses as the source, but also include their Host Identity using the Host Identity Payload (HIP) protocol [73]. Furthermore, the payload of the packet is signed with the corresponding private key. Hence, the recipient can discern not only the originating end point, but also its current network attachment point, *and* verify that the packet was actually sent by the end point. This approach obviates the need for any form of tunneling, but still requires some mechanism for locating a mobile host's current attachment point.

2.2.4 IP redirection

Teraoka, Yokote, and Tokoro proposed a Virtual Internet Protocol (VIP) that extends the IP protocol to consider two distinct addresses: a virtual network address (VN) and a physical network address (PN) [132]. The virtual network address serves as an EID, while the physical network address is the traditional IP address. As in Mobile IP, hosts are assumed to have a home network where the VN and PN are identical—the VN can be thought of as corresponding to the home address and the PN the care of address. Routers within the network maintain an address mapping table (AMT) that caches recently observed VN→PN pairs. A packet with identical destination VN and PNs (*i.e.*, addressed to a mobile host on its home network) can be redirected by any router with an entry in its AMT for the VN.

In the VIP scheme, a mobile host notifies a router on its home network of its new attachment point (PN) when it attaches to a foreign network. This router then acts as a home agent for the host, forwarding any packets addressed to the mobile hosts' VN to its current PN by replacing the original (home) PN with the current (care-of) PN. As the packet travels from the home network toward the mobile host all the on-path routers cache the current PN of the mobile hosts VN. Hence, any of these routers will serve as a home agent for the mobile host when they observe a packet destined for the mobile host on its home network. AMT cache entries are expired in order to avoid routing loops and misdirected packets. While more efficient than Mobile IP, VIP requires changes not only to the IP layer of the end hosts, but network routers as well. Also, as the number of mobile hosts grows large, AMTs at interior routers may grow prohibitively large.

Gupta and Reddy propose a redirection mechanism for IPv4 that can support mobility in a fashion similar to Mobile IP with routing updates [41]. Originally proposed to support service replication, the IP Redirection Protocol (IPRP) allows hosts to maintain and update care-of addresses for remote hosts. A redirector performs the functions of a home agent in Mobile IP. Hosts wishing to contact the mobile host first contact the redirector, which uses IPRP to provide the mobile host's current care-of address. Binding updates are accomplished by cascading redirections—after receiving a new care-of address from the mobile host (using a mechanism not described in the original proposal), the redirector would further redirect all correspondent end points to the new care-of address. The proposed implementation of IPRP requires the modification of all IP stacks to support managing care-of addresses.

Yalagandula *et al.* propose a similar redirection-approach that allows a mobile host to serve as its own redirector [143]. When a mobile host moves to a new attachment point, it sends binding updates to all of its correspondent hosts (which it tracks on an active partner list). The absence of a home agent requires correspondent nodes to obtain a mobile node's current attachment point on their own, however.

2.2.5 NAT & VPN-based solutions

Standard NAT [122] software can accomplish similar redirection without modifying the IP stack. By interposing a NAT between an end point and its network attachment point, NAT software can translate home IP addresses into appropriate care-of addresses. This approach, termed Virtual Network Address Translation (VNAT), was recently proposed by Su and Nieh [128]. VNAT does not use a redirector or home agent, however, and does not discuss how the address of the mobile host's current network attachment point is obtained.

The indirection provided by NAT can also be provided through Virtual Private Network (VPN) products. Several commercial products (like Columbitech's WVPN solution [22]) use Transport Layer Security (TLS) [27] to create secure VPN tunnels between mobile hosts and their home agents. Rather than forward packets using IP encapsulation, as in Mobile IP, home agents in these schemes use TLS tunnels and VPN address re-mapping to affect the same triangle routing.

2.2.6 Multicast mobility

Mysore and Bharghavan propose an approach to network-layer mobility that avoids the need for a home agent or a new protocol for binding updates entirely [76]. They issue each mobile host a permanent Class D IP multicast address [25] that serves as an end-point identifier. If multicast were widely deployed, this approach might hold promise; because a multicast EID has the benefit of being directly routable by the routing infrastructure, it removes the need for an explicit care-of address. Instead, it places the burden of managing updates of end point bindings squarely on the routing infrastructure. The binding issue remains the same, however. The mobile node must send a binding update—it just takes the form of a multicast group join message. Similarly, the home agent functionality is replaced by whatever entity is in charge of multicast tree rendezvous. In this scenario, the multicast distribution tree for a host's EIDs must be reconstructed each time a node moves, requiring an extremely agile and efficient tree-building protocol. A later proposal by Helmy [43] uses a traditional, non-multicast IP address as a mobile host's home address, but uses multicast for packet delivery. As noted by the designers, however, both schemes require a secure, robust, scalable, and efficient multicast infrastructure for a large number of sparse groups—a hypothetical protocol not yet available in the Internet.

2.2.7 Summary

Supporting changes in attachment point inside the network has one main advantage: end points need not be concerned with the current network attachment point of remote end points—the network will deliver packets appropriately regardless of an end point's current location. However, these host mobility schemes have several significant limitations:

1. Network-layer mobility schemes constrain the granularity of mobility. In particular, multiple, distinct applications and services may exist on a host with only one attachment point. Under the traditional IP addressing model, all of these end points share the same IP address; hence, network-layer mobility schemes would require these end points to move in concert. Increasing support for application migration [38, 63, 83] and service redirection [85, 119] ensures that the end points may in fact move independently, however. This fine-grained mobility requires each end point to have its own EID or home address, resulting in an increase in the number of addresses that must be managed by the IP routing infrastructure and severely stressing the scaling properties of many of the schemes. Indeed, the developers of the Amoeba operating system—which supported process migration—cite the need to name individual process end points as one of the main motivations for the development of FLIP [53], a network-layer protocol that can assign location-independent EIDs to individual processes. FLIP maps EIDs to network attachment points upon packet transmission.

2. Many network-layer mobility schemes incur unnecessary packet routing overhead (in terms of increased latency, additional bandwidth usage, or end point processing). Some mobility support schemes add additional packet addressing or routing overhead for all packets, regardless of their destination [18, 72, 76, 128]. In home-agent-based schemes, mobile end points incur overhead when attached to a foreign attachment, regardless of whether or not they ever move from that attachment point [41, 49, 89]. Ideally, overhead would only be incurred immediately proceeding, during, or following a change in attachment point—once an end point has "settled" at a new attachment point, an efficient scheme would treat it similarly to an end point that had never moved.

3. Since network-layer mobility schemes conceal changes in attachment point inside the network layer, it is often difficult for end points to detect mobility events, which often significantly impact their operation. For example, a TCP sender attempts to estimate the properties of the network path for the connection. A significant change in the network attachment point often implies that previously discovered path properties are invalid, and need to be rediscovered. This consequence is not limited to classical TCP congestion management. For example, many Internet services are replicated at various locations throughout the Internet; these services often attempt to serve clients from "near-by" servers (where "distance" is measured between the client's and server's attachment points). If either a client or a server changes attachment point, the service may wish to change the client-to-server assignments. Other applications perform even more sophisticated adaptation to changing network conditions. For example, Odyssey [81] demonstrated significant performance gains by allowing applications to adjust their *fidelity* in response to a change in available network bandwidth—a common side-effect of a change in attachment point.

4. Disconnection is not addressed. In particular, packets addressed to a currently disconnected mobile host are discarded. Hence, end points must be prepared to handle extended periods of disconnection that may accompany changes in attachment point.

In light of these limitations, many researchers continue to question whether network-layer mobility solutions are appropriate [21, 60, 83, 148] or sufficient [40, 45, 81, 128, 129, 145]. Cheshire and Baker examined the various network-layer mobility approaches available to a mobile Internet host [21] and noted that none were suitable for all classes of applications. Zhao, Castelluca, and Baker implemented a system to allow mobile hosts to select between mobility schemes (*i.e.*, utilize Mobile IP or not) on a case by case basis through a Mobile Policy Table at a mobile host [148]; the Mobile Policy Table specifies whether or not to employ network-layer mobility support between a particular pair of end points.

Regardless of their strengths or limitations, none of the network-layer mobility proposals have yet found widespread deployment. Hence, for the remainder of this dissertation, we will assume the absence of such schemes. Instead, we explore methods that operate in the absence of network-layer mobility and, in many cases, provide improved functionality and performance compared to the network-layer schemes described above. In particular, this dissertation addresses all four of the limitations listed above. Our approach:

- Allows end points to be arbitrarily fine-grained,
- Imposes minimal overhead on sessions that do not change attachment point, and almost all of the overhead is incurred upon a change in attachment point, not afterward,

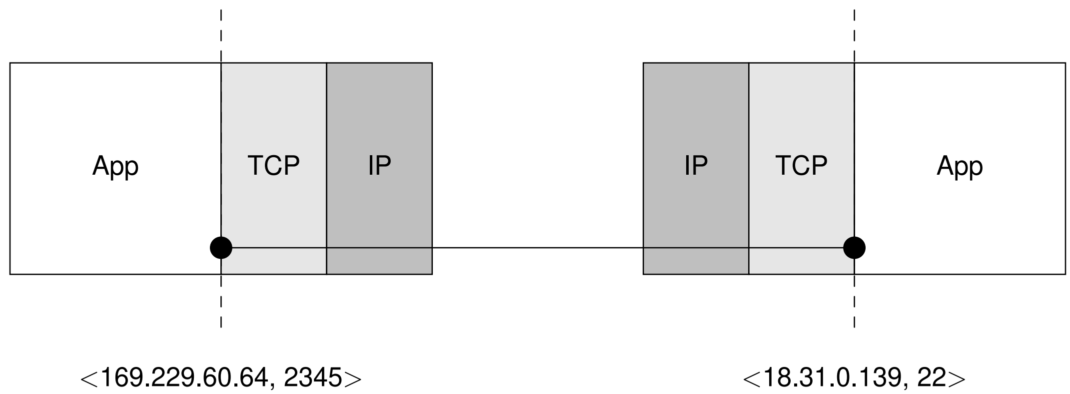

Figure 2-3: An Internet transport layer connection. In this example, a TCP connection has been established between applications at IP address 169.229.60.64 and 18.31.0.139; the connection uses port 2345 on the former and 22 on the later.

- Exposes changes in network attachment point to interested applications,
- Provides applications with sophisticated means of adapting to periods of disconnection, and
- Can function in conjunction with network-layer techniques when present.

2.3 Connection migration

When two end points wish to exchange data packets, they employ a *transport layer* protocol to manage delivery. Connection-oriented transport protocols establish a communication channel, or *connection*, between two end points and exchange packets over the connection; packets sent by one end point are delivered to the other, and *vice versa*. One of the main tasks of a transport protocol is to multiplex communication channels between end points—that is, to provide end points with more than one channel at a time and demultiplex incoming packets onto different channels. Some transport protocols may provide additional services such as reliability for connections. For example, Transmission Control Protocol (TCP) connections reliably deliver packets to applications in the order in which they were transmitted [97].

Conceptually, an application forms a connection between two end points. Due to the constraints imposed by the layered nature of the Internet stack, transport protocols must interface with the network layer using addresses understandable by that layer—IP addresses. Hence, connections in the Internet are communication channels between two network attachment points, *not* end points. Multiplexing is supported through the use of *ports;* a port is simply a tag used to separate incoming traffic into distinct channels. Hence, an Internet transport connection end point is specified in terms of a port on an attachment point, or ⟨*IP address, port*⟩ pair, as shown in Figure 2-3. This naming mechanism proves problematic if an end point changes network attachment point.

Once a connection is established between two attachment points, the local end point will only accept packets addressed from the remote attachment point. Hence, connections established by an end point at one attachment point cannot be used at another attachment point, *even* by the same end point. In order to continue communications between the same two end points, a new connection must be established between the new attachment points. Creating a new connection gives rise to two complications. First, each end point much discover a new attachment point and somehow communicate it to the remote end point. Second, after both end points agree on the new attachment points, the end points must abort the old connections and establish new ones. For reliable transport protocols, the process of changing attachment points may result in the loss of packets that were not yet successfully transmitted on the initial connections.

A number of researchers have proposed mechanisms to allow connections to adapt to changes in attachment points. The act of moving a connection from one attachment point to another is referred to as *connection migration*. Most connection migration proposals have focused on TCP, and have either introduced a higher-level mechanism to stitch together multiple separate TCP connections into one *virtualized* connection [60, 64, 82, 99, 145, 147] or extended the TCP protocol itself [37, 44, 129]. In addition, a number of recently-proposed transport protocols support connection migration as a standard feature [57, 126]. We briefly describe the essence of these approaches below.

2.3.1 Virtualized connections

Zhang and Dao proposed a Persistent Connection model for TCP where the connection end points are described in terms of location-independent EIDs [147]. In their model, a Persistent Connection exists between two end points, not their attachment points. The mappings between global EIDs and current network attachment points are stored in a global clearinghouse. When an end point changes network attachment point, it notifies the clearinghouse, which in turn notifies all end points in the system of the change in attachment point. To avoid unnecessary notifications, a mobile end point can provide to the notification service the set of correspondent end points to notify. Despite this optimization, Zhang and Dao observe that their global EID scheme continues to suffer from poor performance and scalability properties due to its reliance on a single, centralized clearinghouse.

In the Persistent Connection model, changes in attachment point are handled by a Persistent Support Module at each end point. The support module uses the new attachment points to establish replacement TCP connections for existing connections invalidated by the change in attachment point. Unfortunately, data that has not yet been delivered on previous connections is lost—violating TCP's reliable delivery semantics. Qu, Yu, and Brent later proposed a similar virtualization scheme that preserves TCP's delivery semantics [99]. In their Mobile TCP solution, end points use a proprietary interface to the operating system to extract from the original connection any bytes that have not yet been successfully delivered to the remote end point, which it then retransmits [100]. Okoshi *et al.* proposed a functionally similar solution called MobileSocket that does not require access to operating system buffers [82]. Implemented in Java, MobileSocket buffers all outgoing data at user level and establishes a separate control channel between connection end points. End points in MobileSocket exchange application-layer acknowledgments as the TCP connection progresses, allowing both ends to remove data from their MobileSocket buffers as it is successfully received at the correspondent end point.

In all three virtualization approaches—Persistent Connections, Mobile TCP, and MobileSocket—connection reestablishment is managed by a software library interposed between applications and the operating system. The library conceals the connection virtualization from the application, making it appear as if the original connection continues uninterrupted. This *interposition* approach was recently revisited by Reliable Sockets (Rocks) [145]. Similar to the previous approaches, Rocks allow TCP connections to support changes in attachment points. In contrast, Rocks preserve TCP's reliable delivery semantics without requiring any changes to the operating system or explicit application-layer acknowledgments, and safely inter-operates with end points that do not support Rocks. Rocks were developed concurrently with our work, and, indeed, many of the mechanics of Rocks implementation bear considerable similarity to the TCP virtualization approach described in Chapter 4. Unlike our approach, however, Rocks impose TCP-like delivery semantics on connections and cannot provide unreliable, out-of-order, or unconnected delivery semantics such as those provided by UDP.

The home agent approach has also been applied at the transport layer. MSOCKS [64] proposes using a SOCKS *proxy* [61] to forward transport connections to a mobile end point. Correspondent end points establish *front-end* TCP connections with the proxy; the proxy then establishes a separate, *back-end* connection with the mobile end point. The proxy *splices* the two connections together, copying all incoming traffic on the front-end connection to the back-end connection and *vice versa*. If an end point moves, it establishes a new back-end connection with the proxy, which splices it to the old front end connection. Hence, the mobile end point's change in attachment point is concealed from the correspondent end point. As before, the SOCKS library conceals the connection virtualization from applications on the mobile end point. NetMotion has a similar commercial product that uses a proprietary proxy to support host mobility [79].

2.3.2 Modified TCPs

Some researchers have proposed modifying TCP itself to support changes in attachment point. Huitema proposed ETCP [44], an extended TCP protocol that includes a flow identifier in the TCP header. By assigning each TCP connection a unique *connection identifier*, end points can associate incoming packets with the appropriate connection regardless of what attachment point was used to transmit them. Hence, while mobile hosts continue to require a home agent to forward along initial packets from a new remote end point, route optimization can be performed implicitly by the correspondent end points. Correspondent end points respond to mobile end points by simply addressing subsequent packets to the address used most recently by the mobile end point. If the mobile host fails to respond, packets can always be retransmitted to the home agent.

An alternative TCP extension, TCP-R sends explicit attachment point updates to correspondent end points [37]. If a mobile end point using TCP-R changes attachment point, it sends a special "Redirect Request" or `RD_REQ` message containing the IP address of its previous attachment point and the IP address of its current attachment point. Because of TCP-R's explicit specification of IP addresses and inability to change port numbers, it cannot work across NATs. Furthermore, many firewalls and stateful proxies may not properly handle redirected TCP-R connections because they do not conduct a traditional TCP connection establishment between the new attachment points. We present a TCP connection migration scheme in Chapter 4 that remedies these deficiencies as well as several others. After its original development as an approach to handle host mobility [120] and subsequent extension to support service fail-over [119], other researchers adapted (*e.g.*, M-TCP [129]) and extended (*e.g.*, support for concurrent migration [136]) our approach.

2.3.3 New transport protocols

A number of proposed transport protocols have begun to incorporate rudimentary support for mobile or even multi-homed end points but have not yet found wide-spread acceptance. The Datagram Congestion Control Protocol (DCCP) [57] allows end points to notify correspondent end points of a new network attachment point. An end point simply sends a special *DCCP-Move* packet from its new attachment point, and the remote end point addresses all further packets on that connection to the new attachment point; further packets from the previous attachment point are ignored. As noted by DCCP's authors, however, "DCCP's support for mobility is intended to solve only the simplest multi[-]homing and mobility problems. For instance, DCCP has no support for simultaneous moves. Applications requiring more complex mobility semantics, or more stringent security guarantees, should use an existing solution like Mobile IP or [the one presented in this thesis]." [57, pp. 52]

The Stream Control Transport Protocol (SCTP) [126] also allows end points to change attachment points. It further allows end points to use multiple attachment points simultaneously. In its current form, however, SCTP requires end points to specify all the attachment points they wish to use during the duration of a connection at connection establishment. A proposed extension [125] allows end points to introduce new attachment point bindings on-the-fly (*i.e.*, after a change in attachment point) in order to make SCTP more useful in mobile environments [105].

2.4 Session abstraction

Many Internet applications are architected using the notion of *sessions*—long-term relationships between application end points that typically span multiple transport connections. Examples of this approach include interactive Web sessions, file transfer applications, interactive log-in sessions, and multi-modal conferencing sessions. Sessions typically consist of one or more transport connections and provide a convenient abstraction with which to manage coordinated application state between end points. In general, a session can encompass any number of end points; in this dissertation, we will consider sessions encompassing exactly two end points.

Application-layer protocols often use the session abstraction to efficiently manage application state relevant to multiple connections. Similar to connections, sessions create a context for packet exchange that allows end points to coordinate state relating to the packets they are exchanging. While connection state is restricted to issues of sequencing and retransmissions with a particular connection, applications often use session state to provide a framework for additional services that may span multiple connections, such as checkpointing, compression, authentication and confidentiality (*e.g.*, TLS [27]), unified congestion management (*e.g.*, the Congestion Manager (CM) [7]), and multi-party conferencing (*e.g.*, the Session Initiation Protocol (SIP) [108]). Some sessions last only long enough to send a short packet in one direction; others last for much longer periods of time during which the end points exchange a significant amount of information. As the length and complexity of a packet exchange increases, the session abstraction becomes increasingly powerful. Providing enhanced services via a session abstraction often results in resource savings; by defining service parameters at session establishment, end points amortize the cost of any required negotiation across the duration of the session. Other session services enhance the user experience. For example, SSH agents [144] manage user credentials for the duration of an interactive session. A user authenticates herself to the agent only once (*e.g.*, by typing a password); the agent then performs all further authentication requests on behalf of the user, freeing her from tedious, repeated password entry.

The Internet protocol stack provides no explicit support for the session abstraction. The now-defunct Open Systems Interconnection (OSI) communications model [149], however, defines an explicit *session layer* to provide synchronized message exchange [48]. In addition to the setting up and tearing down of session associations, token exchange, and duplex negotiation, the OSI session layer allows for the definition of synchronization points within the session, the interruption of a session, and session resumption from an agreed upon synchronization point.

2.4.1 Session-layer mobility

Because of the inability of connections to cope with changes in end point attachment point, applications have been forced to develop their own application-specific mechanisms for handling mobile end points. Many applications have found it convenient to leverage the session abstraction to coherently aggregate multiple connections from disparate network attachment points into one relationship. HTTP cookies [35] for Web-based applications is a good example of this approach: By including HTTP cookies in both HTTP requests and responses, Web clients and servers exchange session state that survives the termination of a particular HTTP exchange or transport connection.

In fact, researchers have previously proposed providing mobility as a general session-layer service [60, 139], although the design and implementation are significantly different from our *Migrate* architecture. One of the main issues in designing a session-layer mobility scheme—indeed, *any* mobility scheme—is deciding how to identify the end points. In the Session Layer Mobility (SLM) scheme [60], end points use a new global naming service—a so-called User Location Server—to provide an end point's current attachment point; end points change connection attachment points through an undocumented TCP-specific protocol extension, presumably similar to the virtualization approaches described above. Other researchers have proposed extensions to the IETF-standardized Session Initiation Protocol (SIP) [108], originally developed to help establish telephone and conference sessions, to track end-point attachment points [139]. SIP uses email-like addresses (such as `snoeren@lcs.mit.edu`) to define end points.

2.4.2 End-point naming

By *binding* end-point names to IP addresses, a *naming system* allows an end point to be identified by a more descriptive name than its network attachment point [109]. An end point wishing to establish a session with another end point must first resolve the remote end point's name into an IP address to use for communication. A session is said to be *bound* when the names of its end points are resolved to network attachment points.

Networked applications have long eschewed IP addresses in favor of their own naming systems to describe remote end points. For example, Internet applications often refer to end points using Domain Name System (DNS) hostnames (*e.g.*, `www.lcs.mit.edu`) [68], service records (*e.g.*, DNS "MX-records"), or content-based schemes like Intentional Naming System (INS) intentional names [1] and distributed hash table keys [127]. Each of these naming systems uses a different namespace, but they all provide a *resolution* mechanism that returns the binding for a particular name (*e.g.*, DNS will resolve `www.lcs.mit.edu` to an IP address like 18.24.10.46). The naming system resolves the end-point name used by the application to an IP address that specifies a particular network attachment point before packets are sent. The IP address can then be used to deliver data to the network attachment point used by the end point of interest. In some cases, a name may resolve to multiple addresses; the semantics of multiple bindings are application-specific, but in most cases any of the addresses can be used.

The end-point binding stored in the naming system could become out of date because the named end point is no longer located at the network attachment point specified by the naming system. Such a binding is termed *inconsistent*, since it is no longer consistent with the current mapping between the end point and its network attachment point. *Dynamic* naming systems like dynamic DNS [141] allow the replacement of inconsistent bindings with new, up-to-date bindings reflecting an end point's current network attachment point.

Naming systems use widely varied methods to store and retrieve name bindings. Internet naming systems usually use a distributed set of resolvers (often called *name servers*) that store (or are responsible for computing) subsets of the mappings. An end point that wishes to resolve a particular name must then contact these name servers across the network. Most end points attempt to avoid the cost of repeated queries (both in terms of time and consumed bandwidth) by caching bindings. Hence, multiple packets destined for the same end point are addressed using the IP address resulting from only one resolution.

Cached bindings may become inconsistent in a dynamic environment. Despite the ability of a dynamic naming system to update its bindings to reflect changes in an end point's network attachment point, end points using a cached binding will remain oblivious to the change. Hence, some mechanism must be employed to invalidate the inconsistent cached binding and re-resolve the end point name to its new network attachment point.

2.4.3 Avoiding inconsistency

In general, there are at least three ways to avoid binding inconsistency:

- The first approach is for applications to use a *late binding* technique, where the end-point name is not resolved to a network attachment point until as late as possible—just before a packet must be sent—and is *not cached*. In its simplest form, the remote end point's name is freshly resolved for each new packet to be sent.

- A second approach is for applications to resolve end-point names at the beginning of a session and cache the bindings for the duration of the session. This technique introduces a window of vulnerability when session bindings can become inconsistent, but the window can be kept small by using only short sessions. When sessions are sufficiently short, and the interval between them correspondingly large with respect to the frequency of changes in attachment point, an end point is unlikely to change network attachment points during a session. Hence, a binding obtained at the beginning of a session remains consistent throughout the session's lifetime.

- The third approach is for end points to asynchronously update the end-point bindings upon changing network attachment points. Rather than periodically polling for new bindings as described in the first approach, a *binding update* can be used to notify end points of the inconsistency of the previous binding, and to provide the current, consistent binding.

Each approach is most appropriate under different conditions, depending primarily on the frequency with which end points change attachment points. Researchers typically consider two classes of mobility and describe them in terms physical displacement, but they correspond equally well to frequency of change:

- *Micro-* or *link-local* mobility refers to the case when end points move inside a confined local area, typically behind a single base station or access point, generally using the same network technology and provider. Time scales can be quite small, even on the order of a round trip time (RTT).

- *Macro-* or *wide-area* mobility, in contrast, deals with the more general case of movement across subnets, providers, and network technologies. Changes typically occur on coarser time scales (*i.e.*, 10s of RTTs or more) than micro-mobility.

Late binding avoids end-point binding inconsistency by delaying resolution until the last possible moment and re-resolving the binding at every possible opportunity. Hence, late binding is appropriate in the micro-mobility case, when end points change attachment point frequently. Late binding was employed by Adjie-Winoto *et al.* in the Intentional Naming System (INS) [1]. INS integrates name resolution and message routing to track highly mobile services and nodes. The TRIAD architecture [40]) proposes a similar approach. Similarly, wireless networks often conceal extremely rapid changes in attachment point across a homogeneous link technology from the network layer through late binding at the physical and link layers (*e.g.*, via link-layer bridging and roaming [46]).

The significant polling overhead typically incurred by late binding can be avoided, however, when changes are less frequent (*i.e.*, multiple consecutive resolution operations return the same binding). When attachment points are stable enough to allow bindings to remain consistent for some period of time, end points can resolve the remote end point name only at the beginning of a session and keep session lengths short; the likelihood of an end point changing network attachment point during the session is low. Unfortunately, this approach eliminates much of the benefit of sessions—the shorter the session duration, the less benefit there is to negotiating session services, as their costs will not be favorably amortized. While there may be little penalty for those applications that do not use session services, there remains the possibility that an end point may occasionally change attachment points during even short sessions, leaving the session in an inconsistent state.

Studies suggest that Internet end points do not move frequently in the wide area, even when hosts are mobile [131]. Binding updates are the best choice in this case, as they use caching to avoid the polling overhead of late binding but enable sessions to recover from the occasional change in attachment point. (In situations when attachment points change rapidly in a confined area, such as a cellular network, and the binding update overhead may become excessive, binding updates may be usefully combined with late binding micro-mobility approaches [16].)

2.4.4 Managing updates

Issues with a binding update scheme include determining when, to whom, and from where to send updates. Clearly, end points should issue binding updates immediately following a change in attachment point. With regard to whom to send updates, end points can be classified into two categories with respect to a mobile end point: those that have cached the end point's binding, and those that have not. The first class must receive a binding update; the second, however, need not be notified so long as it can be assured that any future bindings will be resolved to the new network attachment point. If bindings are resolved by a naming system that allows dynamic updates, it is necessary only to notify those end points that have cached the mobile end point's binding; updating the naming system suffices to handle the rest.

The last issue is determining from where the binding updates should be sent. Since updates need to be sent to the set of end points currently in communication with a mobile end point, the mobile end point itself is in the best position to generate the updates. A mobile end point can *directly notify* all correspondent end points upon any change in attachment point by sending them a binding update. Directly updating correspondent end points of changes in attachment point has two advantages:

1. The binding update *shares fate* with session connectivity. The binding update can be delivered to the remote end point if and only if the session can continue. An end point may not always have connectivity to the naming system; hence, direct binding updates ensure that established sessions can continue if at all possible.

2. Direct binding updates can be *implicit*. The updates need not explicitly specify a new attachment point; instead, the attachment point may be inferred from source IP address of the binding update itself. This implicitness is especially important on the Internet when NAT is

used to provide global connectivity for private networks [24]. For example, packets originating from a network attachment point with an IP address from a private address space [103] may be rewritten by a NAT to have a globally-routable address before being forwarded to the Internet. Hence, the end point using that attachment point believes it to have one IP address (*e.g.*, 192.198.1.10) while remote end points must address packets to it using a different IP address (*e.g.*, 24.147.17.155).

Further, some network address translation schemes (*e.g.*, AVES [80]) are not "stable," meaning the network attachment point used to communicate with one remote network attachment point cannot be used to communicate with a different remote attachment point. In this case, the IP address seen by each correspondent end point may be different from those seen by other correspondent end points. Hence, the only way to ensure that end points determine the correct IP addresses is to send binding updates directly between the two communicating end points.

The major limitation of direct updates is the inability to recover from concurrent changes in the attachment points of both end points. If an end point changes attachment point before the delivery of a binding update from the remote end point, neither end point has an up-to-date binding of the other end point. The end points cannot recover without a third party to broker the binding updates. While a reachable third party is strictly necessary in this case, the naming system suffices. The end points can recover the session by re-resolving the remote end point using the same naming system used during session establishment.

Assuming that each end point resolves a remote end point's name at the beginning of a session and caches the binding for the duration of the session, the set of end points that have cached a given mobile end point's binding is *at least* the set of end points currently engaged in a session with the mobile end point. Because of a race condition between naming system updates and resolutions, there may be additional end points that have already resolved the mobile end point's name to its previous attachment point but not yet succeeded in initiating a session. These end points will not see any updates to the naming system.

When the naming update arrives after the binding has been resolved by the naming system, the querying end point will unsuccessfully attempt to initiate a session to the remote end point's old attachment point. End points can keep the number of such cases small by performing resolution at the last possible moment and not caching bindings across sessions. Ultimately, to ensure correctness, end points should attempt to re-resolve a remote end point's name if initial session establishment fails. In practice, the trend toward dynamic naming systems has already caused such retries to find their way into applications—for instance, current FreeBSD telnet and rsh applications try to contact alternate network attachment points if the naming system returns multiple bindings for an end point.

2.4.5 Controlling change

So far, we have assumed that changes in attachment point were an inevitable occurrence, and have not explored why they happen. In practice, deciding when an end point should change its attachment point, and where it should move to, are complicated issues. Often, these decisions are based on user policy rather than network constraints (*e.g.*, a user may prefer a less expensive network attachment point or one with higher bandwidth). This decision is especially complicated for multi-homed end points, as different attachment points often have varying characteristics.

Inouye, Binkley and Walpole observe that the metrics of interest are application-dependent, and vary greatly both in terms of dimension (*e.g.*, bandwidth, latency, loss rate) and acceptable ranges. They propose Physical Media Independence (PIM) [45], an architecture for supporting multiple physical interfaces on multi-homed Internet hosts. PIM supports policy-based attachment point selection but relies on Mobile IP to provide network-layer mobility support for applications unable to explicitly handle address changes.

As an additional complication, an end point may wish to select a new remote end point when the local network attachment point changes—perhaps because the new network attachment point is ill-suited for continued communication with the previous remote end point (*i.e.*, the network path has limited bandwidth, high loss rate, or long latency). This intervention is particularly appropriate for end points communicating with replicated servers. In such cases, clients may often select between multiple potential remote end points, some more capable of providing efficient service to particular network attachment points than others. Hence, a practical session abstraction must incorporate a notion of connectivity quality and allow for arbitration between multiple potential network attachment points, both local and remote.

2.5 Disconnection

One of main the challenges for applications operating in a mobile environment is an end point that disconnects [111]. A significant amount of research has focused on allowing mobile clients to continue to function while disconnected. Applications that do not require network connectivity clearly require no additional support. Other applications do not explicitly utilize network resources themselves but access files stored on a networked file system. In this case, the networked file system must support disconnected operation. For example, the Coda file system [74] allows disconnected clients to use locally cached copies of files and reconciles any conflicts upon reconnection. Coda proactively *hoards* copies of files that are likely to be needed while disconnected, allowing most file operations to proceed even while disconnected.

Network applications—those that explicitly communicate with remote application end points—require a far greater level of support. In some instances, applications can defer communication yet continue to provide a user with some level of functionality. In particular, applications based on the Remote Procedure Call (RPC) model [14], where each communication is a request-reply exchange, have been successfully adapted for disconnected operation using the Rover toolkit [51], which queues RPC for later delivery. In addition to queuing RPCs, the Rover toolkit can emulate Coda's hoarding process; instead of copying or relocating files, Rover relocates the remote session end point to the mobile host through the use of dynamic objects. When both end points are located on the same mobile host, no network communication is needed. Similar ideas appear in the HTTP-based Mobile Extensions proposed by Dahlin *et al.* [23], which allow HTTP session end points to be hosted at proxies throughout the network, or in the case of disconnection, at the mobile client itself.

2.5.1 Suspend/resume

Unfortunately, the hoarding approach does not apply when the remote end point cannot be cached or relocated to the disconnected host. We are particularly interested in session-based applications whose remote end points cannot be easily relocated. Hence, we focus on resuming the session once connectivity is reestablished. In our model, sessions are *suspended* when the end points become disconnected, and *resumed* when connectivity returns.

To enable a form of suspend/resume operation, some network-layer mobility schemes [128] and connection migration proposals [37, 60, 82, 145] conceal periods of disconnection from applications. In addition to being problematic to implement, concealment often yields sub-optimal performance, and results in significant amounts of wasted resources. In particular, concealing disconnectivity becomes extremely difficult when applications require periodic communication or keep-alive messages. Further, if concealment is successful, the application is unable to adapt to changes in network conditions and continues consuming system resources (CPU, memory, kernel buffers, timers, file descriptors, etc.) while disconnected. Many of these resources are scarce, and cannot be efficiently multiplexed.

Systems like Rover and Coda assume an asymmetric resource balance between client and server, and view "servers being the true home of data and clients merely being caches." As an increasing portion of Internet hosts become mobile and themselves resource-poor, and the notion of peer-to-peer computing expands, this asymmetry assumption becomes increasingly tenuous. It is becoming unreasonable to assume the correspondent end point of a mobile end point is neither mobile nor resource poor. Hence, we consider resource conservation on both of the hosts involved in the session.

Applications are traditionally suspended by creating snapshots, or *checkpoints*, of their process execution state, which can later be restored in order to resume process execution from the same state. This technique has been applied to migrate applications from one host to another [63], or to restore applications after a system crash. Unfortunately, many applications handle several sessions inside of one process; traditional process-based checkpointing does not allow individual sessions to be independently suspended or resumed.

2.5.2 Session management

Researchers have proposed several specialized techniques to enable the management of individual sessions within a process. For example, the Java Servlet Specification v2.3 [130] supports the notion of explicitly storing application state inside session data structures, which can be individually passivated, resumed, and shuttled between replica servers using the native Java serialization and RMI mechanisms. Servlet sessions include only application state, however, and do not reference any network connections or system resources (*e.g.*, files, locks, timers, etc.) that may be needed.

In contrast, Resource Containers [10] and Scout paths [71] both provide mechanisms to associate application and system resources, allowing system resources to be charged to individual sessions. IO-Lite [86] goes one step further by blurring the distinction between system and application state. Network buffers in IO-Lite are at once application and system state—only one physical copy of transmitted data exists in memory. None of these approaches allow system resources to be suspended or removed from active use. Hence, while they may be charged to a particular application session, they cannot be safely released when a session is suspended.

2.5.3 Pervasive computing platforms

Mobility and disconnectivity support are two aspects of the larger vision of pervasive computing in which both communication and computation migrate across heterogeneous platforms. In order to support such powerful operations, pervasive computing platforms typically define specific programming and inter-process communication (IPC) models. *One.world* [39] defines a Java-based environment in which to build pervasive applications and supports both host and fine-grain application or session mobility through the use of specific RPC mechanisms [38]. *One.world*'s tuple spaces allow names to be dynamically bound to different values, depending on the current environment; all bindings are invalidated when end points change location and are resolved again from the new location. This dynamic binding is similar to the concept of contextual objects, introduced by

Kermarrec *et al.*, which allow one name to correspond to a variety of data objects based upon the current context [56]. This concept also exists in Active Names, which map names to a chain of mobile programs that can customize how a service is located [138].

The main drawback of pervasive computing platforms is that they require a complete redesign of existing applications. This dissertation focuses on providing many of their benefits, in terms of support for mobile and disconnected session end points, while preserving the traditional POSIX API. By doing so, legacy applications can realize much of the benefits and programmers can design new, mobile-aware applications in traditional, session-oriented style by leveraging our expanded API.

A worker may be the hammer's master, but the hammer still prevails.
A tool knows exactly how it is meant to be handled,
while the user of the tool can only have an approximate idea.

— Milan Kundera

Chapter 3

A System Session Abstraction

This chapter presents the first component of the *Migrate* mobility architecture: a system session abstraction. *Migrate* elevates the session relationship from an internal application construct to a first-class entity described by the application but managed and maintained by the system. Applications employ a novel Application Programing Interface (API) to name two session end points using any naming system of their choice, and *Migrate* preserves the relationship and communications between them in the face of any changes in network attachment point. This preservation is a two step process: First, the session relationship must be maintained by ensuring each end point has an accurate understanding of the other end point's current network attachment point. Second, any established communication channels between the end points must continue to operate in the face of changes in attachment point. This chapter focuses on maintaining the session relationship; preserving communications is discussed in the following chapter.

The rest of this chapter is organized as follows. It begins by introducing *Migrate*'s session abstraction, its API, and accompanying control protocol in Section 3.1. Section 3.2 discusses the additional security concerns introduced by the session layer. The chapter concludes in Section 3.3 with a demonstration of how *Migrate* sessions enable *host* mobility.

3.1 A session layer

Migrate introduces a session layer to the Internet protocol stack. This layer presents a simple abstraction—a *session*—to the application to handle changes in network attachment points. In *Migrate*, a session is an association between two communicating end points, consisting of one or more connections. Figure 3-1 shows a session consisting of three connections. Rather than name end points by their attachment points—as connections do with IP addresses—or prescribe some novel naming system, *Migrate* allows applications to name session end points using an arbitrary naming system of their choice (*e.g.*, Domain Name System (DNS) hostnames); they simply provide *Migrate* with the names and a method to resolve them. *Migrate* sessions track the end points as they change network attachment points, maintaining the end-point relationship requested by the application.

Migrate separates the task of initial network attachment point resolution from end point tracking, or session *maintenance*. *Migrate* employs an application-chosen naming system to initially resolve end point names to attachment point addresses; end points adapt to changes in session attachment points themselves. This separation is key to *Migrate*'s ability to support flexible end point naming

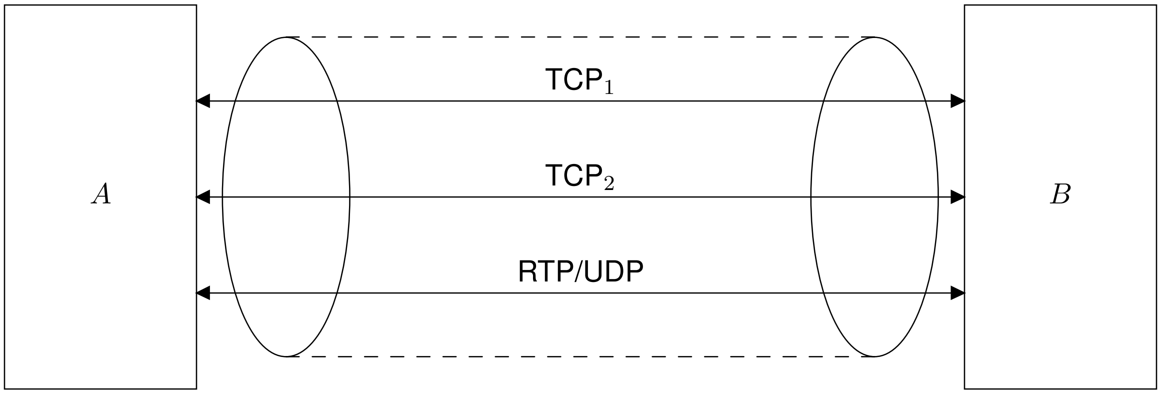

Figure 3-1: A session between end points A and B containing three separate connections: two TCP connections, TCP_1 and TCP_2, and an RTP/UDP stream.

Method	Description
Session **session_create**(int fd, int flags)	Create a session
Session **get_session**(int fd)	Retrieve a connection's session
int **session_close**(*Session* s)	Destroy a session
void **set_lookupfunc**(*Session* s, *LookupFunc* f)	Set remote lookup function
void **set_lookupname**(*Session* s, *LookupFunc* f)	Set local lookup identity
int **register_handler**(*Session* s, *Handler* h)	Register a mobility handler
int **migrate**(*Session* s, struct sockaddr *addr)	Migrate to new attachment point
int **add_connection**(int fd, *Session* s)	Add a network connection
int **remove_connection**(int fd)	Remove a network connection

Table 3.1: Session API exported by the *Migrate* session layer

while efficiently managing changes in attachment point; *Migrate* leverages existing dynamic naming systems yet enables robust, efficient session maintenance—end points can change attachment points even in the absence of a reachable naming system. *Migrate* performs session maintenance between the communicating peers themselves using a novel session control protocol.

3.1.1 Session API

Applications specify their notions of a session by explicitly joining together related transport-layer connections (or destinations in connection-less protocols). Sessions are instantiated using the `session_create()` function. `session_create()` takes two parameters: an initial, established connection to become part of the session, and a set of flags, which specify various default behaviors listed in Table 3.2 and are discussed as appropriate below. Since the connection has already been established by the application, it identifies the desired remote end point. Once established, a session is identified by a locally-unique token or *Session ID*.

The session layer exports a session abstraction to the application, manages constituent connections as a group, and adapts to changes in network attachment points as needed. These adaptations include tracking the remote end point as it changes network attachment points, moving the local end point to another attachment point if the current one becomes unusable, and suspending the session entirely if it becomes disconnected. The conditions under which *Migrate* performs these actions on a particular system are dictated by a policy file, discussed in Chapter 6 and detailed in Appendix A.

Flag	Meaning
M_ALWAYSLOOKUP	Rebind the remote end point on any change in attachment point
M_AUTOCLOSE	Terminate the session when all constituent connections have closed
M_DONTMOVE	Do not automatically adjust to changes in local attachment point availability

Table 3.2: The flags that may be passed to a `session_create()` call. `M_ALWAYSLOOKUP` and `M_DONTMOVE` may not be passed simultaneously.

The selection of local network attachment point (in the case of multi-homed end points) and connection transport protocols remains completely under the application's control. Applications retain control over the characteristics of their transport connections (*e.g.*, port number, protocol options, buffer size, etc.) regardless of whether or not they are part of a session.

Figure 3-2 shows how a typical network application would use the session abstraction. Applications tell *Migrate* how to name remote end points through the `set_lookupfunc()` call. *LookupFunc* is a structure (shown in Figure 3-3) containing both a naming system resolver and an end-point name (to be passed into the function)—*e.g.*, `gethostbyname()` and `foo.bar.com`. Normally, when an end point changes attachment point, *Migrate* resumes sessions by contacting correspondent end points at the same IP addresses as before. Occasionally, however, both end points may move simultaneously causing such a resumption to fail. In that case, *Migrate* invokes the resolution function provided by the application to refresh the remote end point binding.

In many client-server applications, the client must be able to initially locate the server, but it is not usually necessary for the server to asynchronously locate the client using a naming system. Hence, such applications typically do not explicitly exchange any end point information other than IP address. In cases when both the client and the server may change attachment points, however, this name exchange becomes necessary as the server may need to invoke the resolution function if the attachment points change simultaneously. In such instances, a client may call `set_lookupname()` with its own name, which is passed to the remote end point upon session establishment. This identity is then used at the remote end point as an argument to the registered *LookupFunc* function if necessary.

Some naming systems use the same name for multiple attachment points and resolve the binding differently based upon the network attachment point of the end point requesting resolution [2]. This technique is often used to direct clients toward a particular network attachment point of multi-homed end points; end points with network attachment points in one portion of the network receive a binding to one network attachment point, while end points with network attachment points in another portion of the network will receive a different binding. Typically, the naming system arranges to return a binding to the network attachment point that can provide the “best” service where “best” is based on some particular application-specific metric. Hence, an end point may wish to consult the naming system to reevaluate its choice of remote attachment point after any change in local attachment point. In that case, applications can pass the `M_ALWAYSLOOKUP` flag to the `session_create()` call. *Migrate* will then rebind the remote end point anytime the local attachment point changes regardless of whether the remote end point has changed attachment point or not.

In some cases, end points may wish to take application-specific action upon changes in attachment point, either local or remote. Hence, *Migrate* provides applications with the opportunity to register a *handler* function, which *Migrate* invokes any time a session end point changes attachment point.

```
char hostname[256];
char name[] = "foo.bar.edu";
int fda, fdb;
struct hostent *dhost;
struct sockaddr_in *daddr;
migrate_session *sid;
migrate_lookupfunc lf;
migrate_handler mobhandler;

/* Deterine local hostname */
gethostname(&hostname, 256);

/* Find a remote end point */
dhost = gethostbyname(name);

/* Validate remote end point */
daddr = valid_address(dhost);

/* Establish a connection */
daddr->sin_port = htons(PORTA);
connect(fda, daddr, sizeof(struct sockaddr));

/* Create a new session including connection fda */
sid = session_create(fda, flags);

/* Specify end-point discovery mechanisms */
lf->func = gethostbyname;
lf->arg = &name;
set_lookupfunc(sid, lf);
lf->arg = &hostname;
set_lookupname(sid, lf);

/* Establish a second connection */
daddr->sin_port = htons(PORTB);
connect(fdb, daddr, sizeof(struct sockaddr));
add_connection(fdb, sid);

/* Register interest in changes */
register_handle(sid, mobhandler);
```

Figure 3-2: A sample *Migrate*-aware application using the session abstraction

```
struct migrate_lookupfunc_t {
  struct hostent * (*func)(const char *); /* Function to call */
  const char *      arg;                   /* Parameter to pass */
};

typedef struct migrate_lookupfunc_t LookupFunc;
```

Figure 3-3: The C type signature of a *Migrate LookupFunc* structure

Flag	Meaning
M_LOCAL	There has been a change in local attachment point
M_REMOTE	The remote end point changed attachment point
M_INSTANT	The change in attachment point appeared instantaneous

Table 3.3: The flags that may be passed to a *Migrate* handler function

Applications register a handler function with the `register_handler()` call. *Migrate* calls a handler function with two parameters: the *Session ID* of the session experiencing a change in attachment point and a set of flags. Table 3.3 lists the flags and their meaning.

The first two flags are set according to which end point experienced a change in end point—remote, local, or both. Applications interested in learning the new location of an end point after session resumption can do so through the standard operating system calls (*e.g.*, `getsockname()` and `getpeername()` on UNIX platforms). In many instances, a change in attachment point appears instantaneous in that the first notification that the remote end point is no longer at its previous attachment point is a binding update describing the end point's new attachment point. In such instances, *Migrate* sets the `M_INSTANT` flag.

In other cases, however, an end point may disconnect for some period of time, meaning it is no longer reachable at its last known attachment point, but has yet to update either the local end point or the naming system with its new location. Because certain applications may want to know of the disconnection as soon as possible, *Migrate* invokes the registered handler as soon as *Migrate* detects disconnection. For example, a streaming media server application may wish to begin buffering live streams for later replay. *Migrate* provides a number of services to assist applications in dealing with periods of disconnectivity. These extensions to the session abstraction are presented in Chapter 5. Unless an application takes special action using these extensions, *Migrate* invokes the handler again when session connectivity is restored.

Normally, *Migrate* manages changes in both local and remote attachment points for applications. On occasion, however, applications may wish to explicitly move sessions from one local attachment point to another. This functionality is provided through the `migrate()` call. Applications can pass the name of an alternate local attachment point to use, just as they would in a standard connection `bind()` call. Applications that wish to manage local end point selection entirely by themselves may pass the `M_DONTMOVE` flag on session creation, which will prevent *Migrate* from reacting to any changes in local attachment point availability. When the `M_DONTMOVE` flag is set, *Migrate* suspends sessions using attachment points that no longer provide the needed connectivity. This flag overrides the default behavior specified by system policy, which would typically migrate such sessions to another attachment point.

