APPLICATION MOBILITY

BY

VICTOR CHARLES ZANDY

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin—Madison

2004

© Copyright by Victor C. Zandy 2004
All Rights Reserved

APPLICATION MOBILITY

VICTOR CHARLES ZANDY

UNDER THE SUPERVISION OF PROFESSOR BARTON P. MILLER

AT THE UNIVERSITY OF WISCONSIN — MADISON

Application mobility is the ability for an application to travel with its user, moving between computers or moving with a computer between locations. We describe new techniques that enable unmodified applications to move independently of each other without requiring changes to infrastructure or actions by administrators. The techniques are based on three new abstractions that overcome the obstacles to application mobility unresolved by previous work in process migration and mobile computing.

First, we introduce *reliable network connections*, an abstraction that automatically detects network connection failures caused by movement, and that recovers from these failures transparently. We introduce an *enhancement detection protocol* that enables the use of this abstraction in environments where not all applications support it. This protocol is a general-purpose solution to the problem of safely detecting, at user-level, the presence of remote support for any type of network communication enhancement.

Second, we introduce the abstraction of a *window session*, a transportable representation of the state of a graphical user interface, that allows the user interface of a running application to be moved, either with or independently of the application process, from one display to

another. This abstraction is sufficiently general to support the additional functionality of replicating user interfaces across multiple displays.

Third, we allow applications to access files through the abstraction of a name space that appears static, but which changes its file access strategies as the application moves. We developed the *flac* language to specify such name spaces. Flac enables the concise description of file access strategies in terms of services that provide access to file name spaces, and name space composition operators that capture mobile file access semantics. These same descriptions can be used to prescribe the run-time behavior of an application's file name space by handing them to a system that implements the semantics of flac.

We have implemented systems that demonstrate that these abstractions can be built entirely from user-level code with no application modifications and with generally imperceptible overhead, and can support the mobility of ordinary programs across administrative boundaries and over extended periods of disconnection.

Acknowledgments

It is a great pleasure to thank the people who helped me with my graduate career.

Bart Miller, my thesis advisor, is precisely the sort of teacher, critic, and editor that I wanted and needed. He patiently taught me about ideas, writing, presenting, reading, hacking, and the power of incremental progress. I was very, very lucky to have worked so closely with him.

Miron Livny probed the boundaries of my ideas at every stage, exposing dimensions I had not seen and still have not adequately explored. He has planted seeds for years of future work.

Marv Solomon generated insightful questions and comments at every talk I gave on my work, offered unprovoked encouragement and affirmation, and carefully reviewed this dissertation with his amazing eye for intellectual, technical, and stylistic quality.

Paul Barford provided invaluable early technical review of the network component of this work, thoughtful feedback on the final result, and years of insight into the mind of the network researcher and general career guidance.

Vadim Shapiro, my first advisor at UW, gave me advice about research and life that despite all appearances I could not stop repeating to myself. He also introduced me to many beautiful ideas: the quad-edge is still my favorite data structure.

Andrea Arpaci-Dusseau lent thoughtful review to the early stages of the work, and offered encouragement throughout.

My committee members exemplify the deep commitment of the entire faculty and staff of UW Computer Sciences Department to providing a comfortable and resourceful intellectual environment for graduate students.

Phil Roth, my friend and colleague, provided emotional, intellectual, and logistical support without which I could not have succeeded.

Other friends, particularly Jeremy Zauderer, Maurice Zauderer, Andrew Prock, and Graham Betts, repeatedly gave emotional support at crucial moments.

Alex Ryskin led me astray from physics, gave me access to computers and problems to work on, debugged and critiqued my terrible programs, and otherwise started all this trouble.

A parade of shrinks and awesome psychopharmacology preserved mental hull integrity.

Anna Zandy, my sister, has the unusual burden of being a younger sibling idolized and admired by her older brother. Her patience and tolerance for the travails of education and daily renewal of her enthusiasm for learning inspire me through my darkest struggles. Also, I profoundly depend on her to explain professional sports to me.

Janet and William Zandy, my parents, have supported my intellectual development above all other considerations, providing infinite love, understanding, and access to an immaculately clean bedroom.

Fran and Bill Twiddy, Meg's parents, have opened their hearts and embraced me with an unbelievable level of kindness.

Meg Twiddy, whom I love intensely and wildly, was my backbone. I would have collapsed without her.

Contents

Abstract . . . . . **i**
Acknowledgements . . . . . **iii**
Contents . . . . . **v**
Figures . . . . . **viii**
1Introduction **1**
1.1 Reliable Network Connections 3
1.2 Checkpoints of User Interfaces 5
1.3 Mobile File Access 6
2 Related Work **9**
2.1 Process Migration 10
2.2 Network Connection Mobility 12
2.3 User Interface Mobility 14
2.4 File I/O 16
2.4.1 File Systems 16
2.4.2 Name space operations 17
2.5 Desktop Mobility 18
3 Reliable Network Connections **21**
3.1 Network Connection Failure Model 24
3.1.1 TCP Review 24
3.1.2 Events Leading to Aborts 26
3.2 Detecting Socket Enhancements 26
3.3 Reliable Sockets 30
3.3.1 Rocks Overview 30
3.3.2 Experience 35
3.4 Reliable Packets 38
3.5 Security 43
3.6 Process Checkpointing 44

3.7 UDP 46
3.8 Performance 47
3.8.1 Throughput and Latency 48
3.8.2 Connection 50
3.8.3 Reconnection 51
3.9 Portability 51
4 User Interfaces 57
4.1 System Overview 60
4.1.1 Initialization 61
4.1.2 GUI Migration 62
4.1.3 GUI Replication 64
4.1.4 GUI+Process Migration 64
4.2 Implementation 65
4.2.1 Hijacking the Application 65
4.2.2 Finding the Window Server 67
4.2.3 Synchronizing Communication 69
4.2.4 Retrieving and Regenerating GUI Resources 70
4.2.5 Maintaining Transparency 72
4.3 Evaluation 74
4.3.1 Detach and Re-attach Latency 74
4.3.2 Interactive Overhead 75
4.4 Security 76
5 File I/O 79
5.1 The Language 82
5.1.1 Services and Specification 82
5.1.2 Path operations 83
5.1.3 Overlay combinator 84
5.1.4 Select Combinator 85
5.1.5 Transfer Combinators 88
5.2 Additional Examples 90

5.2.1 Replica selection 91
5.2.2 Importing Environments 92
5.2.3 Copy-on-write unions 94
5.3 Flac Run-Time Implementation 95
5.3.1 Tracking Context 97
5.3.2 Responding to Location Changes 98
5.3.3 Transfer Combinator 99
5.3.4 Process Migration 100
5.3.5 Intercepting I/O operations 101
5.4 Evaluation 103
6 Conclusion 108
6.1 Contributions 108
6.2 Perspectives 110
References 113

Figures

3.1 An established TCP connection. . . . 24
3.2 The enhancement detection protocol. . . . 28
3.4 The reliable socket state diagram 31
3.3 The reliable sockets architecture. . . . 31
3.5 The reliable packets architecture. . . . 40
3.6 Average rocks and racks throughput and latency. . . . 49
3.7 Average TCP connection establishment time. . . . 50
3.8 Windows Sockets API (Version 2) calls. . . . 53
4.1 The elements of a GUI-based application. . . . 58
4.2 Initializing the application process. . . . 61
4.3 Detaching a GUI from Desktop A. . . . 62
4.4 Re-attaching a GUI to Desktop B. . . . 63
4.5 Replicating a GUI on Desktop B. . . . 64
4.6 Migrating an application process and its GUI from a laptop to a desktop computer. . . . 66
4.7 Average detach and re-attach latency. . . . 75
4.8 Breakdown of detach latency for Netscape. . . . 75
4.9 Average round trip time for a minimal X protocol request and reply. . . . 76
5.1 File access methods change with application context. . . . 80
5.2 Flac abstractions. . . . 81
5.3 Path operators. . . . 84
5.4 Identity relationship of overlay and path operators. . . . 86
5.5 Two ways to test software. . . . 93
5.6 Testing on several operating system versions. . . . 93
5.7 Collecting operating system versions. . . . 94
5.8 Architecture of prototype flac run-time. . . . 96
5.9 Environment of our flac experience. . . . 104
5.10 Flac specification for Emacs. . . . 105

Chapter 1

Introduction

Application mobility is the ability for an application to travel with its user, moving between computers or moving with a computer between locations. Application mobility combines issues in *process migration* with issues in *mobile computing*. The issues in process migration center around the problem of moving a running program off one machine and resuming its execution on another. The issues in mobile computing center around the problem of providing users who move with their computers uninterrupted access to their work from locations where resources are not available or must be accessed by different means. This dissertation unifies the concepts common to these two areas. It describes new techniques that enable unmodified applications to move independently of each other across machines and locations without requiring changes to the infrastructure or actions by the administrators.

These techniques contribute to the area of process migration by expanding the set of program state that can be migrated and the contexts in which migration can occur. The state includes network connections, user interfaces, and access to file systems; we enable these resources to be migrated for programs running on unmodified operating system and network

infrastructure. These techniques contribute to the area of mobile computing by enabling interactive applications to tolerate disconnection and often remain available to their users during periods of disconnection, and to automatically respond to changing resource access requirements.

We draw these areas together by identifying three common goals for supporting application mobility:

- **Transparency to applications.** It should be possible to move any existing application without preparations such as re-programming or re-linking the application code with mobile-aware functionality. Such transparency increases the opportunity for users to benefit from mobility functionality — they are not restricted to certain applications or encumbered with program modifications.
- **Independence of movement.** The unit of mobility should be individual applications in execution. This level of mobility gives users the freedom to move only the applications that they want or need at their new location, leaving other applications behind or free to move elsewhere. An important benefit of this freedom is the ability to leave behind programs that would be inappropriate to move, such as games on a home machine or a program that accesses proprietary data on an office machine.
- **No modification to infrastructure.** Users should be able to move their applications within and between administrative domains without special cooperation from administrators such as access to modified operating system kernels, special file system configurations, or proxy services. A solution to this goal can enable application mobility to be deployed entirely at the convenience of its users.

The key challenges in enabling application mobility constrained by these goals is the ability to capture and migrate the state of application resources that are external to the process abstraction. These resources include network connections, user interfaces, and file systems.

No previous work addressing the movement of applications that use these resources has achieved all of these goals.

The contribution of this dissertation is a collection of new techniques for moving these three types of resources. The rest of this chapter previews these techniques.

1.1 Reliable Network Connections

The first part of this dissertation addresses the question of how to move applications that use network connections. Most network communication protocols are not designed to support communication between end points that move or disconnect for extended periods. Rather, these actions cause ordinary network connections to fail, raising errors from which applications that are oblivious to mobility do not recover. Existing protocols that do support mobility, such as Mobile IP [55], require kernel modifications and lack support for extended periods of disconnection and independent application movement.

Our idea is to provide to ordinary applications the abstraction of a network connection that never fails — a *reliable* network connection. This abstraction, from the application's perspective, is indistinguishable from an ordinary network connection. It maintains during movement and disconnection the illusion of being temporarily blocked but otherwise viable, while transparently locating and re-establishing communication with its peer on behalf of the application. In addition, it *interoperates* with applications that use conventional network connections, silently reverting to ordinary network connection functionality when it detects the absence of compatible functionality in its peer.

The challenges in designing a *user-level* reliable network connection include finding effective, efficient, and application-transparent mechanisms to detect connection failures,

preserve in-flight data and other communication state, and locate and re-connect with lost peers. In addition, interoperability requires a technique to detect peers that support reliable network connections that is not hostile to peers that do not.

We have defined two new user-level implementation models for reliable network connections, and implemented systems, *reliable sockets* (*rocks*) and *reliable packets* (*racks*), to validate and evaluate them. The models differ in how they intercept network communication; each represents a different set of trade-offs between implementation complexity and performance. We also have designed a new *enhancement detection protocol*, a safe, user-level, general-purpose protocol that can be used by any system with the need to remotely detect the presence of enhanced network connection functionality.

Reliable network connections and the rocks and racks implementations make the following contributions to network connection mobility:

- Failure detection: They automatically detect network connection failures, including those caused by link failures, extended periods of disconnection, change of IP address, and process migration, within seconds of their occurrence.
- Automatic recovery: They automatically recover failed connections without loss of in-flight data, even when one end (it does not matter which one) changes its IP address. For the rare cases in which both ends move at the same time, rocks and racks provide a callback interface for a third-party location service, one of several value-added interfaces provided to rocks- or racks-aware applications by the *rocks-expanded API* (RE-API).
- Interoperability: When an application using either rocks or racks establishes a new connection, it safely probes its remote peer for the presence of a rocks- or racks-enabled socket, and falls back to ordinary socket functionality if neither is present.
- User-level implementation: Users can use rocks and racks without re-compiling or re-linking existing binaries and without making kernel modifications, and rocks can be installed by unprivileged users.

1.2 Checkpoints of User Interfaces

The second part of this dissertation addresses the question of how to move the graphical user interface (GUI) of a running program. No existing window system that we know of has functionality to detach a user interface from one display and move it to another. The interface to the window system provides only operations to create, destroy, and manipulate the individual resources of an application's user interface.

Our idea is to define an abstraction, called a *window session*, that represents precisely the user interface state of an individual application. The ability to manipulate window sessions engenders operations for detaching, re-attaching, and replicating an application's user interface.

The main challenge is to provide access to window sessions without making modifications to the application or window system. Applications interact with window systems using protocols whose messages contain server-dependent information, such as handles for identifying windows and other resources. These messages must be transformed to reflect the new handles and resources after a window session is moved. In addition, window servers lack critical primitive operations, such as the ability to enumerate all of the user interface resources belonging to an application, that are necessary to isolate the window sessions of individual applications.

To validate the usefulness of window sessions for application mobility, we have developed a system, called *guievict*, that introduces the window session abstraction to an existing window system. In addition to showing that window sessions can effectively support user

interface movement for application mobility, our system makes the following contributions to user interface mobility:

- Migration occurs at application granularity: Users can select and move the GUIs of individual applications from their desktop, leaving the GUIs of other applications behind or free to move elsewhere.
- No modification to applications: Any application program, including those based on legacy toolkits, can be migrated without modifications such as re-programming, re-compiling, or re-linking.
- Unpremeditated migration: Users do not need to run their applications in a special way, such as by redirecting their GUI communication through a proxy.
- No modification to window systems: The new functionality is encapsulated in a window server extension, implemented on top of the window server extension API, that is loaded in the server when it is started, and a library that is loaded in the application at run-time.
- Improvement to a widely-used window system: Our implementation is based on the X window system [67], whose architects have long regretted not designing support for user interface movement [27]. We have characterized and implemented the essential functionality, and we have been invited to contribute our extension to the X window server code base.

1.3 Mobile File Access

The third part of this dissertation addresses the question of how to provide an application that is moving with uninterrupted access to its files. Movement may force an application to change its method for accessing a file system. It may also make access to a file system impossible, unreliable, or slow, prompting users to copy their files to another file system.

We would like mobile applications to access files through a name space that automatically and transparently performs these adaptations. We provide access through a variety of existing file systems, rather than building a new file system. Our idea is to create a language for

specifying the semantics of such a name space. This language serves two roles: it enables concise *description* of semantics with which users can easily modify and compare strategies for mobile file access, and it enables unambiguous *prescription* of the semantics to a system that automatically manages file name spaces to applications.

We have designed a new language, called *flac* (*file access*), with abstractions and operators for specifying mobile file access strategies over existing file systems. The main challenges in its design were identifying simple primitive operators that can be composed to express a variety of practical access strategies. We have implemented a prototype interpreter for flac, and used it to validate both the idea that a language can describe file access strategies and prescribe their automatic implementation.

The flac language and our prototype implementation of a flac run-time make the following contributions to file access for mobile applications.

- A descriptive language: Flac can concisely describe the file name space of a mobile application. It provides service abstractions to represent the methods by which an application accesses file systems, composition operators to express strategies for combining and changing services during movement, and operators to bind services to an application's file name space. We demonstrate the descriptive power of the language by specifying and comparing a variety of new and old file name space semantics.
- A prescriptive language: Name space specifications written in flac are executable. Users of mobile applications describe their file access strategies in a single, maintainable flac specification that they hand to a flac translator. This translator, after checking the specification for errors and possibly verifying properties such as file consistency guarantees, generates an implementation of the name space semantics for the user's applications.
- User-level implementation: We have built a prototype run-time interpreter for the flac language that operates entirely at user level. It requires no special network services,

operating system features, or administrative support, and it operates transparently to applications. We show that it can provide uninterrupted access to file systems in multiple administrative domains to unmodified applications that move across different machines and different networks.

The rest of the dissertation is organized as follows. Chapter 2 describes related work, where we show the void in process migration and mobile computing research into which we seat application mobility, and explain how previous approaches to moving network connections, user interfaces, and access to files fall short of our goals. Our contributions are presented over the next three chapters. Each of these technical chapters presents the ideas, implementation, and evaluation of one of our results: reliable network connections (Chapter 3), user interface mobility (Chapter 4), and file access mobility (Chapter 5). They may be read in any order. Chapter 6 shows how these components collectively support our vision for application mobility, reviews contributions, and offers perspectives on the results.

Chapter 2

Related Work

We describe previous work related to application mobility. The overall picture is that previous work in mobility, both process migration and mobile computing, has targeted mobility problems that are different from application mobility. In some cases, particularly process migration, we can borrow some existing techniques to support application mobility. In others, particularly the mobile computing work to support mobile network connections, user interfaces, and file access, the existing techniques are largely incompatible with the issues in application mobility, prompting the need for the new techniques of this dissertation.

We first discuss the foundation of process migration technology upon which we can build application mobility support, highlighting how previous work in process migration omits necessary support for general mobility of resources — network connections, user interfaces, and file access — external to the process abstraction. We then discuss previous efforts to support the mobility of these resources, and show how they fall short, largely because they address different problems. Finally, we review how these other types of mobility problems

most closely related to application mobility, which we characterize as *desktop mobility* problems, have led to weaker functionality than that of application mobility.

2.1 Process Migration

Process migration is the ability to move a program in execution from one machine to another [46]. Process migration functionality is central to application mobility because it enables applications in execution to follow their users to new machines.

Process migration has been studied as an operating system function since the early 1980s. It was a feature in DEMOS/MP [61] (1983), LOCUS [81] (1984), V [80] (1985), and Sprite [19] (1991). These research systems had the advantage that they could implement kernel mechanisms for process migration. These systems were based on network of machines in a common administrative domain, so movement was limited to other machines in the same domain. In some cases, a process that migrated to a new machine had *residual dependencies* on its previous machines, access to which was necessary for the transparency of the movement.

Despite this kernel-level research, process migration is not a feature of commodity operating systems today. Instead, many systems for user-level process migration have been developed, including Condor [40], Libckpt [58], Process Hijacking [88], and others [13,15,74,75]. These systems involve user-level code, linked into the program, that can be invoked during execution to create a snapshot of the state of the running program, called a *checkpoint*. Migration is completed by moving the checkpoint to a new machine and restarting it. In general, most previous user-level process checkpointing systems have provided the ability to checkpoint the state of the basic process abstraction — the memory and register state

of the process — but not the state of additional resources used by the program, including network connections, file access, and user interfaces. The Condor system is an exception. It provides migrating processes with access to the file system of a fixed machine by redirecting I/O requests made by the process to a proxy on the machine. This service is one type of file access mobility that is useful for mobile applications, but mobility scenarios can require other types of file access, depending on the mobile application's context, such as access to files while disconnected from a network. The goal of our file access research is to concisely specify these different types of file acess services and the various situations in which they are used.

Zap [50] is a recently developed process migration system with similar goals to application mobility. Zap presents abstraction of a *pod*, a collection of running programs that are migrated as a unit. The design and functionality of Zap has several differences with our techniques for application mobility. First, Zap is based on a set of kernel modifications to virtualize the system call interface for the applications of a pod, enabling the pod to be moved to a new machine transparently to the applications. Second, the pod abstraction is rigid — it is not possible to move running programs into or out of a pod. Third, Zap has less flexible and general mechanisms for network, file access, and user interface mobility than those we have developed. In particular, Zap uses a packet re-writing system similar to racks and MSOCKS [41] (see Section 2.2) to support network connections, but cannot handle extended periods of disconnection; it supports mobile file access only for pods that move among machines that share the same network file system; and it depends on a desktop mobility system (see Section 2.5) to move user interfaces.

2.2 Network Connection Mobility

Network connection mobility is the ability for applications to move with open network connections. The basic mobility functionality of a system providing this service is to transparently preserve the application-level abstraction of an open connection across a change of network address of the machine on which the application is running. This functionality can be added to the network at the IP level [59], to the implementation of the transport protocol (such as TCP [60]), or to the application's interface to the the network stack (most commonly the sockets API [76]).

Persistent Connections [90] and Mobile TCP [63,64] are user-level systems that interpose a library between application code and the sockets API that preserves the illusion of a single unbroken connection over successive connection instances. Persistent Connections does not preserve data that is in-flight at the time of reconnection, so its functionality is safe to use when both ends of the connection are quiessent and all their previous messages have been received at the remote end. Mobile TCP has a mechanism to preserve in-flight data, but it requires a special kernel interface to access the contents of the TCP send buffer (such interfaces are neither common nor standard) and can fail during extended periods of disconnection. Both of these systems depend on a third party to locate and re-establish contact with a disconnected peer, and neither interoperates safely with ordinary applications.

The MobileSocket [49] system provides a transparent wrapper to the Java sockets API package that is similar in design to Mobile TCP sockets, but with a user-level in-flight data buffer that avoids the limitations of Mobile TCP sockets. However, its buffering strategy can degrade application data flow and, as our implementation experience demonstrates (see Section 3.3), is unnecessarily complicated. In addition, MobileSocket cannot be used by non-

Java applications and MobileSocket-based applications cannot communicate over TCP sockets with applications that do not use MobileSocket.

The TCP Migrate option [69] is an experimental kernel extension to TCP. It introduces a new state to the TCP state machine that an established connection enters when it becomes disconnected and returns from when the connection is re-established. The Migrate option is designed to interoperate with applications running on machines that do not have Migrate option installed, falling back to ordinary TCP behavior. However, it addresses a mobility model less general than that of application mobility. Specifically, it does not support extended periods of disconnection or the migration of an application with an open network connection to a new machine. A mechanism to migrate applications with open network connections was developed to use the Migrate option in a web server reliability system [70], but like other similar kernel modifications [4,52,84,89], it was a research experiment that has not become a standard feature in any commodity operating system.

Descending one level in the network stack to the IP level, Mobile IP [55] routes all IP packets, including those sent by TCP and UDP, between a mobile host and ordinary peers by redirecting the packets through a *home agent*, a proxy on a fixed host with a specialized kernel. Mobile IP is designed for host mobility and is not well-suited for application mobility. Specifically, at the IP level, the home agent does not have a per-connection view of the data it is forwarding, which is necessary to support the independent movement of individual connections and to protect connections against failure during extended periods of disconnection. A Mobile IP home agent *could* overcome these limitations by emulating the TCP state machine — requiring functionality to interpret the transport layer of packets, maintain per-connection associations, and buffer and acknowledge TCP packets on behalf of

disconnected endpoints — but no such extension has been reported. We demonstrate that the same functionality can be achieved at user-level with systems like rocks or racks.

MSOCKS [41] has architectural similarities to both rocks and racks. MSOCKS enables a client application process to establish a mobile connection with an ordinary server by re-directing the connection through a proxy running on a special kernel. The proxy is based on a kernel modification called a *TCP splice* that allows the client, as it moves, to close its end of the connection and establish a new one without affecting the server. Like the racks system, the TCP splice translates the state of the original connection held open by the server to the state of the current connection held by the client. Like the rocks system, it uses an in-flight buffer to preserve data sent from the client to the server. However, unlike our systems, MSOCKS has no automatic failure detection and reconnection and does not support migration of applications with open network connections to new machines.

2.3 User Interface Mobility

Our goal in user interface mobility is to enable the graphical user interface (GUI) of an application to be moved to a new display, either with its associated application process or independently of it.

The xmove system [72] is the only other existing system with similar goals. Xmove enables the GUIs of unmodified X windows [67] applications to be moved to new displays while they are running. It is based on a proxy that emulates the X window server. Instead of connecting to the actual window server, the user re-directs their application to an instance of this proxy when starting their application. The proxy forwards the application's window operations to the actual window server, simultaneously recording the state of windows as they

are created and changed. When the user wishes to move the user interface, they notify the xmove proxy of their intent, and the proxy deletes the windows from the current server and recreates them from the recorded state on the new window server.

The main drawback of xmove is that it was designed to support only the movement of the user interface to a new display, not the migration of both the GUI and application process. The difference is that the xmove proxy must be migrated with the application or left behind as a residual dependency, and the communication channel between the xmove proxy and the application must be preserved. Process migration of GUI-based applications with xmove is *possible:* we built a system that checkpointed the state of the xmove proxy and (using rocks) the communication between the application and the xmove proxy, which successfully extended the scope of xmove's functionality to process migration. However, the resulting system is architecturally cumbersome: there is an extra process to checkpoint and restart on each migration. We show in Chapter 4 that our abstraction of the *window session* can deliver the same functionality more efficiently and cleanly; we also show that it is not necessary to redirect the application to a proxy prior to movement.

Another approach to user interface mobility is to integrate mobility support into the window system libraries from which user interfaces are composed [28]. This functionality has been included in the design of a recent user interface library, GTK [54]. The advantage of this approach is that the application contains its own mobility support, and no external mechanism like the xmove proxy is required to facilitate movement. However, it has the limitation that mobility is available only to those applications built on the library. Legacy applications and applications built with other new libraries (such as KDE [77]) cannot be directly run over GTK; they must be ported.

Systems have been developed to support user interfaces mobility at the granularity of the *desktop* [16,39,50,65,66,68,82], the set of all applications running on a machine. Systems facing similar issues have been developed to *replicate* desktops across multiple machines [1,2,4,6,8,9,25,31,34]. This unit of mobility is coarser than that of application mobility, and different techniques are required to support application mobility. However, our application mobility techniques can be used to reproduce desktop mobility functionality, including replication. We discuss this topic separately in Section 2.5.

2.4 File I/O

We want mobile applications to have uninterrupted access to an apparently unchanging file name space, including across movements to different administrative domains and over periods of disconnection. Our approach is based on a language for describing file access strategies depending on location and other contextual information. No other system has tried this approach before; we compare our functionality to that of file systems that support mobility, and discuss operating system file name space operations that influenced the design of our language.

2.4.1 File Systems

The basic idea behind all file systems that support mobility is to distribute replicas of files to various locations, and to enable applications that are clients of these systems to access files from any available replica. The unit of replication may be either an individual file or a container of files such as a disk volume. These systems are important to application mobility because, when they are available, they provide a type of file system that a mobile application may use for file access. However, these systems do not fully address file access requirements

of mobile applications because they are not available in all administrative domains, and most require administrator assistance to control their policies and configuration.

The Coda file system [37] manages a replica of a subset of a user's files, called a *hoard*, on a mobile computer that the user can access when they are disconnected from the network. Coda requires kernel-level support on the file system client, and files must be served from a Coda file server. Other research file systems, such as that of Locus [81], the Ficus [51] file system, and systems based on Bayou [56], are similar to Coda in that they provide access, using specialized operating systems, to file replicas for mobile program; the main conceptual differences among them are the consistency models under which the replicas are managed.

Although Coda is not widely deployed, similar functionality has been introduced in commodity operating systems. The *Offline Files* feature in Microsoft Windows [45] provides a hoarding system for files from a remote file server. However, this feature can only be used with Windows file servers, and its availability to users depends on policies that are configured by the administrator of the file server. Likewise, recent versions of Apple's OS X [3] provide a file system hoarding feature also similar to Coda. However, this feature only works with the iDisk file system, a premium network file system service controlled by Apple; users of OS X cannot harness its functionality for other file systems.

2.4.2 Name space operations

Our language for expressing file system access strategies is based on file *name space composition* operators. A file name space is the set of files provided by a file system; our language provides operators for manipulating and combining the name spaces of different file systems to provide a unified, unchanging name space to a moving application. While our

language is the first to provide name space operators for mobility, the semantics of these operators were influenced by operators developed for other purposes.

The Plan 9 operating system [57] provides a per-process name space and a set of operations for manipulating the name space. Its name space operators are designed to connect services, represented as file systems, to a process. They include a *union* operator that mounts a set of a file systems to the same point in the name space of the application, allowing the files of multiple file systems to be available under the same sub-tree of the file system hierarchy. This feature that inspired the semantics of a similar operator (`overlay`) in our language. At a higher level, however, Plan 9 lacks name space functionality designed for mobility, such as the ability to transparently switch, in response to mobility events, the file system that serves a portion of an application's name space.