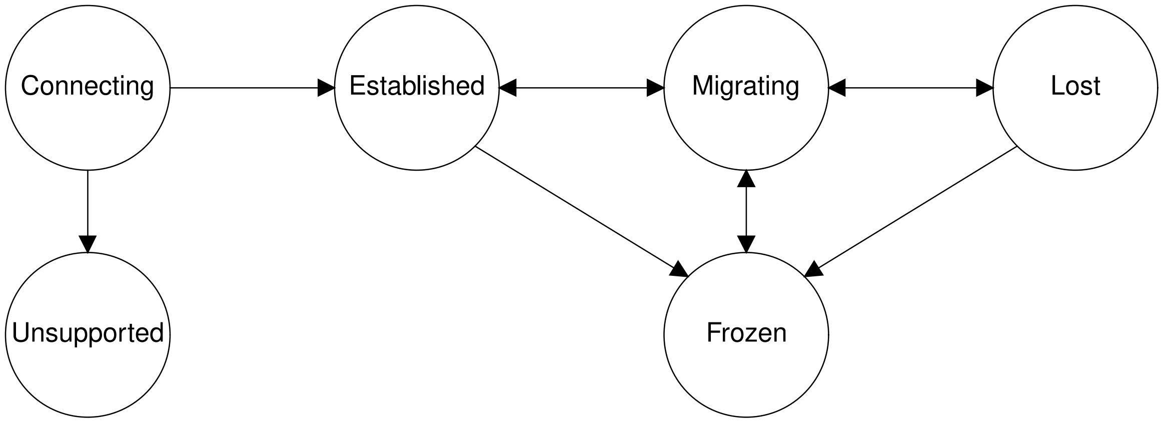

Figure 3-4: The session Finite State Machine (FSM). Sessions cannot be migrated or suspended until they are successfully established.

In order for the *Migrate* system to manage session communication, it must know which network connections are associated with each session. Applications describe this relationship using the `add_connection()` call. To support legacy applications, if a socket has not been associated with a session before being connected, *Migrate* transparently instantiates an anonymous session and associates the new connection with it.

Connections that are no longer associated with a session (either because the connection has terminated or now belongs to another session) can be removed through the `remove_connection()` call. Removing all of the network connections in a session does not necessarily close the session, as application semantics may dictate that a session span multiple, non-overlapping network connections. If, however, the application desires that session share fate with its network connections, it can pass the `M_AUTOCLOSE` flag to `session_create()` when the session is instantiated; the session will then be terminated once all its constituent connections finish. This flag allows *Migrate*-aware applications to pass network connections to non-aware applications and not need to perform any cleanup after termination. Regardless of the `M_AUTOCLOSE` flag, applications may explicitly close a session with the `session_close()` call. Any and all open connections contained within the session are closed simultaneously.

3.1.2 Session control protocol

Migrate manages session establishment, tear-down, and maintenance through a session control protocol. The main task of the session control protocol is to manage changes in network attachment point. When an end point changes attachment point, it must notify the remote end point of all open sessions of its new network attachment point.

The main difficulty with updating the remote end points is authenticating the end points—ensuring that the end point requesting the update is the original end point. Verifying an update is much simpler than authenticating the end point. *Migrate* does *not* provide authentication; it relies on the application to authenticate the original end point. Instead, *Migrate* ensures that update requests can be made only by the original end point (or one authorized by that end point). Security is discussed in detail in Section 3.2.

Sessions can be in one of several possible states (shown in Figure 3-4), depending on *Migrate*'s understanding of the remote end point's attachment point. A session starts in the *connecting* state as the initiating end point attempts to establish a session control channel with the remote end point. If *Migrate* successfully establishes a control channel, both end points exchange cryptographic keys that will be used to secure any further updates to the session. The session then moves to the *established* state, and communication between the application end points proceeds. Alternatively, if *Migrate* negotiation fails, the session transitions to the *unsupported* state, and the session cannot survive changes in network attachment points.

Upon notification of a change in local attachment point, *Migrate* moves the session into the *migrating* state and attempts to deliver a binding update to the remote end point. In order to assure the remote end point of the validity of the request, *Migrate* signs the binding update using the previously negotiated cryptographic key. Upon receipt of the update, the remote end point first verifies the signature. If the signature verifies, *Migrate* honors the request by updating the network attachment point bindings and migrating the associated connections. Upon acknowledgment of its request and successful connection migration, the mobile end point moves the session back to the *established* state. Chapter 4 presents the details of connection migration.

On occasion, however, a mobile end point may be unable to contact the remote end point either because it lacks connectivity to the remote end point's attachment point from its new network attachment point or because the remote end point is not responding. In the latter case, the session moves into the *lost* state and *Migrate* rebinds the remote end point through the resolution mechanism provided by the application. Upon successful completion of the resolution operation, the session returns to the *migrating* state and resumes as before. If, at any point in the process, *Migrate* is unable to return to the established state due to lack of connectivity, failure of the supplied naming system, or local policy restrictions, *Migrate* moves the session into the *frozen* state and suspends it. Details of session suspension and resumption are discussed in Chapter 5.

3.2 Attack-equivalent security

A concern with any new protocol is that it may introduce new security vulnerabilities; *Migrate*'s session control protocol is no exception. Due to the simplicity of our end-to-end approach, however, there is only one function—binding an end point to a new network attachment point—that could give rise to vulnerabilities.[1] *Migrate*'s security model is designed to make it easy to show that any attacks against this feature can be reduced to attacks launched in the absence of this feature. We term this property *attack-equivalence*, since the set of attacks on an application using *Migrate* is precisely the same as the set of attacks against that application without *Migrate*. The only caveats are that *Migrate* may increase an application's window of vulnerability to existing attacks or reduce the number of times an attacker needs to launch a given attack to affect a desired result. We first explain why this property holds and, then, discuss its implications.

Our fear is that a third-party attacker could somehow use *Migrate* to either impersonate a given end point or expose otherwise confidential information being passed between two end points. Practically, this means an attacker is either able to subvert an application's authentication scheme or hijack an existing session and rebind an end point to itself or a co-conspirator. We claim that applications using *Migrate* are exposed to these vulnerabilities if, and only if, they are vulnerable without *Migrate*.

The crux of our argument rests on the style of authentication mechanisms employed. *Migrate* uses an *anonymous* authentication scheme to ensure that binding updates are in fact sent by the remote

[1] Mobility handlers are executed inside the application processes that provide them, so there is no danger of *Migrate* executing foreign code in a privileged environment. See Chapter 6 for implementation details.

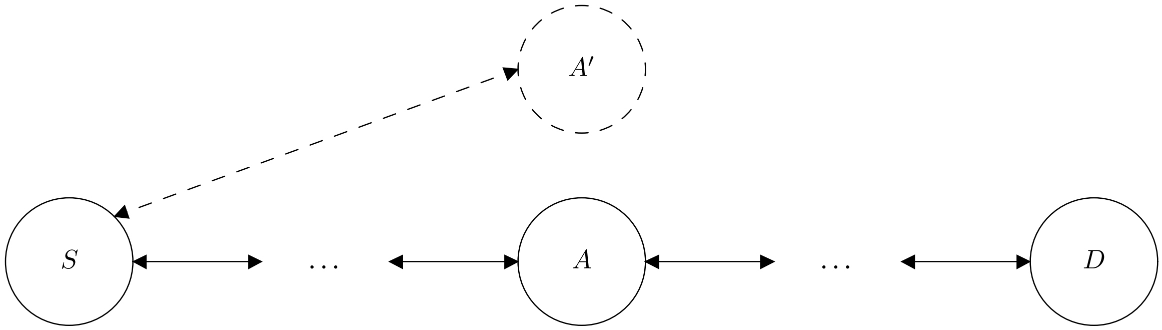

Figure 3-5: A man-in-the middle attack. A masquerades as D to S and *vice versa*. A can then impersonate D to S and bind D to a new network attachment point, A'.

end point. Initial authentication of the remote end point is performed by the application before a *Migrate* session is even established. Because *Migrate* neither assists or interferes with an application's end-point authentication mechanism, it cannot be used to subvert initial authentication. Therefore, all attacks that leverage *Migrate* reduce to those that hijack an open session, *i.e.*, convince an end point to direct a pre-existing session to a new attachment point.

3.2.1 Session hijacking

By design, anonymous authentication has one vulnerability: it is subject to man-in-the-middle attacks. Figure 3-5 depicts an open session between S and D. Because *Migrate* does not authenticate the end points, it is possible for an attacker, A, that is on the path between the two end points S and D to masquerade as either S or D. To do so, A must be able to replace packets between S and D with its own. (We describe the details of such an attack in Section 6.6.) If A is able to launch such an attack, it can sign binding updates to S that appear to be signed by D, and *vice versa*. These binding updates could update the session end-point bindings to any network attachment point A chooses and trigger the migration of all constituent connections. However, since A was already able to launch a man-in-the-middle attack (which requires the ability to read traffic, block traffic, and insert traffic) against S and D, the ability to sign end-point binding updates (and, therefore, migrate connections) adds no new powers to her arsenal. Traffic on the session between S and D was *already* being directed to A.

Applications that wish to prevent on-path attackers like A from reading or modifying traffic that passes through them must employ some sort of protection scheme that provides privacy and non-malleability. This typically takes the form of encryption. If the traffic between S and D is encrypted, it remains protected, even if it is hijacked and mis-directed to A'. While a hijacked session will not deliver traffic from S to D, A must have been able to prevent packets from being delivered to launch a man-in-the-middle attack, so this is no more powerful. It is the case, however, that A may only need to prevent *one* packet from being delivered to launch a man-in-the-middle attack. Once successful, A can then use *Migrate* to ensure any later packet is not delivered.

Similarly, *Migrate* may suspend sessions for varying lengths of time. Some applications may expect sessions or individual network connections to last for only a very short time. Hence, these communications may be secured using relatively insecure cryptographic techniques. In normal operation, the communications do not last long enough for attacks against such schemes to be effective. If, however, communications are suspended by *Migrate* for a long enough period of time, an attacker may be able to launch an off-line attack in the meantime and break the scheme by the time communications resume. Hence, it is of paramount importance that the strength of any cryptographic

Method	Description
`int` **`session_rate`**`(`*`Session`*` s, int num)`	Allow at most `num` changes in remote end point's attachment point per minute.
`int` **`session_length`**`(`*`Session`*` s, int len)`	Allow a session to be suspended for at most `len` seconds.

Table 3.4: Extensions to the API to support policy-based resource control

technique is appropriate for the entire duration of the communication session it is securing, accounting both for the time the session is active and the time it may be suspended. The next session presents a mechanism that allows applications to tell *Migrate* for how long a given session may be suspended.

3.2.2 Denial-of-service

As with any Internet service, *Migrate* is susceptible to denial-of-service (DoS) attacks that attempt to prevent legitimate users from using the service by depleting server resources. These attacks can come in several flavors:

- **Invalid session requests —** Attackers may send arbitrary session establishment packets to a *Migrate*-capable host in an attempt to exhaust the host's resources; a successful attack forces a host to consume all its resources processing invalid requests, leaving no opportunity for valid requests to be processed. This attack is analogous to a SYN flood attack against a host's TCP stack [20].
- **Invalid migration requests —** Attackers may attempt to exhaust a host's resources by sending arbitrary migration request packets that *Migrate* will attempt to validate. For the moment, we assume it is cryptographically unfeasible for an attacker to actually hijack a session using such requests. We will consider the validity of this assumption in Chapter 6.
- **Abandoned (valid) sessions —** One or more attackers may establish large numbers of open sessions with a *Migrate* host without any intent to use them for communication in an attempt to exhaust the host's resources.
- **Excessive (valid) migration requests —** An attacker may request a large number of migrations of an open session with the intent to consume resources on the remote host.

Migrate can prevent each of these attacks through judicious implementation and effective policy control mechanisms. The practicality of the first two attacks depend greatly on the cryptographic techniques employed by *Migrate* to secure the session, which are described at length in Section 6.6. Preventing the latter two attacks requires guidance from the applications running at the end host.

DoS attacks consisting of multiple, valid requests are difficult for *Migrate* to defend against, since they are in fact correct operations from the point of view of *Migrate*. Instead, it is up to the applications running at the end host to dictate the amount of resources that should be expended on a particular session. Table 3.4 lists the API calls that applications can use to describe the appropriate resource limits for a *Migrate* session. The `session_rate()` function specifies how many times a remote end point may change attachment points in any particular minute. If a session exceeds this rate, further binding updates will be ignored until the rate returns to within the allowed range. Any non-zero rate will eventually allow further changes in attachment point, so it is never possible for a remote end point to be disconnected due to excessive migration requests; the session may just be suspended for some period of time.

Applications interested in controlling the length of time a session is suspended can use the `session_length()` call. Sessions suspended for longer than `len` seconds will be considered permanently disconnected. This allows time-sensitive applications or those using fixed-strength cryptographic techniques to bound their windows of vulnerability. It also ensures abandoned sessions will not be preserved by *Migrate* for an unbounded amount of time. We will return to this point in Section 5.4 when we discuss garbage collection of session continuations.

3.3 An example: Host mobility using DNS

As an example of how *Migrate* sessions can be used, we describe how mobile Internet hosts can use *Migrate* in combination with a standard naming system that supports dynamic updates to provide transparent support for *host* mobility—that is, when Internet hosts change physical locations in the network. The dynamic variant of the Domain Name System (DNS) can provide the necessary level of indirection between a host's current attachment point and an invariant end-point identifier—namely, its hostname. Network applications on mobile hosts can use DNS in combination with the *Migrate* session abstraction to handle attachment point changes with little additional overhead compared to the non-mobile case. *Migrate* takes advantage of the fact that a hostname lookup is already ubiquitously performed by most applications that originate communication with network hosts and uses the DNS name as session end point. This approach has two benefits: A DNS name identifies a host and does not assume anything about the network attachment point to which it may currently be attached, and the indirection occurs only when the initial resolution is done via name resolution (*i.e.*, a DNS lookup).

When attaching to a network, a mobile host uses a locally-obtained network attachment point valid on the current network to communicate with other Internet hosts. The issue of obtaining a network attachment point and corresponding IP address is separate from locating and communicating with other mobile hosts. Any suitable mechanism for address allocation may be employed, such as manual assignment, the Dynamic Host Configuration Protocol (DHCP) [31], or an auto-configuration protocol [135].

3.3.1 Implementation

Applications desiring mobility support need only describe their communications in terms of sessions. In other words, for each connection they open to a remote host they must create an encompassing session using the `session_create()` call. Further, they need to inform *Migrate* of the name of the remote host through the `set_lookupfunc()` call. Since DNS is the naming system of choice, the application would pass the operating system-provided DNS resolution function (`gethostbyname()` on UNIX platforms) and the remote hostname as parameters. As mentioned previously, mobile clients should also use the `set_lookupname()` call to inform the remote end point of the local hostname. Assuming the applications were content to have *Migrate* handle all changes in attachment points, no further additions or changes are needed; passing the `M_AUTOCLOSE` flag to `session_create()` suffices to close sessions once the connections have terminated. Because these simple extensions are sufficient to provide mobility support for a

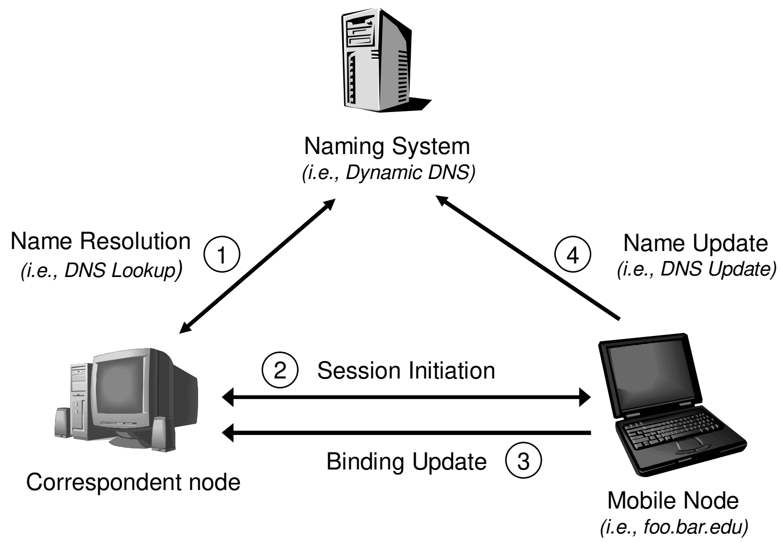

Figure 3-6: Supporting Internet host mobility using DNS as the naming system. 1) The application uses a DNS server to resolve the desired end point (host) name to a network attachment point on the mobile node. The local end point is bound implicitly by the host. 2) The application establishes a session between itself and an application running on the mobile node. 3) If the mobile node moves, it notifies the correspondent node with a binding update, and 4) updates the DNS server with its new network attachment point.

large number of applications, we have implemented a mechanism to transparently insert these *Migrate* calls into pre-complied legacy applications. This transparent interface is described along with the rest of the *Migrate* implementation in Chapter 6.

In addition to modifying applications to use the session extraction, hosts using *Migrate* in concert with DNS to handle host mobility must also ensure their DNS record is kept up-to-date as their network attachment point changes. A mobile host must detect changes in its attachment point and update its hostname-to-address ("A-record") binding in the DNS accordingly. Both tasks are easy to implement, the former through monitoring the status of local network interfaces, and the latter by using the standard secure DNS update protocol [140, 141]; we have written a Perl script that handles both tasks on UNIX platforms. In fact, some DHCP servers today issue a DNS update at client boot time when handing out a new address to a known client based on a static MAC-to-DNS table; this behavior augurs well for incremental deployment of *Migrate* as a solution to host mobility since DNS update support is widely available.

Figure 3-6 illustrates how binding updates would work in conjunction with the DNS. Once a mobile host obtains an IP address, there are two ways in which it can communicate with correspondent hosts. First, as a client, the mobile end point actively initiates sessions to a correspondent host. In this case, no additional tasks need to be performed initially; using the DNS as in the static case continues to work. However, if the mobile host moves to another network attachment point during a session, a binding update must be sent to the remote end point of each session currently active on the host. If a mobile host is a client, then no updates need to be made to the naming system or any third party. Second, if the mobile host functions as a server, or wishes to receive unsolicited packets from other hosts, it must also update the DNS with the IP address of its new attachment point.

3.3.2 Address record caching

DNS provides a mechanism for name resolvers to cache name bindings for some period of time, as specified in the time-to-live (TTL) field of the A-record. To prevent an inconsistent binding from remaining in the cache, the time-to-live (TTL) field for the A-record of the name of the mobile host should be set to zero, which prevents the record from being cached. While negating any caching efficiency for the individual record, setting the TTL to zero does not cause a significant scaling problem [52]; name lookups for an uncached A-record do not have to start from a root name server because, in general, the "NS-record" (name server record) of the mobile host's DNS name is cacheable for a long period of time (many hours by default). Starting most name lookups at the name server of the mobile host's domain scales well because of administrative delegation of the namespace and DNS server replication in any domain. Several content distribution networks for Web server replication of popular sites use the same approach of small-to-zero TTL values to redirect client requests to appropriate servers (*e.g.*, Akamai [2]). There is no central hot spot because the name server records for a domain are themselves cacheable for relatively long periods of time [52]. We will discuss this issue in more depth in Chapter 7.

3.3.3 Benefits

The key benefit of using a naming system such as DNS to support host mobility is that doing so preserves the topological properties of IP addresses. That is, the IP address used to send and receive data continues to reflect the location of the network attachment point in use, which is instrumental to the scalability of Internet routing. This approach is different from other proposals, such as Mobile IP [49, 89], that suggest Internet hosts communicate using *foreign* IP addresses—IP addresses that do not accurately reflect the location of the current network attachment point. In addition to preserving the scaling properties of traditional routing protocols (without requiring home agents), maintaining the semantics of IP addresses is also important for security reasons.

Despite the fact that IP addresses denote only a network attachment point in the Internet and say nothing about the identity of the host that may be connected to that attachment point, IP addresses have implicitly become associated with other properties, such as administrative domain (*e.g.*, a machine having an IP address 18.31.0.100 is quite likely to exist on a network managed by MIT). For example, IP addresses are often used to specify security and access policies as in IPsec Security Associations [55] and ingress filters used to alleviate DoS attacks [34]. A name-based approach to host mobility works without violating these properties.

He said, "You gonna follow me?"
I said, "I've never thought about that before!"
He said, "When you're not following me, you're resisting me."

- Bob Dylan

Chapter 4

Connection Migration

One of the main difficulties in supporting mobile end points is preserving on-going communication channels. Internet transport protocols specify connection end points in terms of their network attachment points; connections cannot currently follow end points as they move from one network attachment point to another. This chapter describes two different schemes for transport connection *migration*: a protocol-agnostic connection *virtualization* approach, and transport protocol extension based on *rebinding*. Virtualization solutions are presented for both the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), while rebinding is discussed only in the context of TCP.

The remainder of this chapter is organized as follows. The first section describes how virtualization can be used to overcome many of the shortcomings of existing Internet transport protocols by synthesizing a logical persistent connection through the use of multiple transport connections. Section 4.2 presents an alternative to the virtualization approach using transport-layer binding updates to support connection migration for TCP, which allows a single TCP connection to survive a change in network attachment point. Section 4.3 presents a mechanism to secure TCP migration and discusses new vulnerabilities introduced by connection migration. Section 4.4 is a brief sidebar on transport protocols that do not impose a connection abstraction and discusses the issues raised when end points using unconnected transport protocols change attachment points. Finally, the chapter concludes in Section 4.5 with a discussion of various issues raised by the use of connection migration.

4.1 Connection virtualization

Applications bind transport protocol end points, often termed *sockets*, to individual ports on a network attachment point during connection establishment. The resulting connection can then be uniquely identified by a 4-tuple: $<$*source IP address*, *source port*, *destination IP address*, *destination port*$>$. Packets addressed to a different IP address, even if delivered to the appropriate port, must not be demultiplexed to a connection established from a different address. This separation is crucial to the proper operation of servers on well-known ports. For example, Web servers typically serve all of their clients from the same local port—80—and use the source IP address to differentiate between client packets.

In a mobile environment, however, an application may wish to continue an existing connection from a new attachment point. One well-known method [60, 82, 99, 145, 147] of allowing changes in

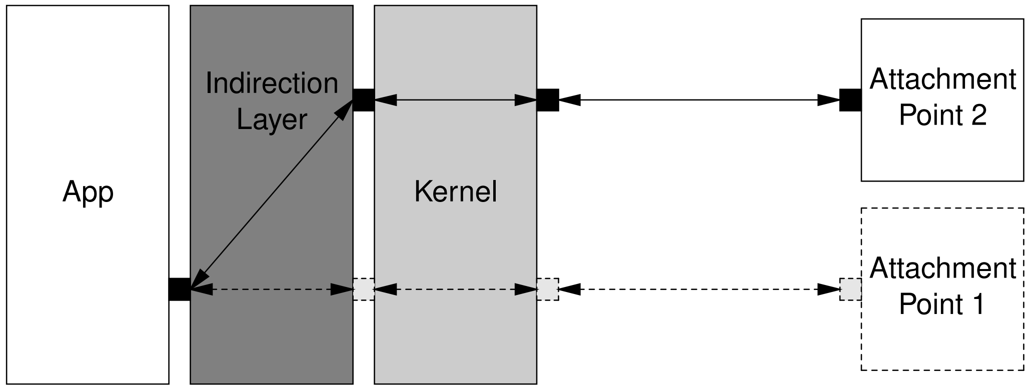

Figure 4-1: A virtualized connection. The virtual socket is dynamically re-mapped to ephemeral network sockets by an indirection layer. A new network connection is established for each change in attachment point. Here, the indirection layer has created a new network connection to attachment point two, and destroyed the old connection to attachment point one.

connection attachment points is to virtualize a logical connection by transparently stitching together multiple physical connections. This section first describes connection virtualization and then shows how *Migrate* virtualizes both UDP and TCP connections.

4.1.1 Indirection layer

In the virtualization approach, shown in Figure 4-1, the application is presented with a virtual socket that is dynamically mapped to an ephemeral network socket, which is connected to the remote attachment point. An *indirection layer* creates and destroys network sockets as necessary to support different attachment points; the layer creates a new network socket and transport connection each time a new attachment point binding is required. The advantage of this approach is that it can be carried out entirely at the user-level, requiring no extensions or modifications to the Internet protocol stack; the major drawback is the virtualization overhead. We present a virtualization approach in the context of UDP connections below, and then extend the technique to handle TCP connections.

Implementing a virtualized connection requires inserting an indirection layer between applications and the operating system kernel. When an application attempts to establish a connection, the indirection layer does *not* directly connect the socket to the requested destination; instead, it creates a *socket pair* and connects one end to the application socket. It then creates another, separate network socket at the same end point as the original and connects it to the destination requested by the application. Data received on the network socket is sent along, or *spliced*, to the application via the socket pair, and *vice versa*.

When notified of a change in remote attachment point (*e.g.*, through the session-based binding update mechanism described in the previous chapter), the indirection layer creates a new network socket, connects it to the new remote attachment point, and destroys the old one. Any further data from the application (socket pair) is then spliced to the new socket; data received on the socket is delivered to the application as before—any further data received on the old socket is ignored. This process is symmetric—both end points must virtualize the connection in the same fashion.

4.1.2 Port mapping

A major difficulty with connection virtualization is negotiating which ports to use for the new connection: Connected sockets cannot be multiplexed; hence, the current port cannot be atomically

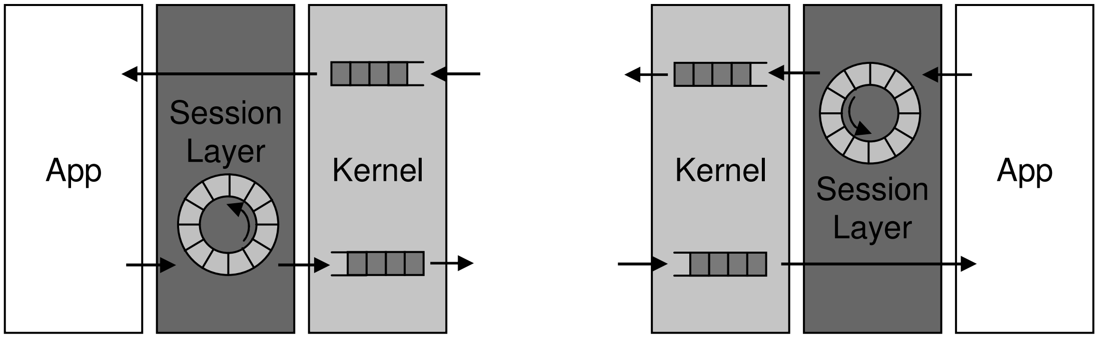

Figure 4-2: A double buffer

reused. (While the application could first close the port and then attempt to reopen it, there is a race condition in which another application could claim the port first.) The problem is similar to the service rendezvous required for applications based on Sun RPC, which Sun RPC addresses with the portmapper [121].

In the particular case of *Migrate*, the session negotiates new ports during the session migration process. Recall that the session migration process is not entirely symmetric—one end point actively initiated the migration by contacting the other end point. For the purposes of discussion, consider the first end point "active" and the latter "passive." After the session has successfully completed a binding update and the new network attachment points are known, the "passive" end point requests a new port for each open connection (*e.g.*, using the `bind()` system call on UNIX platforms). Once a port is assigned, *Migrate* communicates it to the "active" end point, which then acquires a new port on its current attachment point and establishes a connection between the new ports.

4.1.3 Double buffering

An additional complication is introduced by the presence of kernel socket buffers—buffers containing data that has either been sent by the application but not yet delivered to the remote end point, or received by the local end point but not yet delivered to the application. The main difficulty arises because of the opaqueness of these socket buffers; there is no standard way to access the buffered data from user space—it is either successfully delivered to the receiving application or discarded when the socket is destroyed.

Hence, when an indirection layer establishes a new connection, it is possible that outstanding data remains buffered in the socket buffers of the previous connection. The indirection layer must, therefore, *double-buffer* transport connections [60, 82, 99, 145]. In *Migrate*, the session layer keeps a copy of all *potentially* outstanding bytes on a connection as shown in Figure 4-2. After migration, the receiver needs only inform the sender of the last successfully received byte (assuming bytes are delivered in-order); the sender can then replay any remaining outstanding bytes from its buffer into the kernel, which then delivers them as before.

4.1.4 Virtualized UDP

The User Datagram Protocol (UDP) [95] provides an unreliable datagram service. Unlike most transport protocols which impose a connection abstraction, UDP can operate in two modes: a standard, *connection-oriented*, mode, or an *unconnected* mode. Unconnected UDP sockets can be used to send data to arbitrary attachment points and may receive datagrams from any attachment points.

The lack of explicit end-point relationships with unconnected UDP sockets makes them problematic to handle in the face of mobile end points, since the appropriate semantics vary from application to application; we discuss this issue further in section 4.4. Presently, we will assume UDP is operating in a connection-oriented fashion.

UDP's unreliable delivery semantics significantly ease the double-buffering requirement described above. Because UDP connections make no attempts to ensure reliable datagram delivery, applications using UDP expect and can recover from packet loss. There is no need for *Migrate* to double-buffer UDP connections—instead, any outstanding data in kernel buffers can simply be discarded; applications requiring reliable delivery of these datagrams will provide their own retransmission schemes.

Unfortunately, because UDP does not provide any sequencing or reliability guarantees, UDP connections do not normally exchange any connection establishment messages; the act of establishing a connection does not cause the transmission of any data between connection end points. This lack of explicit connection establishment can complicate the port mapping process in the presence of Network Address Translators: an end point behind a NAT cannot explicitly specify the new port to use since NATs generally assign ports only when data is actually transmitted [24].

Hence, after both end points have created a new UDP sockets, *Migrate* immediately sends a short *hello* packet from the "passive" end point's socket to the "active" end point (whose new port is addressable and communicated to the "passive" end point as described above) to complete the connection binding. To ensure successful connection reestablishment, the receipt of a *hello* packet must be acknowledged. Only after successful receipt of the *hello* packet or its acknowledgment, when both end points have a consistent view of the other's current attachment point, does *Migrate* consider the UDP connection established and splice it through to the application.

4.1.5 Virtualized TCP

Unlike UDP, TCP requires careful double-buffering to preserve its byte-stream semantics. TCP connections provide reliable, in-order delivery; the sender buffers all bytes until they are acknowledged as being successfully received by the receiver—bytes that remain unacknowledged after a period of time are retransmitted. Similarly, data received by an end point is buffered in the kernel until delivered to the application. Hence, *Migrate* employs the double-buffering scheme described above to preserve outstanding data on TCP connections.

The key issue, then, is determining which bytes remain outstanding. TCP itself generates acknowledgments as data is received, but those are inaccessible at user level. One approach is simply to buffer *all* bytes on each connection, but this technique becomes impractical as connections age and the buffer becomes large. Another approach is to create a session-layer acknowledgment scheme, wherein the session layer occasionally notifies the remote end point as data is received [37]. Notification has the drawback of adding additional complexity and overhead, however.

Migrate maintains bounded double buffers without explicit acknowledgments; to do so, *Migrate* determines the maximum amount of data that can possibly be outstanding at any one time, and then stores bytes in a circular buffer. A circular buffer of size n stores data in a wrap-around fashion: data is copied in starting at the beginning and wraps around to the beginning when the end is reached. This process results in a buffer that always contains the last n bytes of data (or all of the bytes ever transmitted, if that is less than n). *Migrate* can efficiently set the size of the buffer because it is possible to bound the amount of potentially outstanding data as the sum of the transmit and receive buffers at the end points. All data that has been transmitted by one end point and not yet delivered to the application at the remote end point must exist in either the transmit buffer, the receive buffer, or both. This observation is true because TCP does not discard data from the transmit buffer until it has been acknowledged by the receiver (and, therefore, exists in the receive buffer). It also does not remove data from the receive buffer until it has been delivered to the application.

Migrate exchanges the sizes of receive buffers at connection establishment and uses that in combination with the current size of the local transmission buffer to size the circular buffers for each connection. Because transmit and receive buffers can be re-sized during the connection, an end point must notify the remote end point when it increases the size of its receive buffer (changes in the size of the transmit buffer only affect the size of the local circular buffer and, therefore, need not be communicated to the remote end point). Because receive buffers are generally set to a standard size and changed only infrequently, *Migrate* uses a simple optimization: Rather than exchange receive buffer sizes on every connection establishment, it instead assumes a default size, and only notifies the remote end point if an end point extends the size of a connection's receive buffer (reductions in buffer size are rare, and do not affect correctness).

Despite these optimizations, there remain several inefficiencies. The most obvious is the need to double buffer the data, which could be avoided if the contents of the transmission buffer were available at user level, as has been proposed in various unified buffering schemes such as IO-Lite [86]. Yet, even if the buffers were available, two other inefficiencies remain: the session layer needs an additional round trip time to exchange sequence number and port information, and any received out-of-order data must be discarded (since the new TCP connection must still deliver data in order, it is forced to retransmit the received, out-of-order data). Both of these drawbacks could be avoided if TCP itself were capable of rebinding the end points of an open connection; we discuss such an extension next.

4.2 Migrate TCP: A rebinding approach

This section describes an extension to TCP to support the secure migration of an established TCP connection across an IP address change. TCP allows the specification of *options*—extra header fields—to provide extended functionality. TCP implementations can insert options without interfering with the operation of connections with end points that don't implement the option; TCP specifies that end points that do not understand an option simply ignore it [97]. We define a new TCP Migrate option that allows an end point to continue a TCP connection from an IP address other than the one it used to establish the connection. This migration is transparent to an application that expects uninterrupted, reliable communication with the peer. It requires no third party to authenticate migration requests, thereby allowing the end points to use whatever authentication mechanism they choose to establish a trust relationship. Although details are only provided for TCP migration, the idea is general and can be implemented in a like manner for specific, connection-oriented UDP-based protocols such as the Real-time Transport Protocol (RTP) [114] to provide migration support for those as well. A similar mechanism can be deployed for any transport protocol that enforces a connection-establishment handshake.

4.2.1 TCP options

TCP uses SYN segments to synchronize end points during connection establishment. We propose a new *Migrate* TCP option for inclusion in SYN segments that identifies a SYN packet as part of a previously established connection, rather than a request for a new connection. This Migrate option contains a *token* that identifies a previously established connection on the same destination

$\langle$*IP address*, *port*$\rangle$ pair. Connection end points negotiate the token during initial connection establishment through the use of a *Migrate-Permitted* option. After a successful token negotiation, TCP connections may be uniquely identified by either their traditional $\langle$*source IP address*, *source port*, *destination IP address*, *destination port*$\rangle$ 4-tuple, or a new $\langle$*source IP address*, *source port*, *token*$\rangle$ triple at each end point.

An end point may restart a previously-established TCP connection from a new attachment point by sending a special Migrate SYN packet that contains the token identifying the previous connection. The remote end point will than re-synchronize the connection with the original end point at the new attachment point. An ARQ protocol, TCP uses cumulative acknowledgments (ACKs) to confirm the receipt of sequential data bytes. To facilitate this process, TCP maintains a control block that records the sequence number of the last successfully received sequential byte of data, as well as a significant amount of other connection control state. A migrated TCP connection maintains the same control block and state (with a different attachment point, of course), including the sequence number space, so any necessary retransmissions can be requested in the standard fashion. This preservation also ensures that SACK [66] and any similar options continue to operate properly. Furthermore, any options negotiated on the initial SYN exchange remain in effect after connection migration and need not be resent in a Migrate SYN.[1]

Since SYN segments consume a byte in the TCP sequence number space, Migrate SYNs are issued with the same sequence number as the last acknowledged byte of data. The reuse of the sequence number results in *two* bytes of data in a migrated TCP connection with the same sequence number (the new SYN and the previously-transmitted actual data). Normally, a sequence number refers to exactly one byte of data; hence, there is no ambiguity about which bytes a cumulative ACK is acknowledging. In this case, there are two distinct bytes, which would lead to ambiguity if both needed to be explicitly acknowledged. Fortunately, the Migrate SYN segment need never be explicitly acknowledged. Any packet received from the remote end point by a migrating end point at its new network attachment point that has a sequence number in the appropriate window for the current connection implicitly acknowledges the Migrate SYN. Similarly, any further segments from the migrating end point provide the remote end point an implicit acknowledgment of its SYN/ACK. Thus, there is exactly one byte in the sequence space that needs explicit acknowledgment even when the Migrate SYN is used. Further, after the receipt of a Migrate SYN, an end point will ignore any further packets received from the connection's previous attachment point to insure that connection migration is atomic despite possible packet reordering.

4.2.2 An example

Figure 4-3 shows a connection where a mobile end point connects to a remote end point and later moves to a new attachment point. The mobile end point initiates the TCP connection in standard fashion in message 1, including a Migrate-Permitted option in the SYN packet. The values k_m and T_m are parameters used in the token negotiation, described in Section 6.5.2. The remote server with a Migrate-compliant TCP stack indicates its acceptance of the Migrate-Permitted option by including the Migrate-Permitted option in its response (message 2). The client completes the three-way handshake with message 3, an ACK. The connection then proceeds as any other TCP connection would until message 4, the last packet from the remote end point to the migrating end point at its current attachment point.

[1] They can be, if needed. For example, it might be useful to renegotiate a new maximum segment size (MSS) reflecting the properties of the new path. We have not yet explored this extension in detail.

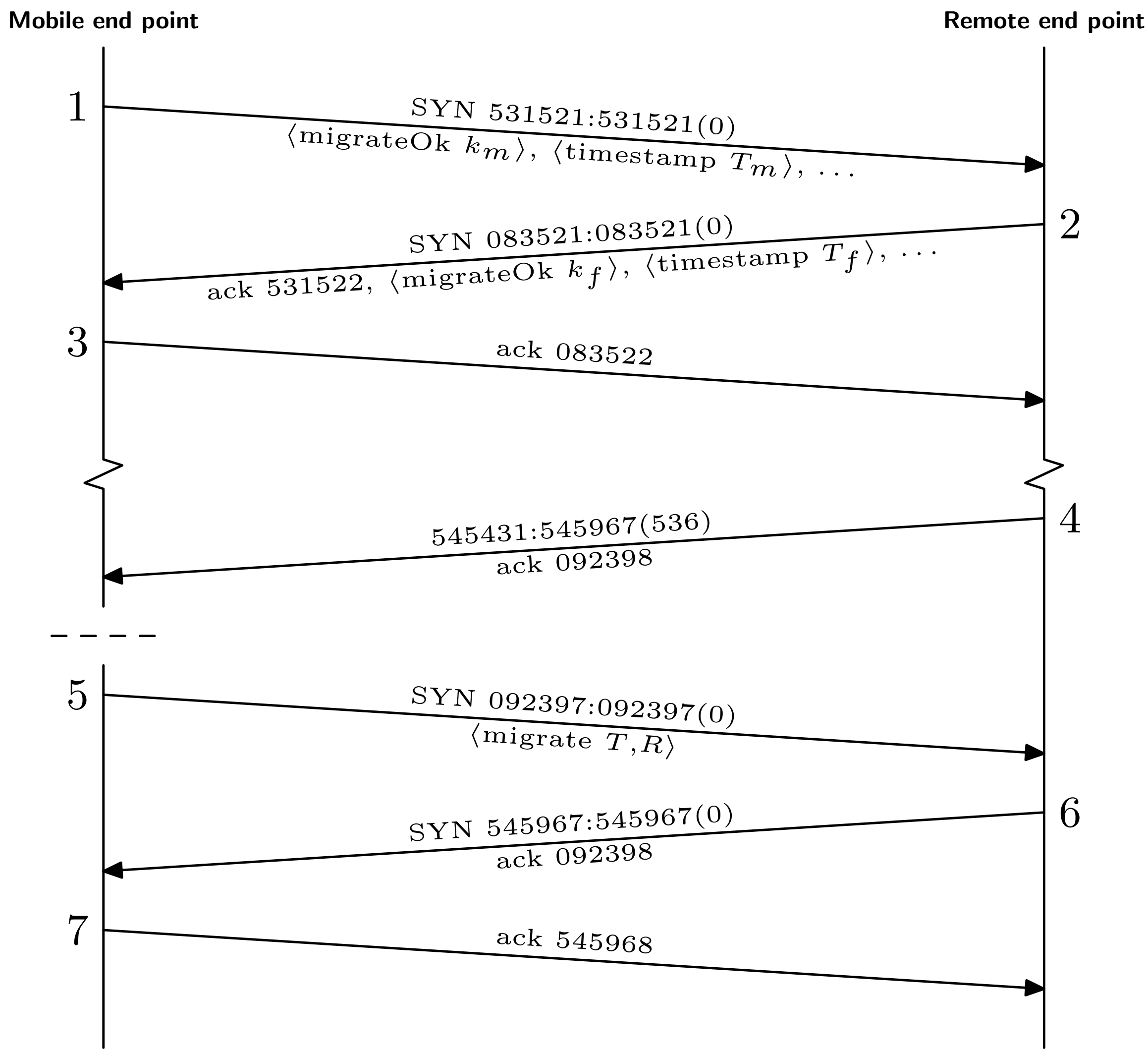

Figure 4-3: TCP Connection Migration. Time flows downward. The migrating end point initiates migrateable TCP connection in message 1. The server accepts the Migrate-Permitted option in message 2. The client completes the three-way handshake with message 3, an ACK. The connection then proceeds until message 4, the last packet from the remote end point to the migrating end point at its current attachment point. At some time later the migrating end point sends a Migrate SYN (message 5) from a new attachment point, including the previously computed connection token. The sequence number of the Migrate SYN is the same as the last acknowledged byte of data. The server responds in message 6 with a SYN/ACK using the sequence number of its last acknowledged byte of data.

At some time later, the mobile end point moves to a new network attachment point and notifies the remote end point by sending a SYN packet from its new address in message 5. This SYN includes the Migrate option, which contains the previously computed connection token as part of a migration request. The sequence number of this Migrate SYN segment is the same as the last byte of acknowledged data. The server responds with a SYN/ACK in message 6, also using the sequence number of its last acknowledged byte of data. The ACK is from the same sequence space as the previous connection. While, in this example, it acknowledges the same sequence number as the SYN that generated it, it could be that segments were lost during a period of disconnection while the mobile end point moved, and that the ACK will be a duplicate ACK for the last successfully received in-sequence byte. Since it is addressed to the migrating end point's new attachment point, it serves as an implicit ACK of the Migrate SYN as well. Upon receipt of this SYN/ACK, the migrating end point similarly ACKs in the previous sequence space and the connection resumes as before. All of the options negotiated on the initial SYN, except the Migrate-Permitted option, are still in effect and need not be replicated in this or any subsequent migrations.

4.2.3 Cascaded migration

Because end points may change attachment points multiple times during the lifetime of a TCP connection, connections may be migrated multiple times by sending multiple Migrate SYNs—one corresponding to each new attachment point. In the normal case, when attachment point changes occur infrequently, the Migrate SYN exchange successfully completes at each attachment point, and the migrations all proceed similarly to steps 5–7 in Figure 4-3. Occasionally, however, end points may change attachment points two or more times in rapid succession, before the Migrate SYN exchange has had time to complete.

Therefore, care must be taken to ensure that Migrate SYNs can be ordered regardless of their time of receipt. This ordering ensures that both end points always have a consistent view of the connection's current attachment points. Each Migrate SYN contains a monotonically increasing *Request Number*. An end point that issues a Migrate SYN from a new attachment point *must* increase the request number. Duplicate Migrate SYNs (*i.e.*, those sent due to the lack of receipt of a SYN/ACK packet) can use the same request number. Hence, any two Migrate SYNs can be ordered, making the migration process robust to packet reordering. Each end point records the greatest request number it has seen so far for each open connection; any Migrate SYNs with request numbers less than that are discarded. Migrate SYNs with the same request number are duplicates, and should be treated appropriately (an indication that the SYN/ACK packet was lost). The receipt of a Migrate SYN with a request number larger than all previous Migrate SYNs received on this connection should immediately transition the connection to `SYN_RECV`, regardless of its current state—*i.e.*, abort any partially completed Migrate SYN exchange in progress. This procedure allows end points to change attachment points repeatedly in rapid succession, without necessarily completing the migration to any intermediate attachment point.

4.2.4 `MIGRATE_WAIT` state

TCP defines a RST segment, which allows a connection end point to ask the remote end point to abort the connection. Special processing of TCP RST messages is required with migrateable connections, since a mobile end point's old IP address may be reassigned before the end point has issued a Migrate SYN to the correspondent end point. TCP specifies that an end point receiving a packet it was not expecting (*i.e.*, that does not belong to a connection currently established at the end point) should generate a RST segment to notify the sender of the inconsistency. In rare cases,

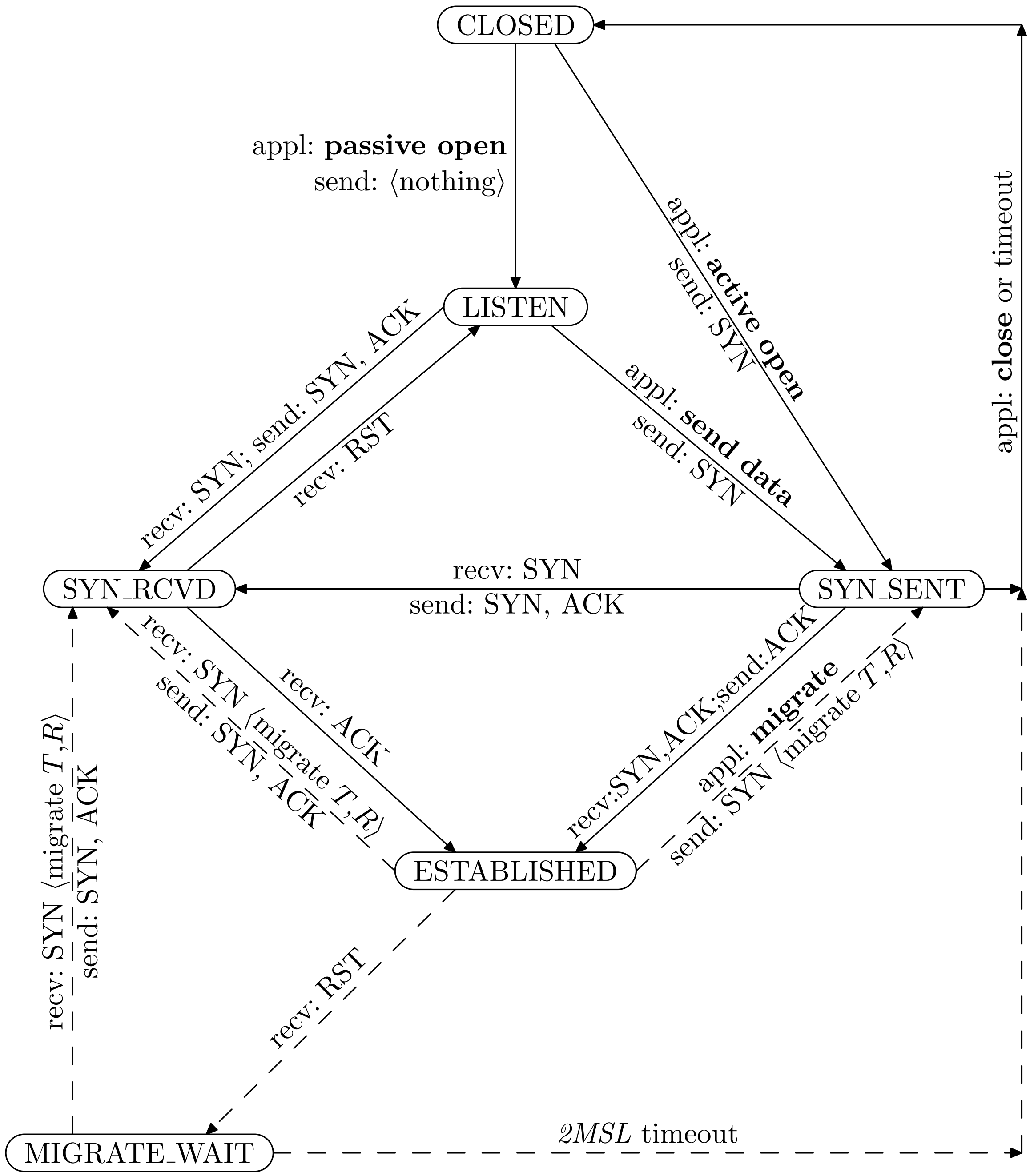

Figure 4-4: Partial TCP state transition diagram with Migrate transitions (adapted from [123, Figure 18.12])

a new end point may assume a migrating end point's old attachment point and receive a packet intended for the migrating end point. Following the TCP specification, the new end point will emit a RST segment to the correspondent host [97]. Figure 4-4 shows the modified TCP state transition diagram for connections that have successfully negotiated the Migrate-Permitted option. The receipt of a RST that passes the standard sequence number checks in the `ESTABLISHED` state does not immediately terminate the connection because it may have been generated by a new end point that just assumed the remote end point's attachment point. Instead, the connection is placed into a new *`MIGRATE_WAIT`* state.

Connections in the `MIGRATE_WAIT` state function as if they were in the `ESTABLISHED` state, except that they do not emit any segments (data or ACKs), and are moved to `CLOSED` if they remain in `MIGRATE_WAIT` for a specified period of time. We recommend using twice the maximum segment lifetime (2MSL), the same period of time specified for the `TIME_WAIT` state. (The TCP specification [97] defines an MSL as 2 minutes, but common implementations also use values of 1 minute or 30 seconds for MSL [123].)