The 3-D file system [38], the Translucent File System [32], and the BSD union mount [53] each provide a name space operator similar to Plan 9's union operator. This operator provides additional semantics for copying files among the file systems comprising a union mount point. The original purpose of these semantics was to support version control for software development. However, the ability to move files between file systems is essential to mobile file access, and our language provides a file transfer operator (`transfer`) that, while significantly more general than this union operator, was inspired by its semantics.

2.5 Desktop Mobility

Mobile access to applications at the granularity of the *desktop*, the set of all applications with which a user interacts at a machine, is a more thoroughly studied alternative to application mobility [16,39,50,65,66,68,82]. Despite the superficial similarity of having the

goal of providing mobile users access to their applications, desktop mobility approaches have several important differences from application mobility. The fundamental difference is that desktop mobility does not allow mobile access to be controlled on a per-application basis: either all applications on a desktop can be accessed, or none. This model precludes mobile users from accessing any of their applications when security policies set by administrators prohibit mobile access to select applications.

The remaining differences depend on how desktop mobility is implemented. There are two basic approaches, with significantly different mobility properties. First, *remote desktop access* systems enable a mobile user to access a single desktop as they move among different computers [50,65,66,68,82]. In these systems, only the point from which the users accesses their desktop changes; the applications running on the desktop do not move with the user. The main limitation of these systems is that they are based on remote access. Disconnected users have no access to their applications. Users of networks with high latency, low bandwith, or a high rate of packet loss can experience a user interface that feels unresponsive.

Second, *desktop migration* systems enable the entire desktop to be migrated with the user [16,39]. The basic approach is to run the operating system and applications in a virtual machine whose state can be checkpointed and moved when the user moves. The main limitation of these approaches is that the amount of state to be moved can be large (on the order of gigabytes), since it includes the memory, processor, and device state of the machine being emulated. This adds significant latency to beginning and end of movement operations as the user waits for the machine state to be checkpointed or restored, and it requires the ability to transport a larger amount of data than most networks or portable storage devices can

support [39]. This overhead is excessive for users who want to move only one or a few applications.

Chapter 3

Reliable Network Connections

An application may move while it has open network connections. Movement can introduce disconnection for an extended period of time, change of IP address, or migration to a different machine, all of which are circumstances under which ordinary network connections fail. We need a way to preserve the state of an application's connections in these circumstances. Such a system should be implemented at user level, so that it can be used by any user. *Reliable sockets* (rocks) and *reliable packets* (racks) are the systems we developed to address this problem. The idea is to provide a transparent layer over the application's network communication interface that provides a *reliable* network connection, one that automatically detects and recovers from failures including those caused by movement.

Rocks and racks are based on enhancements layered over the communication interface on both side of a network connection. This design raises the problem of determining whether a communication peer supports a particular enhancement. We solve this problem with a new *enhancement detection protocol* (EDP). This protocol is based on a carefully chosen sequence of user-level operations performed during the establishment of a connection that should be

harmless to unenhanced peers and whose outcome has a low probability of confusing an unenhanced peer with an enhanced one.

Rocks and racks are distinguished from previous work by several major features:

- Failure detection: They automatically detect network connection failures, including those caused by link failures, extended periods of disconnection, change of IP address, and process migration, within seconds of their occurrence.
- Automatic recovery: They automatically recover failed connections without loss of in-flight data, even when one end (it does not matter which one) changes its IP address. For the rare cases in which both ends move at the same time, rocks and racks provide a callback interface for a third-party location service, one of several value-added interfaces provided to rocks- or racks-aware applications by the rocks-expanded API (RE-API).
- Interoperability: When an application using either rocks or racks establishes a new connection, it safely probes its remote peer for the presence of a rocks- or racks-enabled socket, and falls back to ordinary socket functionality if neither is present. Remote detection of rocks and racks is accomplished by the enhancement detection protocol, which can be used by any type of socket enhancement (not just rocks and racks) and can be implemented at user level.
- User-level implementation: both systems can be used without re-compiling or re-linking existing binaries and without making kernel modifications, and rocks can be installed by unprivileged users.

In addition to mobility, the functionality of rocks and racks can be used for other applications. For example, the ability to transparently replace a network connection could be used by a performance tuning system to change the static parameters of a network connection, such as buffer sizes, that are otherwise impossible to change after a connection has been initialized.

The main difference between rocks and racks is the way that they track communication. Rocks are based on a library that is interposed between the application code and the operating

system. The library exports the sockets API to the application, permitting it to be transparently dropped into ordinary applications, and modifies the behavior of these functions to detect connection failures and mask them from the application. While this implementation model is convenient for user-level process migration, it cannot be used with statically-linked or privileged programs and to be completely transparent to applications it requires several complicated mechanisms not related to mobility. Racks, in contrast, are based on a separate user-level daemon that uses a kernel-level packet filter to redirect the flow of selected packets to the daemon. Racks do not introduce any code into the processes that use them, making them usable with any program and enabling a simpler implementation than that of rocks. The price for this convenience is an approximately 50% increase in latency and 15% decrease in throughput.

This focus of this chapter is on the Linux implementation of rocks and racks. Although the notion of a reliable network connection is general, implementing rocks and racks involves features that vary among platforms, including available mechanisms for tracking communication and the API and I/O models provided for network programming. We also describe the portability issues that emerged during our initial port of rocks from Linux to Microsoft Windows.

The remaining sections of the chapter are as follows. Section 3.1 discusses the TCP failure model as it relates to mobility. Section 3.2 presents the enhanced socket detection protocol. Section 3.3 presents the architecture and functionality of rocks and describes our experience with its implementation model. Section 3.4 presents racks. Section 3.5 discusses the security issues of rocks and racks. Section 3.6 relates rocks and racks to process checkpointing and migration. Section 3.7 discusses our approach to the reliability and mobility of UDP.

Section 3.8 evaluates the performance of the Linux implementation of rocks and racks. Section 3.9 discusses portability.

3.1 Network Connection Failure Model

Rocks and racks extend the reliability of TCP by detecting failures to TCP connections and preventing applications from becoming aware of them. We review the essential background on TCP, then turn to the relationship of its failure modes to mobility events. Although the failure modes discussed here can be derived from the specification of the protocol, this is the first analysis of these failures from the perspective of mobility.

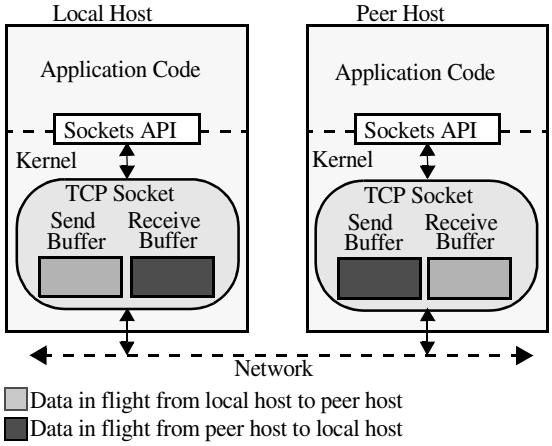

Figure 3.1: An established TCP connection.

3.1.1 TCP Review

TCP provides reliable bi-directional byte stream communication between two processes running on separate hosts called the local host and the peer host (see Figure 3.1). The operating system kernels on each host maintain the state of their end in a TCP socket. A TCP socket is identified by an internet address comprised of an IP address and a port number; a pair of such addresses identifies a TCP connection. Applications manipulate sockets through calls to the sockets API.

The TCP reliability mechanism is based on a pair of buffers in each socket and a scheme for acknowledging and retransmitting data. When the local application process writes to the socket, the local kernel copies the data to the socket's send buffer before transmitting it to the peer. This data remains in the send buffer and is periodically retransmitted until the kernel receives an acknowledgement of its receipt from the peer. When the local kernel receives data from the peer, it copies it to the destination socket's receive buffer and sends back an acknowledgement. The receive buffer holds data until it is consumed by the application process. A process cannot pass more data to TCP than can be stored in the combined space of the local send buffer and the peer receive buffer. This limits the maximum amount of in-flight data, data that has been passed from the process to the kernel on one end but not yet consumed by the process from the kernel on the other end, that can exist at any time over a TCP connection.

TCP connection failures occur when the kernel aborts a connection. Aborts primarily occur when data in the send buffer goes unacknowledged for a period of time that exceeds the limits on retransmission defined by TCP. Other causes for an abort include a request by the application, too many unacknowledged TCP keepalive probes, receipt of a TCP reset packet (such as after the peer reboots), and some types of network failures reported by the IP layer [10]. Once an abort has occurred, the socket becomes invalid to the application.

3.1.2 Events Leading to Aborts

Mobile computer users routinely perform actions that can lead to the abort of TCP connections. These actions include:

- *Disconnection*: A mobile host becomes disconnected when the link becomes unreachable (such as when the user moves out of wireless range), when the link fails (such as when a modem drops a connection), or when the host is suspended.
- *Host changing its IP address*: A host might move to a new physical subnet, requiring a new IP address, or a suspended host might lose its lease on its IP address provided by a service like DHCP [20] and get a new one. This change of IP address will lead to a failure of the TCP connection the next time one endpoint attempts to send data to the other.
- *Process changing its host*: Process migration can cause two types of failures. First, it changes the IP address of the process. Second, unless the process migration mechanism can migrate kernel state, it will separate the process socket descriptor from the underlying kernel socket. The original kernel socket will be closed or aborted while further use of the descriptor by the process will refer to a non-existent socket.
- *Host crashing*: The peer of the crashed host will either reach its retransmission limit while the host is down or, after the host reboots, receive a reset message in response to any packet it sends to the host. Rocks and racks can handle host crashes when then are combined with a process checkpointing mechanism, in which case the failure mode reduces to that of either one of the previous two actions.

3.2 Detecting Socket Enhancements

When establishing a new connection, acting as either a client or a server, rocks and racks must determine whether the remote peer supports either one of these enhancements. Existing techniques for exchanging such information, such as TCP options, are reserved for privileged users and require kernel modifications. The novelty of our enhancement detection protocol is that it provides a user-level technique for exchanging enhancement information that does not have negative effects on unenhanced peers. All non-trivial costs of the protocol are borne by

the client, so it is reasonable to deploy in production servers, and it does not interfere with network infrastructure such as network address translation (NAT) devices [21] or firewalls [22]. The protocol is general purpose: it can be used by other mobility systems such as MSOCKS [41] or mobile TCP sockets [63,64], and it can support systems that enhance sockets with other functionality such as compression, encryption, or quality-of-service. For example, the EDP is an alternative to the common practice of reserving a new port for encrypted versions of legacy network services such as IMAP [18].

It is not obvious how to remotely distinguish an enhanced socket from an ordinary one using only user-level mechanisms. The problem for the enhanced socket is to unmistakably indicate its presence to remote processes that use enhanced sockets without affecting those that do not. The socket enhancement code cannot simply announce its presence when the connection is established, as others have suggested [63,64], since an unenhanced process is likely to misinterpret the announcement. It is also problematic to use a separate connection, because any scheme for creating the second connection could conflict with other processes that do not participate in the scheme. And despite the multitude of socket options, it is not possible to use socket options to construct a distinct socket configuration that could be remotely sensed.

In fact, we believe that it is generally impossible to perform user-level enhancement detection successfully in all cases. Our approach instead was to find a way that is successful given our observation and understanding of how most programs use the sockets API. The EDP is based on an extremely unusual combination of socket operations initiated by the client that is safe to perform on unenhanced servers.

The protocol is in five steps as follows (see Figure 3.2):

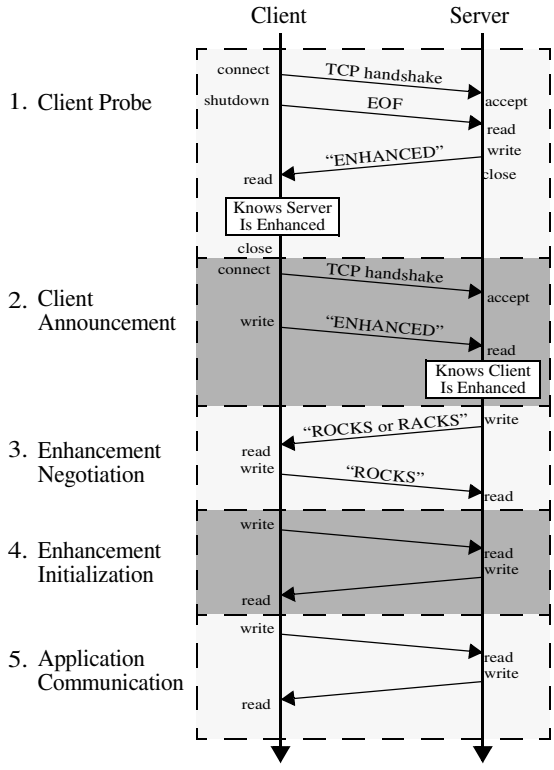

Figure 3.2: The enhancement detection protocol.

1. Client Probe: The client and server perform a four step protocol to establish to the client that the server is enhanced:

   a. The client opens a connection to the server.

   b. The client closes its end of the connection for writing, using the sockets API function `shutdown`.

   c. The server detects the end-of-file and sends an *enhancement announcement* in response that indicates its enhanced condition to the client, then closes the connection.

   d. The client receives the announcement and now knows the server is enhanced. It closes the connection.

2. Client Announcement: The client opens another connection to the server and sends an enhancement announcement. Now both the server and the client are mutually aware of

being enhanced.

3. Enhancement Negotiation: The client and server exchange messages to agree on which enhancements they will use.
4. Enhancement Initialization: The client and server perform enhancement-specific initialization.
5. Application Communication: The protocol is complete; the application code begins normal communication.

The enhancement announcement must be a pattern that is extremely unlikely to be produced by unenhanced clients or servers. We use a fixed, long (1024 byte) string of random bytes as the announcement.

There are a few interesting cases to consider. First, when an enhanced client connects to an unenhanced server and performs Steps 1 and 2 of the protocol, an ordinary server, depending on how it has been programed, usually will close or reset its end of the connection, obliviously send data, or do nothing. In any case, the client does not receive the announcement from the server, and so it aborts the protocol and reverts to ordinary behavior. However, in the case that the server does nothing, the client needs some way to know it should abort. Although only a strange server would quietly leave open a connection that has been closed for writing by its client (we have not seen it happen), should this ever happen the client has a timeout to prevent it from hanging. The timeout period is a multiple of the time it took for connect to succeed, a conservative estimate of the time for several round trips.

Second, if an unenhanced client that is connected to an enhanced server happens to perform the first two steps of the protocol, which includes reading from a half-closed connection, then it will unexpectedly receive the announcement generated by the server.

However, this client behavior is too bizarre to be worth accommodating; for example, although some remote shell client implementations may use shutdown in a similar way [76], they always send commands and data to the server beforehand, so they do not apply to this case.

Finally, the two client connections may reach two different servers if the server application is replicated behind a device that distributes incoming connections to multiple server machines. However, this arrangement only affects the protocol when the replicated servers are non-uniformly enhanced, which is a problem with deployment, not the protocol.

3.3 Reliable Sockets

Reliable sockets are implemented as a library interposed between the application process and the kernel at both ends of a TCP connection (see Figure 3.3). The library exports the sockets API to the application code to enable it to be transparently dropped into ordinary applications. The library also exports the rocks expanded API (RE-API), which enables mobile-aware applications to set policies for the behavior of the reliable sockets library and to manually control some of its mechanisms.

We give an overview of the reliable sockets architecture and operation and then describe our experience with rocks, particularly issues that are pertinent to any system that attempts to interpose user-level functionality between application code and the kernel.

3.3.1 Rocks Overview

The operation of a reliable socket can be summarized by the state diagram shown in Figure 3.4. A reliable socket exists in one of three states: CLOSED, CONNECTED, or SUSPENDED. Note that these states correspond to reliable socket behavior that affects the

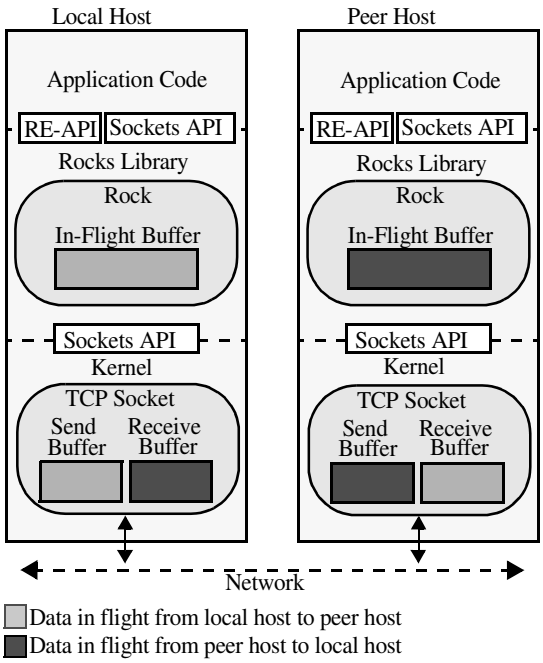

Figure 3.3: The reliable sockets architecture.

process, not the internal TCP socket state maintained by the kernel. A reliable socket begins in the CLOSED state.

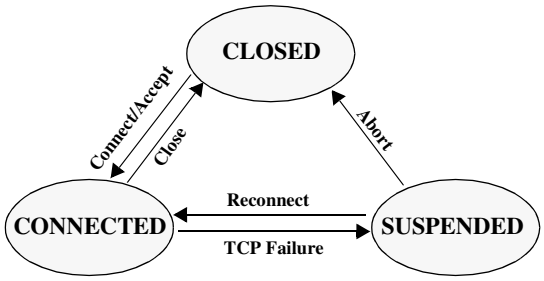

Figure 3.4: The reliable socket state diagram

To establish a reliable socket connection, the application code makes the usual sequence of sockets API calls to create a TCP connection. Instead of being handled by the kernel or the system sockets library, these calls are handled by the rocks library, which performs the following steps:

1. Test for interoperability: The rocks library performs the EDP and reverts the socket to

ordinary socket behavior if the peer does not support rocks or racks.

2. Establish the data connection: The data connection is a TCP connection that, once the reliable socket connection is established, is used for application communication.
3. Initialize: The rocks establish an identifier for the connection based on the addresses of the connection endpoints and a timestamp, perform a Diffie-Hellman key exchange [43] for later authentication, and exchange the sizes of their kernel socket buffers (which are available from the sockets API).
4. Establish the control socket: The control socket is a separate UDP socket that is used to exchange control messages with the peer. It is mainly used to detect the failure of the data connection.

Following these steps the rock changes to the CONNECTED state. Once connected, the application can use the rock as it would use any ordinary socket.

The rock buffers in-flight data as it is sent by the application. The size of the in-flight buffer is the sum of the size of its TCP send buffer and the size of its peer's TCP receive buffer, the maximum number of bytes that can be in flight from the rock to its peer. When the application sends data, the rock puts a copy of the data into the in-flight buffer, and increments a count of bytes sent. Older data in the in-flight buffer is discarded to make room for new data; the choice of size for the in-flight buffer guarantees that data that has not yet been received by the peer remains in the buffer. When the application receives data, the rock increments a count of bytes received.

Connection failures are detected primarily by heartbeat probes that are periodically exchanged between the control sockets. Unlike the TCP retransmission mechanism, heartbeats detect connection failures within seconds instead of minutes, their sensitivity can

be tuned with the RE-API on a per-connection basis, and they work even if the connection is idle. Although the TCP keep-alive probe can also detect failures of idle connections, it is poorly suited for reliable sockets because its two hour minimum default period generally cannot be lowered on a per-connection basis, only on a system-wide basis by privileged users. A rock switches to the SUSPENDED state when it detects that it has not received several successive heartbeats (the number can be adjusted using RE-API).

The use of a separate control socket is motivated by the difficulty of combining application data and asynchronous rocks control data on the same TCP connection. When both flow over a single connection, there must be a way to transmit heartbeat probes even when ordinary data flow is blocked by TCP, otherwise rocks would suspend perfectly good connections. TCP urgent data is the best available mechanism for this type of communication, but it has several limitations. First, although sockets can receive urgent data out-of-band, sending the heartbeat over the same connection as application data would interfere with applications, such as telnet and rlogin, that make use of urgent data. Second, on some operating systems, including Linux, when new out-of-band data is received, any previously received urgent data that has not been consumed by the application is merged into the data stream without any record of its position, possibly corrupting the application data. Since it cannot be guaranteed that a heartbeat is consumed before the next one arrives, this corruption cannot be prevented. Third, on some operating systems, including Linux, when the flow of normal TCP data is blocked, so is the flow of both urgent data and urgent data notification. A separate control socket avoids all these problems.

A suspended rock automatically attempts to reconnect to its peer by performing the following four steps:

1. Establish a new data connection: Each rock simultaneously attempts to establish a connection with its peer at its last known address, the IP address and port number of the peer end of the previous data connection. Whichever connection attempt succeeds first becomes the new data connection.
2. Authenticate: The rocks mutually authenticate through a challenge-response protocol that is based on the key they established during initialization.
3. Establish a new control socket: The new control socket is established in the same manner as the original control socket.
4. Recover in-flight data: The rocks perform a go-back-N retransmission of any data that was in-flight at the time of the connection failure. Each rock determines the amount of in-flight data it needs to resend by comparing the number of bytes that were received by the peer to the number bytes it sent.

Rock reconnection is a best-effort mechanism: it depends on the ability of one end (it does not matter which one) to establish a new connection to the other. A rock cannot establish a new connection if (1) the other end has moved during the period of disconnection, (2) it is blocked from establishing a new connection by a firewall, or (3) the last known address of the peer was masqueraded by a NAT device. Although ordinary applications cannot recover from these cases, the RE-API provides an interface to support mobile-aware applications with alternate means of recovery. Mobile-aware applications can receive a callback when a connection is suspended or when its reconnection timeout expires, and they can specify an alternate last known address. Suspended rocks attempt to reconnect for three days, a period of time that handles disconnections that span a weekend; the RE-API can be used to change or

disable the period on per-rock basis. Rocks suspended longer switch to the CLOSED state and behave to the application like an ordinary aborted socket.

A rock closes gracefully when the application calls close or shutdown. If the application attempts to close a suspended rock, the rock continues to try to reconnect to the peer, and then automatically performs a normal close once it is reconnected, preserving in all but one case the close semantics of TCP. The outstanding case is that an application attempt to abort a TCP connection is converted by the rocks library to an ordinary close, because rocks uses the abort mechanism to distinguish connection failure from intentional application-level close. We have yet to see an application that depends on abort semantics, but should one exist, rocks could use the control socket to indicate an application abort.

Rocks include two programs, rock and rockd, that make it simple to use rocks with existing applications. rock starts a program as a rock-enabled program by linking it with the reliable sockets library at run time. rockd is a reliable socket proxy that enables rock-enabled applications to move while they have open connections to applications that do not support rocks. This mobility is enabled by re-directing the communication between the applications through the rockd. Although the connection between the rockd and the ordinary application is not reliable, the rocks-enabled application can freely move. To simplify the use of rockd, the RE-API has an option to redirect a new connection through a rockd when the enhancement detection protocol detects a peer that does not support rocks.

3.3.2 Experience

The rocks implementation functions well for many interactive programs including ssh, telnet, and X windows clients such as GNU Emacs and Adobe Framemaker. On hosts where

we have root access, we have modified the startup scripts for the corresponding servers to use rocks; on other server hosts, we establish rocks connections through rockd. Using rocks by default generally works well since the EDP switches to ordinary sockets when necessary. However, problems sometimes arise when trying to use rocks with a new application. The main issue is maintaining application transparency; new applications can exhibit behavior that interferes with the rocks library in unanticipated ways. The point of this section is to illustrate the major problems that must be handled by a system that uses user-level interposition to maintain application transparency: reliably intercepting function calls, sharing resources among processes and invocations of programs, and virtualizing various OS resources.

We usually link an application with the reliable sockets library by using the preloading feature of the Linux loader, a commonly available mechanism that enables a library to be linked with binaries at execution time. Preloading has several problems. First, not all binaries support preloading: it cannot be performed on static binaries, since it depends on the dynamic linker, and for security reasons it is usually disabled for setuid binaries. Second, system libraries do not always correctly support preloading: the name resolver in the Linux C library, for example, contains static calls to `socket` that cannot be trapped by the preloading mechanism. Rocks works around this problem by patching the C library with corrected calls at run time, but this technique requires knowledge of the problematic functions, which may change with new library versions. Third, the rocks library may not be the only library that the user wants to interpose in their application. For example, they may also link those used by Kangaroo [78], Condor [40], or those created by the Bypass toolkit [79]. Multiple library interposition requires a sensible ordering of the libraries, linkage of intercepted function calls through each library, and consistent management of file descriptors and other kernel resources

virtualized by the libraries, none of which happens automatically. Although libraries generated by Bypass can co-exist with other interposed libraries, most others just assume they will be placed at the layer closest to the kernel.

Since the rocks state resides in user space, it is not automatically managed by the kernel when the application calls `fork` or passes the underlying socket descriptor to another process over a Unix socket. When a rock becomes shared in either of these ways, the rocks library does several things. First, it moves the state of the rock to a shared memory segment, and forces all sharing processes to attach to the segment. Second, it makes one of the sharing processes responsible for monitoring the rock's heartbeat and triggering its reconnection in the event of failure. Third, in the other sharing processes, it periodically verifies that the responsible process is still running and, if it is not, chooses another process to resume its responsibilities.

Another problem stemming from rock sharing is that a server daemon that hands off incoming client connections to subprocesses may find itself accepting reconnection attempts from past connections. Rocks maintains in shared memory a table mapping rock identifiers to processes. When a server rock accepts a reconnection attempt, it uses this table to locate the process that has the other end of that suspended connection, and passes the new connection to it.

A similar problem with the user-level state of a rock occurs when the application calls `exec`. If left alone, `exec` would expunge from the process all rocks state, including the rocks library, but retain the underlying kernel sockets. When the rocks library intercepts an application call to `exec`, it creates and shares its rocks with a temporary process, sets the environment variables used to preload the rocks library, and then allows the `exec` call to

execute. If the call fails, the library kills the temporary process. If the call succeeds and the rocks library is loaded, the library transfers the state of the rocks from the temporary process during its initialization. If the call succeeds but, because the preloading does not work in the new binary, the rocks library is not loaded and the temporary process eventually times out and closes the rocks.

Maintaining transparency requires virtualizing additional mechanisms, including: (1) emulating polling, asynchronous I/O, and non-blocking I/O semantics for reliable sockets, since data may be available to read from a user-level buffer in the rocks library; (2) multiplexing the timers and signal handlers set by both the application and the heartbeat mechanism; and (3) virtualizing process control interfaces such as wait, kill, and SIGCHLD to isolate processes created by the application from those created by the rock library.

None of these issues alone is particularly difficult, but in aggregation the mechanisms we have introduced to preserve transparency are nearly as substantial as the socket enhancements they support and they create additional operating system dependencies that diminish the portability of rocks.

3.4 Reliable Packets

Seeking an alternative to the application transparency problems created by the rocks library, we developed reliable packets with the goal of supporting network connection mobility outside the kernel without the need to execute in process' address space. The main idea is to use a packet filter to manipulate the packets that are exchanged between the kernel-level socket endpoints of the connection, instead of trying to control the behavior of the sockets API seen by applications. This idea is similar to the use of packet manipulation in the

TCP migrate option [69] and the TCP splice of the MSOCKS proxy [41]. The main differences are that racks perform packet manipulations without kernel modifications and they provide additional functionality including interoperability, long-term connection suspension, and automatic failure detection and reconnection.

A packet filter [48] is a kernel mechanism that enables a user process to select packets that traverse the host network stack. Packet selection is based on a set of matching rules, such as a particular combination of TCP flags or a range of source IP addresses, that the process dynamically passes to the packet filter. Early applications of packet filters included user-level implementation of network protocols, trading the performance penalty of passing network traffic over the user-kernel boundary for the convenience of kernel independence [48]; racks follows this tradition. However, as the primary use of packet filters turned to network monitoring [42,47], the kernel functionality that enabled packets to be redirected through user space became replaced with more efficient mechanisms for passing copies of packets, making systems like racks difficult to develop. Recently the ability to control packets from user space has returned to some operating systems, primarily to support firewall software. Our implementation is based on the Linux netfilter technology [7], but it also could be built using FreeBSD's divert sockets [17].

Racks are implemented in a daemon, the *rackd*, that uses a packet filter to insert itself in the flow of packets between local sockets and the network (see Figure 3.5). The job of the rackd is to prevent local TCP sockets from aborting due to connection failure. Since the rackd executes as a separate process outside of both the kernel and application processes, the rackd lacks the ability to change the binding of kernel sockets to other processes. If it allowed sockets to abort, as the rocks library does, it could not recover the connection.