Migrate TCP processes any segments received while in the `MIGRATE_WAIT` as if they were received in the `ESTABLISHED` state, except that it does not generate ACKs. Receipt of a Migrate SYN with the corresponding connection key is the only event (other than a timeout) that can remove a connection from the `MIGRATE_WAIT` state. Upon receipt of such a segment, the connection end point responds in the same fashion as if the connection were in the `ESTABLISHED` state. (A similar, but far less likely, situation can occur if a connection end point receives a RST segment while the connection is in the `FIN_WAIT1` state—the application at the end point has closed the connection, but there remains data in the connection buffer to be transmitted. For simplicity, these additional state transitions are not shown in Figure 4-4.)

The `MIGRATE_WAIT` state prevents connections from being inadvertently dropped if the address allocation policy on the migrating end point's previous network reassigns the end point's old IP address before the end point has reconnected at a new attachment point and had a chance to migrate the connection. It also prevents the continued retransmission of data to an unreachable end point. This passive approach to disconnection discovery is better than an active, mobile-initiated *squelch* message—a message requesting the remote end point suspend communication on the connection and await resumption from a new attachment point—because any such message could be lost (and any guaranteed-reliable transmission mechanism could take unbounded time). Furthermore, a migrating end point may not have sufficient (if any) notice of an impending change in network attachment point to issue such messages.

4.2.5 Performance enhancements

Several enhancements can be made by implementations to improve TCP connection throughput during connection migration. For instance, end points can prevent the transmission of segments to old attachment points by using a RST segment as a squelch message; end points aware of an impending migration may transmit a RST segment to the remote end point, which will force the connection into `MIGRATE_WAIT`, preventing additional packet transmissions. End points should use RST as a squelch message with great care, however, since it invokes the strict 2MSL time bound on the allowable delay for end point relocation and connection migration. After transmitting an RST segment, an end point must request connection migration within 2MSL or the remote end point will abort the connection.

Perhaps, the most obvious additional performance enhancement is for migrating end points to issue three duplicate ACKs immediately after connection migration [15], thereby triggering the fast-retransmit algorithm and avoiding the possibility of entering timeout. By preempting a TCP timeout, the connection also avoids dropping into slow-start and congestion avoidance. Such techniques should be used with care, however, since they assume the available bandwidth of the new path between the end points is on the same order-of-magnitude as the previous path. For migrations across homogeneous link technologies this assumption may be reasonable. However, when moving from local to wide-area wireless networks, there may be order-of-magnitude discrepancies in the available bandwidth. Hence, we do not include the duplicate ACK optimization in the TCP Migrate specification and leave it to particular implementations to responsibly evaluate the circumstances and provide behavior compatible with standard TCP.

4.3 Securing the migration

It is possible for an attacker to partially hijack TCP connections by guessing the sequence space being used by the connection [70]. (Current TCP implementations take care to ensure sequence numbers are not easily guessable to prevent such attacks [11].) Using the Migrate options, however, an attacker who can guess both the sequence space and the connection token can hijack the connection completely. Furthermore, the ability to generate a Migrate SYN from *anywhere* greatly increases the set of places from which an attack could be launched: Ingress filtering (see 2.2.2) can normally be used to prevent connection hijacking by attackers not on the path between the end points, but such filtering is inappropriate for Migrate SYNs.

We must, therefore, take special care to secure Migrate SYNs. We do so by requiring a Migrate SYN to be signed by its issuing end point. Connection end points negotiate keying material during connection establishment and use the resulting keys to sign any Migrate SYNs on that connection. A variety of mechanisms exist for selecting keying material secretly; we describe two in Chapter 6. Unfortunately, signing and validating Migrate SYNs are cryptographic operations that may take non-negligible amounts of computational resources. Possible attacks against the Migrate TCP options, therefore, include both denial-of-service (DoS) attacks and methods of migrating connections away from their appropriate attachment points. As in the previous chapter, our goal is to provide *attack-equivalent* security: a TCP connection using the Migrate options should be no less and no more secure than a normal TCP connection. We discuss these attacks below and either show why the Migrate options are not vulnerable or explain why the attack presents no additional threat in relation to standard TCP.

4.3.1 Denial of service

SYN flooding is a common form of DoS attack, and most modern TCP implementations have taken great care to avoid consuming resources at the passive end point until the three-way handshake completes [12]. To validate a Migrate SYN, the passive end point must perform a significant computation, which implies we need to be especially vigilant against DoS attacks that attempt to deplete the CPU resources of a target end point. We guard against this attack by first checking that the Migrate SYN includes a valid token. An end point does not attempt to validate a Migrate SYN unless the token is correct, implying the sender is either the original remote end point or an attacker who succeeded in guessing—or intercepting—a valid token. Since end points generate RST messages if the token is incorrect or the SYN did not validate, an attacker has no way to identify when it has found a valid token. Because a would-be attacker would, therefore, have to issue roughly 2^{63} Migrate SYNs in expectation to force a SYN validation, we argue that it is infeasible for an attacker to guess the token. More worrisome, however, is the prospect that an attacker might intercept a token and replay it. End points can guard against replay attacks by changing the token after each use; we describe such an approach in the subsequent section.

While the Migrate SYN does not create a new avenue for DoS attacks, the Migrate-Permitted option sent on initial SYN segments may add complications to traditional protections against SYN flooding. Modern operating systems do not create any connection state upon receipt of a SYN segment. Instead, they issue a SYN cookie [12] in the SYN/ACK, which must be echoed by the remote end point in its ACK. Only after the completion of the three-way handshake (indicating at least some level of resources has been dedicated by the remote end point) will the passive host establish the TCP connection. Hence, SYN-flood attacks that blindly issue SYN packets, but do not examine the returned SYN/ACKs, will fail to reserve resources at the would-be victim.

Some methods of negotiating secret keying material using the Migrate-Permitted option (such as Diffie-Hellman) require a passive end point to record keying material immediately upon receipt of a SYN packet. This requirement is incompatible with SYN cookies. We examine this issue further in Chapter 6, where we discuss alternative cryptographic mechanisms and present an approach based on one-way functions that does not require the storage of any keying material until after the three-way handshake completes.

4.3.2 Connection hijacking

In order to prevent replay attacks, a Migrate SYN signature covers both the SYN segment's sequence number and request number. A replayed Migrate SYN segment can, therefore, only be used until either a new byte of data or another Migrate SYN is sent on the connection. If, however, a migrating end point moves rapidly to a another new location—before completing the Migrate SYN exchange at its current attachment point—a replayed Migrate SYN could be used to migrate the connection back to the migrating end point's previous attachment point, which may have been subsequently assumed by an attacker. In order to prevent this attack, the Migrate option processing ignores the source address and port in duplicate Migrate SYNs, since a valid request from a new attachment point would include a higher request number.

More worrisome, however, is the fact that once a Migrate SYN has been transmitted, the token may be observed by any nodes along the new path and DoS attacks could be launched by sending bogus Migrate SYNs with valid tokens. This can be mitigated by computing a new token. One approach is for migrating end points to include a new Migrate-Permitted option in their Migrate SYNs. After the completion of the three-way handshake, the connection would be rekeyed, resulting in a new token. The window of opportunity when the previous connection token can be used (if it was snooped) is then quite small—only until the new three-way handshake is successfully completed.

The security of TCP connections, migrateable or not, continues to remain with the authentication of end points, and the establishment of strong session keys to authenticate ongoing communication. Although we have taken care to ensure the Migrate option does not further decrease the security of TCP connections, the latter are inherently insecure, since IP address spoofing and sequence number guessing are not very difficult. Hence, we strongly caution users concerned with connection security to use additional application-layer cryptographic techniques to authenticate end points and the payload traffic.

4.4 Unconnected sockets

Unconnected UDP sockets are not associated with particular remote end points; they can be used to send datagrams to and receive datagrams from arbitrary attachment points. Hence, it would appear that no additional support is required in the face of mobile end points. Unfortunately, it's not quite that simple. Many applications multiplex internal, application-layer sessions over a single, unconnected UDP socket and use the remote attachment point address to internally demultiplex received datagrams. Any change in remote end point bindings will confuse applications that depend on a consistent remote attachment point.

4.4.1 Maintaining consistency

Migrate is capable of providing a consistent view of remote end points even for unconnected sockets. When a datagram is sent to a remote attachment point, *Migrate* attempts to establish a session with the end point currently using that attachment point. If successful, *Migrate* then employs the mechanisms described in the previous chapter to track the remote end point's location. If the location ever changes, *Migrate* transparently redirects datagrams destined for the original attachment point to the end point's new attachment point. Hence, applications continue to specify the original attachment point when sending data, but the datagram is actually directed to the current attachment point.

Similarly, the attachment point indicated upon receipt by the remote end point is rewritten appropriately. If a datagram is received from an end point that was previously communicating from another attachment point, the datagram is supplied to the application as if it arrived from the original attachment point, not the current one.

4.4.2 Name collision

The combination of these two mechanisms provides consistent end point location, assuming the application never communicates with two distinct end points at the same remote attachment point. The procedure is significantly complicated by the potential of attachment point reuse. Consider an end point, A, that initiates communication to a remote end point, B, from attachment point 1. At some point in time, A migrates to a new attachment point, 2. Afterward, a third end point, C, is assigned attachment point 1. If C were already communicating with B, packets from C would be rewritten and appear to come from some other attachment point. If, however, C initiates communication to B from attachment point 1, B could become confused. From the application's point of view, packets received from C are indistinguishable from packets from A, as both appear to be simultaneously occupying attachment point 1—A and C are said to *collide* because the application uses the same name, 1, for them both.

The problem of name collision is quite difficult to avoid. One obvious alternative is to have *Migrate* assign C a new, distinct attachment point $1'$, and rewrite all packets appropriately. This solution has two unfortunate side effects. First, the application may depend on knowing C's actual attachment point, 1—providing the application some false attachment point, $1'$, may lead to unpredictable behavior. Further, introducing $1'$ pollutes the address space, and C may collide later with a legitimate end point at $1'$. Finally, such forging requires *Migrate* to rewrite packets for end points that have not changed attachment points. It is desirable to restrict the overhead of supporting dynamic end points to only those connections that require it.

Similarly, as described above, *Migrate* must filter all incoming datagrams on the unconnected socket for possible rewriting, adding processing overhead. To avoid this problem, *Migrate* uses *connected* UDP sockets whenever an end point changes attachment point. In other words, once a session is established between two end points, any ongoing UDP connections are migrated as if they were connected, regardless of whether communication was originally through an unconnected socket or not. These connected UDP sockets are multiplexed together with the original, unconnected socket, which continues to deliver data from other end points. Hence, the application only sees one socket, but *Migrate* is able to easily differentiate between those packets that require rewriting (received on connected sockets) and those that do not (received on the original, unconnected socket). Unfortunately, because *Migrate* needs to provide one virtualized socket to the application there is no similar way to transparently avoid the multiplexing overhead on write operations.

4.5 Deployment issues

This section discusses several issues that arise when employing connection migration and how *Migrate* copes with them.

4.5.1 Changes in local attachment point

By default, *Migrate* does not attempt to support changes in local attachment point for applications that explicitly request a particular attachment point. Applications that do so have expressed an interest in controling their own multi-homing choices. Hence, sockets that have been bound to a specific attachment point may cease to function if the local end point changes attachment point. In such instances, *Migrate* suspends the connections until the application requests their migration to alternate attachment points. Typically, when applications bind a socket to a network port, they don't care what network attachment point is used. Where system calls require the application to explicitly specify a particular attachment point, most applications indicate their ambivalence through the use of a wild card specifier (*e.g.*, `INADDR_ANY` on UNIX platforms). In some cases, however, applications go out of their way to explicitly specify a local attachment point. If, for whatever reason, that attachment point becomes unavailable, data can no longer be received on that socket. Because applications must go out of their way to request such behavior, it is not clear precisely what behavior applications may desire in the face of changing local attachment points.

4.5.2 IPsec

There are several issues raised when the Migrate options are used in conjunction with IPsec [55]. While it becomes unnecessary to secure the connection migration (because IPsec authenticates *all* incoming packets), IPsec's use of IP addresses to specify Security Associations (SAs) can be problematic. When a connection with an associated SA is migrated, a new SA must be established with the new destination address before communication is resumed. If the establishment of this new SA conflicts with existing policy, the connection is dropped. This seemingly unfortunate result is actually appropriate. Since IPsec's Security Policy Database (SPD) is keyed on IP network address, the policies specified within speak to a belief about the trustworthiness of a particular portion of the network.

If an end point attaches to a foreign network, any security assumptions based on its normal point of attachment are invalid. One of the limitations of home-agent schemes like Mobile IP is that they obscure the true location of the remote end point. When communicating with a mobile host, IPsec at a correspondent host will apply security policies based on the location of the home agent, which may be inappropriate or insufficient for the current location of the mobile end point. We argue that if a mobile end point continues to have sufficient credentials independent of its point of attachment, an end-to-end authentication method should be used, and a secure tunnel established for communication over the untrusted foreign network.

4.5.3 Transparency

A final issue that arises with virtualized connections is the level of transparency desired. In particular, it is normally possible for applications to discover the current local and remote attachment points in use through standard system calls (*e.g.*, the UNIX `getsockname()` and `getpeername()` system calls). The system should provide applications capable of handling dynamic end points with current, accurate responses; *Migrate* defaults to this behavior. Some legacy applications, however, may be confused by inconsistent, dynamic responses to these calls. Hence, *Migrate* is capable of virtualizing these system calls as well, always responding with the initial attachment points regardless of the current ones.

I keep renaming my motives, but continue doing the same things.

– Mason Cooley

Chapter 5

Session Continuations

One of the main challenges in a mobile environment is a host that disconnects. Frequently this disconnection is unexpected and the duration unknown; session-based applications that rely on connectivity to remote end points have great difficulty functioning in this environment. Previously, we described how *Migrate* uses the session abstraction to seamlessly restore network connections after connectivity is restored. Additionally, *Migrate* can notify mobile-aware applications about disconnection events, facilitating applications' adaptation to changes in connectivity. However, this notification does not provide applications with any mechanism to conserve resources during periods of disconnection.

In this chapter, we develop a generic mechanism that allows application sessions to gracefully pause during periods of disconnection, adapt to changes in the environment upon resumption, and conserve resources during disconnection. In our model, sessions are *suspended* when end points become disconnected, and *resumed* when connectivity returns. We present *session continuations*, an abstraction that enables end points to manage the suspension and resumption of disconnected sessions while allowing host operating systems to reclaim application and system resources during disconnection.

The rest of this chapter is organized as follows. We first motivate our design by introducing the continuation concept in its original, programming-language context before applying it to sessions in Section 5.1. Section 5.2 presents extensions to the *Migrate* API to support session continuations and explains their use in network applications. Section 5.3 discusses how session continuations can decrease the utilization of several key system resources during disconnection. The chapter concludes in Section 5.5 with a brief summary and preview of our evaluation of session continuations.

5.1 Continuations

A *continuation* is a conceptual representation for "the rest of a computation." Its initial development in the late 1960's and early 1970's is variously credited to van Wijngaarden, Strachey, Wadsworth, Morris, and others [104]. Continuations were motivated by the need to formally express sophisticated control structures in programming languages, such as non-local exits (*e.g.*, `break` and `continue` in C), unrestricted jumps (*e.g.*, `goto`), and various forms of exception handling (*e.g.*, Java's `throw` and `catch`). These control structures had previously been difficult to express, as they required passing the thread of control between execution contexts, such as naming scopes and time-dependent states of various mutable entities.

5.1.1 Formal description

In order to understand continuations, it is helpful to first briefly examine the mathematical modeling of programming languages through the use of abstract program elements. *Denotational semantics* formally describe the meaning of a program through a calculus of elements representing various aspects of a computer system.[1] In particular, denotational semantics define an evaluation function, $\mathcal{E}$, which computes the $v \in Value$ or meaning of a given program expression, Exp. Hence, we can define $\mathcal{E}$ as a function from expressions to values:

$$\mathcal{E} : \text{Exp} \rightarrow Value$$

Most useful programming languages support identifiers, which are defined in a naming context $e \in Environment$. These identifiers are typically scoped and may have different meanings at different points throughout the program, requiring the evaluation function to take the current environment as a parameter. Similarly, identifiers typically refer to variables, which can often be assigned time-dependent values. The current state, s, of these variables is recorded in the $Store$, which is also a parameter to the evaluation function. Hence, for languages supporting mutable variables, the evaluation function can be defined as:

$$\mathcal{E} : (\text{Exp} \times Environment \times Store) \rightarrow (Value \times Store)$$

Since an expression may have side effects, the evaluation function must return not only the value of the expression, but the resulting possibly mutated store. Denotational semantics without a notion of continuations are known as *direct* semantics.

Direct semantics have difficulty elegantly expressing the difference between the meaning of a statement in different control contexts (*e.g.*, a statement inside a loop and the same statement outside a loop). A control context provides a mechanism to describe alternate program flows that may occur because of control statements such as conditionals or jumps. A continuation provides exactly this context, allowing the explicit specification of where the program goes afterwords.

Denotational semantics that include the notion of continuations are typically called *standard* semantics. A continuation is then a function from an intermediate value and store to final value and store:

$$k \in Continuation : (Value \times Store) \rightarrow (Value \times Store)$$

Hence, in standard semantics, the evaluation function becomes a function from some input expression and naming, control, and state contexts to a result:

$$\mathcal{E} : (\text{Exp} \times Environment \times Continuation \times Store) \rightarrow (Value \times Store).$$

5.1.2 Continuation passing style

Function calls are one of the most common mechanisms for passing control flow between contexts. Compilers are charged with the task of managing the various environments such that function calls

[1]Operational semantics for mobile applications have been explored previously [17].

return to the appropriate place in the program with the correct state intact. This task is complicated, as *activation records* containing the necessary mutable state must be stored and managed. Since function calls can be nested, there may be a large number of activation records to manage at any time.

Of course, the whole problem could be avoided if there were nothing to do after a function returned. Calls placed at the very end of a function are known as *tail calls*. If a function ends with a tail call, there is no need to record any state about the current function, as its computation is complete. In fact, if the last statement of every function were a nested function call, there would never be a need to return to the previous function. In such a situation, a function call is equivalent to a `goto` that passes arguments.

Unfortunately, such a rigid tail-call regime requires each function to know exactly where to jump to when it finishes. However, there are many functions that are used repeatedly in a program with different return points. This ambiguity can be avoided by passing the function an additional parameter, specifying what other function to invoke when it completes. This parameter is a *continuation*—an explicit representation of the rest of the computation. Hence, in continuation passing style (CPS), all function calls, and, indeed, any control structure, can be modeled as a function call with a continuation parameter. When a function is done, it simply calls the continuation.

5.1.3 Application to sessions

CPS, where the thread of control is explicitly passed from one continuation to another along with any necessary context, has been shown to be beneficial in several domains including process management and inter-process communication (IPC) [30, 36]. The key advantage of CPS is that any state or context necessary for the continuation is specified explicitly, and control never returns to the entity calling the continuation. This approach significantly simplifies state management, as context must never be transparently saved and restored—all state that persists across function calls is explicit.

Our insight is that sessions can be handled in the same way. By making the notion of the "rest of the session" explicit, *session continuations* enable graceful handling of session disconnection, reconnection, and rebinding. Upon disconnection, each session end point generates a continuation specifying how to resume the session and includes any state and system resources necessary for the continuation. With the continuation safely stored, the previously communicating hosts are free to reclaim any resources previously allocated to the abandoned session. If the end points ever wish to resume the session, they each need only invoke their continuation to continue processing.

Unlike a process checkpoint, a session continuation is *not* simply a snapshot of the current local state (although we support such snapshots for legacy applications not using the continuation API). A state snapshot is problematic to capture and restore and is likely to be inconsistent with current conditions at the time the continuation is invoked. A session continuation is a function from the quiescent state (*i.e.*, the state of the end point at session resumption time) to the remainder of the session. In order to adapt to dynamic conditions, continuations take a set of input parameters reflecting both the current network state and optionally that of the remote end point (as discovered during session reconnection). Specifically,

> A *session continuation* is a function from the state of the session end points and network conditions at reconnection time to a context sufficient for control to be returned to an application that effectively continues the session from where it was suspended.

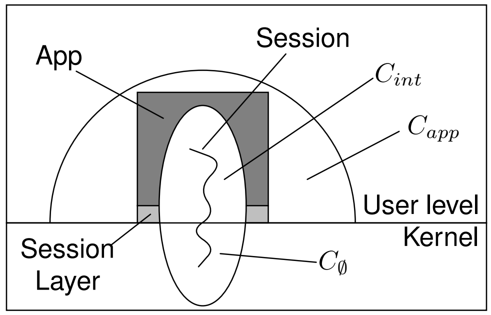

Figure 5-1: Three separate continuations make up a complete session-continuation: a base continuation, $C_{\emptyset}$, an internal continuation C_{int}, and one that restarts the entire application, C_{app}.

In most cases, the required state and system resources may be a subset of or similar to the original state and set of resources. We say similar because in some instances a continuation may provide an application with an alternate, equivalent resource. For example, *Migrate* may replace the original application process with a new copy (see the discussion of complete continuations in Chapter 6). In other cases, changes in end point or network conditions may dictate that the session context be different, reflecting the new network attachment point as well as current network and end point characteristics (*e.g.*, link bandwidth, security policies, etc.). For example, in the scenario described in Chapter 1, a user moved her laptop from a relatively secure network to a highly insecure network. A *Migrate*-aware SSH application would likely want to ignore or discard the previous session key (stored as part of the session continuation) in favor of re-keying the connection with a much stronger encryption key.

5.1.4 Continuation classes

We need to express a session continuation such that it can be marshaled, stored, unmarshaled, and executed in an efficient fashion. The level of detail required by a continuation depends on the context in which it will be evaluated. We find session continuations fall naturally into two classes, corresponding to the architecture of the particular application:

- **Internal continuations** continue the session within its hosting application. For applications that service many sessions at once, the disconnection of one session generally does not lead to the suspension of the entire application.
- **Complete continuations** continue the session outside of its original application (*e.g.*, by starting another instance of the same application). Many applications start a new process for each session, which can be discarded if the session suspends.

The complexity of a continuation depends greatly on the state of the system. For example, if a continuation will execute in the same process that generated it, and no associated state has changed during disconnection, the continuation simply needs to restore the system resources required by the session and identify the point in the application to which control should be returned. The application can then perform whatever additional application-specific functions are required upon resumption. We call this simple preservation continuation the base continuation, $C_{\emptyset}$.

In more useful scenarios, the continuation will be run in the application in which it was created, but with changes in both local and remote state. In that case, the internal continuation C_{int} needs to contain sufficient application state to allow the session to continue, and a function that restores the context required by the session.

Sometimes, the hosting application itself may need to be restarted. In this case, not only must system resources be established appropriately, but the appropriate application must be continued (through its own continuation, $C_{app}()$) before the internal continuation can be run. The session continuation is then a composition of continuations as shown in Figure 5-1. The more complete the state representation, the more portable the continuation. In the extreme, a continuation could be run on a system different from the one that originated the session.

While generating a continuation, an application may also export a small amount of state to be communicated by *Migrate* to the remote end point upon reconnection and passed as a parameter to that end point's continuation. This ability facilitates continuation of sessions whose remote state would otherwise be difficult to determine.

The above discussion shows continuations are composable; complete continuations often depend on several internal continuations. For example, a session whose end points moved, processing suspended, and hosting application terminated would have a complete continuation that is the composition of the examples given above, where s is the current state of the network and session end points:

$$C = C_{int} \circ C_{app} \circ C_{\emptyset}(s)$$

Evaluation order is important, as each successive continuation may depend on the previous one. In particular, an internal continuation requires a hosting application, and a hosting application may require an open network connection to associate with the session. Hence, the system needs to first instantiate the appropriate network connections before calling the application continuation. While continuations can be viewed in a layered model, where each horizontal layer of the network stack creates its own continuation, this view is not strictly accurate. More precisely, each continuation is a vertical swath through the stack, which depends on previous continuations to preserve any dependencies.

5.1.5 Generation

The composable nature of continuations makes them amenable to incremental calculation, allowing end points to generate additional continuations on demand. Because executing a complete continuation as described above, where the system must invoke an entire new application to continue the session, is quite expensive, *Migrate* attempts to keep the continuation overhead as small as possible given current resource constraints. *Migrate* can generate session continuations in two ways, either *eagerly* or *lazily*, depending on the particular session's needs (as specified by the application or a combination of user and system preference).

- **Eager generation.** Some applications require (nearly) immediate notification of disconnection. Examples are applications that need to begin to buffer outstanding data, claim or release locks, or reset timers. *Migrate* supports these applications through eager continuation generation, where continuations are generated immediately upon disconnection detection. Generation of the continuation may be an ongoing process, however. In particular, some continuations may contain buffered or distilled data from the disconnection period and, therefore,

Method	Description
`int freeze(Session s)`	Suspend a session
`int register_handler(Session s, Handler h)`	Register a mobility handler
`int return_cont(Session s, Continuation c)`	Provide a continuation
`int store(Session s, char *a, void *v, int len)`	Store an attribute/value pair
`int export(Session s, char *a)`	Export to remote end point
`int store_size(Session s, char *a)`	Return size of a stored value
`int retrieve(Session s, char *a, void *d)`	Restore an attribute value

Table 5.1: Extensions to the *Migrate* session API to support session continuations and an attribute/value store.

cannot be fully generated until the resumption point is known. For example, a live streaming media application needs to know the exact moment of resumption before it can finish generating a continuation that replays only the missed portion of the stream.

- **Lazy generation.** In some cases, the application can persist largely unaffected during disconnection and the continuation can be generated at any time. In this case, to avoid unnecessary overhead, the continuation is not generated until the system decides it needs to reclaim resources currently being used by the disconnected session.

  In the extreme, the continuation need not be generated until called. This situation may occur on systems with vast amounts of resources or after extremely short periods of disconnection. For example, in a fast handoff situation, it may be the case that a continuation is required before being generated (*i.e.*, the mobile host moved before *Migrate* detected any disconnection). In this case *Migrate* generates the continuation on the fly.

Fundamentally, what *Migrate* tries to do is release resources when necessary, but allow the session to continue with minimal perceived delay when connectivity is restored.

5.2 Continuation API

As introduced in Chapter 3 the *Migrate* API allows an application to define a session by specifying a remote end point and instantiating some number of connections between itself and the remote end point. *Migrate* then ensures that the connections track the remote end point if it changes network attachment point and uses the best available local attachment point according to system policy (see Appendix A). Occasionally, however, the end points may become disconnected, and *Migrate* will suspend the session. When *Migrate* suspends a session, it notifies the application and expects to receive a continuation to call when connectivity resumes. This section presents extensions to the *Migrate* API that allow applications to construct and manipulate session continuations. The individual API calls are listed in Table 5.1.

5.2.1 Disconnection handling

As discussed in Chapter 3, sessions inform *Migrate* of their interest in mobility events through the `register_handler()` call. The provided handler function is invoked by *Migrate* any time a disconnection is detected. Applications that wish to generate a session continuation in an eager fashion may do so inside their handler and pass it to *Migrate* using the `return_cont()` call.

```
typedef struct {
  fd_set          fds;      /* IPC to preserve                         */
  cont_func       cont;     /* Continuation function                   */
  const char *    argv[];   /* (complete only) Command line arguments */
  const char *    envp[];   /* (complete only) Environment variables  */
  const char *    cwd;      /* (complete only) Working directory      */
} migrate_continuation;
```

Figure 5-2: A *Migrate* continuation structure contains a set of file descriptors that must be preserved (commonly pipes to other applications), an attribute/value store, and the continuation function itself. Complete continuations also specify several parameters used when restarting the application process.

Alternatively, applications may choose to generate continuations in a lazy fashion by calling `return_cont()` at a later time. If an application has not yet registered a continuation by the time the session is resumed, *Migrate* will simply invoke the handler function again as described previously, thereby notifying the application of restored connectivity.

The type declaration of a *Migrate* session continuation is shown in Figure 5-2. This structure is composed of three classes of information. The first is a set of file descriptors that will be needed upon resumption—corresponding to files, devices, UNIX pipes, etc.—in addition to the network connections previously declared to make up the session (through the `add_connection()` call). File descriptors included are no longer available to the application and will be restored only upon execution of the continuation; the same is true of the session's network connections. The second component of the continuation structure is the continuation function itself, a function defined within the application process that *Migrate* should call upon resumption of connectivity (instead of the normal mobility handler specified through `register_handler()`).

Applications that handle each session in an individual process will likely provide a third class of information in the continuation—context used to reinvoke the application itself. This context is operating-system dependent. In our UNIX implementation it includes things like command line arguments, environment variables, and the current directory. Continuations that specify this context are termed *complete*, as they provide *Migrate* with all the information necessary to restart the application when the session resumes.

Applications returning a complete continuation exit immediately after calling `return_cont()`, thereby freeing all resources associated with the application process not explicitly preserved inside the continuation structure—those included are preserved by *Migrate* during disconnection and restored prior to the invocation of the application. The continuation function continues to be used in complete continuations; *Migrate* calls it after restarting the application process. This approach eases the task of supporting complete continuations—applications can simply start as they normally would (typically by waiting for a remote end point to contact them) and perform all special handling inside the continuation function itself. This structure also allows the same application to support both internal and complete continuations without modifications.

Applications that handle multiple sessions in one process often do not provide complete continuation information. The resulting *internal* continuation can, therefore, only resume a suspended session if the application process still exists when connectivity is restored. If the application exits before an internal continuation has been called, *Migrate* simply discards the continuation. Because the process continues to exist after the generation of an internal continuation, it is up to the application to discard any remaining internal state and resources associated with the suspended session. In the interest of decreasing latency when the period of disconnection is short, applications may elect to not eagerly free session resources when they generate a continuation, but instead schedule a time to do so later if *Migrate* has not yet resumed the session. In such instances the continuation may first check for the existence of the original session before attempting to resume it using only the state stored in the continuation.

Similarly, some applications may have handlers that must continue to operate during the entire period of disconnection, such as those that record data or events during disconnection. In such instances, the handler can return an internal continuation and arrange for the continuation function to retrieve the necessary information when it is called.

5.2.2 State management

One of the key features of session continuations is the ability to preserve state separately from the hosting application(s). Some sophisticated applications may have internal mechanisms for recording persistant state to secondary storage (*e.g.*, Java serialization [130]), but many existing applications consider session state ephemeral and store it in run-time variables. The task of retro-fitting such applications (as demonstrated in Chapter 7) is made considerably simpler by providing a generic attribute/value store for each session.

Migrate provides an attribute/value store for use by the continuation. Arbitrary state that must survive periods of disconnection can be inserted in the store through the `store()` call. This state is then available through the `retrieve()` function until the session is destroyed—either by the application through the `session_close()` call or by the continuation garbage collector (see Section 5.4). *Migrate* includes a session's attribute/value store in all continuations ensuring it remains available after session resumption. Applications that need to know how large a value is before extracting it from the store (to allocate a sufficiently large buffer, for example) may pass the attribute to `store_size()`.

In certain instances, sessions build up shared state that is not readily observable from the remote end point, but may be necessary for session resumption. Rather than require the application to communicate this state as part of its continuation, *Migrate* supports the exchange of small amounts of state during session resumption. Applications can request the exchange of any attribute in a session's store through the `export()` call. Values placed in the store and exported are then made available to remote end upon reconnection. It is common for *Handler*s to export state which is then used by remote continuations. For example, in the FTP example in Chapter 7, the server exports a `sent_size` attribute (corresponding to the number of bytes it already transmitted) to the client for use by its continuation.

5.2.3 Eliding continuations

In some instances disconnection and reconnection may appear instantaneous, in that by the time *Migrate* has received notice of any disconnection, connectivity has already been restored, albeit at a different location.[2] In such instances, the *Handler* function is called with the `M_INSTANT` flag, as described in Chapter 3 and shown here in Table 5.2. The application may then chose whether to generate a continuation, which would be immediately executed, or to instead handle the

[2] Instantaneous disconnection and reconnection from the same location may occur but, by definition, has no impact on open sessions and is, therefore, ignored.

instantaneous change internally without generating a continuation. This reduction in overhead can lead to decreased handoff latency when minimal changes are necessary. An example of how an application might conditionally generate a continuation is provided in Figure 5-3.

5.2.4 Recursive continuations

Applications often interact with external components such as helper applications. Sometimes these components are modules inside the application itself, but many times they are separate processes. By allowing continuations to specify IPC channels (file descriptors) corresponding to these external entities, *Migrate* is able to keep the channels open during session suspension, thus hiding session suspension from external entities. However, this mechanism is fraught with the same difficulties as transparent network connectivity preservation: The external processes are unaware that the session is suspended, which may lead to undesired behavior and wasted resources.

If the external processes are part of the same application and, in particular, can be modified, then it is possible to design a session continuation that communicates with the external processes to manage resource conservation and ensure proper behavior during disconnection. In the general case, however, external applications are likely to be written by different authors who may have no knowledge that they are acting in concert with other applications. This sort of arrangement is commonplace in UNIX environments where the output of one application is *piped* to the input of another. Hence, it is reasonable to expect that while both applications may be mobile-aware neither was explicitly designed to work with the other.

If both applications are designed to work with the *Migrate* API, they can realize gains even when used together. Consider an application whose input and output is being redirected over a network channel by another application—for example, an editor being executed remotely via SSH. The connectivity state of both applications is then shared: if the SSH session becomes disconnected, the editor becomes disconnected from its input source. When the *Migrate*-enabled SSH receives notice of disconnection, it will generate a session continuation containing the IPC channel connected to the editor. *Migrate*, however, can notice that this channel is in use by another *Migrate*-aware application—the editor. *Migrate* then notifies the editor that it, too, has been disconnected, resulting in a second continuation. This recursive disconnection notice and resulting continuation generation supports arbitrary process chaining.

5.3 Resource continuations

Automatic connectivity notification for chained applications is only one of the possible benefits of the incremental nature of continuation generation. Hosts can also automatically generate continuations that efficiently preserve system resources. When applications include system resources in their continuations, the system may chose to simply maintain those resources during the disconnection, or generate a more sophisticated base continuation that will effectively restore the resources when called, but release them for use by other applications in the mean time. We call base continuations that preserve system resources *resource* continuations. Resource continuations must preserve the normal semantics of the resource. In other words, from the point of view of the application, the resource must appear as if it was simply preserved across the period of disconnection. Similarly, resource continuations must not interfere with other processes that may be sharing the resource. Because resource continuations are completely transparent, all applications can benefit from a system's ability to efficiently suspend and resume individual system resources when not in use.

```
/* This function will be called when connectivity resumes and the process is restarted */
static void mobcont(migrate_session *session);

static migrate_continuation * mhandler(migrate_session *s, int flags)
{
  migrate_continuation * cont;
  fd_set * fds;

  /* Only return a continuation on disconnection */
  if(flags & ~M_INSTANT) {
    cont = (migrate_continuation *)malloc(sizeof(migrate_continuation));
    memset(cont, 0, sizeof(cont));
    cont->cont = mobcont;
    cont->flags = M_COMPLETE;

    /* Preserve stdin/out/err */
    fds = (fd_set *)malloc(sizeof(fd_set));
    FD_ZERO(fds);
    FD_SET(STDIN_FILENO, fds);
    FD_SET(STDOUT_FILENO, fds);
    FD_SET(STDERR_FILENO, fds);
    cont.fds = fds;

    /* If an application doesn't specify an environment or command-line arguments here,
     * they will be copied from the current process by default. The process will exit here. */
    return cont;
  } else {
    /* Handle mobility event without generating a continuation here... */
  }
}

int main(int argc, char *argv[])
{
    /* ...The application first creates a connection to a remote end point... */

    /* Only after validating the end point does it create the session */
    session = create_session(sock);
    /* It can then store session-related state */
    store(session, "sample˙data", (const void *)dataptr, SAMPLE_DATA_LEN);
    /* Register a continuation generation function */
    register_handler(session, mhandler);

    /* Carry on with the session... */
}
```

Figure 5-3: A sample *Migrate*-aware application that exports a complete session continuation upon disconnection but handles instantaneous mobility events directly.

5.3.1 File descriptors

One example of a class of system resources that can be effectively conserved while a session is disconnected is file descriptors or file handles. In a POSIX-conformant operating system, file descriptors are kernel data structures that facilitate all types of I/O, including network sockets, device handles, file pointers, and inter-process communication. To be efficiently implemented, most operating systems are statically configured with a fixed number of file descriptors. High-end database applications and network servers tend to run out of file descriptors under heavy load [6].

As a practical matter, the number of necessary file descriptors associated with network sockets is currently limited by the ability of a server to handle the associated connections. That is, a given server is able to computationally handle only a certain number of concurrent communication sessions. Therefore, it suffices to configure the operation system with enough file descriptors to manage that number of sessions. Such a limit ensures a well-engineered application that does not accept more concurrent client sessions than it can simultaneously process will have sufficient file descriptors. This approach is the one taken by Web servers and other standard Internet servers.

When provided with the ability to suspend sessions, however, servers may be able to handle a significantly larger number of concurrent sessions, since only a small fraction of them may be active at any one time. Hence, it is useful to consider a mechanism that allows suspended sessions to release their file descriptors and reclaim them upon resumption. This restores the previous requirement that there need only be enough file descriptors available to support the maximum number of concurrent, *active* sessions.

Because *Migrate* manages the network connections associated with open communication sessions, it is straightforward to generate continuations that release file descriptors for network connections while suspended. Under the connection virtualization technique described in Chapter 4, open connections are remapped to new connections upon resumption, anyway, so there is no need to keep the old ones around while suspended. Hence, when an application requests the preservation of a network connection as part of its session continuation, *Migrate* disconnects the old connection and generates a continuation that instantiates a new one before calling the application's connection.

Forcibly aborting network connections on suspension can realize considerable resource savings in practice. In addition to the file descriptor data structure itself, network sockets have associated send and receive buffers containing data that has not yet been sent by the operating system or received by the application, respectively. When operating over high-speed links, these buffers may need to be large for maximum performance. Buffer memory in contemporary operating systems is fixed or unpageable, meaning that a suspended sessions connection buffer may decrease the amount of *physical* memory available for active sessions. By aborting connections of suspended sessions, *Migrate* releases this valuable physical memory.

Recall that, in the virtual connection scheme, *Migrate* double-buffers reliable transport protocols in user space so the data is not lost. If *Migrate* were implemented as an extension to the operating system itself, this double buffering could be eliminated, and the socket buffers could simply be moved to user space upon session suspension. Of course, in the best possible case, the operating system would provide a unified buffer cache model like IO-lite [86] and socket data would be permanently maintained in user space, avoiding the whole issue. In such a model, application data would remain associated with the application until explicitly discarded—either by the application itself or by the transport protocol upon receipt of an acknowledgment by the receiver. Such a mechanism would make it easy to identify outstanding data and retransmit it over alternate channels if appropriate.

5.3.2 Memory

Socket buffers are only one type of memory associated with communication systems. In many cases, sessions have a far larger footprint in user space used to store information such as authentication keys, ongoing computations, cached responses, etc. A memory-constrained host may be concerned with the memory footprint of suspended sessions and wish to generate a continuation that frees available memory.

For systems that support virtual memory, this may be unnecessary, as modern virtual memory systems do a good job of removing unused pages from core. The key issue is locality, however. In order for page-based techniques to perform well, data related to the same session must be stored on the same page(s). If session-related data is intermingled with data for other sessions, the suspension of a particular session may not free any individual pages. Hence, we consider how *Migrate* can reduce the memory footprint of suspended sessions regardless of the virtual memory scheme.

We first observe session-related memory can be separated into three classes:

1. **Ephemeral memory.** Temporary scratch space, caches, or other data that is either invalidated by the session suspension or stored elsewhere.

2. **Regenerateable memory.** Data that is likely to be needed upon resumption, but can be recomputed or regenerated at some cost.

3. **Critical memory.** Data that is absolutely essential to the session, and cannot be recreated or replaced without restarting the session.

Using this taxonomy, it is clear that ephemeral memory can simply be freed upon suspension. Conversely, critical data must be stored and cannot be freed. Regeneratable memory could also be freed—provided the function necessary to recompute the values was known. In the general case, an application could explicitly declare all session-related memory as belonging to one of the three classes upon allocation. We call such an approach *colorful malloc*, as data segments can be "colored" according to their type. Unfortunately, such an approach would place a significant burden on the application programmer. The approach taken by *Migrate* is considerably more simplistic, but still quite effective.

In the case of complete continuations, *Migrate* assumes that all memory is ephemeral unless notified otherwise. Applications declare data as potentially critical by including it in a session's attribute/value store. This approach also ensures locality, as *Migrate* is able to allocate this memory itself. Despite the application's claim that the memory is critical, *Migrate* considers the memory regeneratable and applies a compression function to reduce it to a smaller amount of truly critical data. This optimization follows from the observation that compressible data is a specific class of regeneratable data. Because the compression algorithm is well known, this approach avoids the general problem of requiring the application to specify the function necessary to regenerate data.

Using this continuation regime, *Migrate* stores only a compressed version of the data an application described as critical when generating its continuation. While the memory footprint is significantly reduced, the overhead of marshaling and compression the data is non-trivial. We observe, however, that *Migrate* need not generate this memory continuation eagerly, but can generate it lazily—that is, only when memory is scarce. The performance impact of compression is, therefore, only felt when necessary, a trade-off previously shown to be effective in paging schemes for memory-constrained systems [54].

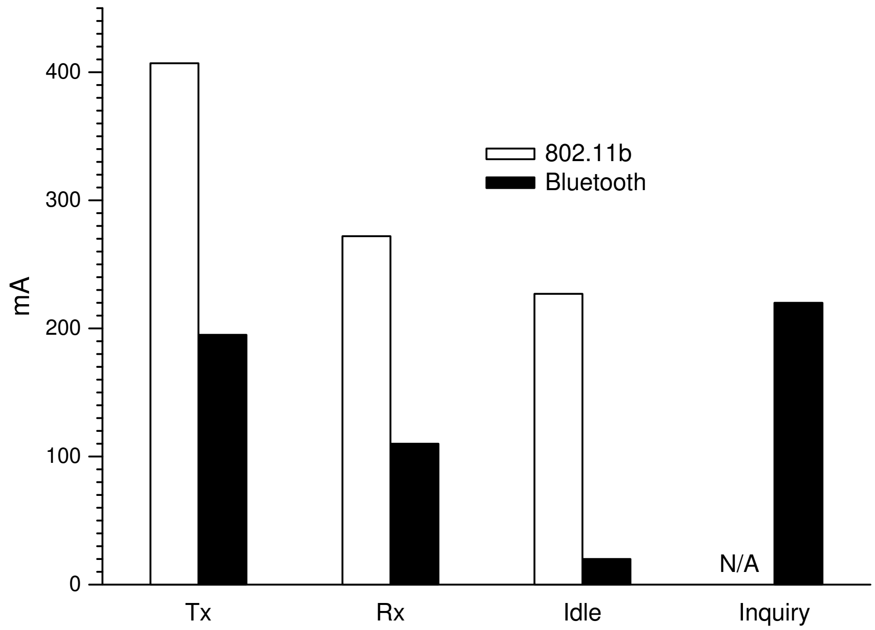

Figure 5-4: The power consumption of common 802.11b and Bluetooth network interfaces. The values shown are currents measured across the PCMCIA bus of an IBM ThinkPad T21 when using a Cisco Aeronet 350 802.11b and a Brainboxes BL-500 Bluetooth interface, respectively.

5.3.3 Other system resources

There are a large number of other system resources that can be conserved by *Migrate*. Just because a resource is not specifically enumerated in a continuation does not mean it cannot be conserved. Consider, for example the case of energy—a very important resource in portable computing devices. Multi-homed hosts often have redundant connectivity to a remote host; applications may prefer one interface to another for their sessions based upon various performance characteristics. If all of the sessions using a particular interface become disconnected, there is no need to keep the interface powered up.

Interface usage is an important consideration when some interfaces are substantially more power hungry than others. Figure 5-4 shows the power consumption of two popular network interfaces technologies in various states of operation. 802.11b provides much higher bandwidth than Bluetooth, but consumes substantially more energy. If a multi-homed host's 802.11b and Bluetooth interfaces provide similar connectivity, there is no need for it to keep the 802.11b interface powered when all sessions using it are suspended; remote end points can still notify it of restored connectivity using the Bluetooth interface. Previous research has demonstrated significant power savings using a similar approach [116]. Hence, a power-constrained host could be configured to power down its 802.11b interface when not in use, but *Migrate* could power it up as part of the base continuation for sessions that would prefer to use it over the Bluetooth interface.

5.4 Garbage collection

While session continuations enable considerable resource savings, they incur costs as well. In particular, each individual session continuation must be stored and managed by both disconnected end points until connectivity is restored. In some cases, however, connectivity may never be restored;

Flag	Meaning
M_LOCAL	There has been a change in local attachment point
M_REMOTE	The remote end point changed attachment point
M_INSTANT	The change in attachment point appeared instantaneous
M_DISCARD	This continuation is being discarded

Table 5.2: The flags that may be passed to a *Migrate* session continuation. When a continuation is selected for garbage collection, the continuation is invoked with the M_DISCARD flag. The M_DISCARD flag is never set at any other time or in conjunction with any other flags.

two end points may never come in contact with each other again. Often, one end point will have crashed or discarded its continuation, leaving the other holding a useless continuation.

Hence, session continuations must be *garbage collected* at some point—continuations that are deemed useless need to be purged from the system. Garbage collection creates two complications: First, at what point is it appropriate to discard an apparently unwanted continuation? Second, since the application that created the continuation expected it to be invoked, how can the continuation be disposed of while ensuring the application is not left in an inappropriate state. The first issue is quite complicated, and remains an area for further study. *Migrate* currently enforces an expiration date on session continuations (calculated by subtracting the session's current age from the permitted lifetime specified by the application through the `set_length()` call). Any continuation that has passed its expiration date is garbage collected. An expiration date is set upon continuation generation, and results from the combination of application (see the `session_length()` call in Section 3.2), user, and system-wide policy.

In order to address the second issue, we introduce a new flag that can be passed to a session continuation. The complete set of flags that may be passed to a session continuation upon invocation is shown in Table 5.2. When *Migrate* determines that a session continuation should be garbage collected, rather than simply discard the continuation, the continuation is invoked and passed the M_DISCARD flag. While all preserved state and IPC is restored, the session itself is placed in a disconnected state, and no communication can be conducted on its connections: Any attempt to read from or write to the session's TCP connections will result in a *connection timed-out* error; any transmitted UDP datagrams are discarded. The application is expected to gracefully terminate the session and exit, thereby released any resources being held by the continuation. Since any attempted network I/O will expose the lack of session connectivity to the application, applications that do not observe the M_DISCARD flag will exhibit the same behavior they would have if *Migrate* were not present when connectivity was lost.

As discussed previously, internal continuations are discarded immediately if the generating process exits regardless of their expiration date since they can no longer be executed. Any included system resources are forcibly released by *Migrate*. File descriptors contained within a continuation are reference counted by the operating system in the same fashion as file descriptors held open by an application. Hence, the closing of a file descriptor by an application or the *Migrate* daemon may not necessarily completely free the associated resource—it may still be in use by another application. Regardless, *Migrate*'s garbage collection procedure ensures that system resources preserved within a continuation will eventually be released; sharing semantics are enforced by the operating system and are not affected by *Migrate*.

5.5 Summary

By generating a session continuation, applications are able to simultaneously manage the suspension and resumption of disconnected sessions while allowing the host operating system to reclaim application and system resources during disconnection. Each *Migrate*-aware application can suspend a session by providing its own session continuation; *Migrate* informs any local applications on the host interacting with the suspended session of its suspension, allowing them to generate their own continuations. Continuations can be generated in both an eager fashion in response to disconnection and a lazy, reactive fashion in response to resource scarcity. *Migrate* reclaims system resources by additionally generating incremental resource continuations that are composed with application-provided continuations. Chapter 7 evaluates the effectiveness of *Migrate*'s resource continuations with respect to system memory, open file descriptors (both those associated with network connections and otherwise), and power consumption.

My religion consists of a humble admiration of the illimitable superior spirit who reveals himself in the slight details we are able to perceive with our frail and feeble minds.

– Albert Einstein

Chapter 6

Implementation

Migrate is implemented as two parts: a daemon and a dynamically loadable library linked against individual applications. The main support infrastructure for the session abstraction is implemented in the *Migrate* daemon described in Section 6.1, which interacts with both local policy engines (Section 6.3) and a suite of connectivity monitors (Section 6.4) to manage open sessions. Applications interact with *Migrate* using the APIs described in Chapters 3 and 5, which are exported by the session-layer library presented in Section 6.2. Figure 6-1 depicts the various components and their interactions. We also present the Linux implementation of the TCP Migrate options in Section 6.5 and discuss the cryptographic primitives used in both TCP connection migration and session migration in Section 6.6.