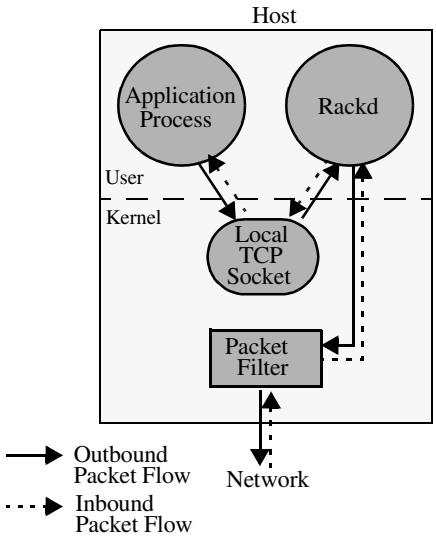

Figure 3.5: The reliable packets architecture.

The rackd inspects packets flowing in either direction and, for each packet, decides whether to discard it or to forward it, possibly modified, to its destination. At any time, the rackd may also inject new packets destined for either end of the connection. Because these operations are privileged on Linux, the rackd needs to run with root privileges.

To be compatible with rocks, the rackd emulates the behavior of reliable sockets, generating a packet stream that is indistinguishable from that induced by the rocks library. However, for connections in which the peer is also managed by a rackd, it takes advantage of the fine control it has over packets to use a simpler enhancement detection protocol and to detect failures without a separate control socket.

The rackd exchanges messages with the rackd or rocks library at the other end of each connection during initialization, authentication, and reconnection. The rackd sends a message by injecting it as if it were data sent by the local socket. It sends the message in a TCP packet whose sequence number follows the sequence number of the last data emitted by the local socket. Once a message has been sent and acknowledged, the local socket and the remote end

no longer have synchronized sequence numbers. The rackd rewrites packets as it forwards them to map sequence and acknowledgement numbers to those expected by the socket.

To establish new connections under the control of the rackd, the rackd configures the packet filter to redirect packets in which only the TCP SYN flag is set; these packets are the first in the three-way handshake of TCP connection establishment. It receives both outbound initial SYN packets (connection attempts issued by local client sockets) and inbound ones (attempts by remote clients to connect to local servers). Since the initial SYN contains the IP address and port number of both ends of the connection being created, it contains all the information necessary for the rackd to select subsequent packets for that connection.

When the initial SYN originates from a local socket, the rackd completes the three-way handshake on its behalf, except it uses a different initial sequence number from the one supplied in the initial SYN and it blocks the local socket from seeing the packets exchanged during the handshake. It performs the EDP client probe over the established connection and then closes it. The rackd then allows the local socket to complete the three-way handshake by sending the original initial SYN packet. If it determined from the EDP probe that the peer is enhanced, the rackd takes control of the connection. Otherwise, it releases the connection by configuring the packet filter to cease redirecting the associated packets; since the local socket connection was established using the original initial sequence number and no messages were exchanged, it can function normally without the rackd.

When the initial SYN originates remotely, the rackd allows the local socket to perform the three-way handshake. The rackd data watches for one of three initial events from the remote client: (1) if the client performs a client probe, the rackd sends the enhancement announcement to the client and closes both ends of the connection; (2) if the client sends an

enhancement announcement, it exchanges reliable sockets initialization messages; otherwise, (3) the rackd releases the connection.

The connection establishment protocol is short circuited if a rackd is present at both ends of the connection. When sending an initial SYN, the rackd modifies the packet to include an unallocated TCP option value that indicates it was produced by a rackd. A rackd that receives an initial SYN containing this option also includes the option in the second packet of the three-way handshake. At this point, both ends of the new connection are mutually aware of their racks support, and immediately following the third packet of the handshake they initialize a reliable sockets connection. As with any other TCP options, the rackd option is ignored on hosts that do not look for it.

Racks detect failures on an established connection using the TCP keepalive protocol instead of a separate control socket. The rackd periodically sends a standard keep-alive probe, a TCP packet with no payload whose sequence number is one less than the last sequence number acknowledged by the peer. When the rackd on the other end receives this packet, it forwards it to the remote socket and in response, the remote socket sends the standard reply to a keepalive probe: an acknowledgement of the current sequence number. To the sending rackd, these acknowledgements serve the role of a heartbeat that asserts the viability of the connection. This technique is unaffected by the use of the keepalive option by the processes on either end of the connection: TCP responds to the probes even if the option is not set and TCP is not affected by the presence of more probes than usual. The keepalive protocol is used when the rackd is connected to another rack; when connected to a rock, the rackd manages a separate control socket.

When it suspends a connection, the rackd must prevent the local socket from aborting, which will happen if there is unacknowledged data in the send buffer or if the application has enabled the TCP keep-alive probe. The rackd sends to the local socket a TCP packet advertising a zero receive-window from its peer. These packets indicate to the local socket that the peer is viable, but currently cannot accept more data. The local socket periodically probes the remote socket for a change in this condition. While the connection is suspended, the rackd acknowledges the probe, leaving the window size unchanged. Although TCP implementations are discouraged from closing their windows in this manner, their peers are required [10] to cope with them and remain open as long as probes are acknowledged.

Racks reconnect in the same way as reliable sockets: each end of the connection attempts to reconnect to the last known address of its peer. When the rackd receives a new initial SYN from a remote socket, it first checks whether it is destined for the previous local address of any suspended racks. If it is, it handles the SYN as an incoming reconnection. To maintain consistency with the local socket, the rackd rewrites the packets of the new connection to match the source IP address, port numbers, and sequence numbers to those expected by the receiving socket, a function similar to that performed by the TCP splice in MSOCKS [41].

3.5 Security

Rocks and racks do not provide additional protection from the existing security problems of network connections. To that end, rocks and racks guarantee that a suspended connection can only be resumed by the party that possesses the key established during initialization. Since it is obtained through the Diffie-Hellman key exchange protocol, the key is secure against passive eavesdropping [43].

Like ordinary network connections, rocks and racks are vulnerable to man-in-the-middle attacks staged during the key exchange or after the connection is established. Resolving this limitation requires a trusted third party that can authenticate connection endpoints. Currently, applications that require this additional level of security can register callbacks with the RE-API to invoke their authentication mechanism during initialization and reconnection. It should be easy to extend rocks and racks to interface with a public key infrastructure, but we are waiting for this technology to become more widespread. Rocks and racks are compatible with existing TCP/IP-based protocols for encryption and authentication, such as SSH [85] and IPsec [35].

In addition, rocks and racks may be more sensitive to denial-of-service attacks because they consume more resources than ordinary connections. Most of the additional memory consumption occurs at user level in the rocks library or the rackd, however additional kernel resources are consumed by the rock control socket and the rackd packet filter rules. These do not represent new types of denial-of-service attacks, but they may lower the requirements for an attacker to bring down a host.

3.6 Process Checkpointing

Existing process checkpoint mechanisms can take advantage of reliable sockets without any modification. When a process linked with rocks is checkpointed, the state of the rocks library is automatically saved with the rest of the process address space. When the process is restarted from the checkpoint, the rocks library detects that the socket file descriptors have become invalid, and initiates their recovery. A process that uses rocks can be migrated with its connections simply by restarting its checkpoint on the new host.

Racks are more complicated to checkpoint. We have experimented with adding racks checkpointing support to a user-level checkpoint library that is linked with the process it checkpoints. A rack checkpoint consists of the state maintained by the rackd and, since the rackd does not buffer in-flight data, the contents of the kernel socket buffers. When directed by the checkpoint library, the rackd induces the socket to transmit the contents of any unacknowledged data in its send-buffer by advertising a large receive window. The checkpoint library obtains the receive-buffer contents by reading from the socket. When restoring a rack checkpoint, the checkpoint library passes the checkpoint to the rackd, creates a new socket, and connects the socket to a special address. The rackd intercepts the packets exchanged during the connection and rather than establishing a normal connection, resumes the connection from the checkpoint. To restore the socket buffers, the checkpoint library sends the send-buffer contents through the socket, and the rackd transmits the receive-buffer contents to the local socket.

The process checkpointing functionality enabled by rocks and racks can be used in several ways. To tolerate the failure of a single node, the process running on that node can be checkpointed and then restarted when the node recovers. The same checkpoint can also be used to migrate the process to another node by restarting the checkpoint on the new node. In the same manner, the entire application can be migrated to a new set of hosts, although this migration must be performed one process at a time to ensure successful reconnection. Alternately, the RE-API could be used to link an arbitrary mechanism for locating migrated processes with the rocks library.

Racks and rocks can also be used to obtain a global checkpoint of a TCP-based parallel application, such as one based on MPI [44] or PVM [26] in direct message routing mode,

from which the application can be restarted after a hardware failure. They free the checkpoint mechanism from any knowledge of the library-level communication semantics of the application, since the rocks and racks recovery mechanisms operate on the underlying sockets, the least common denominator. In contrast, other systems that checkpoint parallel programs, such as CoCheck [74,75] and MIST [13], are explicitly aware of the communication library used by the application. Care must be taken to ensure that the checkpoint of a parallel program is globally consistent. One approach is to stop each process after it checkpoints. Once all processes have checkpointed, the application can be resumed. A more general approach that does not require the entire application to be stopped is to take a Chandy and Lamport distributed snapshot [14].

We have used racks and rocks to checkpoint and migrate the processes of an ordinary MPI application running under MPICH [30]. The application runs on a cluster of workstations using the MPICH p4 device. Once the application is started, each process can be signalled to checkpoint and then terminate or stop.

3.7 UDP

Rocks or racks are not an obvious fit with UDP-based applications. However, the mobility features of rocks and racks can be a benefit to UDP applications by enabling a program to continue its communication following a change of address or an extended period of disconnection. For example, rocks or racks could allow streaming media applications to automatically continue after user disconnection and movement. On the other hand, the reliability features of rocks and racks are not always appropriate for UDP. Although they could simplify the reliability mechanisms of some UDP applications, for others the reliable

delivery of all data may compromise the application's performance or be poorly matched to its reliability model.

Since UDP is inherently unreliable, applications that use UDP must be prepared for lost, duplicated, and out-of-order datagrams. Applications generally use timeouts to trigger retransmission of lost data and to decide that communication should be aborted. It would be difficult for rocks and racks to override timer-based mechanisms, since that would require them to understand the application sufficiently to separate timer events related to communication failure from those that trigger other events such as user-level thread scheduling. Instead, the main benefit of rocks and racks to UDP applications is that they can be a source of information about mobility.

The RE-API provides callbacks through which mobile-aware applications can be notified when a failure has been detected by rocks or racks and when the reconnection mechanism has located the remote peer. These callbacks are not a replacement for reliability mechanisms used by the application, but rather they provide these mechanisms with additional information about communication failures. In the rare cases in which the full reliability features of rocks and racks are appropriate for a UDP application, the RE-API also allows the application to tunnel UDP packets over a rocks- or racks-managed TCP connection.

3.8 Performance

We have evaluated rocks and racks data transfer throughput and latency, connection latency, and reconnection latency over TCP connections between a stationary 500MHz Intel Pentium III laptop and a mobile 700MHz Intel Pentium III laptop both running Linux 2.4.17. Overall, there are few surprises. The additional context switches and copying of redirecting

packets through the rackd makes racks the more expensive of the two systems. The overhead of rocks is noticeable only when data is transferred in small packets, while the performance effects of racks are more significant and occur at larger block sizes. The startup cost of both rocks and racks connection establishment is significantly higher than that of an ordinary TCP connection, but only on the order of 1 ms. Altogether, we feel the overhead is acceptable for the level of mobility and reliability functionality provided by these systems.

3.8.1 Throughput and Latency

We attached the stationary laptop to a 100BaseT ethernet switch in our department network and measured the TCP throughput and latency between it and the mobile laptop from three different links: the same switch, the department 802.11b wireless network, and a home network connected to the Internet by a cable modem (in the uplink direction). We compared throughput and latency of ordinary sockets, rocks, and racks with varying block sizes. Block size is the size of the buffer passed to the socket send system call. We report average measurements over five runs (see Figure 3.6).

The overhead of rocks and racks is most vividly illustrated on the fast link. For blocks of size 100 bytes and larger, ordinary sockets and rocks have comparable throughput that is close to the link capacity (around 90Mb/sec). For smaller blocks throughput drops for all three connection types, however the drop is larger for rocks. The latency overhead of rocks is small (around 10 usec) and independent of block size. We attribute the rocks overhead to the various per-operation costs incurred during data transfer over rocks, including the overhead of copying into the in-flight buffer, periodic heartbeat interruptions, and the rocks wrappers to underlying socket API functions. Racks have more dramatic overhead. While they have

throughput similar to rocks on small blocks, for larger blocks it plateaus at a significantly lower rate (less than 75Mb/sec). There is also a higher per-block latency overhead that, unlike rocks, increases with the block size. We attribute this overhead to the additional per-packet rackd context switches and system calls and the overhead of copying packets in and out of the rackd.