6.1 *Migrate* daemon

The *Migrate* daemon is responsible for managing all open sessions on an end host. Normally, a host will run a single *Migrate* daemon with super-user privileges. Because this configuration is not always feasible, however, the *Migrate* daemon can be run by individual users to support their own applications. In order to co-exist with other *Migrate* daemons on the same system, all *Migrate* components check for a `MIGRATE_PORT` environment variable when starting up; if it is not defined, a default value is assumed. This value is used to differentiate one instance of *Migrate* from another and affects both the local and remote ports used for all of *Migrate*'s control channels. Hence, each application, connectivity monitor, and policy engine can be configured to connect to a particular *Migrate* daemon on a specific port. Both local and remote end points must agree on the port, however, so a user must take care to ensure both end points involved in any sessions are configured to use the same port. For simplicity, we will henceforth assume only one *Migrate* daemon is running on an end host.

We assume the *Migrate* daemon is started upon system boot and runs continuously. If the host system or *Migrate* daemon crashes, all active sessions are lost. In most cases, however, application processes will not survive a system crash, either, so this does not decrease reliability. To improve the robustness of the *Migrate* daemon itself, session state could be periodically logged to stable storage and recovered when the daemon restarts. Unfortunately, this approach cannot prevent the loss of system resources being preserved inside of continuations held by the daemon at the time of a crash. Hence, our current implementation does not attempt to recover any sessions or continuations lost due to a system or daemon crash.

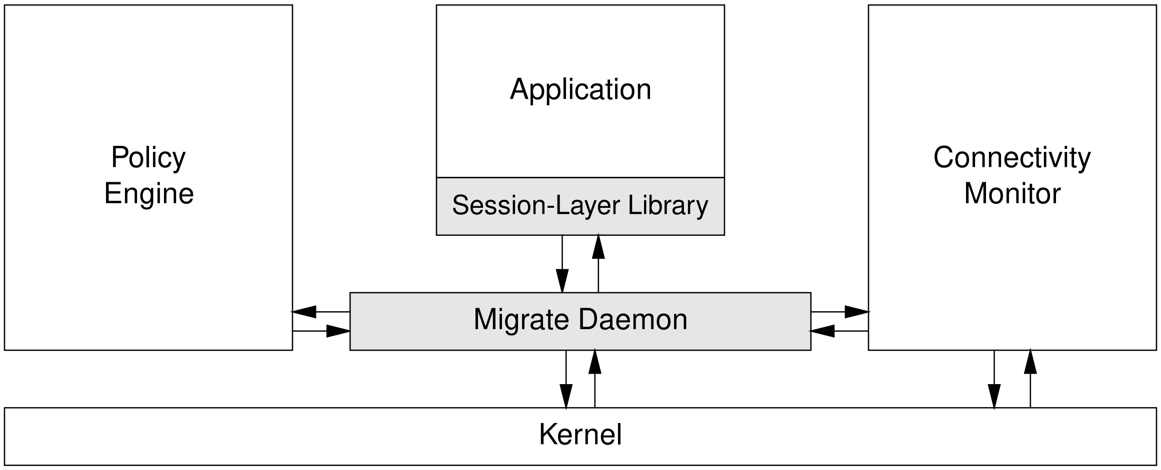

Figure 6-1: The components of the *Migrate* architecture. Applications export sessions, which are managed by the *Migrate* system daemon in collaboration with various connectivity monitors and policy engines.

```
struct migrate_session_t {
  int                  _fd;                   /* TESLA-internal fd for ioctl */
  int                  id;                    /* Local session ID            */
  int                  pid;                   /* ID at remote end point      */
  migrate_state        state;                 /* Session state               */
  struct sockaddr_in   laddr;                 /* Local end point             */
  struct sockaddr_in   paddr;                 /* Remote end point            */
  void *               db;                    /* Attribute/value store       */
  int                  flags;                 /* Session flags               */
  char                 dname[M_MAXNAMESIZE];  /* Local end-point name        */
  char                 pname[M_MAXNAMESIZE];  /* Remote end-point name       */
  migrate_lookupfunc   newf;                  /* Function to find peer       */
  mig_handler          handler;               /* Mobility handler function   */
  size_t               pbufsize;              /* Default peer RCVBUF size    */
};
```

Figure 6-2: The *Migrate* daemon's internal session structure.

6.1.1 Session and connection management

Once operational, the *Migrate* daemon registers each connection associated with a *Migrate* session with the available connectivity monitors and receives any changes in network availability or session connectivity. Upon receiving an event, the daemon consults the policy engine to decide how to handle it. Options include informing the application (the common case), forcing a migration (*e.g.*, when a cheaper network attachment point becomes available), suspending the session, or delaying or ignoring the notification—possibly to avoid excessive flapping in extremely variable conditions.

Figure 6-2 shows the data structure the *Migrate* daemon uses to manage individual sessions. Each daemon issues a locally unique ID to each session. When communicating with the remote end point, sessions are identified by combining their local and remote IDs. For example, a session may be known as 4 on one end point and 7 on the other; the tuple 4:7 then uniquely identifies the session on both end points. In addition to information about the session's end points, state, and application-defined mobility handlers, the daemon maintains additional operational information including the

```
struct migrate_connection_t {
  int                  fd;           /* Virtual file descriptor         */
  int                  rfd;          /* Real file descriptor            */
  migrate_session *    session;      /* Parent session                  */
  struct sockaddr_in   saddr;        /* Initial local attachment point  */
  struct sockaddr_in   daddr;        /* Initial remote attachment point */
  struct sockaddr_in   csaddr;       /* Current local attachment point  */
  struct sockaddr_in   cdaddr;       /* Current remote attachment point */
  int                  type;         /* SOCK_STREAM | SOCK_DGRAM        */
  sync_state           sync;         /* (TCP) Current sync state        */
  unsigned int         sndseq;       /* (TCP) Current sequence number   */
  unsigned int         rcvseq;       /* (TCP) Current sequence number   */
  unsigned int         ackseq;       /* (TCP) Last received byte        */
  struct _ring_t *     ring;         /* (TCP) Ring buffer               */
  unsigned int         refcnt;       /* Connection reference count      */
};
```

Figure 6-3: The *Migrate* daemon's internal connection structure.

IDs of the user and process that own the local session end point, the session's attribute/value store, and the negotiated default size of the remote end point's TCP buffer (see Chapter 4 for a discussion of its use).

When managing a session, if either the owning application or the system policy engine informs the daemon a session should react to an event (by suspending, migrating, or resuming), the daemon contacts the remote end point. It does so by communicating over a separate TCP control channel as described in Chapter 3. This control channel is created during session establishment or when resuming previous sessions after periods of disconnection. Control channel establishment is initiated by the connecting/migrating end point by sending a message to the `MIGRATE_PORT` on the remote host. We assumed that contacting different ports at the same IP address results in connections with the same end point. While this assumption generally holds in the direction of connection establishment, it may be invalidated by various esoteric forms of NAT [80].

Session resumption requires the reestablishment of open network connections, as discussed before. The *Migrate* daemon's connection data structure is shown in Figure 6-3. Connections are named by their initial local and remote attachment points. In order to conserve system resources such as network port space, system file descriptors, and kernel socket buffer space (in the case of TCP) during disconnection, the daemon *aborts* the network connections by setting the TCP linger timer to 0 and closing the connection [123] to ensure that the resources are freed immediately instead of after TCP times out, which could take several minutes. This technique effectively multiplexes all suspended network connections onto one well-known port (`MIGRATE_PORT`). The majority of the state recorded about open connections exists to resume the connection after closing them in this way.

6.1.2 NAT Challenges

The presence of Network Address Translation (NAT) [122] can make it difficult for *Migrate* to explicitly name individual connections inside the same session. In particular, a Port Network Address Translator obscures not only the IP address, but the transport-layer port as well. Further, the mapping from port to original attachment point is known only by the NAT and not to either end point. Therefore, multiple connections destined for the same remote attachment point that pass through a

PNAT are indistinguishable from each other, since the only normally distinguishing feature, the port number, is meaningless.

This behavior causes great difficulty when *Migrate* attempts to resume sessions containing multiple connections to the same remote port (as is common in Web browsing sessions, for example, where a client has multiple TCP connections between itself and port 80 on a remote server). When attempting to establish new connections to map to the old ones, *Migrate* is unable to specify which connection corresponds to which local port, since there are no differentiating factors (visible outside of the kernel, at least), aside from what was actually transmitted on the connection. While Rabin fingerprinting [101] or a similar approach could be used to differentiate based on the content transmitted along each connection, it is possible that two connections may have transmitted exactly the same thing up to a point (standard protocol headers, for example) but will transmit different things after session resumption (meaning they are not actually interchangeable). Hence, our current *Migrate* implementation does not allow multiple connections in the same session to share the same remote port in the presence of a NAT. This shortcoming can be worked around by creating multiple sessions—one for each connection destined to the same remote attachment point.

6.1.3 Continuation management

When an application generates a continuation, either by passing it as a return value from its mobility handler function or through the `return_cont()` function, the *Migrate* library considers whether the continuation should be executed immediately or passed to the *Migrate* daemon, where it may be composed with additional resource continuations (see Section 5.3). If a session is connected (in the *established* state) when an application generates a continuation—an infrequent occurrence—*Migrate* invokes it immediately. Otherwise, the continuation is passed up to the *Migrate* daemon for storage until the session connectivity is restored. *Migrate* currently defaults to lazy continuation generation, and preserves continuation resources *in situ*. If system resources become constrained, however, *Migrate* is capable of composing application continuations with various resource continuations to more efficiently preserve the memory and file descriptors preserved by the continuation. We describe resource continuations in the context of complete continuations in the Section 6.1.5.

To invoke an internal continuation, the *Migrate* daemon must temporarily interrupt the execution of application process that generated the continuation and cause it to pass control to the function specified by the continuation. We emphasize that mobility handlers and internal continuations are always executed inside the process space of the application process that generated them, thereby avoiding any concerns about their security or robustness. In the rare case that the continuation was returned by a mobility handler function while the session is connected, the application process has already been interrupted by the mobility handler. Hence, the *Migrate* library can immediately execute the continuation function in the context of the same signal handler that invoked the mobility handler. In most cases, however, process execution must be interrupted again by a signal, and the continuation function invoked by the signal handler.

In either case, the process signal mask is set to ensure that further mobility events will be queued until the continuation function has returned. It is further expected that continuations are not recursive (*i.e.*, they don't call `return_cont()` internally). These two conditions combined ensure that both mobility handlers and continuations are not re-entrant—freeing programmers from the need for synchronization or mutual exclusion.

6.1.4 Complete continuations

Complete continuations are considerably more complicated than internal continuations. In the case of a complete continuation, *Migrate* terminates the original application process after the continuation is generated and instantiates a new one once session connectivity is restored. In many ways, the challenges faced are similar to those presented in process migration [29, 93, 134], except that the new process executes on the same host as the previous process. In both cases, the system must do one of three things for each resource used by the application process: transfer state from the original process to the new one, arrange for forwarding, or use similar state from the new process and sacrifice transparency [29]. Since a complete continuation executes on the same host as the original process, resource forwarding is straightforward to implement in *Migrate*.

Continuations are designed to limit the scope of the state and resources that must be transferred from the old process to the new and are explicitly *not* transparent. Only critical session state (indicated by its presence in the session's attribute/value store) is transferred to the new process. Similarly, only network connections and those file descriptors include in the continuation are forwarded to the new process. All other state from the original process, including code and data segments, open files, message channels, execution state, and the process control block, is discarded. In the interest of system security, however, some aspects of the process control block, such as the real and effective user ID, priority, and current working directories are transferred to the new process. We now describe the step-by-step process followed by *Migrate* to generate and invoke complete continuations. This process is believed to be fully POSIX-compliant; we have tested it on Linux and FreeBSD.

Upon receipt of a complete continuation from an application, the *Migrate* daemon saves the session's attribute/value store to stable storage (a temporary file). File descriptors contained in the continuation are preserved by using POSIX file descriptor passing to move them into the *Migrate* daemon, where they are held for later forwarding to the new process, and may be additionally processed by individual resource continuations as described below. Finally, various operational aspects of the application process, such as its real and effective user IDs, priority, signal mask, etc., are recorded, and then the process is terminated (by calling `_exit()`, which avoids the invocation of any clean-up functions registered through the use of the `atexit()` call.

Occasionally, an application may ask to preserve a file descriptor that corresponds (unbeknownst to the application) to a network connection that is part of a *Migrate* session. This situation often occurs when applications open connections to local server applications, such as an X server, and the *Migrate* library provides transparent service (as described in Section 6.2.2). In this case, if the connection in question is the only connection in its session the *Migrate* daemon suspends the relevant session, causing the network connection to be suspended and restored when the continuation is executed.

To invoke a complete continuation, the *Migrate* daemon first forks a new process. Ideally, the process control block of the original process would be transferred to this new process. Unfortunately, not all attributes are recoverable without operating system support. In particular, the process ID and parent/child relationships are nonmalleable in a POSIX system, so it is impossible to create a new process with a particular process ID. Similarly, it is impossible to create a process that becomes the parent of an existing orphaned child of a previous process, nor that becomes the child of anything other than the process that created it. While it is possible for *Migrate* to trap system calls such as `getpid()` and `getppid()` in the continuation process to provide the appearance of the original process, doing so could result in unpredictable behavior depending on what the application attempted to do with its process ID (*e.g.*, signal other processes in its process group, etc). Therefore, *Migrate* simply exposes the attributes of the new process to the application, regardless of their relation to the properties of the previous process. Applications that depend on knowledge of such attributes can obtain the appropriate values as part of their continuations.

Instead of preserving transparency, the *Migrate* daemon ensures that the new process exists on its own and is not associated with the daemon process in any way. This separation prevents the new process from having any impact on the *Migrate* daemon's operation, despite any security faults or bugs that may exist in the application. To separate the new process from the *Migrate* daemon, the daemon forks a child process which immediately forks again. The child process then exits, resulting in an orphaned grandchild process, which becomes its own process group leader. By calling `wait()` immediately after forking, the *Migrate* daemon frees the resources that would otherwise be consumed by the zombie child process. The resulting grandchild process then sets its priority and effective user ID to be the same as those of the process that generated the continuation. It also creates a new POSIX process session by calling `setsid()`. It does not, however alter its controlling tty[1]. Rather than deal with the complicated issues of foreground and background process groups, *Migrate* assumes resumed processes will run without a controlling tty. Processes interested in recapturing a tty can do so in their own continuation. Recall that the continuation can preserve arbitrary file descriptors, such as `stdin` and `stdout`, which a process may use to determine an appropriate tty.

File descriptors preserved by the continuation are forwarded to the new process by the *Migrate* daemon through file descriptor passing. Any file descriptors that were released during suspension are recreated by the daemon before passing. Once passed to the new process, the descriptors are restored to their original IDs through the use of `dup2()`, providing needed consistency to the application.

Finally, the new process changes the working directory to that specified in the `cwd` field of the continuation. Immediately before calling `execve()` with the `argv` and `envp` values from the continuation (`argv[0]` is treated as the executable), the process restores the signal mask in place in the original process when the continuation was generated. To assist applications in discovering the identity of the session being continued, the *Migrate* daemon adds two variables to the process' environment: `MIGRATE_CONT` contains a file descriptor corresponding to a resumed connection that is a member of the restored session; applications will likely use this value as the parameter to `get_session()` to recover the session itself. The second variable, `MIGRATE_PID`, contains the process ID of the previous process. Applications may wish to use the ID to communicate with orphaned children of the previous process, which presumably assumed control of the process group of the previous process.

6.1.5 Resource continuations

When file descriptors are passed into the *Migrate* daemon as part of a continuation, *Migrate* considers generating resource-specific continuations to more efficiently preserve them. The daemon determines what resources individual file descriptors correspond to through the use of `getsockname()` and `fstat()`. Once aware of the resources in question, the daemon can generate resource-specific continuations such as those described in Section 5.3. One trivial continuation is to close any redundant file descriptors (*i.e.*, multiple file descriptors to the same resource). Many interactive applications typically have three different file descriptors (`stdin`, `stdout`, `stderr`) that correspond to the same resource (generally a tty). Similarly, depending on the locking semantics in use (record, file, etc.), multiple file descriptors pointing to the same file can often be collapsed into one. Closing the only file descriptor pointing to a particular resource can be dangerous, depending on the

[1] A tty, short for teletype, is an interactive terminal, normally corresponding to a user console.

semantics expected by the application. For example, closing all references to a file may allow the file to be subsequently deleted or overwritten. In the case of NFS, however, these operations can occur anyway, so *Migrate* could release all references without affecting NFS semantics. In order to ensure the operating system semantics are never affected, however, *Migrate* always keeps at least one file descriptor open for each resource contained in a continuation.

Because the *Migrate* daemon is an application just like any other, file descriptors maintained by the daemon are managed by the operating system in the same way they would be if they were being held by the application process that generated the continuation. The daemon neither reads from nor writes to file descriptors in its possession and blocks signals associated with them, so any pending data or events are delivered to the application process by the operating system once the continuation is invoked. The only complication arises in operating systems that perform resource-level accounting [10]. In some cases, resources that should be associated with a suspended application could be mistakenly charged to the *Migrate* daemon while it stores the application's continuation. In such environments, the daemon might need to explicitly indicate resources in its possession that should be charged to another resource principle. We have not yet tested *Migrate* on an operating system with such fine-grained resource accounting.

6.2 Session-layer library

The *Migrate* session-layer library provides stub routines for each of the various *Migrate* API calls and communicates with the *Migrate* daemon using remote procedure calls (RPC) [14]. In addition to relaying application requests, the library receives asynchronous notifications from the daemon of mobility events, which it may either handle internally or pass on to an application-provided handler. The asynchronous notification mechanism is implemented through the use of POSIX signals. By default, *Migrate* uses `SIGUSR2` for its internal notifications but could dynamically adapt to another, unused signal, if the application process installs a handler for `SIGUSR2`.

6.2.1 TESLA

Migrate uses TESLA [110], an interposition agent toolkit, to implement the session-layer library. TESLA is a framework specially designed for network interposition agents, allowing interposition agents to expose their own extended APIs to applications. In addition to adding functionality to applications that are linked against TESLA handler libraries at compile time, TESLA can use the dynamic linker to insert itself between any dynamically liked application and the system C library, *libc*, where it intercepts all network-related system calls and routes them to service handlers such as *Migrate*. The full list of functions intercepted by TESLA is shown in Figure 6-4.

Using TESLA, *Migrate* support can be added to legacy applications at run time. *Migrate* is implemented in two TESLA handlers, `migrate` and `migrateudp` for TCP and UDP sockets, respectively. For example, if a user wishes to enable *Migrate* support when using an unmodified SSH client to connect to `nms.lcs.mit.edu`, a user would simply type:

```
% tesla +migrate ssh nms.lcs.mit.edu
```

If the SSH server were *Migrate*-enabled, either through the use of a *Migrate*-aware server (like the one presented in Chapter 7), or by wrapping the SSH server with TESLA as well (*e.g.*, invoking the server through "`tesla +migrate sshd`"), the user's SSH session would survive changes in client and server attachment points and arbitrary periods of disconnection.

```
/* The following socket functions are redefined by TESLA */

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
int bind(int s, const struct sockaddr *addr, socklen_t addrlen);
int close(int fd);
int connect(int s, const struct sockaddr *addr, socklen_t addrlen);
int listen(int s, int backlog);
int socket(int domain, int type, int protocol);
int dup(int fd);
int dup2(int fd, int newfd);

int getpeername(int s, struct sockaddr *addr, socklen_t *addrlen);
int getsockname(int s, struct sockaddr *addr, socklen_t *addrlen);

int getsockopt(int s, int level, int optname, void *optval,
               socklen_t *optlen);
int setsockopt(int s, int level, int optname, const void *optval,
               socklen_t optlen);

/* These are overloaded by TESLA only for SOCK_DGRAM sockets */

int read(int s, void *msg, size_t len);
int write(int s, const void *msg, size_t len);

int send(int s, const void *msg, size_t len, int flags);
int recv(int s, void *msg, size_t len, int flags);

int recvfrom(int s, void *msg, size_t count, int flags,
             struct sockaddr *from, socklen_t *fromlen);
int sendto(int s, const void *msg, size_t count, int flags,
           const struct sockaddr *to, socklen_t tolen);

/* These should be overloaded by TESLA in the future */

int recvmsg(int s, struct msghdr *msg, int flags);
int sendmsg(int s, const struct msghdr *msg, int flags);
```

Figure 6-4: The library functions wrapped by TESLA. UDP (SOCK_DGRAM) sockets require extra care, as they may demand per-datagram processing (*e.g.*, address rewriting). The current TESLA implementation does not yet support scatter/gather I/O.

The availability of *Migrate* support to an application depends on both the presence of a *Migrate* daemon and the TESLA handlers. Because these conditions can only be validated at run time, we provide a macro, `migrate_avail()`, that allows *Migrate*-aware applications to determine if support is currently available. If the application is currently running with the TESLA *Migrate* handlers, and they are able to communicate with a local daemon, `migrate_avail()` returns one; it returns zero otherwise. The presence of local *Migrate* support does not ensure that the remote end point is also *Migrate* capable, however. Since some applications may demand *Migrate* support, we provide a special socket option that can be used to specify that a network connection *must* be included in a *Migrate* session. Applications can use `setsockopt()` to enable the `MIG_DEMAND` option at the `SOL_MIGRATE` level. When the `MIG_DEMAND` option is set, the `connect()` and `accept()` system calls will fail if the remote end point does not support *Migrate*.

6.2.2 Transparent interface

When legacy applications are linked against the TESLA *Migrate* library (either through recompilation or at run-time through TESLA's dynamic linking), TESLA intercepts calls to the standard POSIX sockets API [67] (`connect()`, `accept()`, `sendto()`, etc.) and invokes the *Migrate* handler, which wraps each network connection in a *Migrate* session. While such applications cannot take full advantage of the services available through the *Migrate* API, system-wide policy may specify default session handling, including the transparent preservation of open network connections and suspension of the application (by blocking any network-related system calls) or buffering of transmitted data (by copying to stable storage) during periods of disconnection. Figure 6-5 shows how the TESLA *Migrate* handler wraps connections established through the standard socket API with an enclosing session abstraction.

One drawback of TESLA, however, is that its architecture has significant performance implications [110]. In particular, TESLA implements the vast majority of its functionality in a separate process, so all applications using *Migrate* will create an additional `teslamaster` process that actually contains the *Migrate* API functionality; only a very thin stub layer is left in the application process. We evaluate the overhead of our TESLA-based implementation in the next chapter.

6.3 Policy engine

Ideally, *Migrate*-aware applications would assist the user in managing her mobility preferences and coordinate them with *Migrate* through a policy API. Such an interface could allow for dynamic adjustments to local policy based on the particular activity the application was engaged in at any particular moment (*i.e.*, support the suspension of important users' sessions for longer periods of time, etc.).

Unfortunately, most applications currently lack direct interfaces to allow users to describe mobility handling preferences. Therefore, *Migrate* provides for a rule-based user policy file in which users can express their preferences in terms of which local network attachment point to use for particular applications. In its current incarnation, *Migrate* requires users to specify sessions based upon the ports their contained transport connections. For example, if a user wished to specify policy for Web browsing, she would insert a policy rule for the TCP protocol on the well-known HTTP server port (80).

Migrate consults the policy file at every mobility event: the creation of a session, a change in the set of available local attachment points, or change in the connectivity status of a particular session end point (tracked as described in the following section). In each case, the policy file answers the

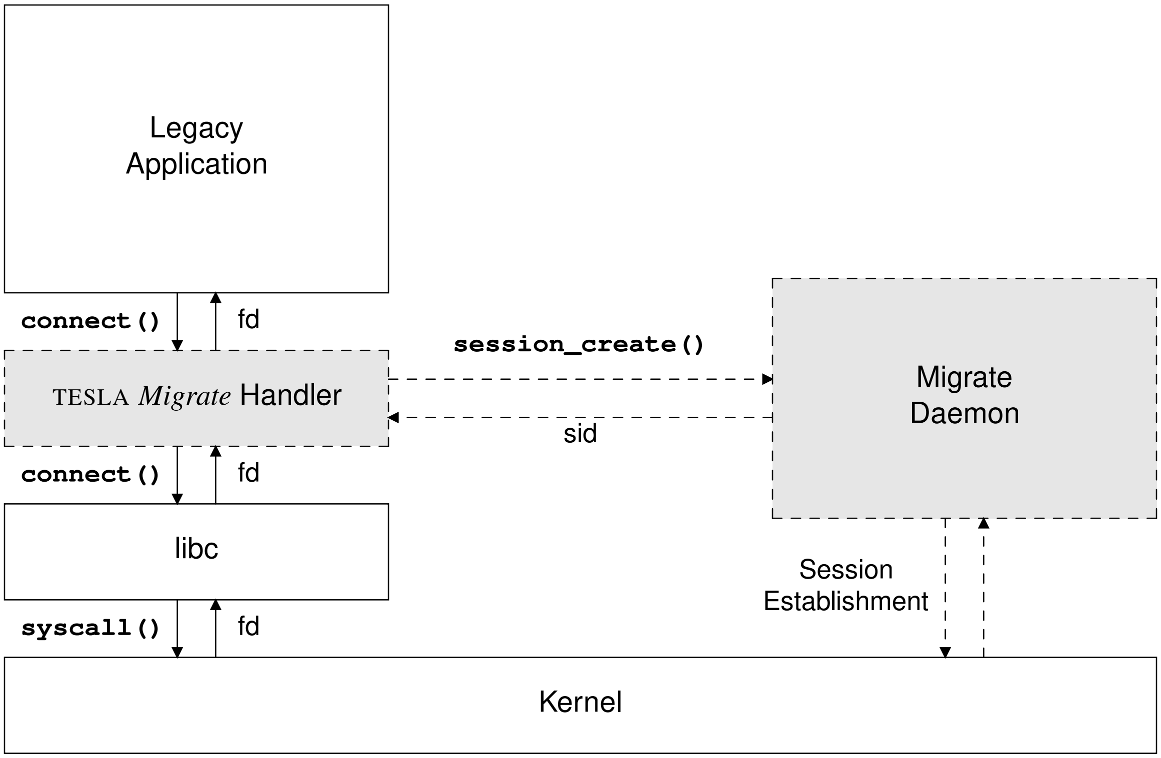

Figure 6-5: Dynamic library interposition for transparent operation with legacy applications. When the session-layer library is interposed between a legacy application and the system (either through relinking or TESLA's run-time library interposition), the *Migrate* handler transparently encapsulates network connections in *Migrate* sessions. These sessions are managed according to local system policy.

question, "Which local attachment point, if any, is best suited for a particular session?" Each session is considered in turn. The input for each invocation is a description of the network connections comprising the session in question—their protocols and current remote and local attachment points, including ports. The file directs *Migrate* to take one of two actions: either migrate the session to a new interface, or leave it alone. For each available attachment point, a score is computed; the attachment point with the highest score for a particular session is selected, provided the score is sufficiently higher than that awarded to the current attachment point. Figure 6-6 shows a sample *Migrate* policy file. A full description of the policy file language and its capabilities can be found in Appendix A.

6.4 Connectivity monitoring

To decide if an end point is connected at its current attachment point, *Migrate* needs to monitor the connectivity of on-going sessions. What constitutes "connected" varies from session to session, however. To be connected, application sessions may require a certain level of connectivity, expressed in terms of available bandwidth, maximum latency, or similar metric. Hence, *Migrate* employs a suite of modular *connectivity monitors* to assist in evaluating the current levels of connectivity being experienced by each session. Ideally, applications could specify what metrics they are interested in, and *Migrate* would select an appropriate connectivity monitor from the available suite automatically on a per-session basis. Unfortunately, we have not yet implemented such a feature.

```
# A sample Migrate policy file

monitor−policy {
   score−interface * 1
   score−interface eth* 10
   score−interface eth0:1 1000
   score−interface ppp* 5

   on proto tcp {
      on remote−ip 18.31.0.4/24
         on port ssh
            score−interface eth* 100
      on l−port 3001−3010 {
         score−interface eth* 200 ;
         score−interface eth0:1 100
      }
      on port http, ftp
         migrate never
   }
}
```

Figure 6-6: A sample policy file. In this example, `eth0` is preferred to other `eth` interfaces, which are preferred to `ppp` interfaces, which are preferred to all others. Sessions containing TCP SSH connections to remote attachment points with IP addresses in the 18.31.0 subnet have an increased affinity for `eth` interfaces, and TCP connections on local ports 3001–3010 actually prefer `eth0` less than other `eth` interfaces. Finally, sessions containing TCP connections to HTTP or FTP server ports are never migrated.

Currently, connectivity monitors must be manually configured and they monitor connectivity for all active sessions, regardless of application.

Connectivity monitors may gather information from a variety of sources, including the physical and network layers (*e.g.*, loss of carrier, power loss, change of address, etc.), the end point applications themselves, or appropriately authorized external entities (*e.g.*, [7]) that may be concurrently monitoring connection state. Since a session may span multiple protocols, connections, and application processes, there may be several sources of connectivity information for any particular session. Regardless of the source, *Migrate* handles connectivity information in the same fashion.

6.4.1 Monitoring API

Whenever a session is established, *Migrate* informs all connectivity monitors that have registered ability in monitoring sessions with matching characteristics (*e.g.*, local and/or remote attachment point and transport protocol match). Figure 6-7 shows the format of registration messages. Each connection is described individually; sessions without any active connections can be monitored through their TCP session-control connection. The connectivity monitors subsequently notify *Migrate* with any connectivity changes using the same message format. Examples of possible connectivity monitors are discussed below.

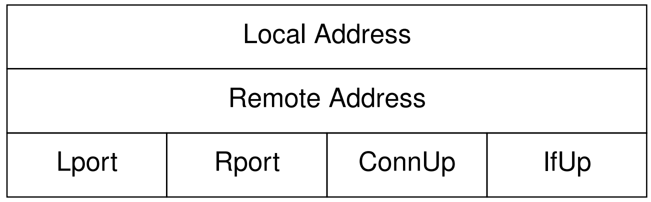

Figure 6-7: Connection status message format. Connectivity monitors use this message to inform *Migrate* of changes in connectivity status for an individual connection. The addresses and ports are the *current* connection end points. ConnUp specifies whether the connection currently appears to have connectivity. IfUp indicates whether the local network interface being used by the connection is currently available.

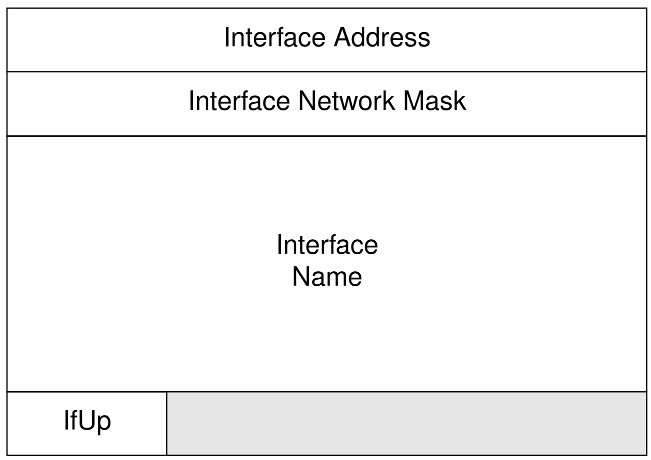

Figure 6-8: Interface connectivity monitor message format. The Interface Name is a 16-character ASCII string reported by the kernel (*e.g.*, `lo`, `eth0`, etc.).

6.4.2 Physical layer

A monitor can glean connectivity information from the network interface itself. For example, most devices report some notion of carrier presence (*i.e.*, whether the cable is plugged in, whether the physical device is operating properly, etc.). The absence of such a carrier indicates complete connectivity failure for all attachment points on that interface. Similarly, many wireless interfaces report signal strength, which a monitor might be able to use to estimate the likelihood of transmission success—the lower the signal strength, the higher the expected loss rate on the link.

The current implementation of *Migrate* provides a connectivity monitor that watches the physical status of network interfaces, and notifies the *Migrate* daemon as network interfaces become operational or lose connectivity due to physical factors. Because many connections often use the same local attachment point (interface), interface monitors can use the message format shown in Figure 6-8 to concisely describe the status of a network interface. This avoids the need for the connectivity monitor to issue status updates for every connection using a local attachment point on the interface in question.

6.4.3 Network and transport layers

There are a variety of ways to monitor end-to-end connectivity [4]; we will not discuss most of them here, except to say that *Migrate* can accept input from any of these methods if the monitor interfaces with the *Migrate* connectivity monitor API. One method in particular, however, is especially useful for TCP connections.

TCP is acknowledgment-based, meaning each transmitted byte must be explicitly acknowledged by the receiver. If a byte is not acknowledged, TCP will attempt to retransmit it after a period of time known as the retransmission timeout (RTO).[2] The lack of a response to subsequent retransmissions leads to an exponential increase in the RTO [87]—a so-called exponential back-off algorithm. Hence, the "health" of a TCP connection can be described by the value of a connection's RTO—the larger the value, the greater the connection's distress. The current *Migrate* implementation provides a connectivity monitor that uses this information to assess the connectivity of sessions containing TCP connections.

Because *Migrate*'s handlers are interposed between the application and the transport protocol, *Migrate* can observe any notifications from the network to the application, such as socket errors like "host unreachable" or "connection reset." Such messages often indicate a lack of connectivity between end points. When used in conjunction with virtualized transport protocols (Section 4.1), *Migrate* intercepts these messages an interprets them as potential notifications of disconnection. *Migrate* immediately attempts to verify connectivity by sending a keep-alive message on the corresponding session's control channel. If this message succeeds, the error condition is passed up to the application. Otherwise, the error condition is masked and treated in the same fashion as a disconnectivity notification from an external connectivity monitor.

6.4.4 Application layer

Applications themselves may gather connectivity information, perhaps by passively monitoring their connectivity, or through active probing (*e.g.*, keep-alive proves). Such applications may interface with *Migrate* directly, sharing their view of the session with *Migrate*. If applications provide such modular connectivity monitors, *Migrate* may be able to use this information to better manage sessions with shared end points in a fashion similar to the Congestion Manager [7]. We have not yet extended any applications to support this feature.

6.4.5 Aggregation

Active connectivity monitoring can be expensive in terms of bandwidth needed for probing and space used to store historical measurements. Network partitions typically effect entire sub-nets as opposed to individual hosts. Thus, it may be beneficial to share connectivity information across clusters of machines [115], not only multiple sessions on one end host [7]. Hosts may wish to interface with non-local connectivity monitors that can provide increased coverage at lower expense. We are considering implementing such a monitor in the context of the Reliable Overlay Network (RON) testbed [4].

6.5 Connection migration

Unlike most of *Migrate*, which is implemented in a generic, POSIX-compliant fashion, the Migrate TCP options must be integrated directly into a system's TCP stack, which often resides in the kernel itself, resulting in an operating-system-specific implementation. We have implemented the TCP Migrate options in the Linux 2.2 kernel. The IPv4 TCP stack has been modified to support the

[2] In fact, a TCP sender may attempt an earlier retransmission in response to a duplicate ACK [124]. The RTO is a persistent, timer-based retry mechanism used as a fail-safe.

```
winchester: >ls -l /proc/net/migrate/

total 0
-rw-r--r-- 1 snoeren root  0 Jul 29 09:32 127.0.0.1:1026->127.0.0.1:22
-rw-r--r-- 1 root    root  0 Jul 29 09:32 127.0.0.1:22->127.0.0.1:1026
-rw-r--r-- 1 root    root  0 Jul 29 09:36 18.31.0.66:22->18.31.0.82:1022
-rw-r--r-- 1 root    root  0 Jul 29 09:38 18.31.0.66:80->18.31.0.82:1023
```

Figure 6-9: A directory listing showing open *Migrate*-capable TCP connections. There are currently three connections: A local SSH connection, a remote SSH login, and an HTTP download.

Migrate and Migrate-Permitted options. We first describe the interface to connection migration and, then, examine the details of the Migrate-Permitted option. The implementation of the Migrate option will be discussed in the following section when we examine the cryptographic alternatives.

6.5.1 Controlling migration

TCP connection migration happens in two ways. Applications with open connections can explicitly request a migration by issuing an `ioctl()` on the connection's file descriptor specifying the local address to migrate to. Most current applications, however, lack a notification method for the system to inform them that the host has moved. Hence, we also provide a mechanism for external entities to migrate open connections, even if they do not have file descriptors corresponding to the connections.

This external migration is affected through the Linux `/proc` file system. Files of the form

source IP address:*source port*->*destination IP address*:*destination port*

are inserted in the `/proc/net/migrate` directory for each open connection that has successfully negotiated the Migrate-Permitted option. These files are owned by the user associated with the process that opened the connection. Any process with appropriate permissions can then write a new IP address to these files, causing the corresponding connection to be migrated to the specified address. A sample directory listing is shown in Figure 6-9. This method has the added benefit of being easily accessible to a user through the command line.

6.5.2 Key negotiation

One of the most complicated aspects of implementing the Migrate TCP options is negotiating secret keying material to sign the Migrate SYNs. End points wishing to initiate a migrateable TCP connection send a Migrate-Permitted option in the initial SYN segment. Upon receipt of an initial SYN with a Migrate-Permitted option, an end point with a Migrate-compliant TCP stack will include a Migrate-Permitted option in its SYN/ACK segment. Similar to the SACK-Permitted option [66], it should only be sent on SYN segments and not during an established connection. Additionally, end points wishing to cryptographically secure the connection token may include up to 200 bits of keying material.

As seen in Figure 6-10, the Migrate-Permitted option comes in two variants—the three-byte-long insecure version and 20-byte-long secure version. The secure version is used to exchange cryptographic keying material and contains an eight-bit *Curve Name* field and a 136-bit *Keying Material* fragment. The Curve Name field selects a set of elliptic curve parameters for use in an elliptic-curve Diffie-Hellman (ECDH) key exchange. The value indicates a particular set of domain parameters

Kind: 15	Length = 3/20	Method Name	Keying M.
Keying Material (cont.)			
Keying Material (cont.)			
Keying Material (cont.)			
Keying Material (cont.)			

Figure 6-10: TCP Migrate-Permitted option

Curve Name	Method	Curve Parameters	Key Length
0x00	None	N/A	N/A
0x01	ECDH	Annex J.4.1, example 1	163 bits
0x02	ECDH	Annex J.4.1, example 2	163 bits
0x03	ECDH	Annex J.4.1, example 3	163 bits
0x04	ECDH	Annex J.4.2, example 1	176 bits
0x05	ECDH	Annex J.4.3, example 1	191 bits
0x06	ECDH	Annex J.4.3, example 2	191 bits
0x07	ECDH	Annex J.4.3, example 3	191 bits
0x08	ECDH	Annex J.4.3, example 4	191 bits
0x09	ECDH	Annex J.4.3, example 5	191 bits
0x0a	ECDH	Annex J.5.1, example 1	192 bits
0x0b	ECDH	Annex J.5.1, example 2	192 bits
0x0c	ECDH	Annex J.5.1, example 3	192 bits

Table 6.1: Defined Curve Name values and their corresponding mechanisms. The table shows the corresponding elliptic curve parameters from the ANSI X9.62 standard [3]. This list may grow to reflect further published elliptic curves with key lengths less than 200 bits.

(the curve, underlying finite field F and its representation, the generating point P and its order n), as specified in [3]. Table 6.1 shows the list of currently defined method name values. Use of the insecure version, which contains only a Curve Name field (which must be set to zero) allows the end points using network-layer security mechanisms such as IPsec [55] to avoid additional cryptographic overhead.

The secure variant of the Migrate-Permitted option also allows the use of the Timestamp [50] option in order to store up to 200 bits of keying material. An additional 64 bits of material can be placed in the Timestamp option. Figure 6-11 shows the Migrate-Permitted option along with a set of other TCP options commonly included on SYN segments; the fields used to store the Migrate-Permitted keying material are shown in gray. The Timestamp option, while often included, is not used on SYN segments. The Protection Against Wrapped Sequence Numbers (PAWS) [50] check is only performed on synchronized connections, which by definition [97] includes only segments after the three-way handshake. Similarly, the round-trip time measurement (RTTM) procedure only functions when a timestamp has been echoed [50]; a timestamp is never echoed on an initial SYN segment. Hence, the value of the Timestamp option on SYN segments is not meaningful to current TCP stacks. Because legacy TCP stacks will never receive a Migrate-Permitted option on a

<table>
<tr><td>MSS</td><td>Len = 4</td><td colspan="2">Maximum Segment Size</td><td>NOP</td><td>Window Scale</td><td>Len = 3</td><td>Scale</td></tr>
<tr><td>SACK Perm.</td><td>Len = 2</td><td>Timestamp</td><td>Len = 10</td><td colspan="4">Timestamp Value</td></tr>
<tr><td colspan="4">Timestamp Reply</td><td>Migrate Perm.</td><td>Len = 20</td><td>Method</td><td>Keying M.</td></tr>
<tr><td colspan="8">Keying Material (cont.)</td></tr>
<tr><td colspan="8">Keying Material (cont.)</td></tr>
</table>

Figure 6-11: One possible set of TCP options. Our Linux Migrate TCP implementation sends these forty bytes of TCP options by default in TCP SYN segments. Four options are requested: maximum segment size (MSS) (four bytes), window scaling (three bytes), selective acknowledgments (SACK) (two bytes), and Migrate (20 bytes). The fields used to store the 200 bits of Migrate-Permitted keying material—64 bits of the Timestamp option and 136 bits from the Migrate-Permitted option—are shaded. One byte of padding is inserted (a NOP option) to preserve 32-bit word alignment.

SYN/ACK, the Timestamp option will be processed normally. Special handling is only required for the SYN/ACK and following ACK segment on connections that have negotiated the Migrate-Permitted option, as Timestamp fields on these segments will not contain timestamps. Thus, the RTTM algorithm must not be invoked for SYN/ACK or initial ACK segments of connections that have negotiated the Migrate-Permitted option.

Once keying material is exchanged, it can be used to sign and validate Migrate SYNs. While the implementations differ, this process is conceptually quite similar to session resumption. In the next section we present an overview of the cryptographic techniques used for both.

6.6 Cryptography

The authentication needs of both session migration and TCP connection migration are essentially identical—both need to authenticate a new remote attachment point as being the same end point that initiated the session/connection. *Migrate* currently provides this service through a Diffie-Hellman key exchange. Using Diffie-Hellman, both *Migrate* sessions and TCP connections using the Migrate options are able to support an arbitrary number of migration events but require non-negligible computation during each event. In this section, in addition to presenting the details of our current, Diffie-Hellman-based authentication scheme, we introduce an alternative authentication method based on one-way functions that entails almost no computational effort during migration in return for additional storage. The main drawback with the one-way-function-based approach, however, is that only a fixed number of migration events can be secured and the number must be specified initially and cannot be extended. Here we examine the cryptographic primitives used in both approaches and discuss our current implementation based on Diffie-Hellman. We have not yet implemented authentication based on one-way functions in *Migrate* but present it here as an attractive alternative for future implementations.

6.6.1 Elliptic curve Diffie-Hellman

An *exponential key exchange*, colloquially referred to as Diffie-Hellman after its inventors [28], is a well-known cryptographic primitive that allows two parties to conduct a public conversation (*i.e.*, eavesdroppers may overhear the messages being exchanged) and arrive at a private secret. That is, it provides a mechanism for secretly exchanging keying material between two unauthenticated parties in the presence of passive adversaries. There are many variants of Diffie-Hellman, each drawing their strength from different cryptographic challenges. One particular variant, ECDH, produces short keys which are thought to have significant cryptographic strength [3]. Due to the limited space in a TCP SYN segments, we use ECDH to secure the TCP Migrate options.

When using ECDH, the keying material negotiated by the Migrate-Permitted option is an ECDH public key, encoded using the ANSI-standard compressed conversion routine [3, Section 4.3.6]. The 136 least-significant bits are stored in the EDCH Public Key field of the Migrate-Permitted option and the remaining 64 bits of the key are encoded in the Timestamp option. These two components constitute the end point's public key, k. If the public key is less than 200 bits, it is left-padded with zeros. For any end point i, k_i is generated by selecting a random number, $X_i \in [1, n-1]$, where n is the order of P, and computing

$$k_i = X_i * P.$$

The $*$ operation is the scalar multiplication operation over the field F. Because the security of the connection hinges on the secrecy of the negotiated key, X_i should be randomly generated and stored in the control block for each new connection. Any necessary retransmissions of the SYN or SYN/ACK must include identical values for the Migrate-Permitted and Timestamp option.

The remote end point, j, similarly selects a random $X_j \in [1, n-1]$ which it uses to construct k_j, its public key, which it sends in the same fashion. After the initiating end point's reception of the SYN/ACK with the Migrate-Permitted and Timestamp options, both end points can then compute a shared secret key, K:

$$K = k_i * X_j = k_j * X_i.$$

6.6.2 Using Diffie-Hellman for validation

As discussed previously, session resumptions are validated through the use of a challenge/response protocol, while connection migrations are signed. This disparity is due to a fundamental trade-off between the techniques. Challenge/response protocols are attractive because they allow the computational burden of validation to be scheduled—the operations need not be carried out at resumption time. Unfortunately, such techniques introduce an additional round-trip delay. Signed connection requests avoid the additional latency inherent this additional round trip but require additional protection against DoS attacks.

When a *Migrate* end point receives a session resumption request, it uses the secret key to encrypt a random challenge, which is returned to the requester. Only after the requesting end point decrypts the challenge and echoes it to the challenging end point is the session allowed to resume and virtualized network connections reestablished. Figure 6-12 depicts the messages sent during *Migrate* session migration.

Because session resumption challenges do not depend on the resumption request, they can be precomputed. Hence, an end point may pre-compute a large number of challenges and support the validation of a large number of simultaneous resumption requests without excessively burdening the host. This property is important, as session resumptions are likely to be correlated—when connectivity is restored after a period of complete disconnectivity, all of the sessions on a mobile host are likely to resume simultaneously.

In TCP connection migration, the secret key is used to sign the Migrate SYN. Specifically, the key is used to sign two separate components in a Migrate SYN: a validation *token* and the migration

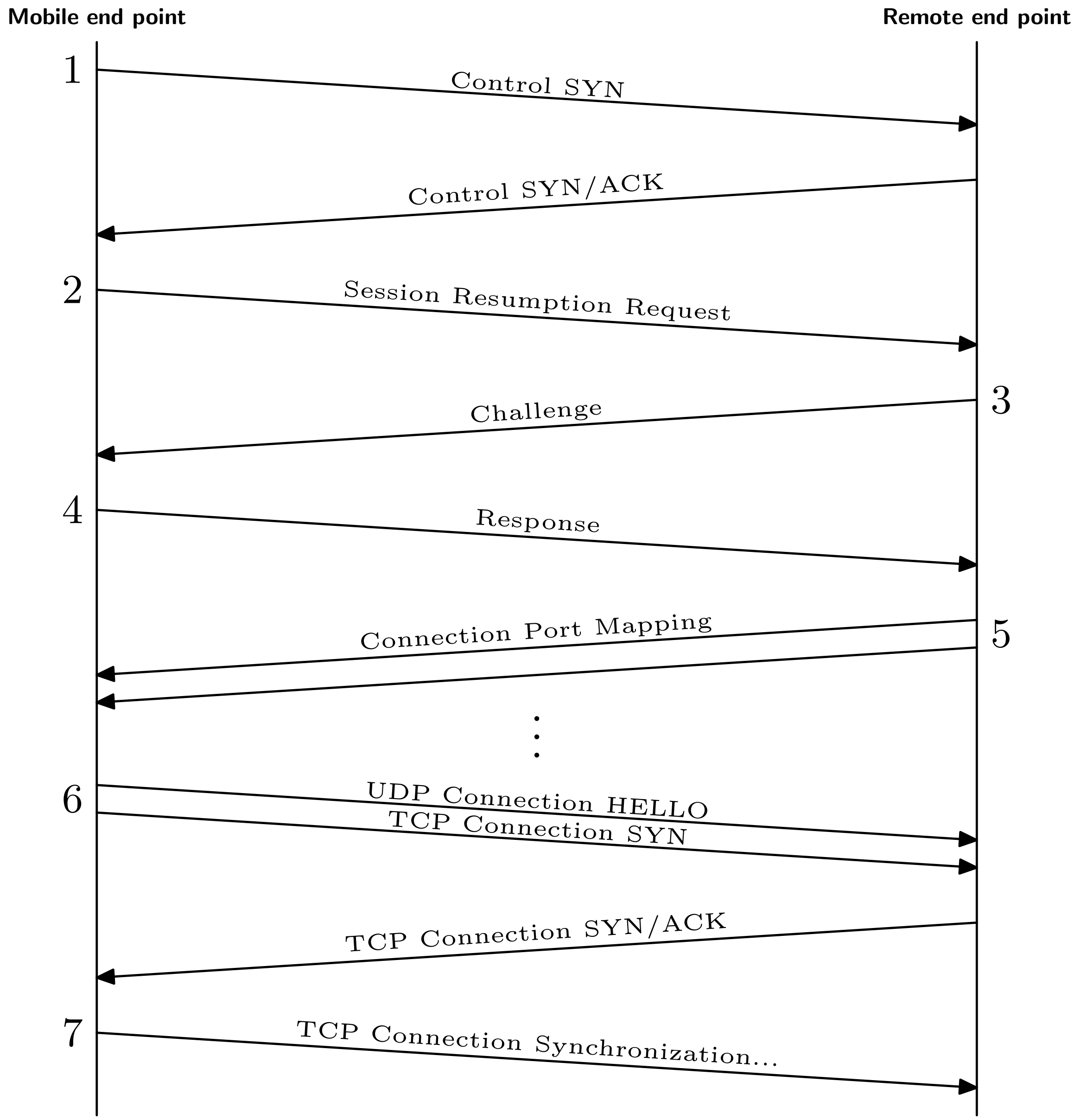

Figure 6-12: *Migrate* session migration with virtualized connections. Time flows downward. The migrating end point establishes a TCP session control channel (step 1) over which it sends a session resumption request (step 2). The remote end point responds with a cryptographic challenge (step 3). The migrating end point authenticates itself by decrypting the challenge (step 4). Upon validation of the response, the remote end point sends a port mapping message for each connection included in the session (step 5). The migrating end point then initiates new data connections as described in Chapter 4 (step 6); virtualized TCP connections require further synchronization (step 7).

<table>
<tr><td></td><td>Kind: 16</td><td>Length = 19</td><td>ReqNo</td></tr>
<tr><td colspan="4">Token</td></tr>
<tr><td colspan="4">Token (cont.)</td></tr>
<tr><td colspan="4">Request</td></tr>
<tr><td colspan="4">Request (cont.)</td></tr>
</table>

Figure 6-13: TCP Migrate option

request itself. The token, T, is computed by hashing together the key and the initial sequence numbers N_i and N_j using the Secure Hash Algorithm (SHA-1) [78] in the following fashion:

$$T = SHA1(N_i, N_j, K).$$

(Recall that end point i initiated the connection with an active open, and end point j is performing a passive open.) While SHA-1 produces a 160-bit hash, all but the 64 most-significant bits are discarded, resulting in a cryptographically-secure 64-bit token that is unique to the particular connection. Since SHA-1 is collision-resistant, the chance that another connection on the same $\langle$*source IP address*, *source port*$\rangle$ pair has an identical token is extremely unlikely. If a collision is detected, however, the connection must be aborted by sending a RST segment. (The end point performing a passive open can check for collisions before issuing a SYN/ACK and select a new random X_j until a unique token is obtained. Hence, the only chance of collision occurs on the end point performing the active open.)

The request, R, is the 64 most-significant bits of a SHA-1 hash calculated from the sequence number of the connection initial sequence numbers N, Migrate SYN segment, S, the connection key, K, and the request sequence number, I.

$$R = SHA1(N_i, N_j, K, S, I)$$

Upon receipt of a SYN packet with the Migrate option, a TCP stack that supports migration attempts to locate the connection on the receiving port with the corresponding token. The token values for each connection were pre-computed at connection establishment, reducing the search to a hash lookup.

If the token is valid, meaning an established connection on this $\langle$*source IP address*, *source port*$\rangle$ pair has the same token, and the request number (*ReqNo*) is greater than any previously received Migrate SYN, the receiving end point then computes $R = SHA1(N_i, N_j, K, S, I)$ as described above, and compares it with the value of the request in the Migrate SYN. If the values are equal, and the request number is larger than those in any previously received Migrate SYNs, the destination address and port[3] associated with the matching connection is updated to reflect the source of the Migrate SYN, and a SYN/ACK packet generated with the ACK field set to the last received contiguous byte of data, and the connection placed in the `SYN_RCVD` state. Upon receipt of an ACK, the connection continues as before.

[3]Migrated connections will generally originate from the same port as before; however, if the mobile end point is behind a NAT, it is possible the connection has been mapped to a different port.

6.6.3 ECDH key security

A connection key negotiated via ECDH is extremely difficult to guess even for attackers that can eavesdrop on the connection in both directions. While hosts that lie on the path between connection end points have sufficient information (namely the two ECDH components) to launch an attack against the elliptic curve system itself, such an attack requires considerable computational power. The best known attack is a distributed version of Pollard's rho-algorithm [92], which Lenstra uses to show that a 193-bit Elliptic Curve system would require $8.52 \cdot 10^{14}$ MIPS years, or about $1.89 \cdot 10^{12}$ years on a 450-Mhz Pentium II, to defeat [62].

While this key strength seems more than secure against ordinary attackers, an extremely well-financed attacker might be able to launch such an attack on a long-running connection in the not-too-distant future. The obvious response is to increase the key space. Unfortunately, we are restricted by the 40-byte limitation on TCP options. Given the prevalence of the MSS (four bytes), Window Scale (three bytes), SACK Permitted (two bytes), and Timestamp (ten bytes) options (of which we are already using eight bytes) in today's SYN segments, the 20-byte Migrate-Permitted option is already as large as is feasible. We argue that further securing the connection key against brute-force attacks from hosts on the path between the two end hosts is largely irrelevant, given the ability of such hosts to launch man-in-the-middle attacks against traditional TCP with much less difficulty.

6.6.4 One-way functions

The Diffie-Hellman authentication approach has two main drawbacks. First, the initial key exchange requires either modular exponentiation or elliptic-curve arithmetic, both of which are computationally expensive. Second, the challenge process is complicated. In the implementation described above, verification requires multiple messages and both an encryption and decryption operation. Much of the computation overhead can be avoided through the use of one-way functions.

A one-way function f is a function closed on a domain X satisfying the following two conditions:

1. Given a value $x \in X$, it is easy to compute $f(x)$.
2. Given some y, it is not feasible to find a value $x \in X$ such that $f(x) = y$.

There are many examples in the literature of one-way functions. Common examples include message digesting functions like SHA-1 or MD5 [106].

It is straightforward to design an authentication mechanism based upon one-way functions. A password authentication scheme based on one-way functions was first proposed by Leslie Lamport [59], and popularized by the S/Key password system [42]. *Migrate* does not authenticate the end hosts (the applications should), however, so a password is unnecessary. Instead, *Migrate* uses an anonymous form of Lamport's scheme. As with Diffie-Hellman, our one-way-function-based approach remains subject to man-in-the-middle attacks; this vulnerability is a property of all protocols that do not authenticate the end points at the outset.

In the anonymous scheme, end point A selects some random value, X, and a number, n, which represents the maximum number of migrations to secure for a particular session. Let $f^2(x) = f(f(x))$, $f^3 = f(f(f(x)))$, and so on. A then computes $f(x), f^2(x), \ldots, f^n(x)$ and sends $f^n(x)$ to B. Henceforth, A can authenticate itself to B by sending $f^k(x)$ for some $k < n$. B need only verify that $f^{(n-k)}(f^k(x)) = f^n(x)$. Clearly, for this scheme to be secure, A can never choose a k greater than a value it already transmitted. B must ensure this invariant holds by disallowing any authentication requests for which k is greater than the smallest k is has already seen. A straightforward enforcement scheme simply has A send $f^{(n-1)}(x)$ on its first migration, $f^{(n-2)}(x)$ on the next, and so forth. B can reduce its required computation by storing only $f^k(x)$ and k for the smallest k it has received. In the common case, B will then need to invoke f only once.

As described, the scheme only allows A to authenticate itself to B. In practice, B similarly selects its own random value X_B and n_B. The one-way function, f, can be the same for both and even well known, since it is infeasible by definition to invert f. Both A and B can independently trade off the amount of computation required to request a migration with the amount of memory they are willing to dedicate to storing pre-computed values of $f^k(X)$ for arbitrary k. Since an end point must initially compute $f^k(X)$ for all $k < n$ in order to compute $f^n(X)$, the end point could simply store all intermediate values, and then need perform no computation at all to request a migration. Hybrid schemes allow for arbitrary tradeoffs; for example, an end point could store every other value in half the space and require one application of f only every two migrations.

One attractive feature of securing the TCP Migrate options with one-way functions is the ability to integrate the Migrate options with SYN cookies [12]. SYN cookies are a popular way to avoid delegating resources at a passive end point during TCP connection establishment until the entire three-way handshake complete, thereby ensuring the remote end point is establishing the connection in good faith. End points employing SYN cookies are thus far more resistant to SYN-flood attacks. Unfortunately, when securing the Migrate options using ECDH, the passive end point must generate and store its keying material upon receipt of a SYN. This is incompatible with the SYN cookie approach, which encodes all information that would be stored by the passive end point into the initial sequence number—which will be echoed back in the final ACK of the handshake. Encoding the passive end point's secret keying material in the SYN/ACK packet is untenable, as there is insufficient space to encode all the necessary state in a cryptographically secure fashion. Even if there were sufficient space to transmit all the necessary information, the computational effort required to encode it would likely offset the savings achieved by not storing the state locally.

Using a one-way function scheme, however, the end points need not select their keying material simultaneously. Because each end point selects its keying material independently, the passive end point could wait until after the completion of the three-way handshake to do so. If end points were allowed to send the Migrate permitted option on any TCP segment, not just those with the SYN flag set, then a passive end point might elect to include it on a later segment—the one immediately following the completion of the three-way handshake, say. We have not yet implemented this extension, however.

If a victory is told in detail, one can no longer distinguish it from a defeat.

- Jean-Paul Sartre

Chapter 7

Evaluating *Migrate*

This chapter evaluates *Migrate* in three different ways. First, we consider the impact of *Migrate* on communication performance in Section 7.1. Our results show that while *Migrate* can impose considerable overhead when virtualizing extremely-high bandwidth ($>$ 350-Mbps) connections, the throughput impact is minor (2% or less for moderate block sizes) for sessions operating over common access link technologies. When used in conjunction with the TCP Migrate options, *Migrate*'s overhead becomes almost negligible and is restricted to session establishment and migration events. Second, Section 7.2 quantifies the cost of migration in terms of the delay associated with both session migration and connection synchronization. We find that the latency of session migration is tied directly to the round-trip time (RTT) between the communicating peers—the total delay is almost exactly four RTTs. Finally, we examine the both the ease of deployment and performance of session continuations in Section 7.3. We show that servers for two popular Internet applications, SSH and FTP, require only small modifications to support session continuations, and that suspended sessions for both applications consume only a few tens of bytes of secondary storage and between one and three file descriptors. Experimental measurements show our sample complete continuations introduce an additional session-continuation delay of less than 100 ms in our configuration, which corresponds to the time required to retrieve the continuation from secondary storage, unmarshall the stored state, and restart the process.

7.1 Overhead

In this section, we characterize the overhead of *Migrate* sessions in terms of their impact on connection performance and end-host resources. Initially, we focus on the overhead imposed upon all sessions, regardless of which end point changes attachment point. This overhead is quantified in terms of throughput degradation, increased connection latency, and additional resource consumption at end hosts. We examine the costs associated with session migration in the subsequent section.

Our experiments measure the C/C++-based *Migrate* implementation presented in Chapter 6 running on versions of two popular UNIX-based operating systems, Linux and FreeBSD. The results show that *Migrate* adds an observable overhead, with the most pronounced impact visible in initial connection latencies and the throughput of small write operations on virtualized connections. Throughput degradation becomes imperceptible as block size increases or available network bandwidth decreases, however, and connection latency overhead is dominated by the expense of the cryptographic session-establishment operations, which need not be performed on subsequent connections.

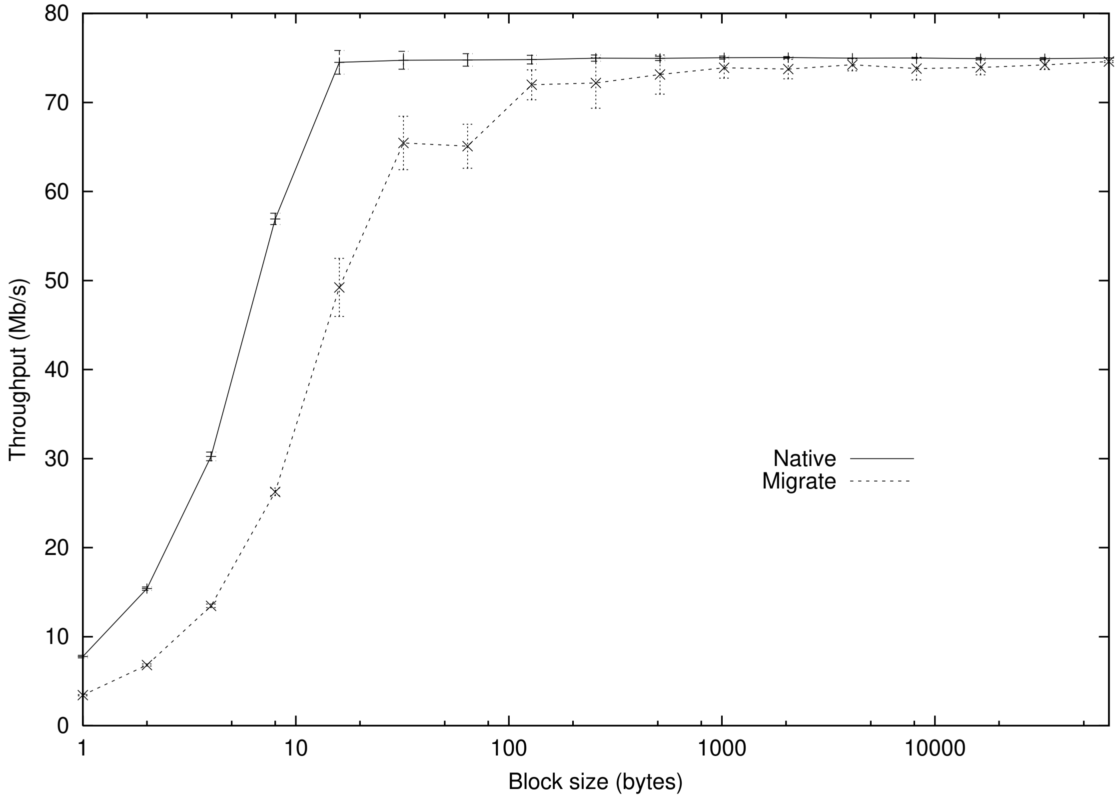

Figure 7-1: Mean TCP throughput with and without *Migrate* on a shared 100-Mbps Ethernet segment, as measured with `ttcp`. The receiver is an Intel 1.5-Ghz P4 running FreeBSD 4.6-STABLE while the sender is an Intel 2.26-Ghz P4 running Linux 2.4.18. Each point represents the average of at least sixteen runs; error bars represent one standard deviation.

7.1.1 Throughput

To evaluate the impact of *Migrate*'s connection virtualization on network communication, we measure the throughput of a virtualized TCP connection that is part of a *Migrate* session. Because the costs of connection establishment can be amortized over the length of the connection, we measure throughput across established connections and consider connection latency in the next section. We measure TCP throughput using the widely-available `ttcp` program [75], which uses the `write()` system call to write blocks of a specified size to the network as fast as possible. On the receiving end, another copy of `ttcp` reads data from the network in 64-KB blocks as fast as possible. Throughput is calculated at the receiver by dividing the number of bytes received by the time elapsed between the completion of connection establishment (when the `accept()` call returns) and the closing of the connection. All experiments were run with 64-KB socket buffers unless otherwise noted and without any socket options (*i.e.*, `TCP_NODELAY` and `TCP_NOPUSH`/`TCP_CORK` were *not* set).

In practice, TCP throughput is governed largely by the available network capacity and the loss rate experienced on the path between sender and receiver. Hence, we measured the throughput across two separate, uncongested local-area networks (LANs): a shared 100-Mbps Ethernet segment and a shared 11-Mbps 802.11b wireless LAN. The mean throughput of a virtualized TCP connection is shown as a function of the sender's block size in Figures 7-1 (100-Mbps Ethernet) and 7-2 (11-Mbps 802.11b). For comparison, we also plot the throughput of the same experiments without *Migrate*.

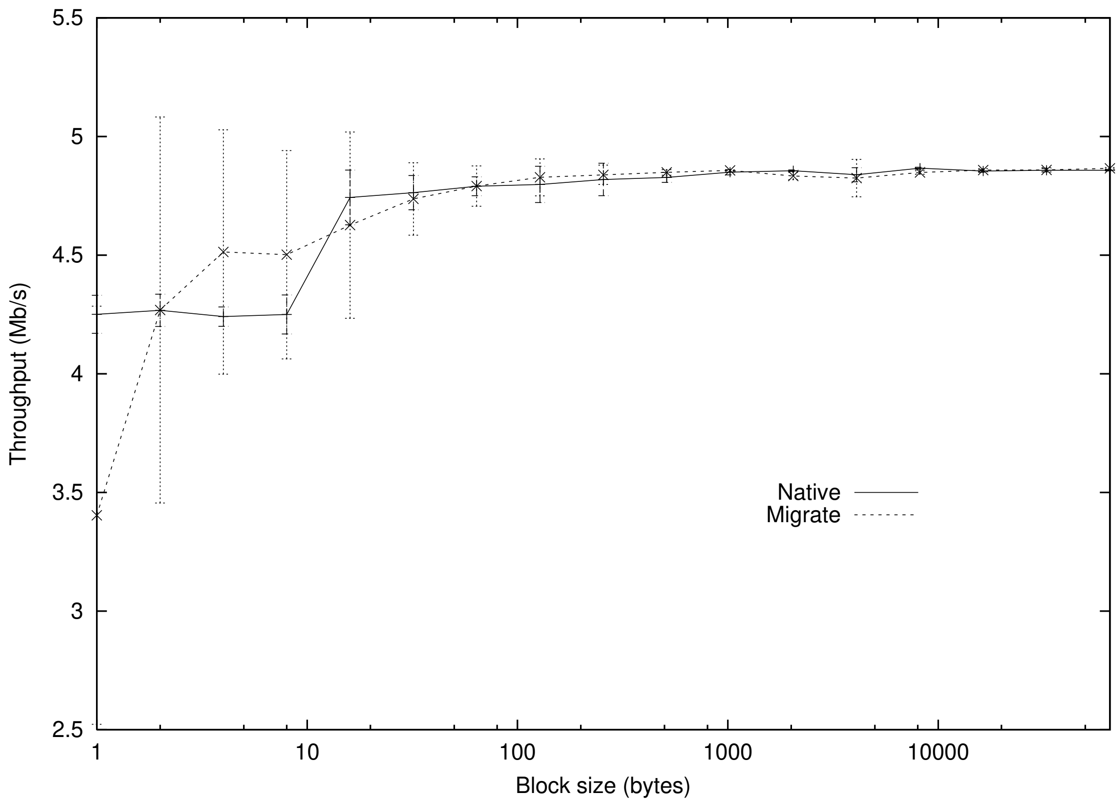

Figure 7-2: Mean TCP throughput with and without *Migrate* on a shared 802.11b wireless LAN, as measured with `ttcp`. The receiver is an IBM ThinkPad T21 (600-Mhz P3) running Linux 2.4.16 while the sender is an Intel 2.26-Ghz P4 running Linux 2.4.18. Each point represents the average of at least sixteen runs; error bars represent one standard deviation.

Not surprisingly, *Migrate* introduces a significant overhead at small block sizes. The overhead is due to several factors including the additional context switch and system call overhead imposed by TESLA and the memory copy operations required by *Migrate* to virtualize TCP connections. Recall that TESLA requires that data first passes to a `teslamaster` process before being written to the network. Similarly, a *Migrate*-enabled receiver first reads the data into its own `teslamaster` process and then passes it along to the `ttcp` application. Once the bottleneck becomes the available network bandwidth (approximately 74 Mbps on the Ethernet segment in this experiment, 4.5 Mbps on 802.11b)—and not the system call overhead—the throughput reduction is less than 2%. In WAN environments, the bottleneck bandwidth is frequently substantially lower, so *Migrate* is even less likely to be the limiting factor.

Far more surprisingly, however, *Migrate* out-performs standard TCP connections for small block sizes on 802.11b networks. This performance improvement is also due to interactions with `teslamaster`. The `teslamaster` process runs in a loop, reading up to 8 KB at a time from the application and writing it to the network as a block. Hence, while an application may be writing data in small blocks, it is likely to be written to the network in much larger blocks, because several blocks will have queued up in the pipe before the `teslamaster` process is swapped in and writes the data to the network. When the network itself is the bottleneck—as is the case regardless of block size in the 802.11b scenario—`teslamaster`'s implicit write batching ensures the kernel

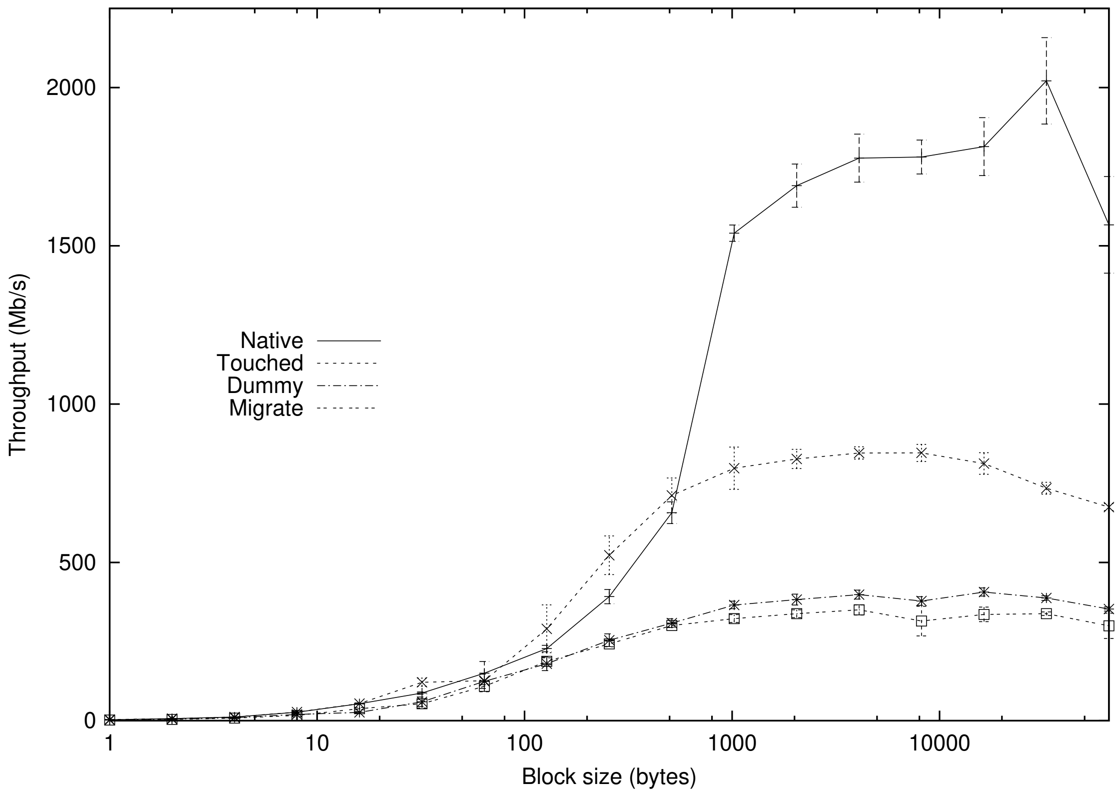

Figure 7-3: Mean TCP throughput across the loopback interface of an IBM ThinkPad T21 (600-Mhz P3) running Linux 2.4.16, as measured with `ttcp`. Results are presented with and without *Migrate*, as well as for a dummy TESLA handler and a receiver that touches every byte received. Each point represents the average of at least sixteen runs; error bars represent one standard deviation.

send buffers have data ready to send even if the `ttcp` application (or the `teslamaster` process, when TESLA is in use) is not currently active. (All experiments were run on uniprocessor machines.)

The relatively large throughput variability of *Migrate*-enabled transfers is due to the increased impact of context switching, cache replacement policies, and other scheduling vagaries on both sender and receiver. This scheduling effect impacts all *Migrate* operations, including data transfer, connection establishment, and session migration, and is visible in all of the graphs presented in this chapter, especially Figures 7-2 and 7-5. Since all network I/O handled by TESLA must pass through a separate `teslamaster` process, performance is dependent on how the application and corresponding `teslamaster` process are interleaved. In an ideal implementation, this variance could be removed by invoking the *Migrate* handler as a co-routine or part of the system call itself.

In addition to affecting performance, scheduling behavior may also cause applications that have set the `TCP_NODELAY` option to emit segments of unexpected size. When the `TCP_NODELAY` option is set, a conformant TCP stack writes data blocks to the network immediately (as opposed to delaying for some period of time in accordance with the Nagle algorithm [77]). As described previously, TESLA queues data in one contiguous buffer and does not respect application block size. `teslamaster`'s buffering policy effectively enforces its own "Nagle algorithm," possibly coalescing consecutive blocks of data into one larger one. Hence, consecutive blocks of data that would normally be sent in separate packets may be sent together.

Because access link bandwidths are likely to increase, we also consider the unrestricted case. Figure 7-3 presents the mean TCP throughput as measured across the loopback interface of an IBM ThinkPad T21 (600-Mhz P3). For comparison, we also present the results of a similar experiment without *Migrate*. *Migrate* imposes a significant overhead, effectively limiting the throughput to 350 Mbps, as opposed to a maximum achieved throughput of 2,020 Mbps without *Migrate*. If the receiver is forced to actually touch every byte received, however—as would be the case in any practical application—native throughput is limited to 845 Mpbs. (By default, a `ttcp` receiver simply discards the data without examining it, reaping considerable savings at large block sizes.) We consider the second, more realistic configuration when calculating the impact of *Migrate*.

In order to separate the overhead of *Migrate* itself from the overhead imposed by our TESLA implementation, we also ran the experiment using a *dummy* TESLA handler [110]. This dummy handler performs no operations at all on the data, but forces the data stream through `teslamaster` in the same way the *Migrate* handlers do. The roughly 50% difference in throughput between the TESLA dummy handler and the native `ttcp` application represents the architectural overhead of TESLA; the difference between the dummy handler and *Migrate* corresponds to double-buffering expense. In the worst case, *Migrate* decreases throughput by an additional 7.5% when compared to the dummy handler. We conclude that the inter-process communication and context switching required to move between the four processes is the most significant factor preventing *Migrate*-enabled `ttcp` processes from achieving a higher throughput. The expense of double-buffering the connection data is non-negligible, but can be avoided through the use of the Migrate TCP options.

As an additional complication, `teslamaster` is configured to read from applications and the network in blocks no larger than 8 KB regardless of the size of blocks being read or or written by the application process. Hence, while the throughput of direct reads and writes (as performed by `ttcp`) may continue to improve as the block size increases, applications using TESLA handlers may be unable to realize these gains. This parameter can be adjusted within TESLA itself if desired, possibly improving performance. Limited experimentation with larger `teslamaster` block sizes in this scenario revealed performance increased slightly (roughly 10%) with the dummy handler, but not with the *Migrate* handlers, leading us to conclude that the increased expense of larger memory copy operations counter-balances the increased efficiency of larger block reads.

7.1.2 Connection latency

In addition to the small overhead on each network operation, *Migrate* may introduce a significant delay in connection establishment. This delay is caused by two factors. Any connection managed by *Migrate* must be brokered by the *Migrate* daemons at each end point, introducing an additional packet exchange. Because the exchange itself proceeds asynchronously it does not seriously impact connection establishment. More significantly, the *first* connection of a session must endure the delay associated with the cryptographic operations necessary to secure the session. To quantify these overheads, we measured the time required to establish a virtualized TCP connection on a single host. By establishing the connection on the loopback interface, we avoid any dependence on the RTT between end points. Increased RTTs will affect native and subsequent virtual TCP connections identically; the initial connection establishment incurs a delay equal to two additional RTTs (or perhaps more, if packet loss occurs) due to session control channel establishment and cryptographic message exchange.

We measured connection latency as the elapsed time between the issuance of a blocking `connect()` system call and its return for each of three different types of connections: a native TCP

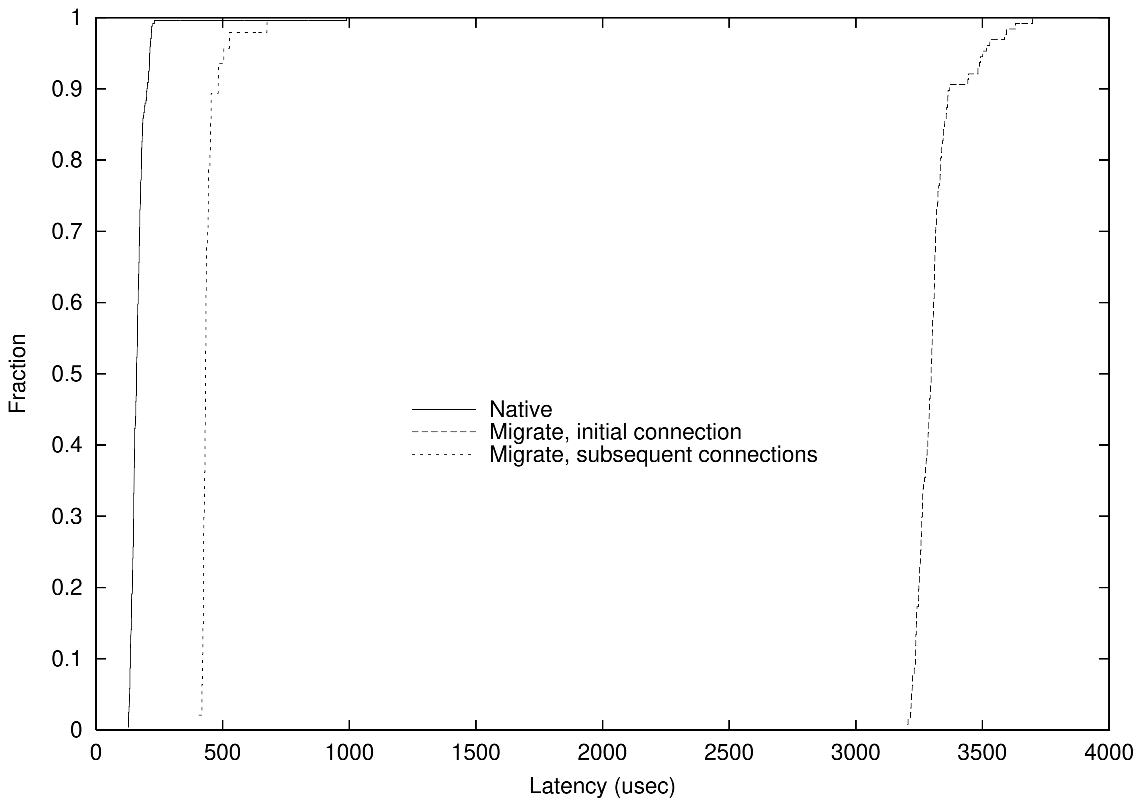

Figure 7-4: Cumulative distribution of the connection establishment latency of a TCP connection on the loopback interface of a 600-Mhz Intel P3 running Linux 2.4.1. Each distribution results from 100 independent trials.

connection, a virtualized TCP connection, and the initial virtualized TCP connection of a *Migrate* session. The median latencies observed over 100 independent runs were 160 μs, 433 μs, and 3297 μs, respectively. (Latencies were measured through repeated calls to `gettimeofday()` which is accurate to within 1–2 μs on Linux.) Figure 7-4 shows the cumulative distribution of latency measurements for the three different connection establishment scenarios. All three curves have similar shape, since the negligible network latency and packet loss means that connection establishment takes essentially constant time. The highest variability is seen in the initial *Migrate* connection establishment due to scheduling variabilities, since both the sending and receiving Migrate daemons may experience context switches during the cryptographic operations.

As expected, the initial connection establishment on a *Migrate* session is significantly slower than subsequent connections, which are marginally slower than native TCP connections due to the TESLA IPC overhead. Cryptographic operations account for the vast majority of the session establishment overhead, requiring just over two ms in this configuration. The sessions in this experiment were secured using a 128-bit key negotiated using Diffie-Hellman. Obviously, longer key lengths will lead to increased session establishment latency. In practice, however, this delay is normally dominated by the connection RTT, which is generally much larger in the wide area. (A similar effect is seen with session migration latencies in Figure 7-6.)

7.1.3 End-host resources

In addition to the per-connection overhead, *Migrate* also consumes a small amount of system resources on the end host. The resident portion of the Migrate daemon is about 1.5 MB in size (1,560 KB on FreeBSD and 1,480 KB on Linux), and each *Migrate*-aware application requires its own `teslamaster` process, which is also about 1.5 MB (1,372 KB on FreeBSD and 1,512 KB on Linux). The memory requirements of both the Migrate daemon and application-specific `teslamaster` processes grow as the number of sessions being managed increases. Each new `teslamaster` also requires two file descriptors: one each to communicate with the daemon and the associated application. Chapter 6 details the data structures used to manage each session.

7.2 Migration

The costs of migration vary greatly, depending on the transport connections in use. We first examine the cost of session migration independent of the constituent network connections, both in terms of migration latency and its impact on application naming systems. We then focus on the additional synchronization latencies associated with virtualized TCP connections. The third experiment combines these two steps and quantifies impact of migration on connection throughput, considering both soft and hard handoffs. We conclude this section by presenting sample traces of TCP connection migration using the Migrate options and measurements of handoff performance.

7.2.1 Session migration latency

Recall from Chapter 3 that session migration consists of four parts: end-point location, authentication, rebinding (including connection port mapping), and connection synchronization. Because the delay associated with end-point location depends on the naming system selected by the application, we assume here that the location of the remote end point is known—which is the case unless both end points move simultaneously. The cost of connection synchronization depends on the transport protocol in use and the loss rate experienced at the previous attachment point. Hence, we consider connection synchronization separately in the next section. Here we quantify the expense of the other two operations—authentication and rebinding—by measuring the migration latencies of sessions containing a variable number of connections. We define *session migration latency*, measured at the migrating end point, as the time between attempting to reestablish the session control connection until the reestablishment of all of the associated network connections.

We first examine session migration latencies over a loopback interface to ignore the effects of network latency. (Migration back to the same interface reduces to a change in ports.) Figure 7-5 presents the cumulative distribution of session migration latencies for sessions containing one, two, and three virtualized TCP connections. The median latencies are 2,135 μs, 2,681 μs, and 3,251 μs, respectively. Assuming the cost for each additional connection is the same, this experiment indicates that end-point authentication accounts for about 1,600 μs and each connection adds roughly 550 μs of delay. The increased delay variability in the two- and three-connection case is due to the increased likelihood of context switching at one or both of the end points between connection mapping messages. Each mapping message is sent separately so they may be processed independently.

Figure 7-6 shows the median time required to migrate a session containing one virtualized TCP connection to an attachment point with a varying RTT between itself and the remote end point. Because session migration is an end-to-end operation, with no dependencies on home agents or similar third parties, the delay depends only on the RTT between the attachment points used by the two end points. As expected, the delay is linear in the RTT and corresponds almost exactly to four

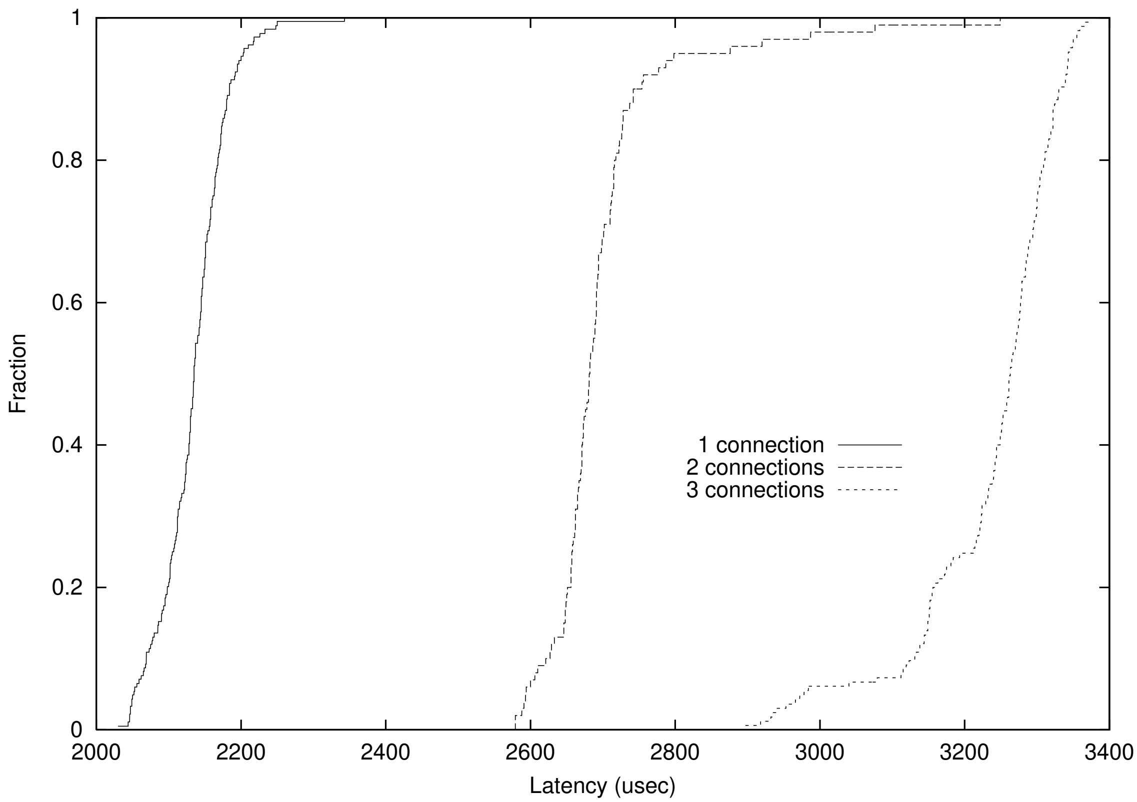

Figure 7-5: Cumulative distribution of the session migration latency of a session with a varying number of TCP connections on the loopback interface of a 850-Mhz Intel P3 running Linux 2.4.2. Each distribution results from 100 independent trials.

round trip times (the 2.7 ms cost shown in Figure 7-5 is negligible compared to the network latency). As shown in Figure 6-12, the reestablishment of contained network connections occur roughly in parallel, so session migration delay is largely independent of the number of network connections contained by the session when the RTT is large.

7.2.2 Connection synchronization

The time required to synchronize a virtualized connection after migration depends on the transport protocol in use and the loss rate experienced by the connection before migration. From the point of view of *Migrate*, which is virtualizing the transport-layer connection protocol, there is no need to replay lost data on UDP sockets; they are instantly synchronized. Connection-oriented, application-layer protocols running on top of UDP that provide additional delivery semantics (*e.g.*, RTP [114]) may need to resynchronize, but they operate at a higher layer than *Migrate* and we do not consider them here. Virtualizing TCP connections, on the other hand, requires all data previously sent by one end point but not received by the remote end point be retransmitted. The greater the congestion and receive windows prior to migration, the more data that may need to be retransmitted. Furthermore, because TCP does not deliver data out of order, data received after a loss must be retransmitted.

To quantify the effect of varying loss rates, we measured the time required to synchronize a virtualized TCP connection over a simple topology. We used Emulab [142] to emulate the topology presented in Figure 7-7; each node is a distinct machine while the links are emulated using Dum-

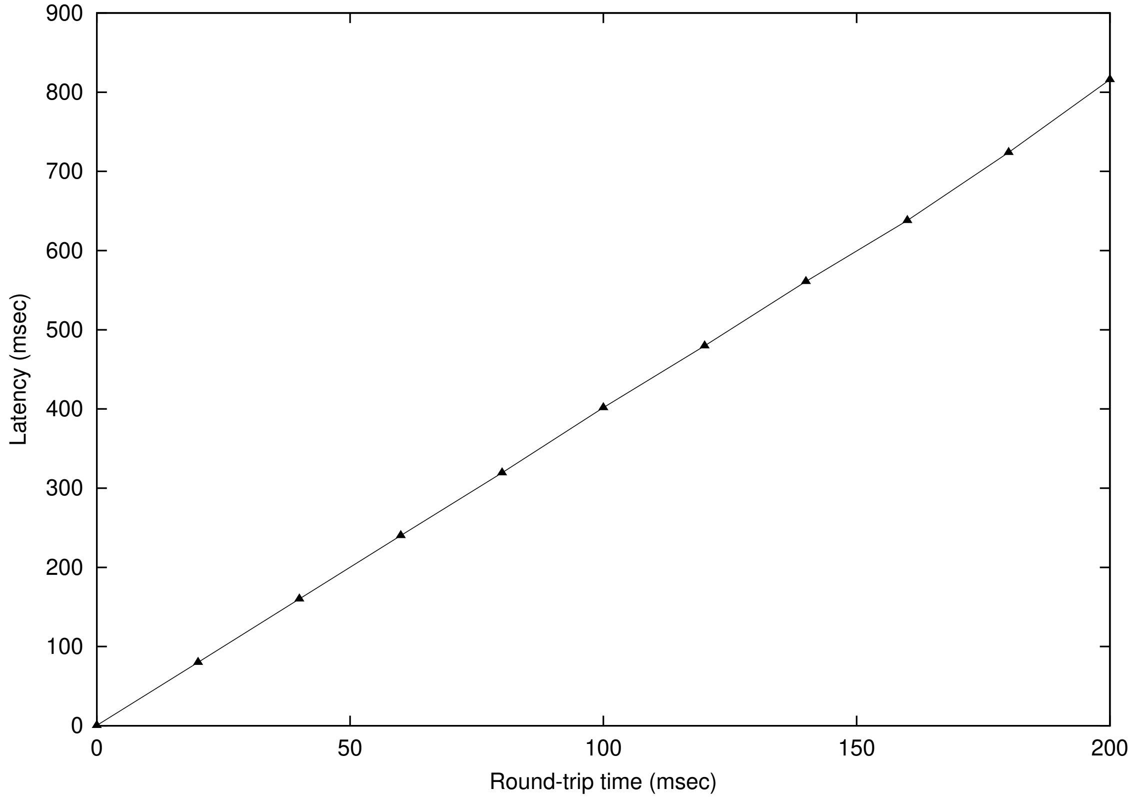

Figure 7-6: Median session migration latency of a session with one TCP connection between two 850-Mhz Intel P3s running Linux 2.4.2 with varying RTTs.

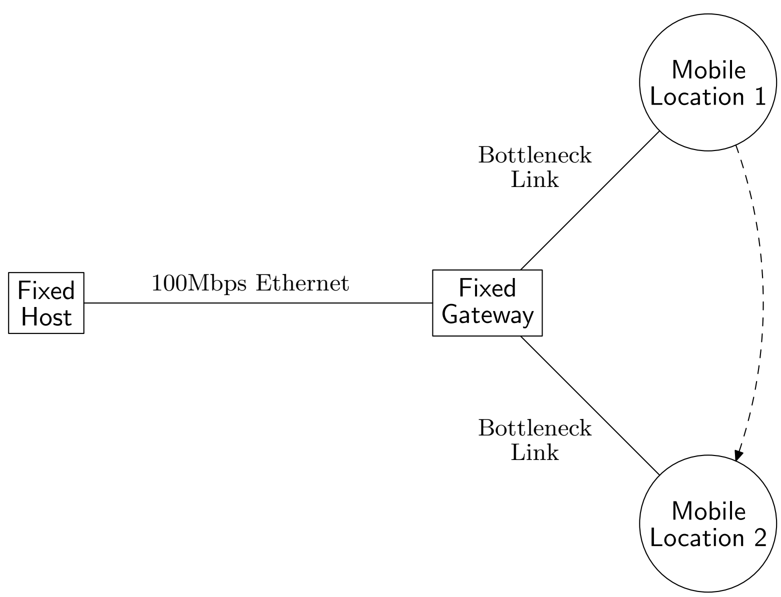

Figure 7-7: Network topology used for virtualized connection synchronization and TCP connection migration experiments. DummyNet [107] is used to emulate 1-Mbps access links with 20ms of delay for the experiments in Section 7.2.2; actual 19.2-Kbps serial lines were used in Section 7.2.5.

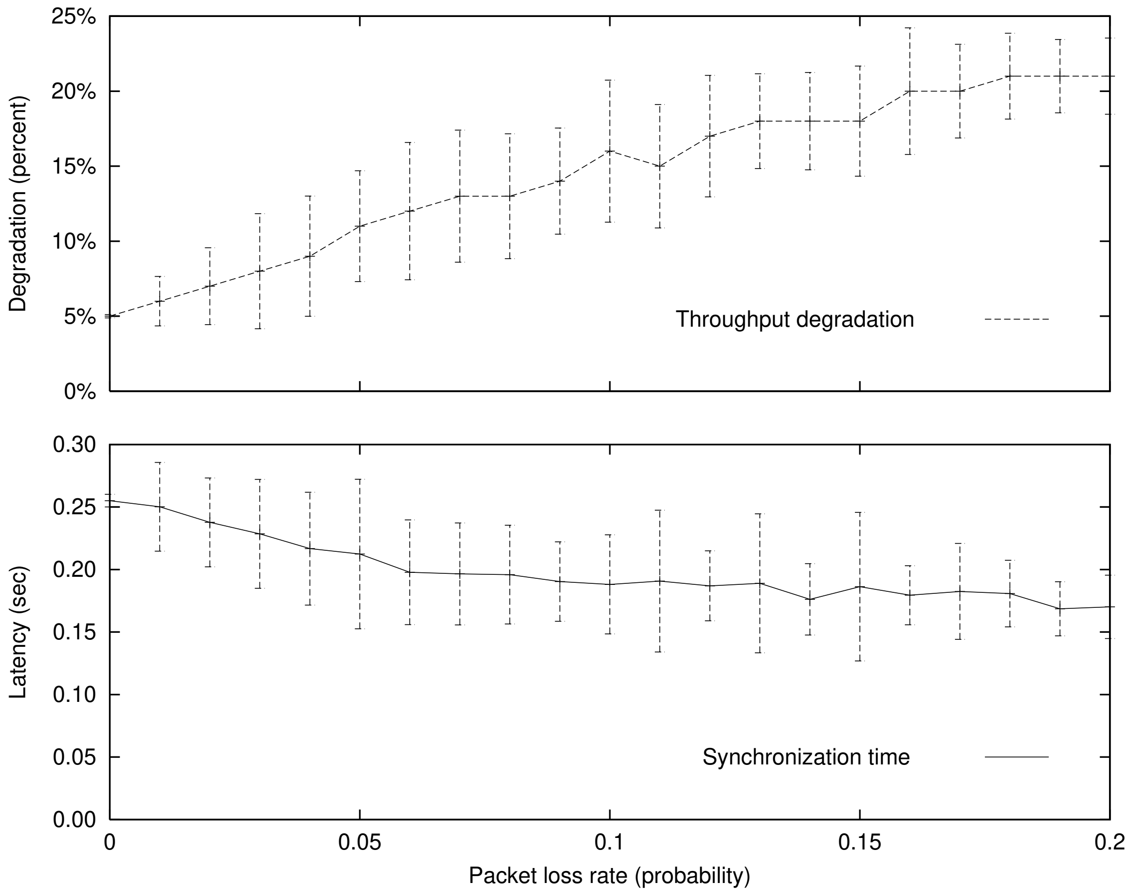

Figure 7-8: Mean connection synchronization latency and throughput degradation of a 524,288-byte virtualized TCP transfer between two 850-Mhz Intel P3s running Linux 2.4.2 over a 1-Mbps link with a RTT of 40 ms. The connection iss migrated to a loss-free link after one second. The initial link loss rate varies from zero to 20 percent. Each point is the average of fifty trials; error bars represent one standard deviation.

myNet [107]. This topology, with different link speeds, is also used to conduct the experiments presented in Section 7.2.5. Both mobile host locations use identical connections: a one-Mbps link with 40 ms RTT to the gateway. The gateway and fixed host are on a 100-Mbps Ethernet segment; the link to the mobile host is, therefore, the connection bottleneck. While our experimental results depend greatly on the bandwidth-delay product resulting from our selection of link bandwidth and latency values, we believe our parameters provide a realistic baseline for evaluation.