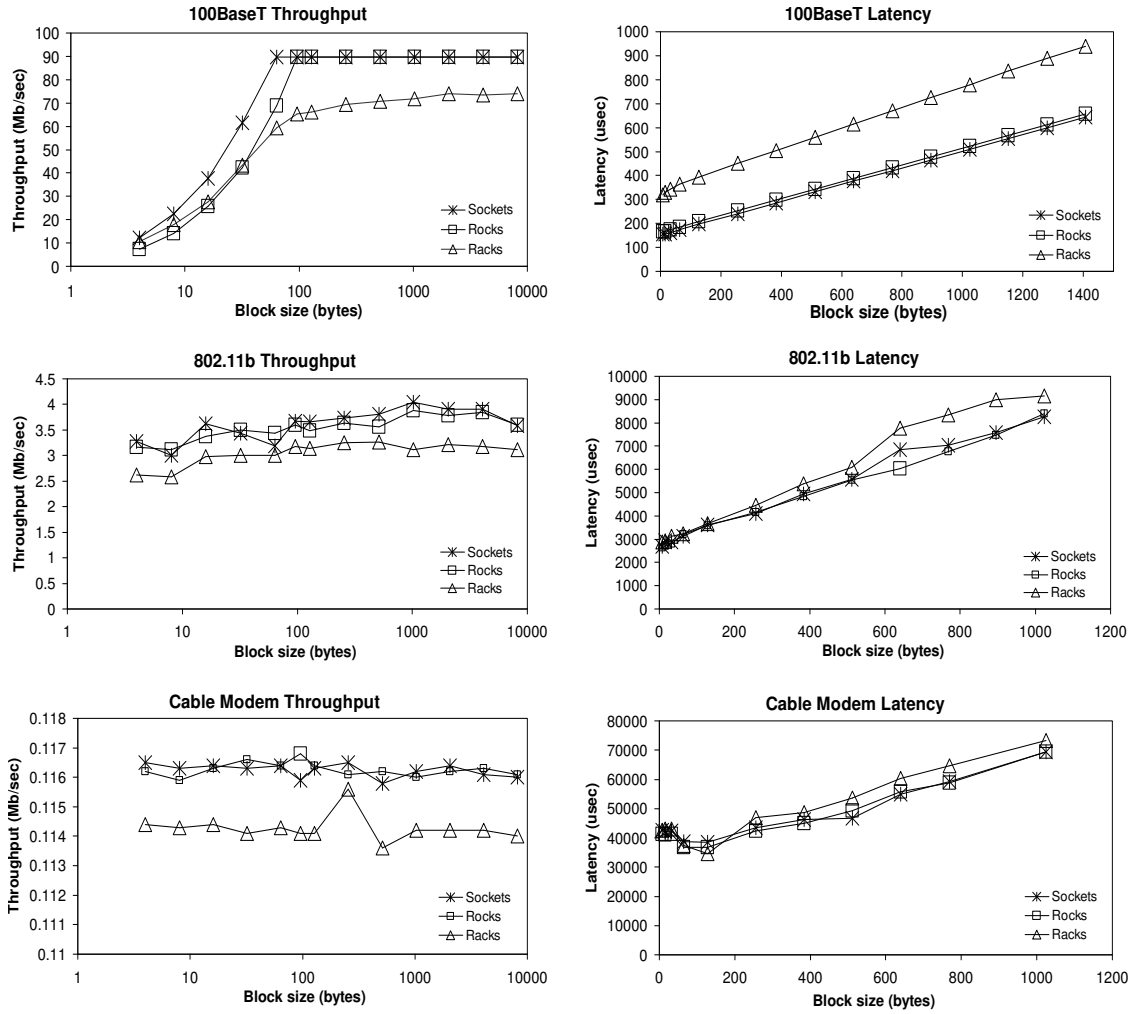

Figure 3.6: Average rocks and racks throughput and latency.
Results shown for 100BaseT, 802.11b, and cable modem links. Note that the cable modem throughput graph has a non-zero y-axis, exaggerating the differences.

The performance effects of racks and rocks are less easily discerned on the slower links. While we had exclusive access to the 100BaseT switch, the measurements on the 802.11b and cable modem networks were subject to the varying conditions of these shared networks, making it difficult to capture clear differences. On the 802.11b link, the standard deviation is about 20% of the average throughput and about 15% of the average latency. On the cable modem, the standard deviation is about 4% of the average throughput and about 40% of the average latency. We conclude that the overhead of racks is still apparent on slower links, but not the overhead of rocks.

3.8.2 Connection

We measured the connection overhead in a rock-to-rock connection and a rack-to-rack connection. We timed 100 application calls to connect and report the average times in Figure 3.7. Rock connection time is about 18 times higher than the time for ordinary socket connection, while rack connection is about 16 times higher. The most expensive aspect of both connections is the key exchange for authentication, an operation that involves large integer arithmetic and takes approximately 2 ms. Although these times are high, connection times are still about 4 ms, which we deem an acceptable cost for the added reliability and mobility.

Connection Type	Time (usec)
Ordinary Sockets	221
Rocks	3908
Racks	3588

Figure 3.7: Average TCP connection establishment time.

3.8.3 Reconnection

We measured the amount of time it takes to reconnect a suspended rock or rack. Reconnection time is the time following a restoration of network connectivity that a rock spends establishing a new data and control socket with the peer and recovering in-flight data. For the experiment, we suspended a connection by disabling the network interface on one machine, then measured the time elapsed from when we re-enabled the interface to when the connection returned to the ESTABLISHED state.

The elapsed time over multiple runs of the experiment were always under 2 seconds. This time is less than the time required to restart most non-trivial applications that would fail without rocks or racks, and small in the time scale of the events that typically lead to network connection failures, such as change of link device, link device failure, laptop suspension, re-dial and connect, or process migration, and so, we conclude, acceptable.

3.9 Portability

An initial port of rocks to Microsoft Windows has been completed. The goals of this port were to test the generality of our techniques and to make the system available to a wider set of users. Because the Windows sockets API (winsocks) contains significantly more calls than the Unix sockets API, the strategy has been to first port the subset of winsocks corresponding to the Unix calls, and then incrementally add support for the remaining calls. The status of the port is that rocks supports enough of winsocks (version 2, the current version) to work with two ssh clients, SSH and SecureCRT, the Windows ftp client, and some simple servers. Three aspects of rocks required major effort to port: intercepting calls to winsocks API; supporting Windows I/O models that are not available on Unix or not supported by rocks on Unix; and

processing heartbeats. The parts of rocks that are based on socket and operating system features common among Unix and Windows, such as the enhancement detection protocol in-flight buffers, connection initialization, and the reconnection algorithm, required little effort to port.

Winsocks is implemented in a library that is loaded in each Windows process that uses it. Rocks needs a way to transparently intercept calls that the application makes to this library. Unfortunately, Windows does not provide a mechanism like the Unix preload loader option for inserting new code at an API layer. Several techniques have been developed by the Windows programmer community to achieve this functionality. The one we decided to use involves re-writing the in-memory *import table*, a jump table in each library or executable for calls to external functions that is patched at run time. Rocks uses a special application launcher that starts the application, allows the operating system to perform run-time linking, loads the rocks library, and then replaces import table entries for winsocks calls with jumps to the corresponding calls in the rocks library. It was easy to integrate this new mechanism into rocks because there is a clean separation between the sockets API implementation in the rocks library and the mechanism that re-directs application calls to it.

Although winsocks provides significantly more calls than the Unix sockets API (see Figure 3.8), more than half of these calls involve operations that are independent of rocks. Calls that rocks can ignore include those involving (1) name services and network addresses, which like their Unix counterparts do not affect connections, (2) multipoint and multicast networks, which are not supported by rocks on Unix, and (3) connect and disconnect data, which is data sent upon connection establishment and termination, an optional transport protocol feature that is not available for TCP or UDP.

Types of Calls		Winsock API Calls
Affected By Rocks	Unix analogues	**accept**, **bind**, **closesocket**, **connect**, **getpeername**, **getsockname**, getsockopt, **ioctlsocket**, **listen**, **recv**, recvfrom, **select**, **send**, sendto, setsockopt, shutdown, **socket,** WSADuplicateSocket, **WSAGetLastError**, WSASetLastError
	Startup and Cleanup	**WSAStartup**, WSACleanup
	Window binding	**WSAAsyncSelect**
	Event object binding	WSACloseEvent, WSACreateEvent, WSAEnumNetworkEvents, WSAEventSelect, WSAResetEvent, WSASetEvent, WSAWaitForMultipleEvents
	Overlapped I/O	WSAGetOverlappedResult, WSAIoctl, WSARecv, WSARecvEx, WSARecvFrom, WSASend, WSASendTo, WSASocket
	Extensions	AcceptEx, ConnectEx, DisconnectEx, GetAcceptExSockaddrs, TransmitFile, TransmitPackets
Not Affected By Rocks	Name Services and Network Addresses	freeaddrinfo, GetAddressByName, getaddrinfo, gethostbyaddr, gethostbyname, gethostname, getnameinfo, getprotobyname, getprotobynumber, getservbyname, getservbyport, htonl, htons, inet_addr, inet_ntoa, ntohl, ntohs, WSAAddressToString, WSAAsyncGetHostByAddr, WSAAsyncGetHostByName, WSAAsyncGetProtoByName, WSAAsyncGetProtoByNumber, WSAAsyncGetServByName, WSAAsyncGetServByPort, WSACancelAsyncRequest, WSAEnumNameSpaceProviders, WSAEnumProtocols, WSAGetQOSByName, WSAGetServiceClassInfo, WSAGetServiceClassNameByClassId, WSAHtonl, WSAHtons, WSAInstallServiceClass, WSALookupServiceBegin, WSALookupServiceEnd, WSALookupServiceNext, WSANSPIoctl, WSANtohl, WSANtohs, WSAProviderConfigChange, WSARemoveServiceClass, WSASetService, WSAStringToAddress
	Multipoint and Multicast	WSAJoinLeaf
	Connect Data	WSAAccept, WSAConnect, WSARecvDisconnect, WSASendDisconnect

Figure 3.8: Windows Sockets API (Version 2) calls.
Ported calls are shown in boldface.

Of the calls that must be ported, the easiest subset to port are those that map directly to the Unix sockets API. Except for minor differences in function call names and parameter types, these calls are identical to their Unix counterparts. They have been ported as needed; we do not expect the remaining calls in this category to be challenging to port.

The more challenging calls are those related to three I/O models that are not supported by rocks. The model that has been ported is that which enables network events to be converted to messages sent to a window. Rocks must ensure that this binding is preserved across disconnection and reconnection events, and it must ensure that messages caused by socket failure are not forwarded to the window. When the rocks library intercepts a call to `WSAAsyncSelect`, the call that establishes this binding, it binds the event to an invisible window whose message dispatcher forwards desirable events to the requested windows, but drops events caused by socket failure, instead switching the rock to the suspended state.

Whenever a suspended rock is reconnected, any previously established event bindings are restored.

The other two types of I/O models require a similar strategy, but they have not been implemented yet. First, network events can be converted to notification of Windows event objects, which are synchronization objects similar to (but not exactly) binary semaphores. When a network event of interest occurs on a socket, this binding mode causes the event object to be signalled, after which a thread waiting on the event is awakened.

Second, overlapped I/O is similar to the Unix asynchronous I/O model in which I/O operations started by the application return immediately and the operating system notifies the application when the operation completes. Use of this model is not widespread in Unix programs [76] and so it has not yet been supported in the Unix version of rocks.

The remaining winsocks calls that need to be ported are functions that act as wrappers for common sequences of socket operations. For example, `AcceptEx` in addition to accepting a new connection returns a copy of the local and remote addresses of the connection and reads an initial block of data. Since these calls can be described in terms of underlying sockets API calls, it should be simple to port them by re-implementing their definition over rocks versions of the same functions.

The final major porting effort involved a new design for the mechanism that sends and receives heartbeats and suspends rocks that have not received heartbeats recently. On Unix, these operations are performed in a signal handler that is periodically called using one of the timers the operating system provides to processes. Such a facility is not available on Windows – there is no way to asynchronously interrupt a thread – so instead heartbeats are handled in a thread, called the *control thread*, that is created for each rock. The control thread is also used

for reconnection, which on Unix is handled in a separate process. This new design is simpler than the Unix design and would be possible to implement on Unix; the advantages of substituting it for the current Unix mechanism would be less platform-specific code and the elimination of potential contention for the Unix timer facility between the rocks library and the application.

Chapter 4

User Interfaces

Interactive applications have graphical user interfaces (GUIs) that a user may wish to move with the application or independently of the application. Both types of movement require the ability to extract the state of a GUI from one machine and regenerate it on a different one transparently to the application. *Guievict* is our system for performing this operation. The main idea behind guievict is to capture the application's *window session*, a transportable representation of its GUI. This abstraction precisely encapsulates the GUI state of the *application* — not individual windows of the application, nor the windows of all applications that are being displayed on a particular machine. In addition to enabling the mobility operations of moving a running application's GUI to a different display and migrating a running application with its GUI to a different machine, window sessions enable GUIs to be *replicated* on multiple displays, a necessary primitive for building collaborative applications [2], without any application-level support for replication.

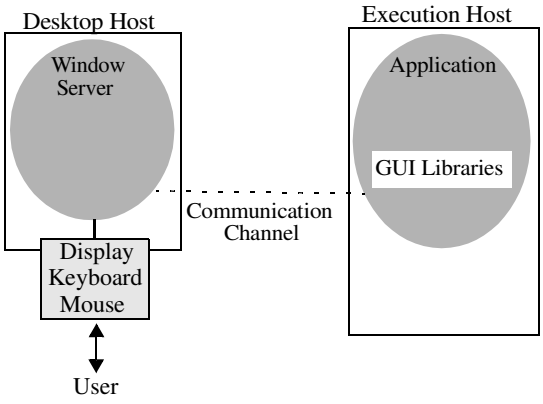

Figure 4.1: The elements of a GUI-based application.
The user interacts with the application through hardware managed by the window server on the desktop host. The application process executes on a possibly different execution host, exchanging GUI-related messages with the window server over a communication channel.

Guievict is distinguished from previous work by the following characteristics:

- Migration occurs at application granularity; users can select and move the GUI of individual applications from their desktop, leaving the GUIs of other applications behind or free to move elsewhere.
- Any application program, including those based on legacy toolkits, can be migrated without modifications such as re-programming, re-compiling, or re-linking.
- Migration can be unpremeditated; users do not need to run their applications in a special way, such as by redirecting their GUI communication through a proxy.
- No modifications to window system code are required; the new functionality is encapsulated in (1) a window server extension, implemented on top of the window server extension API, that is loaded in the server when it is started, and (2) a library that is loaded into the application at run-time.

Figure 4.1 shows the essential elements of a GUI-based application. The application runs in one or more processes on the *execution host* and the user interacts with it from a possibly different *desktop host*. The desktop comprises a display, keyboard, and mouse managed by a window system that multiplexes the desktop for all applications that interact with the user at that host. The window system responds to requests for GUI services (such as creating a window) sent by the application and passes notification of desktop events (such as a mouse

click) to the application over a communication channel such as a network connection or (when the execution and desktop hosts are the same) shared memory. The state of the application GUI is distributed between the window system and a library (often called a toolkit) in the application process.

Guievict has been implemented for the X window system [67] (using the XFree86 implementation) and required overcoming four challenges:

- Dynamically taking control of the window system operations of a running program. Guievict injects code into the application process that discovers the process's communication channel to the window server and synchronizes its communication with the server.
- Retrieving the GUI resources of an application: The guievict X window server extension enables an application, at any time, to determine the identifiers of its GUI resources and the dependencies among these resources, and to extract these resources in a form from which they can be regenerated on a new desktop.
- Regenerating an application's resources in another desktop host: Guievict uses standard X protocol operations to regenerate GUI resources in the new window server.
- Ensuring GUI migration is transparent to the application: Applications whose GUIs have been migrated can be confused by the resulting changes to resource identifiers, message sequence numbers, and display characteristics such as pixel depth. Guievict interposes a filter (called the *guimux*) on the communication between the application that provides the mapping necessary to maintain transparency. The guimux also serves as a multiplexor for GUI replication.

Although other commodity window systems, such as Microsoft Windows and the Quartz system in Apple's OS X, have significantly different architecture and programming models from X windows, they present similar fundamental barriers to capturing window sessions. For example, Windows also lacks an interface for enumerating all of the GUI resources that an application has allocated [86] and, since the operating system rather than the application

assigns identifiers to resources, it also requires guievict-like systems to provide a way to hide changes in GUI resource identifiers from the application. Most OS X applications are based on toolkit libraries provided by the operating system. The toolkits do not provide operations for capturing window sessions, and the interface between the toolkit libraries and the window system is undocumented [3].

The main limitations of guievict are that (1) it requires the user to install the X window server extension on their desktop hosts, (2) it requires the application binary to contain symbol table entries for the window protocol stubs it calls, and (3) it has a large (20 second) overhead in checkpointing font state. We discuss implications and possible workarounds of the last two issues in Section 4.2.

The remainder of the chapter is organized as follows. Section 4.1 presents the architecture of guievict. Section 4.2 describes its implementation. Section 4.3 presents the evaluation of guievict. Section 4.4 identifies the security issues raised by guievict.

4.1 System Overview

A guievict user first *attaches* (see Figure 4.2) guievict to their application, after which they can perform any of three operations: migrating the application's GUI from one desktop host to another, replicating the application's GUI on multiple desktops, and migrating the application process along with its GUI. These operations involve four system components. The *guievictor* is a program that the user runs to control the guievict system. The user runs it on the host running the application process. The *guievict client library* is loaded by the guievictor at run time into the application process and implements application-side GUI migration operations. The *guievict server extension* is the corresponding server-side

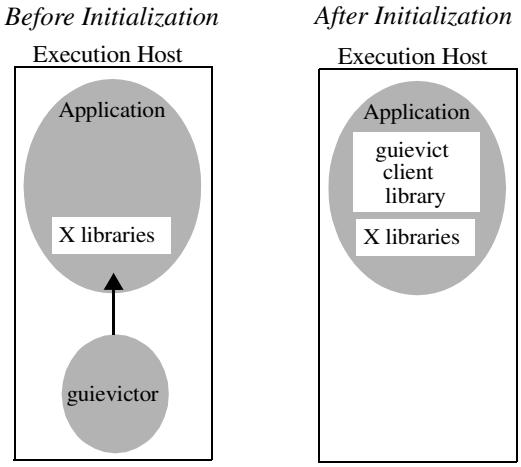

Figure 4.2: Initializing the application process.
The guievictor stops the application process, forces it to load the guievict client library, and exits.

component. The *guimux* is a daemon, started by the client library, that runs on the execution host and ensures that GUI migration and replication is transparent to the application.

The remainder of this section describes the guievict operations in detail. The next section discusses solutions to the technical problems underlying these operations.

4.1.1 Initialization

The user prepares a running application process for the guievict system by invoking the guievictor's *initialize* operation, passing the application process identifier as an argument. The guievictor *hijacks* [88] the application process: it stops the process and forces it to load and initialize the guievict client library (see Figure 4.2). Hijacking is transparent to the application; afterward, the guievictor resumes the application process, allowing it to continue normally. During its initialization the client library establishes a communication channel for future interaction with guievictor. When the user later runs guievictor again to request a guievict operation, guievictor uses the channel to interrupt the application process and send commands to the client library.

The guievict server extension must be loaded and initialized in the window server before the user's first request for a guievict operation. Because the XFree86 window server does not support run-time extension loading (although it contains much of the necessary mechanism), the user must arrange for the extension to be loaded when the server is started, which entails adding a line to its configuration file.

4.1.2 GUI Migration

GUI migration is broken down into two steps.

First, the user requests the application (with a guievictor command) to *detach* its GUI from the window server. In response the guievict client library (see Figure 4.3):

1. Synchronizes the application's communication with the window server and blocks the application from further communication;
2. Retrieves the window session from the server;
3. Closes the connection to the window server.

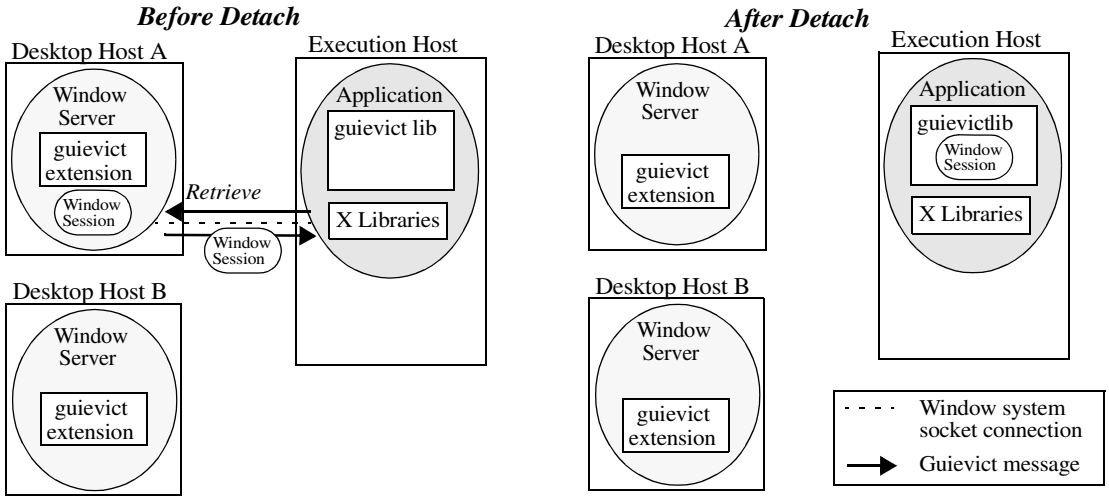

Figure 4.3: Detaching a GUI from Desktop A.
The guievict client library requests the window session from the guievict server extension at Desktop A, then closes the connection.

Second, the user requests the application to *re-attach* its GUI to a new window server (see Figure 4.4). The guievict client library starts a guimux process, replaces the application's socket to the window server with a full-duplex pipe to the guimux process, and then transfers control to the guimux process. Then the guimux process:

1. Opens a connection to the new window server;
2. Regenerates the state of the window session;
3. Signals the guievict client library to resume the application.

The re-attach operation may come an arbitrary period of time after the detach. In the meantime, the client library suspends the execution of application code to prevent the temporary absence of a window server connection from affecting the application. For the common case of a user who wishes to detach from one desktop and attach to another in one logical operation, the guievictor provides a combined detach and attach command.

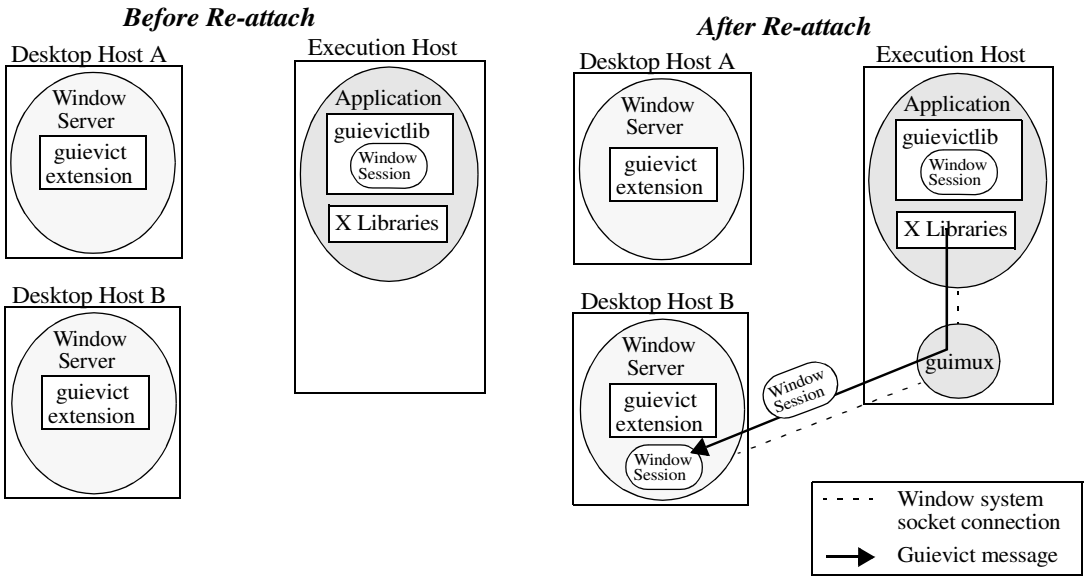

Figure 4.4: Re-attaching a GUI to Desktop B.
The guievict client library establishes a new connection to Desktop B through the guimux, which forwards all window communication between the application and Desktop B, starting with the request to regenerate the window session.

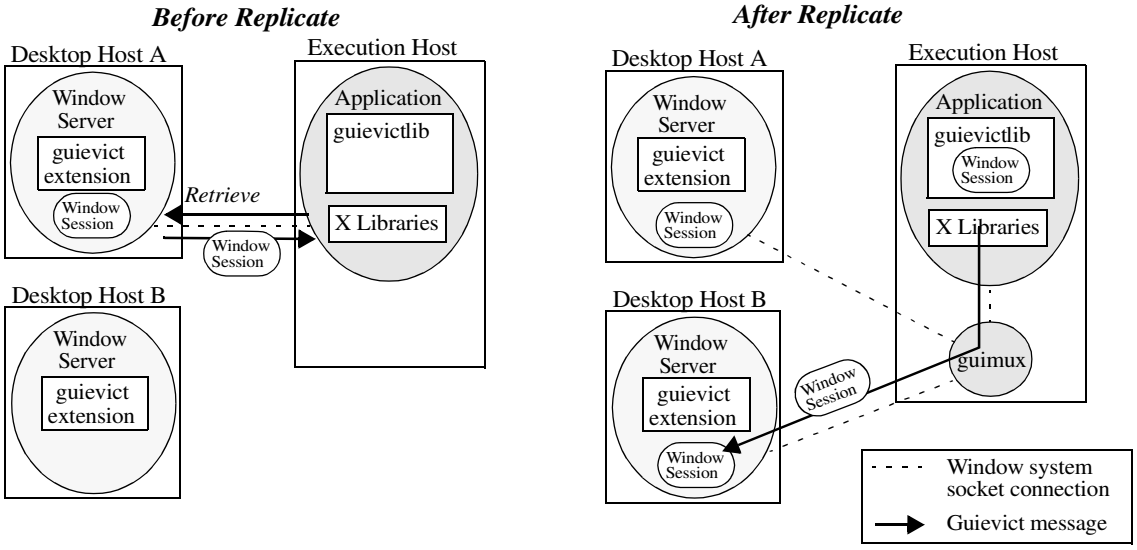

Figure 4.5: Replicating a GUI on Desktop B.
Replication is similar to migration, except that the connection to Desktop A is preserved and multiplexed by guimux with the connections to other desktop hosts.

4.1.3 GUI Replication

GUI replication is a simple variation of GUI migration. The user requests (with the guievictor) the application to *replicate* its GUI on another window server. The guievict client library performs all but the final step of the detach operation. That is, it acquires the current state of the window session, but does not close the connection to the window server. It then performs a normal attach operation to connect the GUI to the additional window server (see Figure 4.5). If this is the first time an attach operation has been performed, it also redirects the original window server connection through the guimux process.

4.1.4 GUI+Process Migration

Guievict supports the simultaneous migration of the application process and its GUI. Figure 4.6 shows a typical scenario for this type of mobility in which the execution and desktop hosts are the same (such as a laptop) and the user wants to migrate their application process and its GUI to another host (such as the computer on their desk). In this scenario, the

user uses guievictor to request the application to *migrate*, providing two arguments: the name of the new X window server and the name of the new execution host. Guievict then:

1. Detaches the application's GUI from its window server;
2. Terminates the guimux daemon (if one is running);
3. Checkpoints the application process, producing a *checkpoint file* [40] containing the state of the application process, including its window session;
4. Exits the application process.

At this point the user must transport the checkpoint file to the new execution host and invoke the guievictor to *restart* the application process. To complete the migration, guievict:

5. Restores all the state of the application process except for its GUI;
6. Attaches the application process to the new window server.

4.2 Implementation

We implemented guievict on x86 Linux using the XFree86 implementation of the X window system. The major technical issues were: hijacking the application, finding the application's connection to the window server, synchronizing this connection, retrieving and restoring GUI resources, and ensuring that GUI migration is transparent to the application.