In this experiment, we began a 524,288-byte (64 8192-byte blocks) `ttcp` transfer from the fixed host to a mobile client (using a 32-KB TCP receive buffer) at Location 1. After one second, the mobile host moved to Location 2 and migrated the session. Figure 7-8 presents both the mean throughput degradation and the mean time to synchronize the TCP connection as a function of the loss rate on the link between the gateway and Location 1. The download time does *not* include session or connection establishment times; time is measured from the moment the receiver's `accept()` call returns until the download completes. To help isolate the effects of packet loss before migration, there is no packet loss on the links connecting the gateway to the fixed host and the mobile host's new attachment point, Location 2. As expected, the total download time increases with

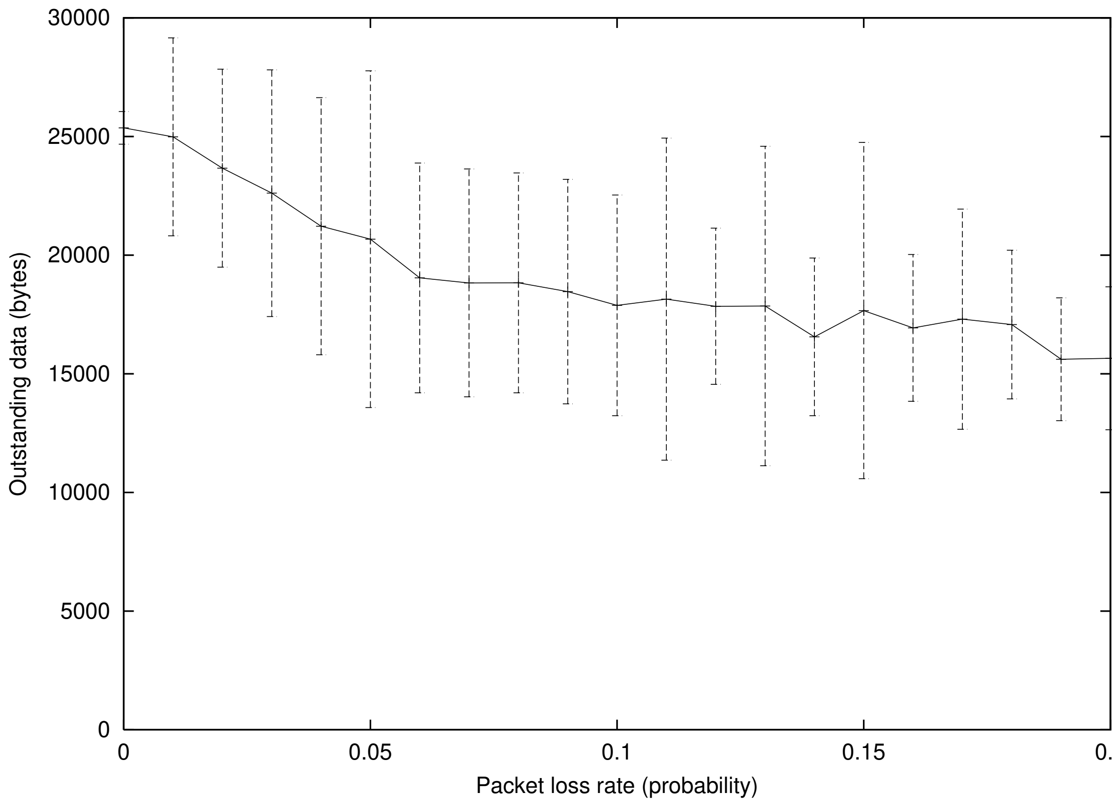

Figure 7-9: Mean number of bytes required to synchronize a virtualized TCP connection over a 1-Mbps link between two 850-Mhz Intel P3s running Linux 2.4.2 over the topology in Figure 7-7. The initial link loss rate varies from 0 to 10 percent. Each point is the average of fifty trials; error bars represent one standard deviation.

packet loss rate. Somewhat surprisingly, however, the connection synchronization time actually decreases as the packet loss rate goes up.

This counter-intuitive result is explained by Figure 7-9, which shows the number of outstanding bytes on the connection at migration time. As the packet loss rate increases, the number of bytes in flight decreases. This effect is due to the fact that TCP is unable to grow its congestion window in the face of packet losses. Hence, the number of bytes that need to be retransmitted to synchronize a connection actually decreases with packet loss rate. While this unfortunately means the impact of migration is greatest when the connection is performing best, the absolute impact of connection synchronization remains small: downloads migrated once between zero-loss links completed in an average of 4.69 seconds as opposed to 4.47 seconds for downloads that were not migrated at all.

The relatively large variance in the number of bytes outstanding and, thus, the synchronization latency, seen in Figures 7-8 and 7-9 is due to two separate effects. First, the TCP congestion window (which, combined with the receiver's receive buffer size, governs the number of bytes in flight) grows sporadically in the face of background loss. Burst losses may lead to time outs which cause the sender to halve the congestion window. The absence of this effect in the zero-loss case results in a significantly tighter distribution. In addition to TCP's congestion window behavior, the inherent variance of the packet loss distribution is amplified by the size of the TCP segments themselves. In this case, all packets are MTU size (1448 bytes) leading to significant quantization effects.

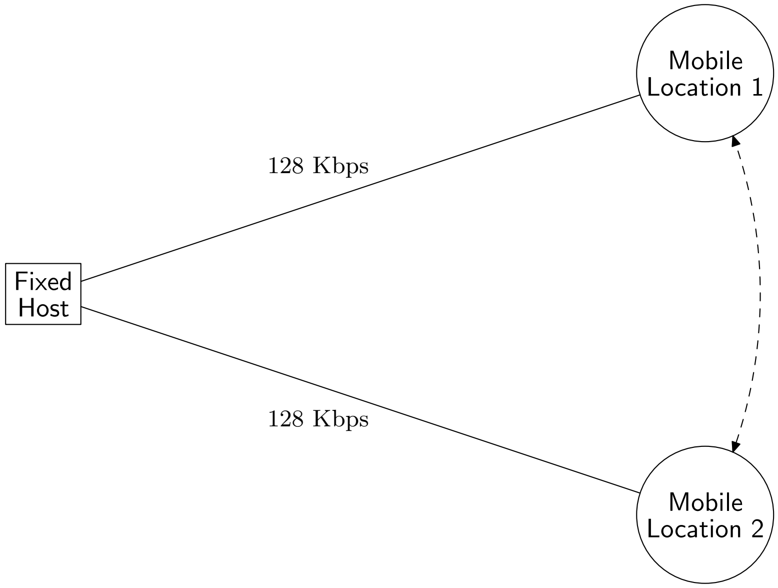

Figure 7-10: Network topology used to measure handoff performance for both virtualized TCP connections (Section 7.2.3) and the Migrate TCP options (Section 7.2.5). DummyNet is used to emulate 128-Kbps links with a one-way delay of 20 ms between the attachment points.

7.2.3 Handoff performance

When an end point changes attachment points, steps examined in the previous two sections, migration and connection synchronization, are both performed. We term their combined impact on a connection's throughput *handoff overhead*. In this section, we quantify handoff overhead by observing the progression of a TCP download as one of the end points continually changes attachment point. We conducted a series of simple experiments where a mobile host migrated between two attachment points using distinct links to a fixed, remote end point. The mobile end point oscillates between the two attachment points with varying frequency. Both attachment points are connected to the remote end point with separate 128-Kbps links with 40-ms RTTs. The experimental topology is shown in Figure 7-10.

We first considered so-called *hard* handoffs, where the end point changes attachment point and any queued communications sent from the previous attachment point are lost. *Migrate* end points experience a hard handoff when a mobile receiver changes attachment points. All in-flight data from a fixed sender continues to be delivered to the mobile end point's old attachment point where it is discarded. Figure 7-11 shows the throughput of a 947,570-byte download subjected to migration oscillations with periods up to 30 seconds. (A download from one attachment point takes just over 60 seconds.) As expected, throughput decreases monotonically as migration frequency increases.

The throughput degradation is clearly visible in Figure 7-12, which shows the progression of a virtualized TCP download subjected to varying rates of oscillation. The absolute impact of a particular migration frequency depends on its relationship to the bandwidth-delay product of the path. We will explain this relationship in more detail when we consider the TCP Migrate options in Section 7.2.5.

Figure 7-13, on the other hand, shows the same experiment, except the transfer proceeds in reverse. The receiver is fixed, and the sender changes attachment points. This movement pattern results in

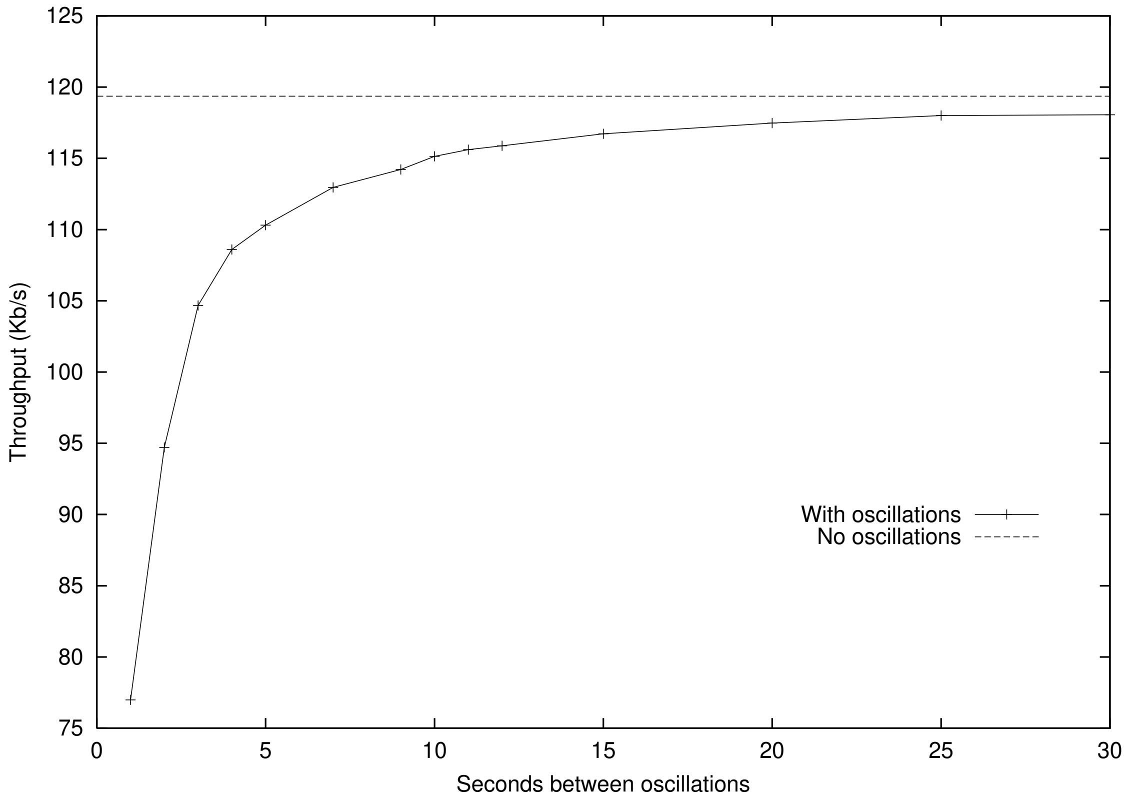

Figure 7-11: Throughput vs. hard-handoff oscillation rates of a virtualized TCP connection. Throughput measured at the receiver by timing the transfer of a 947,570-byte file. A transfer conducted entirely from one attachment point achieves a throughput of 119.4 Kbps.

a *soft* handoff—in-flight data from the mobile sender continues to be received by the fixed receiver, even though the fact the sender has moved. Because *Migrate* does not abort connections to the previous attachment point until a session migration request has been validated, this process continues in parallel with the reception of data from the previous attachment point. Hence, the amount of in-flight data lost due to a change in sender's attachment point is considerably less than when a receiver changes attachment point. The benefits of soft handoffs can be seen in the minimal impact of even high-frequency migrations.

7.2.4 Naming system impact

In addition to its impact on the end hosts themselves, *Migrate* places an additional burden on the naming systems used by application end points. In particular, all naming systems in use by an end point should be updated each time the end point changes attachment point. *Migrate* does not, however, necessarily increase the look-up burden on a naming system, since end points are only looked up during initial session establishment (when they would be any way) and in the event that both end points move simultaneously. Migration events where only one end point moves do not place any demands on the naming system.

Yet the increased update frequency may reduce the effectiveness of look-up caching. Name records with a high update frequency, such as those used to store bindings for mobile end points, do not lend themselves well to caching, as they must be invalidated when the end point moves. Naming

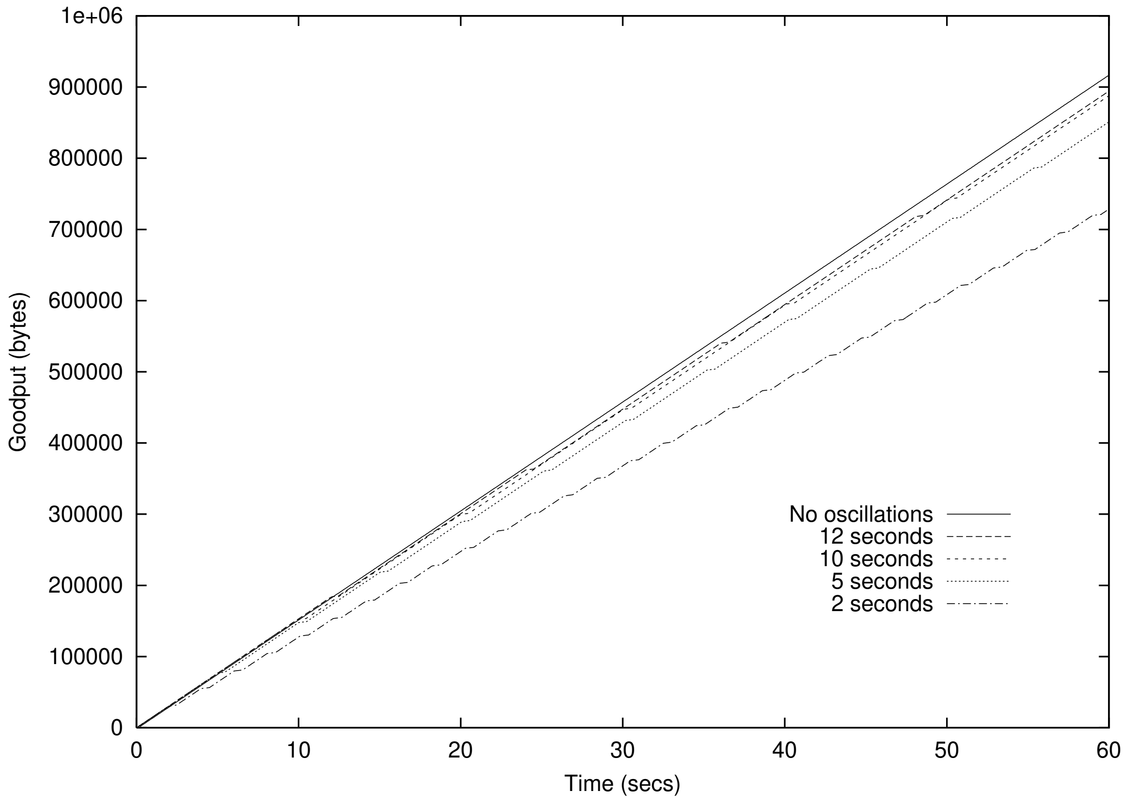

Figure 7-12: Hard session handoff performance. Progress of a virtualized TCP transfer of a 947,570-byte file subjected to varying rates of receiver attachment-point oscillation. The TCP receive buffer is 64 KB.

systems that lack an explicit invalidation mechanism are forced to use a lease-based scheme instead, in which highly dynamic records are expired after a short period of time. We examine the effect on one such popular naming system, the Domain Name System (DNS), and consider how to mitigate the costs of frequent updates.

Studies have shown that the increased look-up load of DNS records with near-zero time-to-live fields (TTLs)—as required by mobile hosts using *Migrate* as well as popular content distribution networks [2]—can be supported by the current DNS infrastructure. Records with small TTLs (on the order of a few minutes) have a relatively small impact on the DNS hierarchy because the location of the authoritative DNS server (the server that stores the end-point binding itself) can be cached [52]. Each additional look-up operation contacts the authoritative server directly, avoiding the complete walk through the domain name hierarchy starting at the root. Hence, the increased burden on DNS of records with low TTLs is limited to the clients that need to look up the record and the authoritative server. Since *Migrate* only looks up the record at session initiation in most cases, there is little additional overhead due to end point mobility. Popular mobile hosts, or those that move frequently and are more likely to move in concert with a remote end point, are easily identified by increased look-up load on their authoritative name servers. These records can either be off-loaded to a less heavily loaded server, or the server can be replicated. Mobile end points that are not servers themselves, and communicate only with remote end points that do not move (as is commonly the case in today's client/server environment) do not need DNS records, and, therefore, need not contribute any additional load to the naming system.

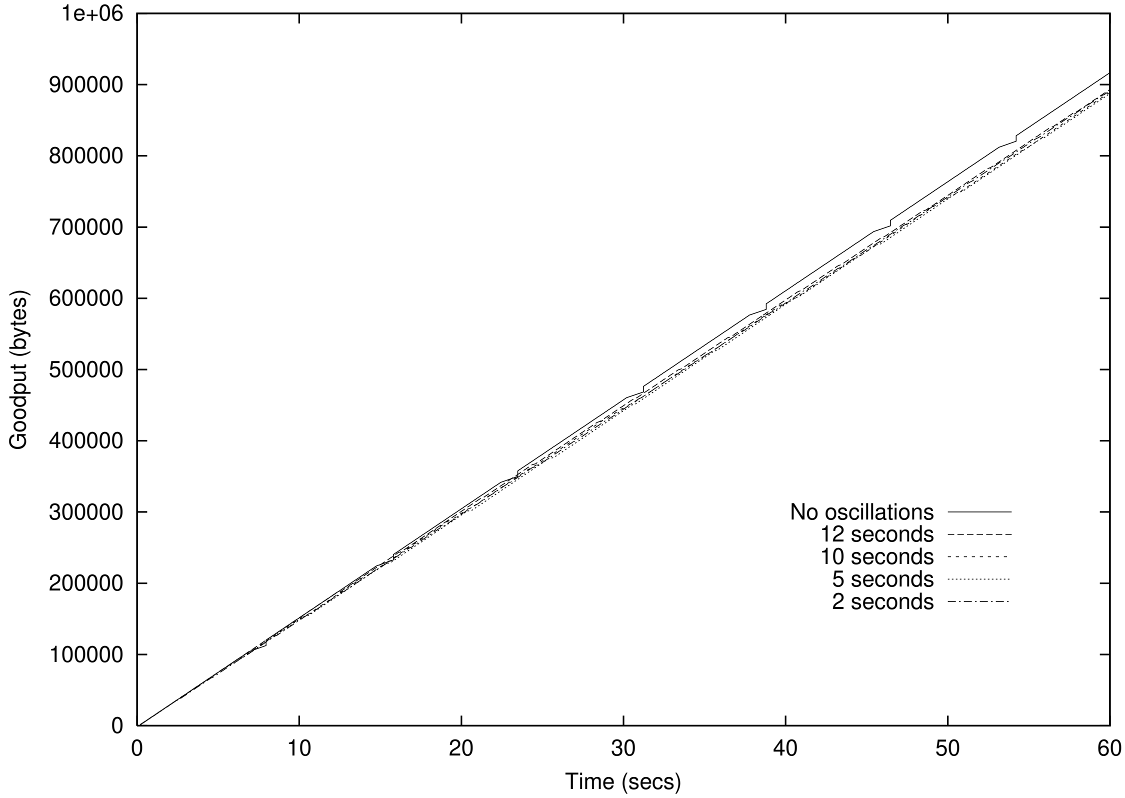

Figure 7-13: Soft session handoff performance. Progress of a virtualized TCP transfer of a 947,570-byte file subjected to varying rates of sender attachment-point oscillation. The TCP receive buffer is 64 KB.

In addition to the increased look-up burden of low-TTL name records, it may be that frequent DNS *updates* could lead to a significant burden on highly utilized name servers. In particular, secured DNS updates [141] require DNS servers to perform a cryptographic signature verification operation for each update. The rate of these operations at any particular server can be controlled by adjusting the number of DNS records it maintains; the more frequently updated the records, the fewer records maintained by a single server. Alternatively, updates can be authenticated by dedicated, trusted update servers and propagated to authoritative servers using traditional domain-transfer techniques [68]. When DNSSEC [32] is in use, updated records must also be signed by an entity holding the signing key for the relevant zone. This operation can also be off-loaded to a trusted agent of it becomes an excessive burden on the DNS server itself.

7.2.5 TCP connection migration

Unlike virtualized TCP connections, whose activities during migration are drastically different than during normal operation, TCP connections using the TCP Migration options perform quite similarly to standard TCP connections even during migration. This section presents traces of migrating TCP connections to demonstrate the effectiveness of the Migrate options and, then, considers the impact of rapid, repeat migration on the throughput of ongoing TCP connections using the Migrate options.

The TCP traces shown in Figures 7-14 and 7-15 were gathered at the gateway depicted in Figure 7-7, which is on the path between the fixed host and both mobile host locations. We conducted TCP

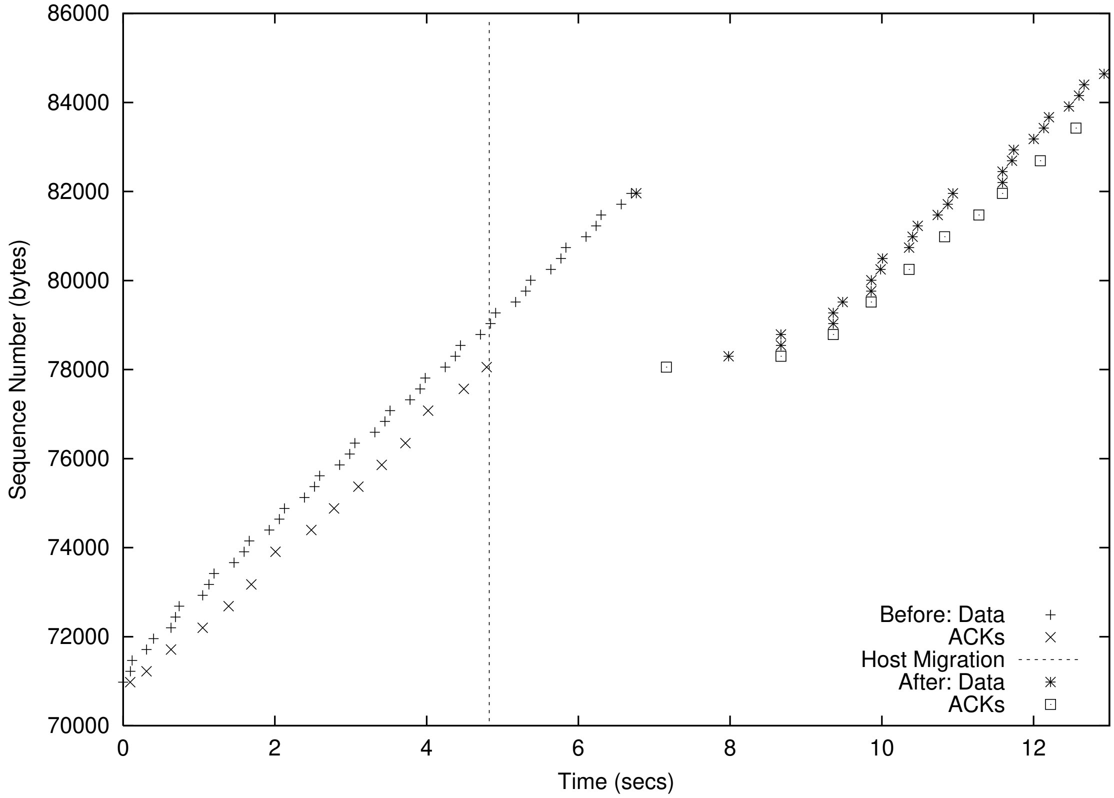

Figure 7-14: A TCP connection sequence trace showing the migration of an established connection transferring data from a fixed server to a mobile client. The Migrate SYN is generated by the migrating receiver; its value is unrelated to the sequence space shown in this graph and is depicted as a dashed vertical line. The Migrate SYN/ACK appears as the first data segment sent after migration.

bulk transfers from the fixed host to the mobile host. The mobile host initiates the connection from Location 1 and migrates to Location 2 at a later time. This topology is intentionally simple in order to isolate the various subtleties of migrating TCP connections, as discussed below.

Figure 7-14 shows the TCP sequence trace of a migrated TCP connection. At time $t \approx 4.2$ s the mobile host moved to a new address and issued a Migrate SYN. Because the trace shown in Figure 7-14 was conducted at the gateway and *not* the mobile end point, the SYN does not appear in the trace until time $t \approx 4.8$ s, as depicted by the dotted line. Since the host is no longer attached at its previous address, all of the enqueued segments at the bottleneck are lost. The amount of lost data is bounded by the advertised receive window of the mobile host. Finally, at $t \approx 6.8$ s the remote host's SYN/ACK passes through the bottleneck and is ACKed by the remote host one RTT later.

The remote host does not immediately restart data transmissions because the TCP Migrate options do not change the congestion-avoidance or retransmission behavior of TCP. The sender is still waiting for ACKs for the lost segments; as far as it is concerned, it has only received two (identical) ACKs—the original ACK, and another ACK with the same ACK value as part of the Migrate SYN three-way handshake. Finally, at $t \approx 7.8$ s the retransmission timer expires (the interval is counted from the arrival of the first ACK, received at the server at $t \approx 4.9$ s), and the remote host retransmits the first of the lost segments. It is immediately acknowledged by the mobile host, and TCP resumes transmission in slow-start after the time out.

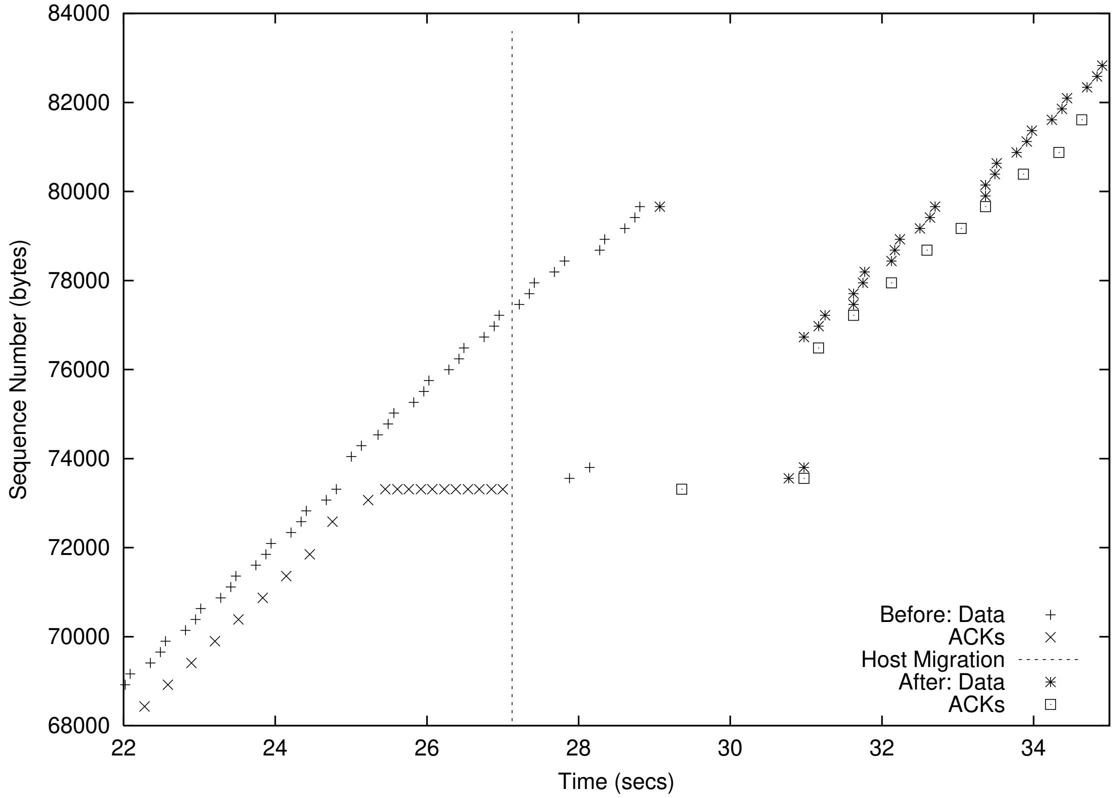

Figure 7-15: A TCP Migrate connection (with SACK) sequence trace with losses just before migration. As before, the Migrate SYN is depicted as a dashed vertical line, and the SYN/ACK is shown as the first data segment after migration.

Figure 7-15 shows the TCP sequence trace of another Migrate TCP connection on the same topology. As before, the dashed line indicates a Migrate SYN was received at the gateway at time $t \approx 27.1$ s, but it was actually issued by the mobile host at $t \approx 26.6$ s. This time, however, there were two additional losses on the connection that occurred just before the migration, as can be seen by the duplicate ACKs at $t \approx 24.9$ s. These two segments are fast-retransmitted [124] using information contained in the selective acknowledgment (SACK) option and pass through the bottleneck at $t \approx 28$ s. Unfortunately, these segments are retransmitted after the mobile host has migrated, so they, along with all the segments addressed to the mobile host's initial address after $t \approx 27.1$ s, are lost.

At $t \approx 29$ s, the Migrate SYN/ACK makes it out of the queue at the bottleneck, and the mobile host immediately generates an ACK from the new attachment point. As in the previous example, however, the fixed host is still awaiting ACKs for previously transmitted segments. It is only at $t \approx 31$ s that the retransmission timer expires and the missing segments are retransmitted. In our Linux implementation, SACK prevents the retransmission of the previously-received segments, only those segments lost due to the mobile host's address change are retransmitted, and the connection continues as before. The success of this trace demonstrates that the Migrate options work well with SACK due to the consistency of the sequence space across migrations.

The behavior shown in Figure 7-15 (preserving the SACK information across a timeout) is actually contrary to that suggested in the SACK specification [66] but leads to improved performance in this case. Hence, we suggest that TCP stacks implementing the Migrate options maintain SACK information across migration events, including those incurring a timeout. Furthermore, TCP stacks supporting the Migrate options may wish to force a timeout after connection migration rather than waiting for the RTO to expire. Automatically triggering a timeout decreases the impact of a hard handoff—where the receiver changes attachment points—bringing its performance up to the case where the sender changes attachment points. We have added this optimization to our current implementation.

As with virtualized connections, we examined handoff performance by measuring the degradation experienced by a connection as a function of the rate at which it is migrated between different attachment points. When using the Migrate options, however, there are no true soft handoffs. Because the Migrate SYN immediately validates the request, the connection is changed over to the new attachment point immediately upon receipt of the request. All in-flight data addressed to the previous attachment point is ignored, even if it is received by the end point. Hence, without loss of generality, we consider a migrating sender that oscillates periodically between two attachment points. This configuration avoids the TCP timeout seen previously since the mobile end point resumes transmitting immediately after migration. All graphs in this section represent data collected at the fixed server.

One might believe that performance degradation would increase steadily as the frequency of oscillations between attachment points increases, as observed previously (Figure 7-11). Recall that the oscillation frequency is deterministic and fixed—the connection is migrated periodically at fixed intervals. Contrary to our initial intuition, however, we find that the degradation is non-monotonic in the oscillation frequency for this experiment. Nonetheless, none of the measured frequencies performed as poorly as virtualized TCP connections subjected to hard handoffs with a two-second oscillation.

Figure 7-16 presents the throughput of a 947,570-byte download subjected to oscillating migrations with periods up to 30 seconds. (A download from one attachment point takes just over 60 seconds.) To illustrate the benefits gained from preserving SACK information across migration events, we present two experiments. In the first, labeled "go-back-n," the receiver discards all non-consecutive blocks received at the previous attachment point(s), forcing the sender to *go-back-n*—resume transmitting from the last successfully received consecutive byte. In the second experiment, labeled "SACK", we show the performance of connections that preserve the SACK information and out-of-order packets across migration events. It is immediately evident that the performance of connections that retain SACK information is generally superior to those that do not.

While the traces and exact numbers we present are specific to our link parameters, they illustrate three important interactions. The first is intuitive: the longer the interval between migrations, the higher the throughput because there is less disruption. This lack of disruption explains the decreasing overall trend and the decreasing magnitude of the "valleys" of the "go-back-n" experiments in Figure 7-16. The second effect is due to the window growth during slow start; if migrations occur before the link bandwidth is fully utilized, throughput decreases dramatically because the connection always under-utilizes the link. This phenomenon occurs at oscillation periods less than about three seconds. The third interaction occurs when migration occurs during non-SACK TCP loss recovery, either due to slow start or congestion avoidance. In this case, TCP's go-back-n retransmission policy during migration causes the connection to retransmit already-received data. This interaction explains the sporadic "valleys" in Figure 7-16 and is discussed in more detail below.

To illustrate the slow-start and loss recovery interactions, Figure 7-17 depicts the progression of five separate "go-back-n" 947,570-byte downloads, each subjected to a different frequency of oscilla-

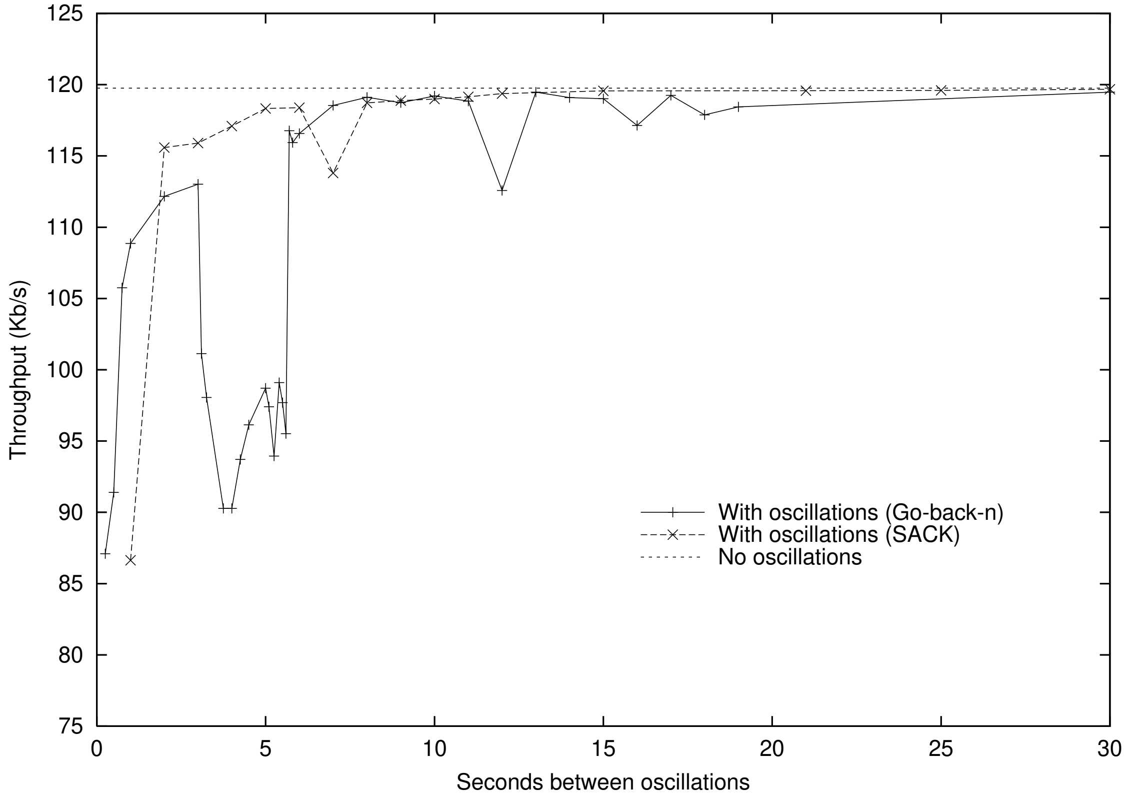

Figure 7-16: Throughput vs. oscillation rate with the TCP migrate options on a TCP connection without SACK. A download conducted entirely from one attachment point achieves a throughput of 119.48 Kbps.

tion. The migration events are often visible as regular pauses. Figure 7-18 examines the sequence traces for the interval from 35 to 40 seconds of the "go-back-n" connections subjected to two- and five-second oscillation periods. At two seconds, the connections are still ramping up their window sizes and have experienced no losses. At five seconds, the connections experience multiple loss events as slow start begins to overrun the bottleneck buffer. Four retransmissions can be observed to be successfully received; the fifth, unfortunately, arrives *just after* the Migrate SYN from the new attachment point. Since the remaining data is non-contiguous, it is flushed at the remote end point in accordance with the *go-back-n* policy, and retransmission resumes from substantially earlier. Regardless of the exact period of interaction, the migration overhead for realistic rates of attachment point changes is almost imperceptible.

7.3 Session continuations

In this section, we consider the effectiveness of session continuations across three metrics: ease of deployment, ability to conserve resources, and speed of session resumption. Our experience shows that session continuations can be integrated with existing Internet server applications with only minor modifications. Measurements of our *Migrate*-enabled server applications show that complete session continuations are able to effectively conserve both system memory and file descriptors during periods of disconnection and increase session resumption latencies only slightly (around 80 ms in our tested configuration).

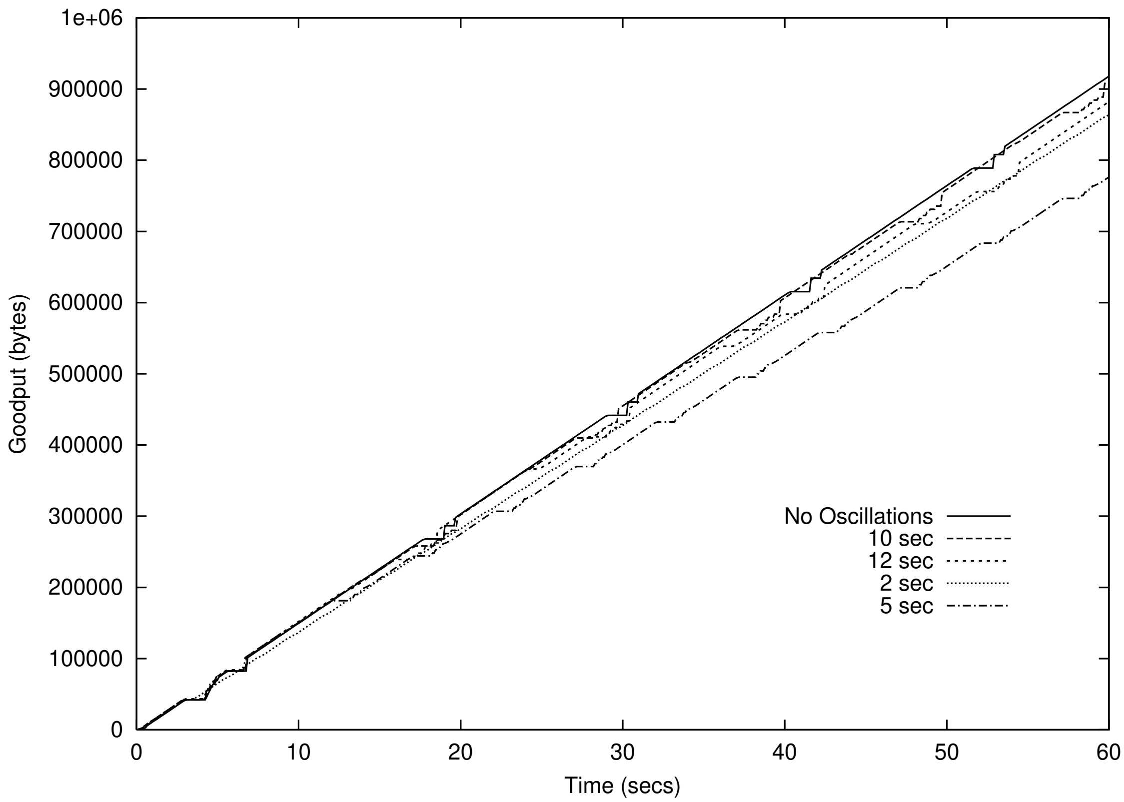

Figure 7-17: Connection ACK traces for varying rates of server attachment point oscillation using the go-back-n policy. The TCP receive buffer is 64 KB.

7.3.1 Ease of deployment

To evaluate the ease with which session continuations can be added to existing, session-based Internet server applications, we selected two common Internet applications to modify: SSH and FTP. We acquired recent versions of popular server implementations, OpenSSH 3.0.2p1 and WuFTP 2.6.2, respectively. The author was not familiar with the implementation of either application prior to its selection for this evaluation. The hope, then, is that the particular architectures are representative of typical session-based Internet servers and are not biased by any constraints or ease of integration with the session continuation abstraction.

Both applications were extended to generate complete continuations upon session suspension. Table 7.1 shows the number of lines of code (LoC) added to each application to support session continuations. The size of the original application is presented for comparison. The modifications

Name	Version	Application Size	Changes Required
SSH	OpenSSH 3.0.2p1	48,967 LoC	262 LoC
FTP	WuFTP 2.6.2	21,147 LoC	358 LoC

Table 7.1: The changes required to add session continuations to two popular Internet server applications. The presented figure includes both the additional code required to generate the continuations and any required changes to existing code.

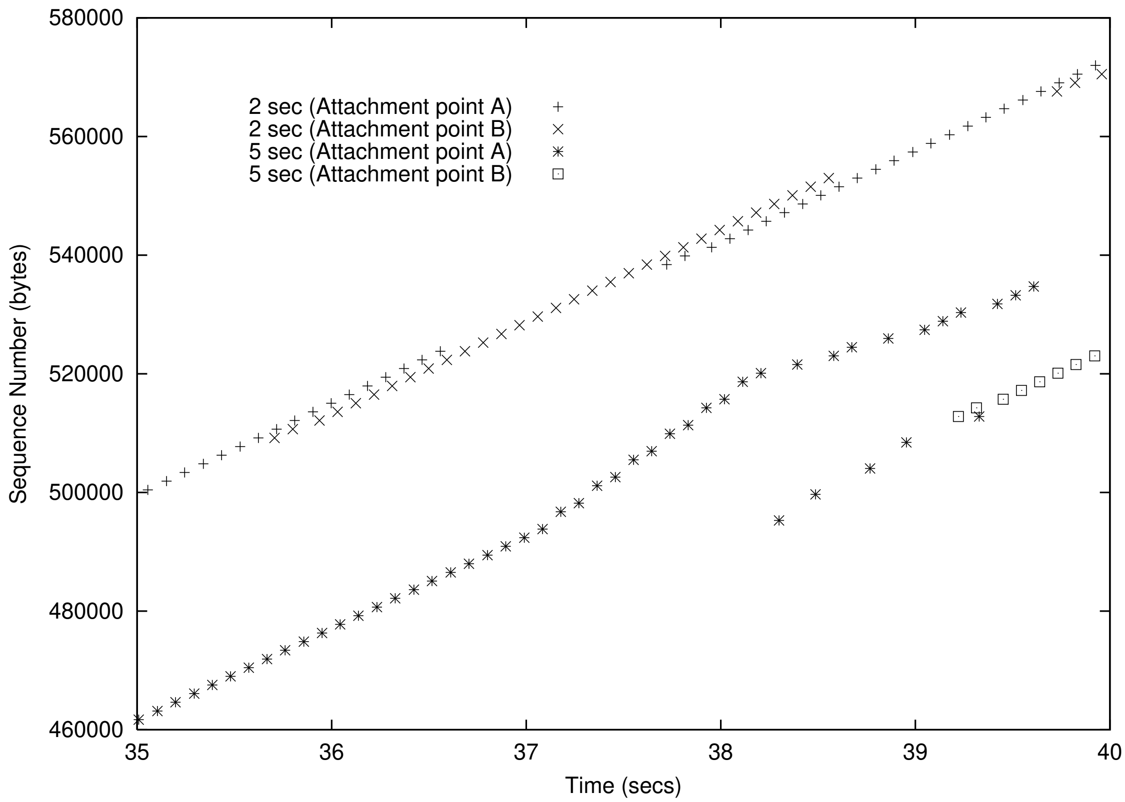

Figure 7-18: Sequence traces of oscillatory TCP migration behavior under the go-back-n policy. These are the same traces shown in Figure 7-17.

represent between 0.5% and 1.5% of the code base in both cases. Due to the problematic nature of quantifying programmer effort, we do not present any measure of the difficulty of making the modifications, other than to note that in both cases the applications were modified in less than two work days. We provide below a brief synopsis of the changes required for each application. Selected portions of the code are presented in Appendix B.

In both cases, continuation generation is simplified by the ability of the application to execute in the absence of network connectivity. SSH and FTP servers are loops: they consume data from a source—local applications or files, respectively—encode it, and delivery it to a remote client, and *vice versa*. Hence, if either application is allowed to run long enough, it is guaranteed to pass a particular point in its execution loop. If a quiescent point exists in the loop, where the application state can be described concisely, then continuation generation is straightforward. Fortunately, both servers are single threaded and block only on external I/O operations, so such quiescent points exist and are easy to identify. In particular, after completing all pending tasks, both applications use the `select()` system call to block awaiting additional input.

The session generation process is, therefore, quite similar in both applications. Even though disconnection may be detected at arbitrary points in the servers' execution, they only generate continuations from a specified, quiescent state. Suspension requests received at any other time are simply recorded by their mobility handlers and queued until the quiescent state is reached again. In each pass through the quiescent state, the applications check to see if a suspension request has arrived. If so, a continuation is generated. Otherwise, the processing loop repeats.

In particular, SSH can continue to process data already received from the network and any data already consumed from child processes until it exhausts the available data. Outstanding data can be (de)encrypted, (de)compressed, and either delivered to the local application or buffered for later network transmission. In any case, the server can run until it quiesces, at which point its state can be succinctly described in terms of the authentication credentials of the remote user, the state of the encryption and compression engines, and any outstanding data buffered for later network transmission. Similarly, FTP can continue reading from its data source and buffer any outstanding data for delivery to the client. Since both applications read data in small, fixed-sized blocks, the amount of data outstanding is limited by the size of this data block.

7.3.2 Resource conservation

One of the main features of session continuations is their ability to allow corresponding hosts to conserve resources during periods of disconnection. We provide experimental evidence of *Migrate*'s ability to conserve two particular resources, system memory and file descriptors, in this section. We measure the resource consumption of the two *Migrate*-aware server applications described in the previous section, SSH and FTP running on Linux 2.4.1, and show that both consume significantly less memory and fewer file descriptors when suspended through session continuations. We also show that *Migrate* is able to conserve power on multi-homed hosts by turning off power-hungry, high-bandwidth interfaces when bandwidth-sensitive sessions are suspended.

Both applications fork separate processes for each session. Hence, our complete continuations are able to release an entire process during periods of disconnection. The resources consumed by an individual Linux process are quite diverse. While not necessarily the most important (a good virtual memory system will swap inactive pages to disk), the most dramatic resource savings that is straightforward to quantify is system memory. Figure 7-19 shows the memory footprints of processes serving active SSH and FTP sessions. For comparison, the processes are shown with our *Migrate*-extensions and without. The TESLA stub required for *Migrate* support increases the memory usage of both applications. The smaller increase experienced by the SSH process is due to the fact that it already loads many of the libraries required by the TESLA stub. The FTP server's library selection is relatively spartan in contrast. In both cases, however, the complete continuations generated upon disconnection are minuscule in comparison—18 KB and 98 KB, respectively.

The difference in the size of the continuations is due to the much larger connection buffers required by the FTP session, as shown in Figure 7-20. For each application, an uncompressed continuation is shown on the left, the compressed version on the right. The FTP session has two active TCP connections, both of which must be double-buffered, in contrast to the SSH session. Further, because the suspended SSH session had just started up, its connection buffers were largely unused, resulting in much more effective compression. The FTP session's connection buffers were filled with less-easily compressed file data, resulting in a substantially larger continuation. Not surprisingly, the SSH continuation contains more session state, since the FTP session need only store the user's ID, settings, and current directory, while the SSH session must preserve the session keys and any as-yet-unsent data in the encryption buffers.

Table 7.2 shows the number of file descriptors required for an individual session in each application. The SSH session is currently logged in to a command shell, while the FTP session is in the middle of a file download. The second column shows *Migrate* adds an overhead of two to three file descriptors per session which are used by TESLA. Only one of the SSH file descriptors is a network connection, while FTP uses two (a control channel and a data channel). Of the remaining file descriptors, our continuations preserve eight and three, respectively. By generating simple resource

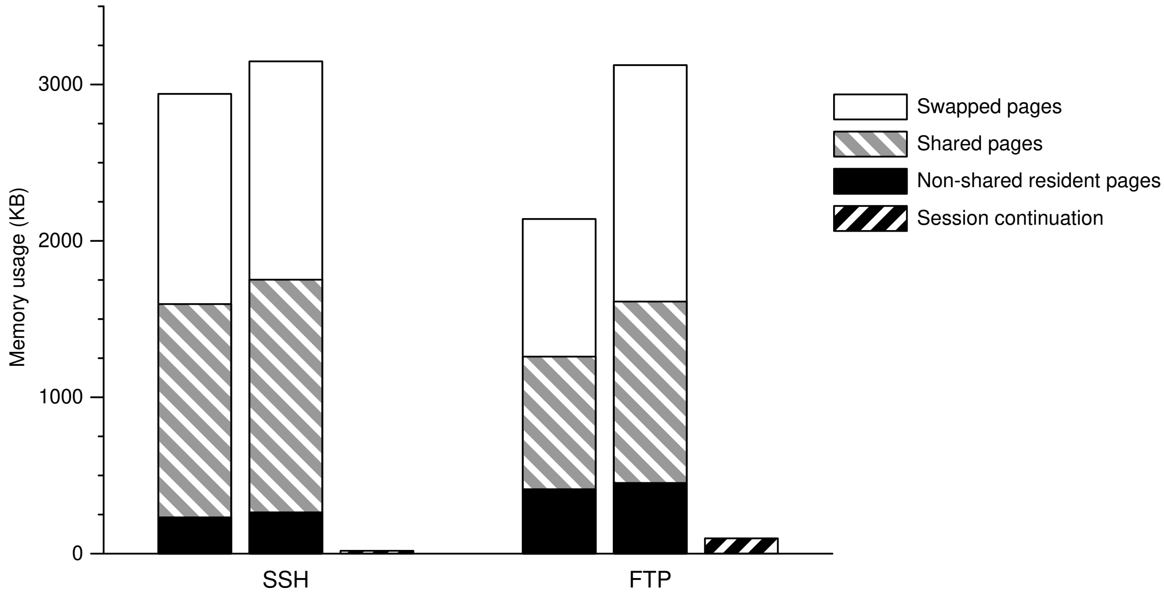

Figure 7-19: The memory footprints of sample *Migrate*-aware servers. We report values observed using gcc version 2.96 with the `-O2` option on a Linux 2.4.1 system with 256 MB of RAM and 512 MB of swap.

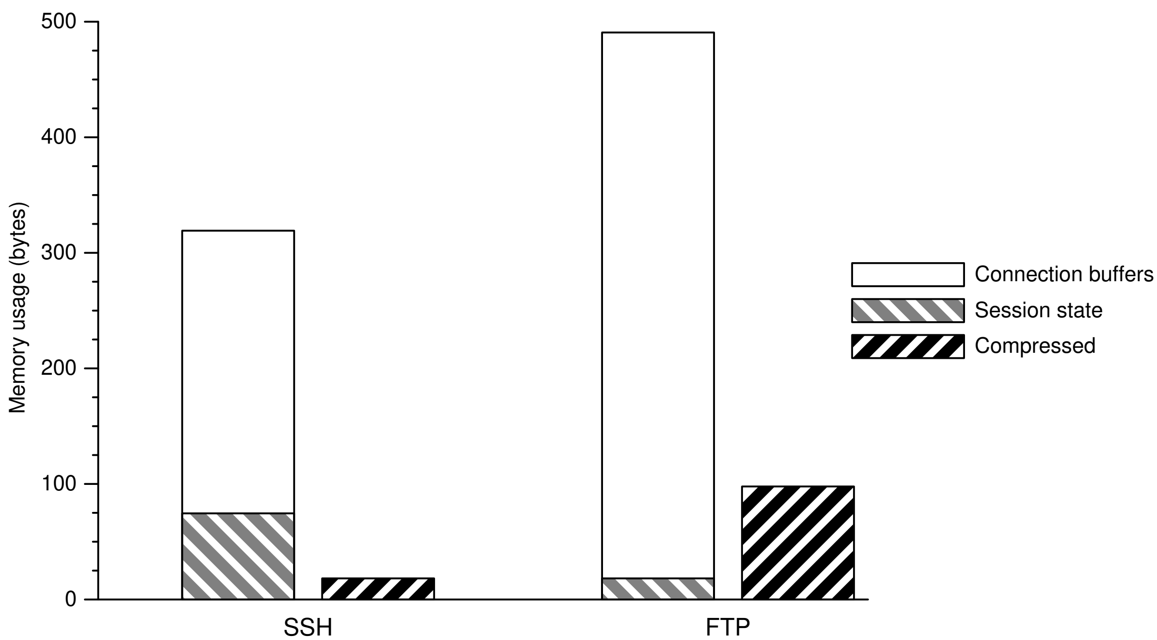

Figure 7-20: Complete continuation sizes. The sizes reported here include persistent application state, buffered network connection data, and all associated *Migrate* control data necessary to invoke the communication.

Name	Native	With Migrate	Network Connections	Suspended	Compressed
SSH	9	12	1	8	3
FTP	8	10	2	3	1

Table 7.2: The file descriptor usage of two popular Internet server applications. The first two columns indicate the number of file descriptors used by an active session before and after enabling *Migrate* support. The third column shows the number of these descriptors corresponding to active network connections. The last two columns present the number of file descriptors required for sessions suspended through a continuation. The "Suspended" column indicates the number of descriptors included within the continuation, and the "Compressed" column shows the actual number held open by *Migrate* during disconnection after generating all available resource continuations.

continuations (closing redundant descriptors, leaving only one descriptor pointing to a particular resource), *Migrate* is able to reduce those numbers further to three and one, respectively. In the case of SSH, the remaining descriptors correspond to the connection to the user's shell, its pseudo-tty, and the `/dev/ptmx` device[1]. FTP, on the other hand, requires only one remaining file descriptor corresponding to the source of content being downloaded. (A more sophisticated FTP session continuation might close the file upon suspension and open it again once the session resumed. Doing so could affect the application semantics, however, as the file might be deleted, modified, or otherwise altered while closed. Keeping the file open during suspension ensures the original application and operating system semantics—whatever they might be—are maintained.)

Finally, we show that by handling multi-homing at the session layer, *Migrate* is able to conserve additional resources during session disconnection. By allowing sessions to migrate back and forth across network interfaces, hosts with multiple network interfaces can trade off cost, performance, and power consumption. It is not always the case that all sessions want to use the same interface. This need has been recognized previously [21], but solutions either rely on a proxy [64] or require all mobile-aware connections to use the same attachment point [21, 148].

Migrate, on the other hand, allows end points to change attachment point on a per-session basis and at any time during the session. We demonstrate the utility of this flexibility by measuring the power consumption of a hand-held PDA with two network interfaces: an 802.11b interface and a Bluetooth interface. Initially, the 802.11b interface is in use by a bandwidth-hungry application. When powered up, the 802.11b interface is preferred even by bandwidth-insensitive applications. Hence, when a long-running, bandwidth-insensitive background transfer starts up, it also uses the 802.11b interface. When all bandwidth-hungry sessions terminate or suspend, however, *Migrate* system policy specifies that the 802.11b interface should be powered down in favor of the Bluetooth interface if Bluetooth coverage is available (802.11b has substantially larger range than Bluetooth).

Figure 7-21 shows the instantaneous current draw of a Compaq iPAQ 3600 over a ten-second period. Initially, a TCP download is being conducted over the 802.11b interface. At $t \approx 5$ s *Migrate* powers down the 802.11b device and migrates the transfer to the Bluetooth interface. Power consumption drops from almost 800 mA to around 450 mA when the bandwidth-insensitive session is migrated to the Bluetooth device, a savings of almost 350 mA. Power consumption is slightly higher for the first second or so while *Migrate* negotiates the change of attachment points, which requires several packet exchanges and significant computation.

[1] `/dev/ptmx` is a device used to control the allocation of pseudo-ttys on Linux. A resource-specific continuation could be written to allow the closing of it as well, but we have not yet implemented such a continuation.

7.3.3 Continuation latency

In exchange for their decreased resource utilization, session continuations introduce additional latency to session resumption over and above the migration latency discussed in Section 7.2. The exact delay depends entirely on the particular session continuation in question. We have measured the time required to resume sessions suspended using the complete continuations presented in this section on a 600-Mhz P3 running Linux 2.4.1. In order to separate delays related to the network, both the clients and servers used in this experiment were run on the same machine; all network connections use the loopback interface.

Figure 7-22 shows the median resumption latency of both SSH and FTP session continuations. For comparison, we also present the initial session establishment times with and without *Migrate*. The *Migrate*-aware session represents the same session that was resumed; the native sessions were separate, but identical. For SSH, the connection time measures the time between invoking the SSH command and completion of the `ntptime` command on the remote host. For FTP, the initial connection time measures the time between the client's `connect()` call and the completion of an automatic USER/PASS login exchange.

In the case of FTP, the additional connection latency due to *Migrate* is significantly greater than that suggested by Figure 7-4. This delay is due to the Network Information Service (NIS)-based authentication mechanism used to authenticate the login. This mechanism results in additional (*Migrate*-enabled) network activity to a remote server. In both cases, the resumption time is similar (roughly 80 ms), corresponding to the cost of starting up a new process and passing it the preserved file descriptors. In the case of SSH, this process is significantly faster than the original session instantiation, which requires several expensive cryptographic operations. In the case of FTP, the resumption operation is relatively more heavyweight. We note, however, that the resumption cost is still small and results in a seamless user experience. Terminating the FTP session and restarting it, while possibly faster (if user intervention can be avoided), requires either an enhanced FTP client or an accommodating user.

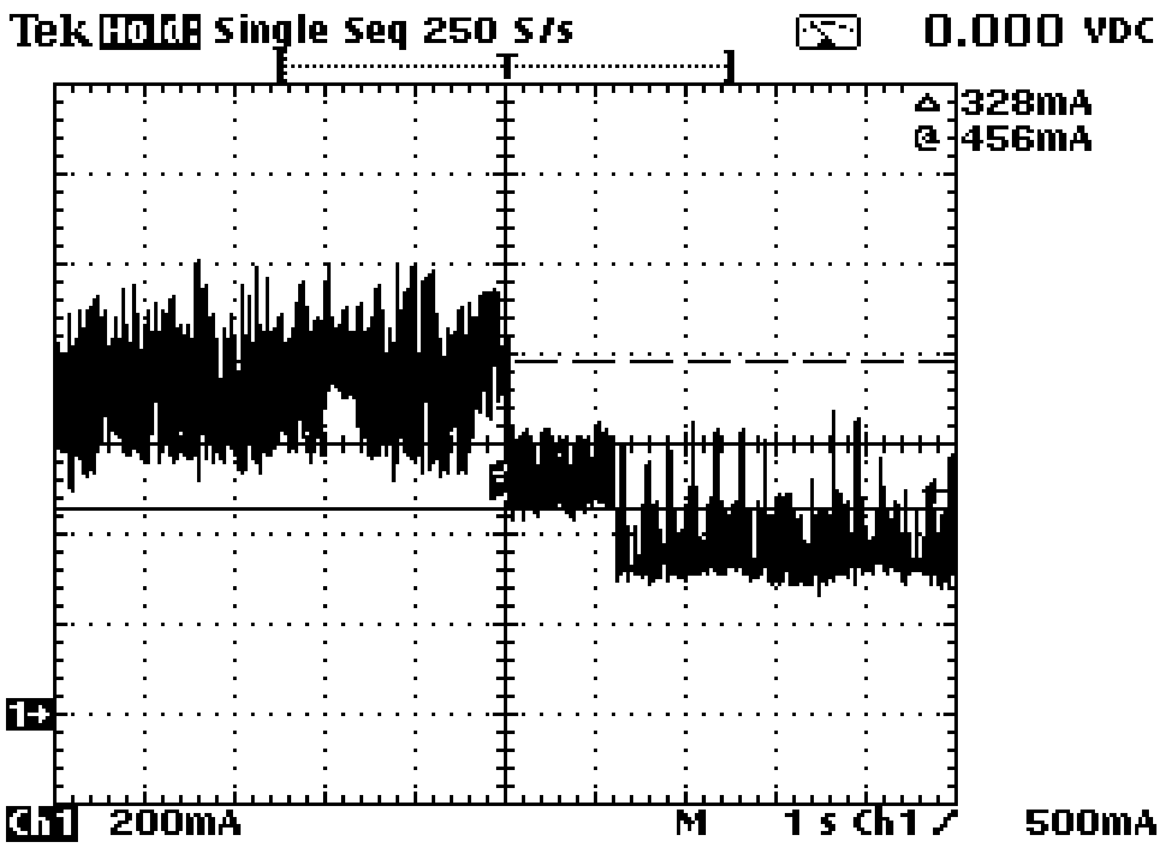

Figure 7-21: The instantaneous power consumption of a Compaq iPAQ 3600 over a ten-second interval. Each grid line on the horizontal axis represents one second; the vertical grid marks are 200 mA. Zero is marked on the vertical axes by the arrow on the left hand side. Initially, the iPAQ is downloading a TCP stream using a Cisco Aeronet 350 802.11b interface. At time $t \approx 5$ s, the transfer is migrated to a Brainboxes BL-500 Bluetooth interface, and the 802.11b interface is powered down. The solid horizontal line was manually placed to illustrate the average power consumption after migration; it corresponds to 456 mA as shown in the upper right. Similarly, the dashed horizontal line roughly corresponds to the average power consumption before migration. The difference between the two lines, as also shown in the upper right, is 328 mA.

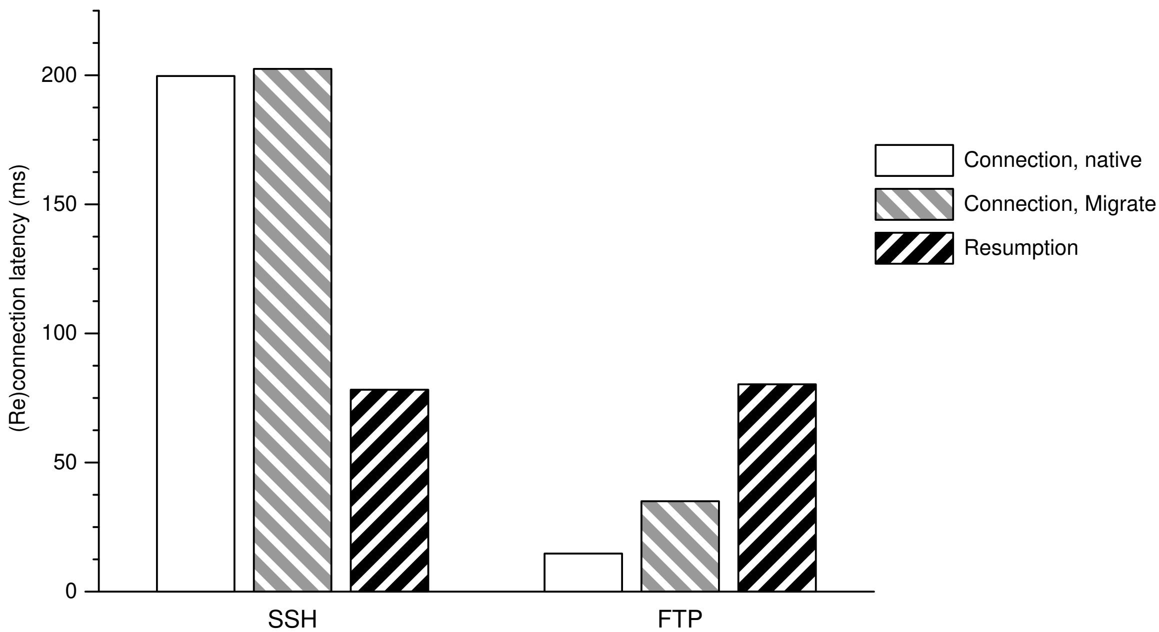

Figure 7-22: The connection and resumption latency for sample *Migrate*-aware applications. The resumption latency measures the time to invoke a complete continuation and restore all suspended network connections. As in Figure 7-5, it does not include the time necessary to resynchronize those connections.

If it were done when 'tis done, then 'twere well it were done quickly.

All's well that ends well.

- William Shakespeare

Chapter 8

Conclusions

This dissertation addresses the problems faced by network applications on intermittently connected mobile hosts. Users of mobile devices have come to expect seamless operation of their local applications despite frequent interruptions, changes in network attachment point, and periods of disconnectivity. This "suspend/resume" model of operation is not well supported for network applications by today's operating systems, however. Traditional network connections are unable to survive changes in attachment points or periods of disconnection, forcing applications to manage all aspects of nomadic computing by themselves. Unfortunately for today's user, many popular network applications do not include this functionality.

We have presented *Migrate*, an architecture to support Internet mobility. Legacy applications can rely on *Migrate* to transparently manage any changes in attachment point or periods of disconnection, providing mobile users with the "suspend/resume" semantics they expect from laptops and PDAs. Mobile-aware applications can use *Migrate*'s extended API to adapt to changing network conditions. In particular, session continuations can significantly decrease the resource footprint of disconnected sessions, enabling Internet servers to support large numbers of open but suspended sessions. If suspended sessions consumed the same amount of resources as active sessions, popular servers would be unable to support *Migrate*'s "suspend/resume" semantics due to resource constraints.

This chapter begins with a summary of our contributions. Section 8.2 presents a set of higher-level guidelines extracted from our work that we believe can assist in the design of future mobile-aware network applications and protocols. Finally, the dissertation concludes with a brief discussion of possible future directions.

8.1 Contributions

In an attempt to design an efficient, flexible, and easy-to-use architecture to support intermittently connected end points in the Internet, this dissertation makes the following specific contributions:

- The development of an *end-to-end* approach to Internet mobility that is based on a system-supported *session* abstraction, along with a specific architecture, *Migrate*, that implements this approach.
- The design and implementation of Migrate options that extend TCP to support the migration of TCP connections to new attachment points.

- *Session continuations*, extensions to the session abstraction that enable application-specific suspend/resume handling. By providing system support for session continuations, hosts can realize significant resource savings during periods of disconnection. Session continuations also have applications to problems outside of host mobility such as load balancing and application migration. Carefully crafted continuations can be executed in environments (hosts) other than those that created them, enabling a form of inter-host session migration.

- An application-agnostic, *attack-equivalence* security model that ensures any attacks enabled by mobility support reduce to ones already present in a non-mobile environment.

- A proposed session API that allows mobile-aware applications to specify intelligent disconnection handling through the use of *continuation-passing style*, and an understanding of how a variety of network-based applications can use the API to provide application-specific functionality.

- An evaluation of the efficiency of our prototype *Migrate* implementation. Our results show that the throughput impact of connection virtualization is small (2% or less for moderate block sizes) for sessions operating over common access link technologies. The overhead can be considerably larger, however, when virtualizing extremely-high bandwidth ($>$ 350-Mbps) connections or those using small ($<$ 200-byte) block sizes. When used in conjunction with the TCP Migrate options, *Migrate*'s overhead becomes almost negligible and is restricted to session establishment and migration events.

- A demonstration of the effectiveness, flexibility, and ease-of-use of our session continuation abstraction. We show that servers for two popular Internet applications, SSH and FTP, require only small modifications to support session continuations, and that suspended sessions for both applications consume only a few tens of bytes of secondary storage and between one and three file descriptors.

We believe that our session-based Internet mobility architecture is suitable to a wide range of operating environments. In particular, our three complimentary approaches to changes in attachment points and periods of disconnectivity—TCP migration, connection virtualization, and session continuations—provide a comprehensive solution spectrum. For frequent, instantaneous changes of TCP attachment points, the TCP Migrate options enable connection resumption in one RTT with very little throughput overhead. For other transport protocols, or attachment point changes accompanied by periods of disconnection, our connection virtualization approach is able to resume session connectivity in four RTTs (plus any necessary connection synchronization time). Finally, for extended periods of disconnection, session continuations can decrease resource utilization on the disconnected hosts while enabling applications to adapt to changes upon resumption.

All software implementations presented in this dissertation are available for download online in both source and executable formats at `http://nms.lcs.mit.edu/migrate`.

The *Migrate* daemon and TESLA-based session library are available for UNIX. While we believe our implementation is portable to almost any POSIX-compliant platform, we have tested it only on Linux and FreeBSD. The TCP Migrate options are only available as a patch for Linux version 2.2.17.

The session continuations for SSH and FTP described in Chapter 7 are available as patch files. Selected sections of the source code are listed in Appendix B.

TESLA can be downloaded from `http://nms.lcs.mit.edu/tesla`.

8.2 Guidelines

In the course of designing and implementing *Migrate*, we have identified several principles that were helpful in making architectural and implementation decisions. We believe these lessons apply more generally to the design of any network application or protocol for use in a mobile environment:

- **Eliminate lower-layer dependence.** The first step in enabling higher-layer mobility handling is to remove inter-layer dependences. In a 1983 retrospective paper on the DoD Internet Architecture, Cerf wrote "TCP's [dependence] upon the network and host addresses for part of its connection identifiers" makes "dynamic reconnection" difficult, "a problem . . . which has plagued network designers since the inception of the ARPANET project in 1968." [19] We presented two distinct solutions to this particular problem in Chapter 4: connection virtualization and rebinding through the Migrate TCP options.

  A host of other problems crop up because of similar linkages. For example, the increasing proliferation of NATs in the middle of the network has caused problems for applications (like FTP) that use network- and transport-layer identifiers as part of their internal states. These problems can be avoided by removing any assumption of stability of lower-layer identifiers. If a higher layer finds it necessary to use a lower-layer identifier as part of its internal state, then the higher layer should allow for it to change and continue to function across such changes. *Migrate* sessions use application-layer names to identify end points; mobile-aware applications using the *Migrate* API presented in Chapter 3 do not depend on the current attachment points in use by either end point.

- **Do not restrict the choice of naming techniques.** Many researchers have observed that the one of the most difficult problems raised by mobility, namely locating the mobile end point, can be addressed through the use of a sophisticated naming system that supports dynamic updates. Hence, many proposals for managing mobility in the Internet attempt to provide naming and location services as a fundamental part of the mobility architecture. Unfortunately, the tight binding between naming schemes and mobility support often causes the resulting system to be inefficient or unsuitable for various classes of applications. Our experience shows that each application is likely to end up using a naming scheme that best suits it (*e.g.*, INS [1], DNS [68], Twine [9], UPnP [137]) rather than suffer the inadequacies of a universal one. *Migrate* allows each application to use a naming system of its choice.

- **Provide services at the end points.** A great deal of previous work in mobility management has relied on a proxy-based architecture, providing enhanced services to mobile hosts by routing communications through a (typically fixed) way-point that is not collocated with the host [8, 23, 40, 64, 89, 146]. It is often easier to deploy new services through a proxy, as the proxy can provide enhanced services in a transparent fashion, inter-operating with legacy systems. Unfortunately, in order to provide adequate performance, it is not only necessary to highly engineer the proxy [64], but locate the proxy appropriately as well.

  Several researchers have proposed techniques to migrate proxy services to the appropriate location, avoiding the need to pre-configure locations [23, 138]. Unfortunately, all candidate proxy locations must be appropriately pre-configured to participate. Further, in the face of general mobility, proxies (or at least their internal states) must be able to move with the mobile host in order to remain along the path from the host to its correspondent peers. This is a complex problem [146]; we observe that it can be completely avoided if the support is collocated with the mobile host itself. Hence, *Migrate* support is implemented at each mobile end point.

- **Optimize for the static case.** When one considers movement patterns with respect to communication session durations, today's mobile devices fall fairly naturally into two categories: those that are basically static, and move only every few minutes or hours (*e.g.*, most laptops or even PDAs), and those that move almost constantly (*e.g.*, cell phones). Those with rapid and continuous movement often require *link-layer* or *micro-mobility* techniques to handle the short time-scales over which their mobility needs to be tracked. Those devices that move rarely relative to on-going communications (of which there are a significant number) should not be forced to pay a large overhead when the chance of moving during a session is low.

  Many mobility schemes add a significant amount of additional baggage to network communications that they significantly impact communication between hosts even when neither of them actually moves during the course of communicating. The mere possibility of movement forces the end points to incur severe performance penalties. These designs stem from an over-generalization; many Internet hosts change attachment points only rarely. While many devices are physically mobile, their network attachment point often changes at much longer time-scales due to techniques like physical-layer bridging. As demonstrated in Chapter 7, *Migrate*'s overhead is often inconsequential when used in bandwidth-restricted environments such as congested, wide-area networks or wireless access links. Furthermore, when the TCP Migrate options are used instead of connection virtualization, the overhead becomes almost negligible and is restricted to session establishment and migration events.

8.3 Open questions

Despite our best intentions, this dissertation raises many more issues than it addresses. We discuss several of the most interesting ones below, as well as possibilities for extensions to the *Migrate* architecture.

8.3.1 Policy interface

Migrate currently lacks an expressive policy interface. In particular, there is no way to describe a particular session other than through its constituent network connections. This limitation is most evident in the system-wide policy file (Appendix A), which is constrained to describe per-session preferences in terms of connection ports and IP addresses—network attachment points, not session end points. An approach that allowed policy to be specified in terms of session end points would be preferred but is complicated by *Migrate*'s flexibility with regard to the naming system used to name any particular session end point. Because there is no canonical name for a particular session end point—indeed, the name of a session end point is entirely dependent on the application to which the session belongs—it is difficult to describe in a general way.

8.3.2 Multi-homing support

Because it operates at user-level, *Migrate*'s support for multi-homed hosts is limited by the operating system's ability to manage multiple network interfaces. In particular, the operating system remains in control of how packets are *routed*, including which interface is actually used to transmit the packet. While *Migrate* can select which attachment point (IP address) is used through the `bind()` system call, the final selection of outgoing network interface and next-hop gateway is made by the operating system using the local routing table. This means that for hosts with multiple interfaces, *Migrate* may not be able to control which interface is used for outgoing traffic on a session.

Most operating systems select the appropriate outgoing interface and next-hop gateway for a packet based on its destination address. Multi-homed hosts, by definition, may have multiple, alternative routes to the same destination. How a particular route is chosen varies from operating system to operating system. For example, KAME/NetBSD will chose among alternate routes based on a hash of the source and destination IP addresses [133]. To enable true support for multi-homing at the session layer (or, indeed, any higher layer than the network layer), an operating system needs to allow *Migrate* to explicitly specify not only which address to place in outgoing packets, but which interface (route) to use as well. We previously implemented such an extension to FreeBSD in support of network-layer inverse-multiplexing [118]. Hopefully, demand from research projects implementing higher-layer multi-homing (as found both in *Migrate* and transport-layer solutions like SCTP [126]) will result in the development of a standard interface. Alternatively, *Migrate* could be configured to establish *ad-hoc* point-to-point IP tunnels for each session on a multi-homed host. We have not examined the performance impact of this approach.

8.3.3 Extensible security

While *Migrate*'s attack-equivalent security model has many benefits, it comes at the cost of ignoring any security provisions already put in place by the application. In theory, *Migrate* could leverage an application's authentication mechanism to authenticate binding updates rather than providing its own, anonymous scheme. The difficulty with relying on the application, however, is that *Migrate* cannot be sure what particular semantics the application's authentication scheme provides. It is well-known that one of the most common sources of security vulnerabilities is the poor implementation or misuse of an otherwise secure mechanism [5]. It would be useful to consider whether a mechanism could be devised to allow applications to describe to *Migrate* how to use their authentication scheme to validate binding updates.

Migrate would also benefit from the ability to use variable length session keys, and re-key open sessions. Currently, session end points negotiate keying material once, at the beginning of a session, and use the same material to secure the session for its entire life time. Our implementation uses the same key length of all sessions, regardless of their expected duration. This leads to increased overhead for short sessions due to the unnecessary key strength. Further, especially long-lived sessions may be vulnerable to off-line attacks. Both vulnerabilities could be addressed by initially keying the session with weaker keys, and re-keying the session if it lasted longer than some cut-off. Sessions that were expected to be long lived at the outset (applications might use the `session_length()` call as a hint) could be keyed with a stronger key initially. Re-keying is problematic, however, because the end points do not authenticate themselves to *Migrate*—only to the remote application end point. Using the old key to secure the exchange of new keying material is obviously ill-advised. Hence, the application would likely need to be involved with any re-keying operation to re-authenticate the end points.

8.3.4 Application structure

This dissertation addresses the needs of session-based network applications that maintain durable state between two end points across multiple packet exchanges or network connections. This is a common architecture used in a large number of applications. There are two major exceptions, however. Some multi-party applications such as multi-media conferencing, online games, and content distribution networks do not maintain one-to-one relationships between application end points.

Instead, multi-party applications may maintain sessions consisting of multiple end points. Because the semantics of connectivity, session suspension, and end-point tracking are unclear in this environment, *Migrate* does not support applications that use multicast [25]. Extending *Migrate*'s notion of a session include more than two end points is an area for future research.

In addition to applications that maintain state across multiple end points, some applications maintain no session state whatsoever. The most notable example of this class of application is the Web. The Hypertext Transfer Protocol (HTTP) [35] was designed as a stateless protocol, and many Web servers maintain no state at all between page requests. This class of application functions well in a mobile environment without explicit support for disconnection. *Migrate* does not interfere with their operation, however, and may safely be used in conjunction with stateless applications. The major benefit of *Migrate* for stateless applications is the ability to resume interrupted transfers—such as large Web downloads, for example. If it was possible to design all applications in a stateless fashion, *Migrate*'s utility would be greatly diminished. How far the stateless architecture can be extended is an open question. It would be interesting to characterize the types of applications that can or cannot be architected in this fashion.

Many Web-based applications do indeed preserve state across requests, however. Such applications can be implemented in two ways: they can either maintain the state on the server using dynamically-generated pages and some sort of back-end database or include all necessary state in each client request. This latter approach is typically implemented using HTTP cookies [58]. Cookies bare many similarities to session continuations in that they provide an end point with sufficient state to resume processing a session where it left off. Unlike continuations, however, which are stored locally at each end point, cookies are transferred to the remote end point and reflected back. Because continuations are usually generated by an end point while it is disconnected, storing the continuation on the remote host is infeasible. It may, however, be stored elsewhere if desired.

8.3.5 Session continuation extensions

Session continuations present a number of intriguing possibilities. In their current form, continuations are generated independently at each end point and are transparent to the remote end point. That is, a session end point is unaware of whether the remote end point has generated a continuation, let alone how it was generated. Continuations are designed to resume sessions from their previous states; they must be invisible to the remote end point by definition.

There may be cases, however, when resources could be more efficiently conserved if continuations were generated in concert by both end points. More generally, it may be sufficient for one end point to know that the remote end point is generating a particular continuation. This knowledge might allow for non-transparent continuations. Such an extension would require the ability to name continuations and, further, some form of versioning to ensure that the continuation actually being generated corresponds to the remote end point's expectations.

Another enticing possibility is to allow end points to off-load continuations. If continuations were sufficiently portable, it might be possible to invoke them in an entirely separate host. In this case, the session is changing not only network attachment points, but system host as well. We have found this approach feasible in the domain of static Web servers [119] and believe it holds more general promise. *Portable continuations*, as we've dubbed them, provide a form of lightweight process migration. A complete implementation would likely face many of the challenges typically faced by full-fledged process migration. Portable continuations appear more general, however, as they enable migration of individual sessions, as opposed to entire processes. For applications that host many sessions in one process, this may be more desirable.

Appendix A

Policy File

The *Migrate* policy file is evaluated each time *Migrate* considers migrating a session between attachment points. The policy file is really a script written in the *Tcl* [84] scripting language. The script is evaluated whenever there is a change in connectivity status for session that *Migrate* is managing. The script is run once for each connection in a session. The input for each invocation is the description of the network connection in question—its protocol and current remote and local attachment points, including ports.

The script can take one of two actions: either migrate the session to a new interface, or leave it alone. Sessions left on an interface that currently lacks connectivity are suspended by the *Migrate* daemon. When migration is selected, the new local attachment point is determined by a set of numerical scores specifying the respective desirabilities of available interfaces for the session. An interface with a higher score is preferable to an interface with a lower score. Interfaces which are down are given a score of zero.

A.1 Commands

Table A.1 lists the commands available to the policy script. There are only two ”primitive” commands, `score-interface` and `migrate`, which provide information to the *Migrate* daemon. The `on` command is used as a control structure to apply certain commands based on properties of the session being examined.

A.1.1 `score-interface` command

`score-interface` *interface-glob score*

Assign the score em score to all interfaces whose labels match the glob *interface-glob*. For example, to assign a score of 100 to all Ethernet interfaces (except *eth1*, which has a score of 150), but a score of only 50 to dialup (PPP) interfaces, one might use this:

Command Name	Arguments	Description
`score-interface`	*interface-glob score*	Score *interface-glob* as *score*
`migrate`	(*theshold* | `if-dead` | `never` | `always`)	Migrate a connection
`on`	(*port* | `l-port` | `p-port`) *port-spec* ...	Consider specific connections

Table A.1: The commands available to a *Migrate Tcl* policy script.

```
score-interface eth* 100
score-interface eth1 150
score-interface ppp* 50
```

A.1.2 **`migrate`** command

`migrate` (*threshold* | `if-dead` | `never` | `always`)

Specifies under what conditions a session should be migrated. The *threshold* parameter specifies that a session should be migrated if another interface has a score that is at least *threshold* higher than that of the session's current interface. The `if-dead` option specifies that migration is desired only if the interface presently being used is down. The `never` option specifies that migration is never desired. The `always` option specifies that migration should be attempted whenever another interface has a score which is even minutely higher.

A.1.3 **`on`** command

The `on` command is used as a control structure to apply certain commands based on properties of the session being examined. In many instances, `on` refers to properties of the connections contained within a session rather than the session itself.

Port selection

`on` (*port* | `l-port` | `p-port`) *port-spec commands* ...

Evaluate commands if the session being examined uses port *port*. With `l-port`, the local port number must match *port-spec*; for `p-port`, the peer port number must match; with neither, either the local or peer port number may match.

port-spec is a comma-delimited list of one or more elements. Each element is either

1. a service name in `/etc/services`,
2. a port number, or
3. a range of port numbers (e.g., 3000–3010).

For example, to assign Ethernet interfaces a score of 50 for connections on port 80 (HTTP), port 8000, and ports 8080–8089, one might use this:

```
on port http,8000,8080-8089 score-interface eth* 50
```

Interface selection

`on` *remote-ip ip-spec commands* ...

Evaluate commands if the session being examined is currently connected to an IP address matching *ip-spec*. *ip-spec* is either an IP address or a CIDR-style IP mask. For example, to assign Ethernet interfaces a score of 50 for sessions to the class-C 18.31.0.* subnet:

```
on remote-ip 18.31.0.0/24 score-interface eth* 50
```

Protocol selection

`on proto ( tcp | udp )` *commands* ...

Executes *commands* if the protocol of the connection being examined is TCP or UDP, respectively.

A.2 Chaining

`on` commands can be chained and nested, e.g.,

```
on proto tcp { on remote-ip 18.31.0.0/24
                  on port ssh  { migrate always }
                  on port http { migrate never }
             }
```

In this example the `migrate always` command applies only to SSH (port 22) connections to the specified subnet.

Since the script is evaluated from top to bottom, each command takes precedence over commands preceding it. More general information should therefore be specified closer to the top of the file and more specific information closer to the bottom. (For an example, see `score-interface`, below, where the *eth1* interface has a different score than the rest of the *eth* interfaces.)

Appendix B

Application Session Continuations

This appendix includes the source code of the session continuation generation code used to produce the session continuations measured in Chapter 7. We include code for our modifications to both the Washington University FTP server and the OpenSSH server. In each instance, we provide the complete listing for our continuation generation code, and selected portions of the existing code that were modified to support our continuations. All modifications are included within `#ifdef`/`#endif` clauses that require the `MIGRATE` conditional compilation flag. In neither case are the all of changes presented. The complete set of patches are available for download on the Web at `http://nms.lcs.mit.edu/migrate`.

B.1 FTPd

```
/*
 * Migrate Session Continuation for wu-ftpd
 *
 * Alex C. Snoeren <snoeren@lcs.mit.edu>
 *
 * Copyright (c) 2002 Massachusetts Institute of Technology.
 *
 * This software is being provided by the copyright holders under the GNU
 * General Public License, either version 2 or, at your discretion, any later
 * version. For more information, see the 'COPYING' file in the source
 * distribution.
 *
 * $Id: ftpd-migrate.c,v 1.1 2002/11/04 06:44:47 snoeren Exp $
 */

#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pwd.h>
#include <syslog.h>
#include <assert.h>

#include "config.h"
#include "proto.h"

#include <tesla/tesla.h>
#include "ftpd-migrate.h"
```

```
static char * __argvz = NULL;
static char * __cwd = NULL;

/* Session state */
extern int logged_in;
extern int askpasswd;
extern int anonymous;
extern int guest;
extern int type;
extern struct passwd *pw;
extern char *home;
extern int use_accessfile;

/* Transfer session state */
int FTPD_transfer_resume = 0;
char * FTPD_transfer_name = NULL;
FILE * FTPD_transfer_fin = NULL;
FILE * FTPD_transfer_dout = NULL;
off_t FTPD_transfer_bsize = 0;
char * FTPD_transfer_buf = NULL;

/* Internal variables */
migrate_session *FTPD_session = NULL;
char * FTPD_chrootpath = NULL;
int FTPD_suspend_now = 0;
jmp_buf FTPD_jmpbuf;
int FTPD_jmpbuf_live = 0;

void
FTPD_cleanup(void)
{
  migrate_session_close(FTPD_session);
}

void
mig_saveargv(char *argv)
{
  __argvz = strdup(argv);
  __cwd = getcwd(NULL, 0);
}

migrate_continuation *
FTPD_handler(migrate_session *session, int flags)
{
  /* Ignore instant movement */
  if (flags & M_INSTANT)
    return NULL;

  /* Remember we've been asked to suspend */
  FTPD_suspend_now = 1;

  /* If we're in a blocking syscall, abort it */
  if (FTPD_jmpbuf_live) {
    longjmp(FTPD_jmpbuf, flags);
  }

  /* Otherwise, don't do anything just yet, wait until we've settled */
  return NULL;
```

```
}

void
FTPD_suspend_input(char *s, char *cs)
{
  if (s != cs) {
    *cs++ = '\0';
    MIGRATE_STRSAVE(FTPD_session, cs);
  }

  syslog(LOG_INFO, "Suspending while awaiting input: %s",
         (s != cs) ? s : "[none yet]");

  FTPD_suspend();
}


void
FTPD_suspend_transfer(char *name, FILE *instr, FILE *outstr,
                      off_t FTPD_transfer_bsize, char *buf)
{
  int FTPD_transfer_ifd = fileno(instr);
  int FTPD_transfer_ofd = fileno(outstr);
  static fd_set cfds;

  syslog(LOG_INFO, "Suspending while transferring: %s, %ld buffered",
         (name ? name : ""), (buf ? FTPD_transfer_bsize : 0));

  if (name)
    migrate_store(FTPD_session, "FTPD_transfer_name", name, (strlen(name)+1));
  MIGRATE_SAVE(FTPD_session, FTPD_transfer_ifd);
  MIGRATE_SAVE(FTPD_session, FTPD_transfer_ofd);
  MIGRATE_SAVE(FTPD_session, FTPD_transfer_bsize);
  if (buf)
    migrate_store(FTPD_session, "FTPD_transfer_buf", buf, FTPD_transfer_bsize);

  FTPD_suspend();
}


void
FTPD_suspend(void)
{
  char * const argv[4] = {__argvz, "-L", "-v", NULL};
  migrate_continuation cont;

  MIGRATE_SAVE(FTPD_session, logged_in);
  MIGRATE_SAVE(FTPD_session, askpasswd);
  MIGRATE_SAVE(FTPD_session, anonymous);
  MIGRATE_SAVE(FTPD_session, guest);
  MIGRATE_SAVE(FTPD_session, type);

  if(logged_in) {
    char * cwd = getcwd(NULL, 0);
    assert(pw);
    MIGRATE_STRSAVE(FTPD_session, pw->pw_name);
    MIGRATE_STRSAVE(FTPD_session, pw->pw_passwd);
    MIGRATE_SAVE(FTPD_session, pw->pw_uid);
    MIGRATE_SAVE(FTPD_session, pw->pw_gid);
```

```
    MIGRATE_STRSAVE(FTPD_session, pw->pw_gecos);
    MIGRATE_STRSAVE(FTPD_session, pw->pw_dir);
    MIGRATE_STRSAVE(FTPD_session, pw->pw_shell);
    MIGRATE_STRSAVE(FTPD_session, home);
    MIGRATE_STRSAVE(FTPD_session, cwd);
    MIGRATE_STRSAVE(FTPD_session, FTPD_chrootpath);
    free(cwd);
  }

  /* Return complete continuation & exit */
  memset(&cont, 0, sizeof(cont));
  cont.cont = FTPD_restore;
  cont.flags = M_COMPLETE;
  migrate_return_cont(FTPD_session, &cont, argv, NULL, __cwd);

  /* Not reached */
  exit(0);
}

void
FTPD_restore(migrate_session *session)
{
  int FTPD_transfer_ifd = -1;
  int FTPD_transfer_ofd = -1;
  char * cs = NULL;

  MIGRATE_RESTORE(session, logged_in);
  MIGRATE_RESTORE(session, askpasswd);
  MIGRATE_RESTORE(session, anonymous);
  MIGRATE_RESTORE(session, guest);
  MIGRATE_RESTORE(session, type);

  if(logged_in) {

    char * cwd;
    char * buf;
    size_t buflen;

    pw = malloc(sizeof(struct passwd));
        pw->pw_name = MIGRATE_STRRESTORE(session, pw->pw_name);
    pw->pw_passwd = MIGRATE_STRRESTORE(session, pw->pw_passwd);
    MIGRATE_RESTORE(session, pw->pw_uid);
    MIGRATE_RESTORE(session, pw->pw_gid);
    pw->pw_gecos = MIGRATE_STRRESTORE(session, pw->pw_gecos);
    pw->pw_dir = MIGRATE_STRRESTORE(session, pw->pw_dir);
    pw->pw_shell = MIGRATE_STRRESTORE(session, pw->pw_shell);

    home = MIGRATE_STRRESTORE(session, home);

    /* Restore any chroot jail */
    if((FTPD_chrootpath = MIGRATE_STRRESTORE(session, FTPD_chrootpath)))
      chroot(FTPD_chrootpath);

    /* Return to appropriate directory */
    if((cwd = MIGRATE_STRRESTORE(session, cwd)))
     chdir(cwd);

    /* Handle interrupted transfers */
    MIGRATE_RESTORE(session, FTPD_transfer_ifd);
```

```
    MIGRATE_RESTORE(session, FTPD_transfer_ofd);
    MIGRATE_RESTORE(session, FTPD_transfer_bsize);
    FTPD_transfer_name = MIGRATE_STRRESTORE(session, FTPD_transfer_name);
    if(FTPD_transfer_ifd != -1) {
      if(!(FTPD_transfer_fin = fdopen(FTPD_transfer_ifd, "r")))
        syslog(LOG_ERR, "Unable to fdopen fin %d: %m", FTPD_transfer_ifd);
      if(!(FTPD_transfer_dout = fdopen(FTPD_transfer_ofd, "w")))
        syslog(LOG_ERR, "Unable to fdopen ofd %d: %m", FTPD_transfer_ofd);
      FTPD_transfer_resume = 1;
      syslog(LOG_INFO, "Resuming transfer of %s",
             (FTPD_transfer_name ? FTPD_transfer_name : "unknown"));
    }
    if ((buflen = migrate_store_size(session, "FTPD_transfer_buf"))) {
      syslog(LOG_ERR, "Restoring a length %d buffer", buflen);
      buf = malloc(buflen);
      FTPD_transfer_buf = migrate_retrieve(session, "FTPD_transfer_buf", buf);
    }
  }

  /* Check to see if we were in the middle of input */
  if(migrate_store_size(session, "cs")) {
    cs = MIGRATE_STRRESTORE(session, cs);
    syslog(LOG_INFO, "Resuming in the middle of input %s", cs);
  }

  /* Respect access file security if enabled */
  if(use_accessfile)
    acl_setfunctions();

  syslog(LOG_INFO, "Migrate complete continuation for %s", logged_in ?
         pw->pw_name : "[no one yet]");
}
```

B.1.1 ftpcmd.y

```
/****************************************************************************
  Copyright (c) 1999,2000,2001 WU-FTPD Development Group.
  All rights reserved.

  Portions Copyright (c) 1980, 1985, 1988, 1989, 1990, 1991, 1993, 1994
    The Regents of the University of California.
  Portions Copyright (c) 1993, 1994 Washington University in Saint Louis.
  Portions Copyright (c) 1996, 1998 Berkeley Software Design, Inc.
  Portions Copyright (c) 1989 Massachusetts Institute of Technology.
  Portions Copyright (c) 1998 Sendmail, Inc.
  Portions Copyright (c) 1983, 1995, 1996, 1997 Eric P.  Allman.
  Portions Copyright (c) 1997 by Stan Barber.
  Portions Copyright (c) 1997 by Kent Landfield.
  Portions Copyright (c) 1991, 1992, 1993, 1994, 1995, 1996, 1997
    Free Software Foundation, Inc.

  Use and distribution of this software and its source code are governed
  by the terms and conditions of the WU-FTPD Software License ("LICENSE").

  If you did not receive a copy of the license, it may be obtained online
  at http://www.wu-ftpd.org/license.html.
```

```
  $Id: ftpcmd.c,v 1.1 2002/11/04 06:44:47 snoeren Exp $

****************************************************************************/
/*
 * Grammar for FTP commands.
 * See RFC 959.
 */

/*
 * Migrate Session Continuation for wu-ftpd
 *
 * Alex C. Snoeren <snoeren@lcs.mit.edu>
 *
 * Copyright (c) 2002 Massachusetts Institute of Technology.
 *
 * This software is being provided by the copyright holders under the GNU
 * General Public License, either version 2 or, at your discretion, any later
 * version. For more information, see the ‘COPYING’ file in the source
 * distribution.
 *
 */

/*
 * getline - a hacked up version of fgets to ignore TELNET escape codes.
 */
char *wu_getline(char *s, int n, register FILE *iop)
{
    register int c;
    register char *cs;
    char *passtxt = "PASS password“r“n";

    cs = s;
/* tmpline may contain saved command from urgent mode interruption */
    for (c = 0; tmpline[c] != '“0' && --n > 0; ++c) {
        *cs++ = tmpline[c];
        if (tmpline[c] == '“n') {
            *cs++ = '“0';
            if (debug) {
                if (strncasecmp(passtxt, s, 5) == 0)
                    syslog(LOG_DEBUG, "command: %s", passtxt);
                else
                    syslog(LOG_DEBUG, "command: %s", s);
            }
            tmpline[0] = '“0';
            return (s);
        }
        if (c == 0)
            tmpline[0] = '“0';
    }
  retry:
#ifdef MIGRATE
    if(FTPD_suspend_now || setjmp(FTPD_jmpbuf))
      FTPD_suspend_input(s,cs);
    FTPD_jmpbuf_live = 1;
#endif
    while ((c = getc(iop)) != EOF) {
#ifdef MIGRATE
        FTPD_jmpbuf_live = 0;
#endif
```

```
#ifdef TRANSFER_COUNT
        byte_count_total++;
        byte_count_in++;
#endif
        c &= 0377;
        if (c == IAC) {
            if ((c = getc(iop)) != EOF) {
#ifdef TRANSFER_COUNT
                byte_count_total++;
                byte_count_in++;
#endif
                c &= 0377;
                switch (c) {
                case WILL:
                case WONT:
                    c = getc(iop);
#ifdef TRANSFER_COUNT
                    byte_count_total++;
                    byte_count_in++;
#endif
                    printf("%c%c%c", IAC, DONT, 0377 & c);
                    (void) fflush(stdout);
                    continue;
                case DO:
                case DONT:
                    c = getc(iop);
#ifdef TRANSFER_COUNT
                    byte_count_total++;
                    byte_count_in++;
#endif
                    printf("%c%c%c", IAC, WONT, 0377 & c);
                    (void) fflush(stdout);
                    continue;
                case IAC:
                    break;
                default:
                    continue; /* ignore command */
                }
            }
        }
#ifdef MIGRATE
        FTPD_jmpbuf_live = 0;
#endif
        *cs++ = c;
        if (--n <= 0 || c == '\n')
            break;
    }

    if (c == EOF && cs == s) {
        if (ferror(iop) && (errno == EINTR))
            goto retry;
        return (NULL);
    }

    *cs++ = '\0';
    if (debug) {
        if (strncasecmp(passtxt, s, 5) == 0)
            syslog(LOG_DEBUG, "command: %s", passtxt);
        else
```

```
        syslog(LOG_DEBUG, "command: %s", s);
    }
    return (s);
}
```

B.1.2 ftpd.c

```
/****************************************************************************

  Copyright (c) 1999,2000,2001 WU-FTPD Development Group.
  All rights reserved.

  Portions Copyright (c) 1980, 1985, 1988, 1989, 1990, 1991, 1993, 1994
    The Regents of the University of California.
  Portions Copyright (c) 1993, 1994 Washington University in Saint Louis.
  Portions Copyright (c) 1996, 1998 Berkeley Software Design, Inc.
  Portions Copyright (c) 1989 Massachusetts Institute of Technology.
  Portions Copyright (c) 1998 Sendmail, Inc.
  Portions Copyright (c) 1983, 1995, 1996, 1997 Eric P.    Allman.
  Portions Copyright (c) 1997 by Stan Barber.
  Portions Copyright (c) 1997 by Kent Landfield.
  Portions Copyright (c) 1991, 1992, 1993, 1994, 1995, 1996, 1997
    Free Software Foundation, Inc.

  Use and distribution of this software and its source code are governed
  by the terms and conditions of the WU-FTPD Software License ("LICENSE").

  If you did not receive a copy of the license, it may be obtained online
  at http://www.wu-ftpd.org/license.html.

  $Id: ftpd.c,v 1.1 2002/11/04 06:44:47 snoeren Exp $

****************************************************************************/
/* FTP server. */

/*
 * Migrate Session Continuation for wu-ftpd
 *
 * Alex C. Snoeren <snoeren@lcs.mit.edu>
 *
 * Copyright (c) 2002 Massachusetts Institute of Technology.
 *
 * This software is being provided by the copyright holders under the GNU
 * General Public License, either version 2 or, at your discretion, any later
 * version. For more information, see the 'COPYING' file in the source
 * distribution.
 *
 */

int main(int argc, char **argv, char **envp)
{

    (void) setjmp(errcatch);

#ifdef MIGRATE
    if(FTPD_transfer_resume) {
      retrieve(NULL, FTPD_transfer_name);
```

```
      FTPD_transfer_resume = 0;
    }
#endif
    for (;;) {
#ifdef MIGRATE
      if(FTPD_suspend_now)
        FTPD_suspend();
#endif
        (void) yyparse();
    }
    /* NOTREACHED */
}


/* Tranfer the contents of "instr" to "outstr" peer using the appropriate
 * encapsulation of the data subject to Mode, Structure, and Type.
 *
 * NB: Form isn't handled. */

int
#ifdef THROUGHPUT
    send_data(char *name, FILE *instr, FILE *outstr, off_t blksize)
#else
     send_data(FILE *instr, FILE *outstr, off_t blksize)
#endif
{
    register int c, cnt = 0;
    static char *buf;
    int netfd, filefd;
#ifdef THROUGHPUT
    int bps;
    double bpsmult;
    time_t t1, t2;
#endif

#ifdef THROUGHPUT
    throughput_calc(name, &bps, &bpsmult);
#endif

    buf = NULL;
    if (wu_setjmp(urgcatch)) {
        draconian_FILE = NULL;
        alarm(0);
        transflag = 0;
        if (buf)
            (void) free(buf);
        retrieve_is_data = 1;
        return (0);
    }
    transflag++;
    switch (type) {

    case TYPE_A:
        draconian_FILE = outstr;
        (void) signal(SIGALRM, draconian_alarm_signal);
        alarm(timeout_data);
#ifdef MIGRATE
        while (!FTPD_suspend_now && !setjmp(FTPD_jmpbuf) &&
               ((FTPD_jmpbuf_live = 1)) &&
```

```
            (draconian_FILE != NULL) && ((c = getc(instr)) != EOF)) {
          FTPD_jmpbuf_live = 0;
#else
        while ((draconian_FILE != NULL) && ((c = getc(instr)) != EOF)) {
#endif
            if (++byte_count % 4096 == 0) {
                (void) signal(SIGALRM, draconian_alarm_signal);
                alarm(timeout_data);
            }
            if (c == '\n') {
                if (ferror(outstr))
                    goto data_err;
                (void) putc('\r', outstr);
#ifdef TRANSFER_COUNT
                if (retrieve_is_data) {
                    data_count_total++;
                    data_count_out++;
                }
                byte_count_total++;
                byte_count_out++;
#endif
            }
#ifdef MIGRATE
            /* XXX: need to trap signals here too */
#endif
            (void) putc(c, outstr);
#ifdef TRANSFER_COUNT
            if (retrieve_is_data) {
                data_count_total++;
                data_count_out++;
            }
            byte_count_total++;
            byte_count_out++;
#endif
        }
#ifdef MIGRATE
        if(FTPD_suspend_now) {
          syslog(LOG_INFO, "Wrote %d bytes so far", byte_count);
#ifdef THROUGHPUT
          FTPD_suspend_transfer(name, instr, outstr, 1, NULL);
#else
          FTPD_suspend_transfer(NULL, instr, outstr, 1, NULL);
#endif
        }
#endif
        if (draconian_FILE != NULL) {
            (void) signal(SIGALRM, draconian_alarm_signal);
            alarm(timeout_data);
            fflush(outstr);
        }
        if (draconian_FILE != NULL) {
            (void) signal(SIGALRM, draconian_alarm_signal);
            alarm(timeout_data);
            socket_flush_wait(outstr);
        }
        transflag = 0;
        if (ferror(instr))
            goto file_err;
        if ((draconian_FILE == NULL) || ferror(outstr))
```

```
            goto data_err;
        draconian_FILE = NULL;
        alarm(0);
        reply(226, "Transfer complete.");
#ifdef TRANSFER_COUNT
        if (retrieve_is_data) {
            file_count_total++;
            file_count_out++;
        }
        xfer_count_total++;
        xfer_count_out++;
#endif
        retrieve_is_data = 1;
        return (1);

    case TYPE_I:
    case TYPE_L:
#ifdef THROUGHPUT
        if (bps != -1)
            blksize = bps;
#endif
        if ((buf = (char *) malloc(blksize)) == NULL) {
            transflag = 0;
            perror_reply(451, "Local resource failure: malloc");
            retrieve_is_data = 1;
            return (0);
        }
        netfd = fileno(outstr);
        filefd = fileno(instr);
        draconian_FILE = outstr;
        (void) signal(SIGALRM, draconian_alarm_signal);
        alarm(timeout_data);
#ifdef THROUGHPUT
        if (bps != -1)
            t1 = time(NULL);
#endif
#ifdef MIGRATE
        while (!FTPD_suspend_now && !setjmp(FTPD_jmpbuf) &&
               ((FTPD_jmpbuf_live = 1)) && (!(cnt = 0)) &&
               (draconian_FILE != NULL) &&
               ((cnt = read(filefd, buf, blksize)) > 0)) {
          if(!FTPD_suspend_now && !setjmp(FTPD_jmpbuf))
            if(write(netfd, buf, cnt) != cnt) {
              FTPD_jmpbuf_live = 0;
              break;
            } else {
              FTPD_jmpbuf_live = 0;
            }
          else {
            break;
          }
#else
        while ((draconian_FILE != NULL) &&
               ((cnt = read(filefd, buf, blksize)) > 0 &&
                write(netfd, buf, cnt) == cnt)) {
#endif
            (void) signal(SIGALRM, draconian_alarm_signal);
            alarm(timeout_data);
            byte_count += cnt;
```

```
#ifdef TRANSFER_COUNT
        if (retrieve_is_data) {
#ifdef RATIO
            if( freefile ) {
                total_free_dl += cnt;
            }
#endif /* RATIO */
            data_count_total += cnt;
            data_count_out += cnt;
        }
        byte_count_total += cnt;
        byte_count_out += cnt;
#endif
#ifdef THROUGHPUT
        if (bps != -1) {
            t2 = time(NULL);
            if (t2 == t1)
                sleep(1);
            t1 = time(NULL);
        }
#endif
    }
#ifdef MIGRATE
    if(FTPD_suspend_now) {
      syslog(LOG_INFO, "Wrote %d(%d) bytes so far", byte_count,
             ftell(instr));
#ifdef THROUGHPUT
      FTPD_suspend_transfer(name, instr, outstr, blksize,
                            (cnt ? buf : NULL));
#else
      FTPD_suspend_transfer(NULL, instr, outstr, blksize,
                            (cnt ? buf : NULL));
#endif
    }
#endif
#ifdef THROUGHPUT
    if (bps != -1)
        throughput_adjust(name);
#endif
    transflag = 0;
    (void) free(buf);
    if (draconian_FILE != NULL) {
        (void) signal(SIGALRM, draconian_alarm_signal);
        alarm(timeout_data);
        socket_flush_wait(outstr);
    }
    if (cnt != 0) {
        if (cnt < 0)
            goto file_err;
        goto data_err;
    }
    if (draconian_FILE == NULL)
        goto data_err;
    draconian_FILE = NULL;
    alarm(0);
    reply(226, "Transfer complete.");
#ifdef TRANSFER_COUNT
    if (retrieve_is_data) {
        file_count_total++;
```

```
            file_count_out++;
        }
        xfer_count_total++;
        xfer_count_out++;
#endif
        retrieve_is_data = 1;
        return (1);
    default:
        transflag = 0;
        reply(550, "Unimplemented TYPE %d in send_data", type);
        retrieve_is_data = 1;
        return (0);
    }

  data_err:
    draconian_FILE = NULL;
    alarm(0);
    transflag = 0;
    perror_reply(426, "Data connection");
    retrieve_is_data = 1;
    return (0);

  file_err:
    draconian_FILE = NULL;
    alarm(0);
    transflag = 0;
    perror_reply(551, "Error on input file");
    retrieve_is_data = 1;
    return (0);
}

void retrieve(char *cmd, char *name)
{
    FILE *fin = NULL, *dout;
    struct stat st, junk;
    int (*closefunc) () = NULL;
    int options = 0;
    int ThisRetrieveIsData = retrieve_is_data;
    time_t start_time = time(NULL);
    char *logname;
    char namebuf[MAXPATHLEN];
    char fnbuf[MAXPATHLEN];
    int TransferComplete = 0;
    struct convert *cptr;
    char realname[MAXPATHLEN];
    int stat_ret = -1;

    extern int checknoretrieve(char *);

#ifdef MIGRATE
    if (FTPD_transfer_resume) {
      syslog(LOG_INFO, "Resuming download at pos %ld",
             ftell(FTPD_transfer_fin));
      if(FTPD_transfer_buf) {
        syslog(LOG_INFO, "Stuffing %ld first", FTPD_transfer_bsize);
        write(fileno(FTPD_transfer_dout), FTPD_transfer_buf,
              FTPD_transfer_bsize);
        free(FTPD_transfer_buf);
        FTPD_transfer_buf = 0;
```

```
    }
#ifdef THROUGHPUT
    TransferComplete = send_data(FTPD_transfer_name, FTPD_transfer_fin,
                                FTPD_transfer_dout, FTPD_transfer_bsize);
#else
    TransferComplete = send_data(FTPD_transfer_fin, FTPD_transfer_dout,
                                FTPD_transfer_bsize);
#endif
    (void) fclose(FTPD_transfer_dout);
    goto logresults;
  }
#endif
```

B.2 SSHd

```
/*
 * Migrate Continuation Handling
 *
 * Alex C. Snoeren <snoeren@lcs.mit.edu>
 *
 * Copyright (c) 2001-2 Massachusetts Institute of Technology.
 *
 * This software is being provided by the copyright holders under the GNU
 * General Public License, either version 2 or, at your discretion, any later
 * version. For more information, see the 'COPYING' file in the source
 * distribution.
 *
 */

#include <stdio.h>
#include <stdlib.h>

#include "ssh.h"
#include "xmalloc.h"
#include "packet.h"
#include "buffer.h"
#include "log.h"
#include "servconf.h"
#include "sshpty.h"
#include "channels.h"
#include "compat.h"
#include "ssh1.h"
#include "ssh2.h"
#include "auth.h"
#include "session.h"
#include "dispatch.h"
#include "auth-options.h"
#include "serverloop.h"

#include "ssh-migrate.h"

extern char **environ;
extern char **saved_argv;

char * migrate_cont = NULL;
int migrate_bs_valid = 0;
sigjmp_buf migrate_bs;
```

```c
char migrate_cwd[255];
int migrate_ssh_sock = -1;

/* Check to see if we should suspend now */
int
migrate_suspend_now(void)
{
  return (!migrate_cont && migrate_ssh_session &&
          (migrate_ssh_session->state == MIGRATE_FROZEN));
}

/* Handle mobility updates */
migrate_continuation *
SSH_handler(migrate_session *session, int flags)
{
  /* Ignore instant movement */
  if(flags & M_INSTANT)
    return NULL;

  /* Break out of the select if we were in it */
  if(migrate_bs_valid)
    siglongjmp(migrate_bs, 1);

  /* We want to finish what we can before generating the continuation */
  return NULL;
}

/* Generate a session continuation */
void
SSH_gen_cont(void)
{
  static migrate_continuation cont;

  /* Pickle the state of the encryption and compression buffers */
  migrate_packet_store(migrate_ssh_session);
x
  /* Return complete continuation & exit */
  memset(&cont, 0, sizeof(cont));
  cont.flags = M_COMPLETE;
  cont.cont = SSH_restart;
  migrate_return_cont(migrate_ssh_session, &cont, saved_argv, environ,
                      migrate_cwd);

  /* Not reached */
  exit(0);
}

void
SSH_restart(migrate_session * migrate_ssh_session)
{
  int fdin_arg, fdout_arg, fderr_arg;
  int sock_in;
  pid_t pid;

  MIGRATE_RESTORE(migrate_ssh_session, fdin_arg);
  MIGRATE_RESTORE(migrate_ssh_session, fdout_arg);
  MIGRATE_RESTORE(migrate_ssh_session, fderr_arg);
  MIGRATE_RESTORE(migrate_ssh_session, pid);
```

```
  MIGRATE_RESTORE(migrate_ssh_session, sock_in);

  /* Revive the encryption and compression engines */
  migrate_packet_restore(migrate_ssh_session, sock_in);

  /* We're all set up, jump back into the server loop */
  server_loop(pid, fdin_arg, fdout_arg, fderr_arg);

  /* The following is the cleanup code from sshd.c since server loop returns here */

  /* The connection has been terminated. */
  verbose("Closing connection to %.100s", migrate_ssh_session->dname);

  migrate_session_close(migrate_ssh_session);
  packet_close();
  exit(0);
}
```

B.2.1 serverloop.c

```
/*
 * Author: Tatu Ylonen <ylo@cs.hut.fi>
 * Copyright (c) 1995 Tatu Ylonen <ylo@cs.hut.fi>, Espoo, Finland
 *                    All rights reserved
 * Server main loop for handling the interactive session.
 *
 * As far as I am concerned, the code I have written for this software
 * can be used freely for any purpose.   Any derived versions of this
 * software must be clearly marked as such, and if the derived work is
 * incompatible with the protocol description in the RFC file, it must be
 * called by a name other than "ssh" or "Secure Shell".
 *
 * SSH2 support by Markus Friedl.
 * Copyright (c) 2000, 2001 Markus Friedl.   All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 * IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 * THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#include "includes.h"
```

```
RCSID("$OpenBSD: serverloop.c,v 1.83 2001/11/09 18:59:23 markus Exp $");

/*
 * Migrate SSHd Continuation Handling
 *
 * Alex C. Snoeren <snoeren@lcs.mit.edu>
 *
 * Copyright (c) 2001-2 Massachusetts Institute of Technology.
 *
 * This software is being provided by the copyright holders under the GNU
 * General Public License, either version 2 or, at your discretion, any later
 * version. For more information, see the 'COPYING' file in the source
 * distribution.
 *
 */

/*
 * Performs the interactive session.    This handles data transmission between
 * the client and the program.    Note that the notion of stdin, stdout, and
 * stderr in this function is sort of reversed: this function writes to
 * stdin (of the child program), and reads from stdout and stderr (of the
 * child program).
 */
void
server_loop(pid_t pid, int fdin_arg, int fdout_arg, int fderr_arg)
{
        fd_set *readset = NULL, *writeset = NULL;
        int max_fd = 0, nalloc = 0;
        int wait_status;    /* Status returned by wait(). */
        pid_t wait_pid;     /* pid returned by wait(). */
        int waiting_termination = 0; /* Have displayed waiting close message. */
        u_int max_time_milliseconds;
        u_int previous_stdout_buffer_bytes;
        u_int stdout_buffer_bytes;
        int type;

        debug("Entering interactive session.");

#ifdef MIGRATE
        MIGRATE_SAVE(migrate_ssh_session, pid);
        MIGRATE_SAVE(migrate_ssh_session, fdin_arg);
        MIGRATE_SAVE(migrate_ssh_session, fdout_arg);
        MIGRATE_SAVE(migrate_ssh_session, fderr_arg);
#endif

        /* Initialize the SIGCHLD kludge. */
        child_terminated = 0;
        mysignal(SIGCHLD, sigchld_handler);

        /* Initialize our global variables. */
        fdin = fdin_arg;
        fdout = fdout_arg;
        fderr = fderr_arg;

        /* nonblocking IO */
        set_nonblock(fdin);
        set_nonblock(fdout);
        /* we don't have stderr for interactive terminal sessions, see below */
        if (fderr != -1)
```

```
		set_nonblock(fderr);

	if (!(datafellows & SSH_BUG_IGNOREMSG) && isatty(fdin))
		fdin_is_tty = 1;

	connection_in = packet_get_connection_in();
	connection_out = packet_get_connection_out();

	previous_stdout_buffer_bytes = 0;

	/* Set approximate I/O buffer size. */
	if (packet_is_interactive())
		buffer_high = 4096;
	else
		buffer_high = 64 * 1024;

#if 0
	/* Initialize max_fd to the maximum of the known file descriptors. */
	max_fd = MAX(connection_in, connection_out);
	max_fd = MAX(max_fd, fdin);
	max_fd = MAX(max_fd, fdout);
	if (fderr != -1)
		max_fd = MAX(max_fd, fderr);
#endif

	/* Initialize Initialize buffers. */
	buffer_init(&stdin_buffer);
	buffer_init(&stdout_buffer);
	buffer_init(&stderr_buffer);

	/*
	 * If we have no separate fderr (which is the case when we have a pty
	 * - there we cannot make difference between data sent to stdout and
	 * stderr), indicate that we have seen an EOF from stderr.	This way
	 * we don't need to check the descriptor everywhere.
	 */
	if (fderr == -1)
		fderr_eof = 1;

	server_init_dispatch();

	/* Main loop of the server for the interactive session mode. */
	for (;;) {

		/* Process buffered packets from the client. */
		process_buffered_input_packets();

		/*
		 * If we have received eof, and there is no more pending
		 * input data, cause a real eof by closing fdin.
		 */
		if (stdin_eof && fdin != -1 && buffer_len(&stdin_buffer) == 0) {
#ifdef USE_PIPES
			close(fdin);
#else
			if (fdin != fdout)
				close(fdin);
			else
				shutdown(fdin, SHUT_WR); /* We will no longer send. */
```

```
#endif
                        fdin = -1;
                }
                /* Make packets from buffered stderr data to send to the client. */
                make_packets_from_stderr_data();

                /*
                 * Make packets from buffered stdout data to send to the
                 * client. If there is very little to send, this arranges to
                 * not send them now, but to wait a short while to see if we
                 * are getting more data. This is necessary, as some systems
                 * wake up readers from a pty after each separate character.
                 */
                max_time_milliseconds = 0;
                stdout_buffer_bytes = buffer_len(&stdout_buffer);
                if (stdout_buffer_bytes != 0 && stdout_buffer_bytes < 256 &&
                    stdout_buffer_bytes != previous_stdout_buffer_bytes) {
                        /* try again after a while */
                        max_time_milliseconds = 10;
                } else {
                        /* Send it now. */
                        make_packets_from_stdout_data();
                }
                previous_stdout_buffer_bytes = buffer_len(&stdout_buffer);

                /* Send channel data to the client. */
                if (packet_not_very_much_data_to_write())
                        channel_output_poll();

                /*
                 * Bail out of the loop if the program has closed its output
                 * descriptors, and we have no more data to send to the
                 * client, and there is no pending buffered data.
                 */
                if (fdout_eof && fderr_eof && !packet_have_data_to_write() &&
                    buffer_len(&stdout_buffer) == 0 && buffer_len(&stderr_buffer) == 0) {
                        if (!channel_still_open())
                                break;
                        if (!waiting_termination) {
                                const char *s = "Waiting for forwarded connections to terminate...\r\n";
                                char *cp;
                                waiting_termination = 1;
                                buffer_append(&stderr_buffer, s, strlen(s));

                                /* Display list of open channels. */
                                cp = channel_open_message();
                                buffer_append(&stderr_buffer, cp, strlen(cp));
                                xfree(cp);
                        }
                }
                max_fd = MAX(connection_in, connection_out);
                max_fd = MAX(max_fd, fdin);
                max_fd = MAX(max_fd, fdout);
                max_fd = MAX(max_fd, fderr);

#ifdef MIGRATE
                if(sigsetjmp(migrate_bs, 1) || migrate_suspend_now()) {
                  SSH_gen_cont();
                } else {
```

```
                migrate_bs_valid = 1;
            }
#endif

            /* Sleep in select() until we can do something. */
            wait_until_can_do_something(&readset, &writeset, &max_fd,
                &nalloc, max_time_milliseconds);
#ifdef MIGRATE
            migrate_bs_valid = 0;
#endif

            /* Process any channel events. */
            channel_after_select(readset, writeset);

            /* Process input from the client and from program stdout/stderr. */
            process_input(readset);

            /* Process output to the client and to program stdin. */
            process_output(writeset);
        }
        if (readset)
            xfree(readset);
        if (writeset)
            xfree(writeset);

        /* Cleanup and termination code. */

        /* Wait until all output has been sent to the client. */
        drain_output();

        debug("End of interactive session; stdin %ld, stdout (read %ld, sent %ld), stderr %ld bytes.",
             stdin_bytes, fdout_bytes, stdout_bytes, stderr_bytes);

        /* Free and clear the buffers. */
        buffer_free(&stdin_buffer);
        buffer_free(&stdout_buffer);
        buffer_free(&stderr_buffer);

        /* Close the file descriptors. */
        if (fdout != -1)
            close(fdout);
        fdout = -1;
        fdout_eof = 1;
        if (fderr != -1)
            close(fderr);
        fderr = -1;
        fderr_eof = 1;
        if (fdin != -1)
            close(fdin);
        fdin = -1;

        channel_free_all();

        /* We no longer want our SIGCHLD handler to be called. */
        mysignal(SIGCHLD, SIG_DFL);

#ifndef MIGRATE
        wait_pid = waitpid(-1, &wait_status, child_terminated ? WNOHANG : 0);
#else
```

```
        /* Hack to act like it exited cleanly */
        wait_status = __W_EXITCODE(0, 0);
        wait_pid = pid;
#endif
        if (wait_pid == -1)
                packet_disconnect("wait: %.100s", strerror(errno));
        else if (wait_pid != pid)
                error("Strange, wait returned pid %d, expected %d",
                    wait_pid, pid);
        /* Check if it exited normally. */
        if (WIFEXITED(wait_status)) {
                /* Yes, normal exit.  Get exit status and send it to the client. */
                debug("Command exited with status %d.", WEXITSTATUS(wait_status));
                packet_start(SSH_SMSG_EXITSTATUS);
                packet_put_int(WEXITSTATUS(wait_status));
                packet_send();
                packet_write_wait();

                /*
                 * Wait for exit confirmation.  Note that there might be
                 * other packets coming before it; however, the program has
                 * already died so we just ignore them.  The client is
                 * supposed to respond with the confirmation when it receives
                 * the exit status.
                 */
                do {
                        int plen;
                        type = packet_read(&plen);
                }
                while (type != SSH_CMSG_EXIT_CONFIRMATION);

                debug("Received exit confirmation.");
                return;
        }
        /* Check if the program terminated due to a signal. */
        if (WIFSIGNALED(wait_status))
                packet_disconnect("Command terminated on signal %d.",
                                  WTERMSIG(wait_status));

        /* Some weird exit cause.  Just exit. */
        packet_disconnect("wait returned status %04x.", wait_status);
        /* NOTREACHED */
}
```

B.2.2 packet.c

```
/*
 * Author: Tatu Ylonen <ylocs.hut.fi>
 * Copyright (c) 1995 Tatu Ylonen <ylocs.hut.fi>, Espoo, Finland
 *                    All rights reserved
 * This file contains code implementing the packet protocol and communication
 * with the other side.  This same code is used both on client and server side.
 *
 * As far as I am concerned, the code I have written for this software
 * can be used freely for any purpose.  Any derived versions of this
 * software must be clearly marked as such, and if the derived work is
 * incompatible with the protocol description in the RFC file, it must be
```

```
 * called by a name other than "ssh" or "Secure Shell".
 *
 *
 * SSH2 packet format added by Markus Friedl.
 * Copyright (c) 2000, 2001 Markus Friedl.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
 * IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 * THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#include "includes.h"
RCSID("$OpenBSD: packet.c,v 1.72 2001/11/10 13:37:20 markus Exp $");

/*
 * Migrate SSHd Continuation Handling
 *
 * Alex C. Snoeren <snoeren@lcs.mit.edu>
 *
 * Copyright (c) 2001-2 Massachusetts Institute of Technology.
 *
 * This software is being provided by the copyright holders under the GNU
 * General Public License, either version 2 or, at your discretion, any later
 * version. For more information, see the 'COPYING' file in the source
 * distribution.
 *
 */

/*
 * Causes any further packets to be encrypted using the given key.  The same
 * key is used for both sending and reception.  However, both directions are
 * encrypted independently of each other.
 */
void
packet_set_encryption_key(const u_char *key, u_int keylen,
    int number)
{
        Cipher *cipher = cipher_by_number(number);
        if (cipher == NULL)
                fatal("packet˙set˙encryption˙key: unknown cipher number %d", number);
        if (keylen < 20)
                fatal("packet˙set˙encryption˙key: keylen too small: %d", keylen);
        cipher_init(&receive_context, cipher, key, keylen, NULL, 0);
```

```
       cipher_init(&send_context, cipher, key, keylen, NULL, 0);
#ifdef MIGRATE
       /* We need to record this data for later */
       if(migrate_ssh_session) {
         migrate_store(migrate_ssh_session, "key", (char *)key, keylen);
         MIGRATE_SAVE(migrate_ssh_session, number);
         MIGRATE_SAVE(migrate_ssh_session, keylen);
       }
#endif
}

/* Pickle packet engine state */
void
migrate_packet_store(migrate_session *s)
{
  MIGRATE_SAVE(s, remote_protocol_flags);
  MIGRATE_SAVE(s, packet_compression);
  MIGRATE_SAVE(s, max_packet_size);
  MIGRATE_SAVE(s, interactive_mode);
  MIGRATE_SAVE(s, extra_pad);

  /* We always fully process these before trying to suspend */
  assert(!buffer_len(&outgoing_packet));
  assert(!buffer_len(&incoming_packet));

  MIGRATE_SAVE(s, input);
  migrate_store(s, "input_buf", input.buf, input.alloc);
  MIGRATE_SAVE(s, output);
  migrate_store(s, "output_buf", output.buf, output.alloc);

  /* Save crypto stuff */
  MIGRATE_SAVE(s, send_context);
  MIGRATE_SAVE(s, receive_context);

  /* We don't handle compression in this version */
  assert(!packet_compression);
}

/* Restore packet engine state */
void
migrate_packet_restore(migrate_session *s, int fd)
{
  u_char *key;
  u_int keylen;
  int number;
  Cipher * storecipher;

  packet_set_connection(fd, fd);

  MIGRATE_RESTORE(s, remote_protocol_flags);
  MIGRATE_RESTORE(s, packet_compression);
  MIGRATE_RESTORE(s, max_packet_size);
  MIGRATE_RESTORE(s, interactive_mode);
  MIGRATE_RESTORE(s, extra_pad);

  MIGRATE_RESTORE(s, input);
  input.buf = malloc(input.alloc);
  migrate_retrieve(s, "input_buf", input.buf);
  MIGRATE_RESTORE(s, output);
```

```
  output.buf = malloc(output.alloc);
  migrate_retrieve(s, "output_buf", output.buf);

  /* Restore crypto stuff */
  MIGRATE_RESTORE(s, number);
  MIGRATE_RESTORE(s, keylen);
  key = malloc(keylen);
  migrate_retrieve(s, "key", key);
  packet_set_encryption_key(key, keylen, number);

  storecipher = send_context.cipher;
  MIGRATE_RESTORE(s, send_context);
  send_context.cipher = storecipher;

  storecipher = receive_context.cipher;
  MIGRATE_RESTORE(s, receive_context);
  receive_context.cipher = storecipher;

}
```

Glossary

address In a network, the name of a *network attachment point*. In the Internet, addresses are specified by IP addresses (*e.g.*, 18.31.0.100).

bind (v.) To map a specified name to a value. In the context of network addressing, an *end-point* binding is a mapping between an *end-point* name and a network *address*.

binding update The act of altering the mapping between a name and its value.

connection A communications channel between two transport-layer *end points* used to send and receive packets. The transport-layer *end points* are *bound* to *network attachment points* at connection establishment.

connection migration The act of modifying one of a connection's *end-point bindings*. In the context of mobile networking, this typically has the effect of moving one end of the connection to a new *network attachment point*.

correspondent end points With respect to a particular *end point*, its correspondent end points are the set of other *end points* currently engaged in communication with it.

end point An application, service, protocol, or other computational agent that uses the network to transfer data to other end points.

inconsistent binding A *binding* from name to value that has become stale due to a change in the value.

multi-homed An *end point* is said to be multi-homed when it is associated with more than one *network attachment point*, each with its own *address*.

naming system A service that stores and *resolves bindings* between names and their values. A dynamic naming system additionally allows *binding updates* to alter the name to value mappings. In the context of the Internet, a naming system typically resolves names to IP *addresses*.

network attachment point The place where the network accepts data from or delivers data to an *end point*. Each network attachment point has an *address* which is typically unique within the network.

network interface A device that physically connects an *end point* to a network. A network interface provides one or more logical *network attachment points*.

resolve (v.) To look up or compute a *binding* between a specified name and its value.

session An association between two *end points* that maintains coordinated state. Sessions often consist of multiple *connections*.

session continuation A *session continuation* is a function from the current *end point* and network conditions to a context sufficient for control to be returned to an *end point* that effectively continues the session from where it was suspended.

socket A transport *connection end point*. Sockets are *bound* to a particular *network attachment point* which they use to transmit and receive data.

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, South Carolina, December 1999.

[2] Akamai Technologies, Inc. Turbo-charging dynamic web sites with Akamai Edge-Suite. `http://www.akamai.com/en/resources/pdf/Turbocharging_WP.pdf`, December 2001.

[3] American National Standards Institute. Public key cryptography for the financial service industry: The elliptic curve digital signature algorithm. ANSI X9.62 - 1998, January 1999.

[4] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert T. Morris. Resilient overlay networks. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 131–145, Lake Luise, Canada, October 2001.

[5] Ross J. Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11):32–40, November 1994.

[6] Apache HTTP Server Version 1.3 Documentation. Descriptors and Apache. `http://httpd.apache.org/docs/misc/descriptors.html`.

[7] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan. An integrated congestion management architecture for Internet hosts. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 175–187, Cambridge, Massachusetts, August 1999.

[8] Hari Balakrishnan, Srinivasan Seshan, and Randy H. Katz. Improving reliable transport and handoff performance in cellular wireless networks. *ACM Wireless Networks*, 1(4):469–481, December 1995.

[9] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proc. International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.

[10] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, Louisiana, February 1999.

[11] Stephen Bellovin. Defending against sequence number attacks. RFC 1948, Internet Engineering Task Force, May 1996.

[12] Dan J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`, 1997.

[13] Pravin Bhagwat, Charles Perkins, and Satish K. Tripathi. Network layer mobility: an architecture and survey. *IEEE Personal Communications*, 3(3):54–64, June 1996.

[14] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[15] Rámon Cáceres and Liviu Iftode. Improving the performance of reliable transport protocols in mobile computing environments. *IEEE Journal on Selected Areas in Communications*, 13(5):850–857, June 1995.

[16] Andrew T. Campbell, Javier Gomez, Sanghyo Kim, Chieh-Yih Wan, Zoltan R. Turanyi, and Andras G. Valkó. Comparison of IP micromobility protocols. *IEEE Wireless Communications*, 9(1):2–12, February 2002.

[17] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[18] Isidro Castiñeyra, Noel Chiappa, and Martha Steenstrup. The Nimrod routing architecture. RFC 1992, Internet Engineering Task Force, August 1996.

[19] Vinton G. Cerf and Edward Cain. The DoD Internet architecture model. *Computer Networks*, 7:307–318, October 1983.

[20] CERT. TCP SYN flooding and IP spoofing attacks. CERT advisory CA-1996-21, September 1996.

[21] Stuart Cheshire and Mary Baker. Internet mobility 4x4. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 318–329, Stanford, California, August 1996.

[22] Columbitech, Inc. Columbitech WVPN technical description. `http://www.columbitech.com/documents/ColumbitechWVPNTechnicalDescription.pdf`, 2002.

[23] Mike Dahlin, Bharat Chandra, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, and Anil Sewani. Using mobile extensions to support disconnected services. CS-TR-00-20, UT Austin, April 2000.

[24] Leslie Daigle. IAB considerations for unilateral self-address fixing (UNSAF) across network address translation. Internet Draft, Internet Engineering Task Force, June 2002. `draft-iab-unsaf-considerations-02.txt` (work in progress).

[25] Stephen E. Deering. Host extensions for IP multicasting. RFC 1122, Internet Engineering Task Force, August 1989.

[26] Stephen E. Deering and Robert M. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460, Internet Engineering Task Force, December 1998.

[27] Tim Dierks and Christopher Allen. The TLS protocol. RFC 2246, Internet Engineering Task Force, January 1998.

[28] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-11:644–654, November 1976.

[29] Frederick Douglis. *Transparent Process Migration in the Sprite Operating System*. PhD thesis, University of California at Berkeley, September 1990.

[30] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, California, October 1991.

[31] Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, Internet Engineering Task Force, March 1997.

[32] Donald E. Eastlake, 3rd. Domain name system security extensions. RFC 2535, Internet Engineering Task Force, March 1999.

[33] Federal Communications Commission. Spectrum study of 2500–2690 MHz band: The potential for accommodating third generation mobile systems. ET Docket No. 00-258, March 2001.

[34] Paul Ferguson and Daniel Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. RFC 2267, Internet Engineering Task Force, January 1998.

[35] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol — HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[36] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullman. Interface and execution models in the Fluke kernel. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 101–115, New Orleans, Louisiana, February 1999.

[37] Daichi Funato, Kinuko Yasuda, and Hideyuki Tokuda. TCP-R: TCP mobility support for continuous operation. In *Proc. IEEE International Conference on Network Protocols*, pages 229–236, Atlanta, Georgia, October 1997.

[38] Robert Grimm, Janet Davis, Eric Lemar, and Brian Bershad. Migration for pervasive applications. `http://www.cs.nyu.edu/rgrimm/papers/one.world.pdf`, 2002.

[39] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Programming for pervasive computing environments. `http://www.cs.nyu.edu/rgrimm/papers/migration02.pdf`, 2002.

[40] Mark Gritter and David Cheriton. An architecture for content routing support in the Internet. In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems*, pages 37–48, San Francisco, California, March 2001.

[41] Sumit Gupta and A. L. Narasimha Reddy. A client oriented, IP level redirection mechanism. In *Proc. IEEE Infocom*, pages 1461–1469, New York, New York, March 1999.

[42] Neil M. Haller. The S/KEY one-time password system. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 151–157, 1994.

[43] Ahmed Helmy. A multicast-based protocol for IP mobility support. In *Proc. 2nd International Workshop on Networked Group Communication*, pages 49–58, Palo Alto, California, November 2000.

[44] Christian Huitema. Multi-homed TCP. Internet Draft, Internet Engineering Task Force, May 1995. (expired).

[45] Jon Inouye, Jim Binkley, and Jonathan Walpole. Dynamic network reconfiguration support for mobile computers. In *Proc. 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 13–22, Budapest, Hungary, September 1997.

[46] Institute of Electronic and Electrical Engineers (IEEE). Wireless medium access control (MAC) and physical layer (PHY) specifications. Standard 802.11, 1999.

[47] Institute of Electronic and Electrical Engineers (IEEE). Carrier sense multiple access with collision detection (CSMA/CD) access method and physical specifications. Standard 802.3, 2002.

[48] International Organization for Standardization (ISO). Information processing systems – Open Systems Interconnection – basic connection oriented session service definition. ISO 8326, August 1987.

[49] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire, Jr. IP-based protocols for mobile internetworking. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 235–245, Zurich, Switzerland, September 1991.

[50] Van Jacobson, Robert Braden, and David Borman. TCP extensions for high performance. RFC 1323, Internet Engineering Task Force, May 1992.

[51] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.

[52] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert T. Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Transactions on Networking*, 10(5), October 2002.

[53] M. Frans Kaashoek, Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. FLIP: An internetwork protocol for supporting distributed systems. *ACM Transactions on Computer Systems*, 11(1):77–106, February 1993.

[54] Scott F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation*. PhD thesis, University of Texas at Austin, December 1999.

[55] Stephen Kent and Randall Atkinson. Security architecture for the Internet protocol. RFC 2401, Internet Engineering Task Force, November 1998.

[56] Anne-Marie Kermarrec, Paul Couderc, and Michel Banâtre. Introducing contextual objects in an adaptive framework for wide-area global computing. In *Proc. ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[57] Eddie Kohler, Mark Handley, Sally Floyd, and Jitendra Padhye. Datagram congestion control protocol (DCCP). Internet Draft, Internet Engineering Task Force, June 2002. `draft-kohler-dcp-04.txt` (work in progress).

[58] David M. Kristol and Lou Montulli. HTTP state management mechanism. RFC 2109, Internet Engineering Task Force, February 1997.

[59] Leslie Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[60] Björn Landfeldt, Tomas Larsson, Yuri Ismailov, and Aruna Seneviratne. SLM, a framework for session layer mobility management. In *Proc. IEEE International Conference on Computer Communications and Networks*, pages 452–456, Natick, Massachusetts, October 1999.

[61] Marcus Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS protocol version 5. RFC 1928, Internet Engineering Task Force, March 1996.

[62] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. `http://www.cryptosavvy.com`, November 1999.

[63] Michael Litzkow, Miron Livny, and Matt Mutka. Condor — A hunter of idle workstations. In *Proc. 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, California, June 1988.

[64] David Maltz and Pravin Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proc. IEEE Infocom*, pages 1037–1045, San Francisco, California, March 1998.

[65] Petros Maniatis, Mema Roussopoulos, Edward Swierk, Kevin Lai, Guido Appenzeller, Xinhua Zhao, and Mary Baker. The mobile people architecture. *ACM Mobile Computing and Communications Review (MC2R)*, 3(3):36–42, July 1999.

[66] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP selective acknowledgment options. RFC 2018, Internet Engineering Task Force, October 1996.

[67] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, Reading, Massachusetts, April 1996.

[68] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 123–133, Stanford, California, August 1988.

[69] Gabriel E. Montenegro. Reverse tunneling for mobile IP, revised. RFC 3024, Internet Engineering Task Force, January 2001.

[70] Robert T. Morris. A weakness in the 4.2BSD UNIX TCP/IP software. Computing science technical report 117, AT&T Bell Laboratories, Murray Hill, New Jersey, February 1985.

[71] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 153–167, Seattle, Washington, October 1996.

[72] Robert Moskowitz. Host identity payload. Internet Draft, Internet Engineering Task Force, February 2001. `draft-moskowitz-hip-arch-02.txt` (expired).

[73] Robert Moskowitz. Host identity payload and protocol. Internet Draft, Internet Engineering Task Force, October 2001. `draft-ietf-moskowitz-hip-05.txt` (expired).

[74] Lily Mummert, Maria Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proc. 15th ACM Symposium on Operating Systems Principles*, pages 143–155, Copper Mountain, Colorado, December 1995.

[75] Mike Muuss. The story of the TTCP program. `http://http://ftp.arl.mil/~mike/ttcp.html`.

[76] Jayanth Mysore and Vaduvur Bharghavan. A new multicasting-based architecture for Internet host mobility. In *Proc. 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 161–172, Budapest, Hungary, September 1997.

[77] John Nagle. Congestion control in IP/TCP internetworks. RFC 896, Internet Engineering Task Force, January 1984.

[78] National Institute of Standards and Technology. The secure hash algorithm (SHA-1). NIST FIPS PUB 180-1, U.S. Department of Commerce, April 1995.

[79] NetMotion Wireless, Inc. A standards-based breakthrough for wireless connectivity. `http://www.netmotionwireless.com/assets/netmotion_standrds.pdf`, August 2001.

[80] T. S. Eugene Ng, Ion Stoica, and Hui Zhang. A waypoint service approach to connect heterogeneous Internet address spaces. In *Proc. USENIX Technical Conference*, pages 319–332, Boston, Massachusetts, June 2001.

[81] Brian Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, October 1997.

[82] Tadashi Okoshi, Masahiro Mochizuki, Yoshito Tobe, and Hideyuki Tokuda. MobileSocket: Toward continuous operation for Java applications. In *Proc. IEEE International Conference on Computer Communications and Networks*, pages 50–57, Natick, Massachusetts, October 1999.

[83] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 361–376, Boston, Massachusetts, December 2002.

[84] John Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, Reading, Massachusetts, 1994.

[85] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proc. 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, California, October 1998.

[86] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-lite; a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

[87] Vern Paxson and Mark Allman. Computing TCP's retransmission timer. RFC 2988, Internet Engineering Task Force, November 2000.

[88] Charles E. Perkins. IP encapsulation within IP. RFC 2003, Internet Engineering Task Force, October 1996.

[89] Charles E. Perkins. IP mobility support for IPv4. RFC 3220, Internet Engineering Task Force, January 2002.

[90] Charles E. Perkins and Pat R. Calhoun. Mobile IPv4 challenge/response extensions. Internet Draft, Internet Engineering Task Force, May 2002. `draft-ietf-mobileip-optim-11.txt` (work in progress).

[91] Charles E. Perkins and David B. Johnson. Route optimization in mobile IP. Internet Draft, Internet Engineering Task Force, September 2001. `draft-ietf-mobileip-optim-11.txt` (work in progress).

[92] John M. Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, July 1978.

[93] Gerald J. Popek and Bruce J. Walker. *The LOCUS Distributed System Architecture*. Computer Systems Series. MIT Press, Cambridge, Massachusetts, 1985.

[94] J. Postel and J. Reynolds. File transfer protocol (FTP). Technical report, Internet Engineering Task Force, October 1985. RFC 959.

[95] Jon Postel. User datagram protocol. RFC 768, Internet Engineering Task Force, August 1980.

[96] Jon Postel. Internet Protocol. RFC 791, Internet Engineering Task Force, September 1981.

[97] Jon Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[98] Jon Postel and Joyce K. Reynolds. TELNET protocol specification. RFC 854, Internet Engineering Task Force, May 1983.

[99] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. In *Proc. IASTED International Conference on Software Engineering*, San Francisco, California, November 1997.

[100] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. TR-CS-97-09, The Australian National University, April 1997.

[101] Michael O. Rabin. Fingerprinting by random polynomials. TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[102] Ram Ramanathan. Nimrod mobility support. RFC 2103, Internet Engineering Task Force, February 1997.

[103] Yakov Rekhter, Robert G. Moskowitz, Daniel Karrenberg Geert Jan de Groot, and Eliot Lear. Address allocation for private internets. RFC 1918, Internet Engineering Task Force, February 1996.

[104] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation: An International Journal*, 6:233–247, 1993.

[105] Maximilan Riegel and Michael Tuexen. Mobile SCTP. Internet Draft, Internet Engineering Task Force, February 2002. `draft-riegel-tuexen-mobile-sctp-00.txt` (work in progress).

[106] Ronald R. Rivest. The MD5 message-digest algorithm. RFC 1321, Internet Engineering Task Force, April 1992.

[107] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, January 1997.

[108] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Caramillo, Alan Jonston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler. SIP: Session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[109] Jerome H. Saltzer. On the naming and binding of network destinations. RFC 1498, Internet Engineering Task Force, August 1993.

[110] Jon Salz, Alex C. Snoeren, and Hari Balakrishnan. TESLA: A transparent, extensible session-layer framework for end-to-end network services. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, Washington, March 2003. To appear.

[111] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 1–7, Philadelphia, Pennsylvania, May 1996.

[112] Robert Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[113] Henning Schulzrinne. Personal mobility for multimedia services in the Internet. In *Proc. European Workshop on Interactive Distributed Multimedia Systems and Services*, pages 143–161, Berlin, Germany, March 1996.

[114] Henning Schulzrinne, Steve Casner, Ron Frederick, and Van Jacobson. RTP: A transport protocol for real-time applications. RFC 1889, Internet Engineering Task Force, January 1996.

[115] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared passive network performance discovery. In *Proc. 1st USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[116] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proc. 8th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 160–171, Atlanta, Georgia, September 2002.

[117] William A. Simpson. The point-to-point protocol (PPP). RFC 1661, Internet Engineering Task Force, July 1994.

[118] Alex C. Snoeren. Adaptive inverse multiplexing for wide-area wireless networks. In *Proc. IEEE Conference on Global Communications*, volume 2, pages 1665–1672, Rio de Janiero, Brazil, December 1999.

[119] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-grained failover using connection migration. In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems*, pages 221–232, San Francisco, California, March 2001.

[120] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 155–166, Boston, Massachusetts, August 2000.

[121] Raj Srinivasan. Binding protocols for ONC RPC version 2. RFC 1833, Internet Engineering Task Force, August 1995.

[122] Pyda Srisuresh and Kjeld B. Egevang. Traditional IP network address translator (traditional NAT). RFC 3022, Internet Engineering Task Force, January 2001.

[123] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, Reading, Massachusetts, 1994.

[124] W. Richard Stevens. TCP tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. RFC 2001, Internet Engineering Task Force, January 1997.

[125] Randall R. Stewart, Michael A. Ramalho, Qiaobing Xie, Michael Tuexen, Ian Rytina, Maria-Carmen Belinchon, and Phil Conrad. Stream control transmission protocol (SCTP) dynamic address reconfiguration. Internet Draft, Internet Engineering Task Force, May 2002. `draft-ietf-tsvwg-addip-sctp-05.txt` (work in progress).

[126] Randall R. Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hanns J. Schwarzbauer, Tom Taylor, Ian Rytina, Malleswar Kalla, Lixia Zhang, and Vern Paxson. Stream control transmission protocol. RFC 2960, Internet Engineering Task Force, October 2000.

[127] Ion Stoica, Robert T. Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, San Diego, California, August 2001.

[128] Gong Su and Jason Nieh. Mobile communications with virtual network address translation. CUCS-003-02, Columbia University, February 2002.

[129] Florin Sultan, Kiran Srinivasan, Deepa Iyer, and Liviu Iftode. Migratory TCP: Highly available Internet services using connection migration. In *Proc. 22nd International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[130] Sun Microsystems, Inc. Java servlet 2.3 specification. Final release, September 2001.

[131] Diane Tang and Mary Baker. Analysis of a local-area wireless network. In *Proc. 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 1–10, Boston, Massachusetts, August 2000.

[132] Fumio Teraoka, Yasuhiko Yokore, and Mario Tokoro. A network architecture providing host migration transparency. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 209–220, Zurich, Switzerland, September 1991.

[133] Dave Thaler and Christian E. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, Internet Engineering Task Force, November 2000.

[134] Marvin Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proc. ACM Symposium on Operating Systems Principles*, pages 2–12, Orcas Island, Washington, December 1985.

[135] Susan Thomson and Thomas Narten. IPv6 stateless address autoconfiguration. RFC 2462, Internet Engineering Task Force, December 1998.

[136] Sameer Tilak and Nael B. Abu-Ghazaleh. A concurrent migration extension to an end-to-end host mobility architecture. *Mobile Computing and Communications Review*, 5(3):26–31, July 2001.

[137] Universal plug and play. `http://www.upnp.org/`.

[138] Amin Vahdat, Michael Dahlin, Tom Anderson, and Amit Aggarwal. Active names: Flexible location and transport of wide-area resources. In *Proc. 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, Colorado, October 1999.

[139] Faramak Vakil, Ashutosh Dutta, Jyh-Cheng Chen, Shinichi Baba, Nobuyasu Nakajima, Yasuro Shobatake, and Henning Schulzrinne. Mobility management in a SIP environment. Internet Draft, Internet Engineering Task Force, December 2000. `draft-itsumo-sip-mobility-req-02.txt` (expired).

[140] Paul Vixie, Susan Thomson, Yakov Rekhter, and Jim Bound. Dynamic updates in the domain name system (DNS UPDATE). RFC 2136, Internet Engineering Task Force, April 1997.

[141] Brian Wellington. Secure domain name system (DNS) dynamic update. RFC 3007, Internet Engineering Task Force, November 2000.

[142] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, Massachusetts, December 2002.

[143] Praveen Yalagandula, Amit Garg, Mike Dahlin, Lorenzo Alvisi, and Harrick Vin. Transparent mobility with minimal infrastructure. CS-TR-01-30, UT Austin, August 2001.

[144] Tatu Ylonen. SSH — secure login connections over the Internet. In *Proc. 6th USENIX Security Symposium*, pages 37–42, San Jose, California, July 1996.

[145] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proc. 8th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 95–106, Atlanta, Georgia, September 2002.

[146] Bruce Zenel and Dan Duchamp. A general purpose proxy filtering mechanism applied to the mobile environment. In *Proc. 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 248–259, Budapest, Hungary, September 1997.

[147] Yongguang Zhang and Son Dao. A "persistent connection" model for mobile and distributed systems. In *Proc. IEEE International Conference on Computer Communications and Networks*, pages 300–307, Las Vegas, Nevada, September 1995.

[148] Xinhua Zhao, Claude Castelluccia, and Mary Baker. Flexible network support for mobile hosts. *ACM Mobile Networks and Application Journal*, 6(2):137–149, April 2001.

[149] Hans Zimmerman. OSI reference model — the ISO model of architecture for open system intereconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.