4.2.1 Hijacking the Application

Process hijacking can be safely implemented with basic dynamic instrumentation mechanisms, such as those provided by the Dyninst API [12]. These include stopping the process, forcing the process to execute code to load and initialize the guievict client library, and then resuming the process. The guievictor contains its own implementation of these

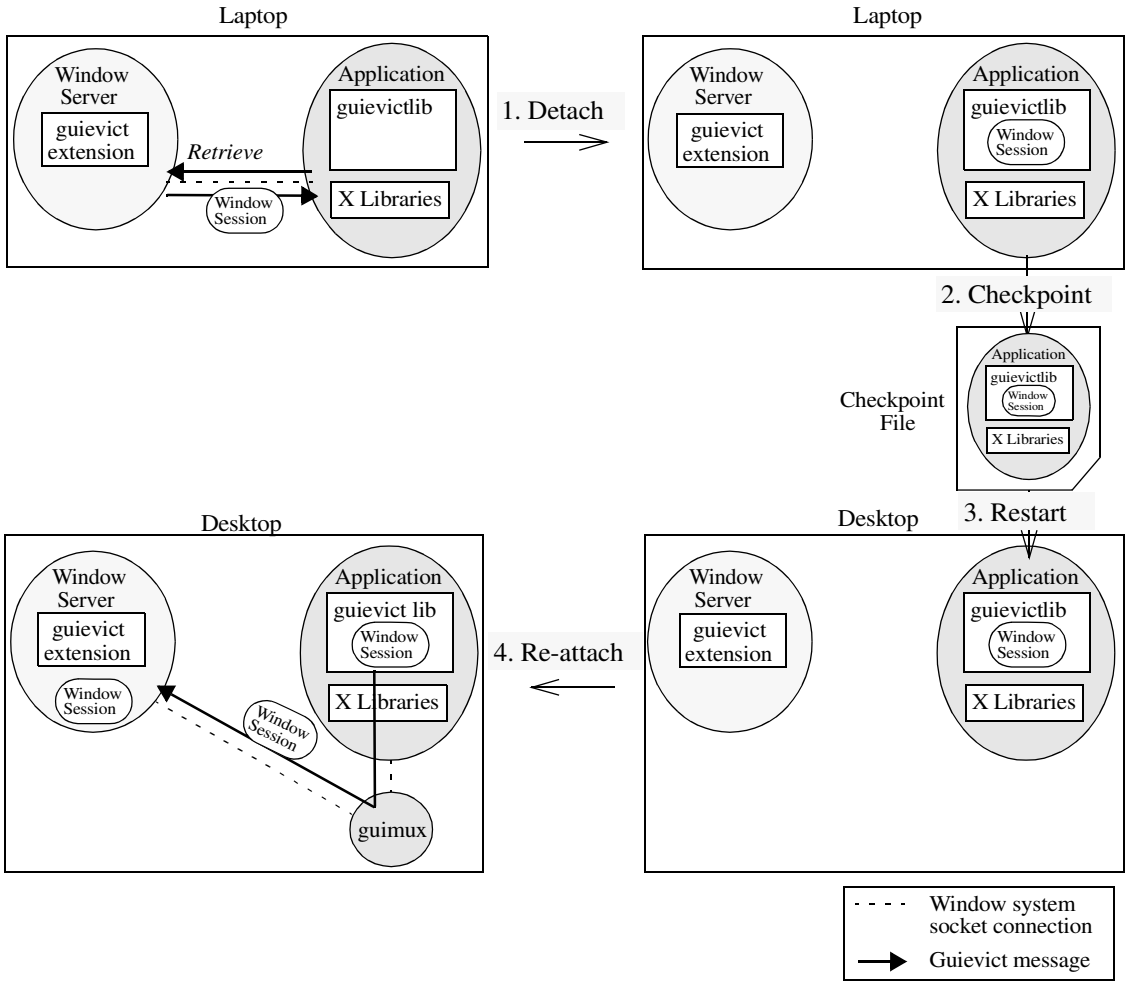

Figure 4.6: Migrating an application process and its GUI from a laptop to a desktop computer. *(1) The guievict client library detaches the GUI from the window server; (2) guievict checkpoints the application process, producing a checkpoint file containing the entire application state including the window session; (3) After transporting the checkpoint file to the desktop host, guievict restarts the application process; (4) guievict re-attaches the GUI to the desktop window server.*

mechanisms to avoid requiring users to install additional software like the Dyninst API (which contains much more functionality than guievict requires).

The guievictor forces the application process to load the guievict client library by injecting code that calls the run-time library loading feature (usually named dlopen) of the process's dynamic loader, a simple technique that is easy to implement for dynamically-linked programs. Guievict introduces hijacking functionality to cope with statically-linked programs.

The idea is to map and initialize a copy of the dynamic loader into the process's address space, essentially by reproducing the initial steps the operating system takes when loading a dynamic-linked program. This instance of the dynamic loader does not control the original code in the process, but rather serves only to provide an implementation of dlopen that guievictor can invoke the same way it does for dynamically-linked programs. Support for statically-linked programs is important because many GUI-based applications are distributed statically to avoid forcing users to have the necessary GUI library dependencies.

The guievict client library creates a named Unix domain socket to act as the communication channel for subsequent interaction with the guievictor. The library deletes the socket when the application exits normally. The socket name is based on the application's process and user identifiers to avoid conflicts with other independently running instances of guievict and to allow stale sockets left behind by abnormal termination to be cleaned up by the user who discovers them. The guievictor gets the attention of the guievict client library by writing a message the socket, causing the application process to receive a signal that is handled by the library.

4.2.2 Finding the Window Server

X windows applications communicate with the window server over a Unix domain or TCP socket. Unlike proxy-based systems such as xmove [72], guievict may be invoked after the creation of the connection to the window server. It must search the file descriptors of the process for sockets connected to a window server. Most operating systems provide a way, for example a `/proc` entry on Linux, to list the open file descriptors of a process; on those that do not, guievict can test each possible file descriptor with the fstat system call. Guievict looks for

a file descriptor that (1) refers to a socket inode (as reported by the fstat system call), and (2) is connected to a window server.

The second condition is difficult to check. In many cases, the `getpeername` system call, which returns the address of the socket on the other side of a connection, is sufficient: guievict looks for a socket whose peer address is one of the well-known X window server TCP ports or Unix domain socket names. This filter can fail if the application is tunnelled to the window server through a proxy, since the peer address of its socket will be the proxy's address. In many common proxy configuration, such as ssh tunnels [85] and firewall port forwarding rules [91], the difference between the proxy address and a normal window server address is a small positive offset in the port number, which is easy to recognize.

In the unlikely event that `getpeername` does not reveal an obvious window server connection, guievict checks whether the peer address of each socket leads to a window server. It creates a new socket, attempts to connect it to the peer address, and, if the connection succeeds, performs the first round of the standard X windows handshake. If the server gives the expected response to the handshake, guievict concludes that it has found a socket connected to an X server. If the probe fails on all sockets, guievict gives up control of the application.

The use of this probe raises two concerns. First, the probe may succeed on non X window servers that happen to respond to the handshake like a X windows server. In practice, the response is distinguishable from that seen in common protocols such as ssh, telnet, ftp, and http, however unfortunately it is not so distinctive to presume that conflicts will not occur in less common protocols. In the event of a false positive, guievict should eventually receive nonsensical messages from the server, after which it will abort. Second, the probe may have a

negative effect on probed servers. Although server implementations should be robust to spurious connections, not all existing ones are, particularly those that have not been hardened for Internet deployment. For users who cannot risk using the probe, the guievictor accepts a command line argument to identify the file descriptor or peer address of the window system connection.

4.2.3 Synchronizing Communication

Guievict must ensure that the state of the window session does not change while it is being retrieved. Changes to window state are caused by messages exchanged between the application and the window server. Guievict synchronizes the communication by finding a point in the message stream where there are no partially sent or unanswered requests, and then blocks further communication.

The synchronization occurs in two steps. First, guievict forces the application process to reach a message boundary in the stream of messages from the application to the server. It does so by examining the application's process stack before starting an operation, searching for X library functions that are stubs for X protocol requests. If such a function appears on the stack, which indicates that the application process may be in the middle of sending a message, guievict sets a timer, resumes the application code for a short period of time, and then re-examines the stack. This procedure repeats until the stack is free of potentially unsafe functions.

Second, guievict sends an X protocol request containing an illegal resource identifier to the server. The only effect of this request is that it elicits an error message from the server. Guievict reads and scans the stream of messages from the server until it recognizes the error,

at which point the client has no unanswered requests and the connection is synchronized. The messages read before the error are buffered and re-sent to the application from the guimux when the application is allowed to resume.

Detecting the presence of X protocol stubs on the application process stack depends on the presence of the symbols for these functions in the application process. This is not an issue for dynamically-linked applications because the symbols must be present to facilitate linkage. However, symbols may be stripped from statically-linked executables. Today, because of this limitation the guievictor refuses to work with stripped static applications. More research is needed to overcome this limitation; one idea is to try to infer message boundaries through some kind of analysis of the byte stream flowing over the socket, perhaps by matching patterns most likely to occur at fixed locations within an X protocol message.

4.2.4 Retrieving and Regenerating GUI Resources

X windows applications create, modify, and destroy GUI resources through the exchange of X protocol messages with the X window server. GUI resources reside in the window server and are indirectly manipulated by the application by 32-bit *resource identifiers*, which are drawn from a namespace that is global across all clients of a window server. Clients choose the low-order bits of the identifier for each resource that they create, but they must set the high-order bits to a fixed *client id* chosen by the server when the application connects to it.

It is generally impossible to locate the resource identifiers in the application process's code and data, so guievict must get them from the X window server. The window server manages a per-client table of allocated resources but, unfortunately, it does not provide external access to the information in this table. Since no previously reported server extension

has addressed this limitation, guievict provides the missing interface. Using our new *GetResources* request, an application can request the server to return an enumeration of all the resources that the application has created. For most types of resources, the resource identifier is sufficient for the application to retrieve the state of the corresponding resource with standard X protocol requests, but there are four exceptions: windows, graphics contexts, cursors, and fonts. The *GetResources* reply for windows includes the background pixel value and the window's cursor identifier. The replies for graphics contexts and cursors include their entire state: for a graphics context, a small array of flags, and for a cursor, its bitmap and geometry.

Fonts are more complicated. X windows fonts are stored at the server. Clients acquire the use of a font by sending a request to the server containing a font resource identifier and the name of the font. The server loads and binds the font to the identifier if it has the font, and otherwise returns an error. Applications can request detailed geometric information about the font associated with a font identifier, but unfortunately there is no request to map a font identifier to the name of the font. (A similar lack of support for reverse mapping an identifier to a name must be overcome by systems that want to determine the name of an open file from its inode number [88].) Strangely, the window server discards the font name after loading a font, so the mapping is not feasible even within the context of a server extension. To generate this mapping, the guievict client library performs during the detach operation a search that indirectly maps each font identifier to a font name. It requests the server to list of all of its stored fonts (a standard X protocol request), and then searches this list for a font whose geometry matches that of each font identifier. Usually, the font name suffices to regenerate the font resource on a new window server, but sometimes the new server does not have a font

with that name. In those cases, guievict searches the font list on the new server and selects the font with the closest matching geometry, using a least-squares font matching algorithm similar to that used by HP Shared X [25]. This complicated and expensive approach to migrating font resources could be eliminated by switching to client-managed fonts, a recently proposed architectural change to the X window system [27] that is beginning to appear in recent distributions. While waiting for this practice to become standard, we minimize the overhead by having guievict cache in the application's file system the names and geometry of the fonts supported by each server it visits.

Sometimes it is not possible to regenerate pixel-based resources identically to their previous instances. Displays can vary by the number of bits per pixel (depth) and the method by which pixels are mapped to colors (visual type), both of which cause the meaning of pixel values to change. Xmove provided an additional translation operation that mapped pixel values from their previous depth and visual to that available on the current server. Recent developments for the X server, however, promise to eliminate the need for such translation. In particular, the R&R extension and shadow framebuffers [29] are server-based mechanisms for virtualizing depth and visual that have been designed with the goal of providing heterogeneity support for migration and replication systems. Guievict complements these developments; together they combine to produce a system for GUI migration that hides display heterogeneity.

4.2.5 Maintaining Transparency

The main role of the guimux daemon is to make GUI migration transparent to the application process by translating resource identifiers and sequence numbers that appear in

the messages exchanged between the application and the window server. The resource identifier mapping is initialized during the regeneration of GUI resources. The guievict client library regenerates a resource by issuing an ordinary X protocol resource creation request containing the original identifier. As guimux forwards these requests, it replaces the identifier with an unused identifier that is valid in the current window server. For subsequent messages from the application to the server, the guimux maps references to resources to their current identifier; it performs a similar reverse mapping on messages from the server to the application. As the application destroys resources, they are removed from the map.

Sequence numbers occur in messages sent from the window server to the application and represent the number of messages the server has processed for the application; they do not occur in messages from the application. The guimux replaces the sequence number of a message with the next sequence number expected by the application process. This procedure is initialized when guievict synchronizes the communication to the window server. At this point, the next sequence number expected by the application process is the sequence number contained in the sentinel error reply.

Replication of windows on multiple displays extends the role of the guimux process. While managing a replicated GUI, the guimux maintains a separate translation map for each window server connection. Messages from the application are translated and sent to each window server. Messages from the window servers are reverse translated and forwarded in series to the application. To control the behavior of replicated GUIs, guimux accepts a set of commands, sent by the guievictor, that act as primitives for setting replication policy. For example, the user can suppress the forwarding of keyboard, mouse, and window modification events from selected desktop hosts to allow users seated at those desktops to observe but not

modify the state of the GUI; more sophisticated policies for managing collaborative work [2,31] can be built over these primitives.

4.3 Evaluation

We have evaluated the performance of guievict's GUI migration functionality. As a point of reference, we compared its performance to the proxy-based xmove system [72]. We measured the time to detach and re-attach a GUI and the impact on interactive response. Measurements were taken on a 700 MHz Pentium-III laptop running XFree86 4.2.0 on Linux 2.4.18 using the most recent release of xmove [71]. Overall, the results are not surprising. As reported in detail below, guievict takes longer than xmove to detach a GUI from a window server, but re-attaches in comparable time. Xmove and guievict (after re-attach) both increase the latency of the communication between the application process and the window server, but not enough to be perceptible to users.

4.3.1 Detach and Re-attach Latency

We measured the latency of detaching a GUI from a window server and then re-attaching it to the same window server for several applications. The guievict detach latency is the elapsed time from the moment the guievict client library receives the detach command to just after it closes the connection to the window server. The guievict re-attach latency is the elapsed time from the moment the guievict library receives the re-attach command to just after it allows the application process to continue. The xmove latencies are analogous. We ran both the application process and the window server on the laptop and detached the application's GUI after its initial windows were drawn but before any user interaction with the GUI. We report average measurements over five runs. The results are reported in Table 4.7.

Guievict has a much more expensive detach operation than xmove. Table 4.8 breaks down the average guimux detach time for one application. The most expensive stage is mapping the font identifiers to fonts names, during which most of the time is spent waiting for the server to return the complete list of the fonts. However, this operation only happens the first time guievict detaches a GUI from a given server, since guievict caches the information in the file system. More generally, guievict takes longer to detach because it retrieves the GUI state when it receives the detach request, while xmove collects the GUI state as it is created. Guievict and xmove have similar re-attach performance, which is to be expected because they perform similar tasks during this stage.

Application	Guievict Latency (msec)		Xmove Latency (msec)	
	Detach	Re-attach	Detach	Re-attach
Xterm	21,042	46	32	134
Xmame	21,100	55	857	96
Emacs	21,198	148	45	230
Ghostview	21,655	379	315	307
Netscape	21,655	667	362	432

Figure 4.7: Average detach and re-attach latency.

Stage	Time (usec)
Font List	21,052,039
Pixmaps	562,205
Windows	31,824
Fonts	6,986
Graphics Contexts	1,977
Cursors	113
Colormaps	63

Figure 4.8: Breakdown of detach latency for Netscape.

4.3.2 Interactive Overhead

To measure the impact of guimux and xmove on interactive response we created a small application that repeatedly sends a request of minimal size (8 bytes) to the window server and waits for a reply of minimal size (32 bytes). We measured the average time for 1000 round

trips for the application by itself, after it has been detached and re-attached with guievict, and through xmove. The results are reported in Figure 4.9.

Guievict and xmove have a measurable impact on the round trip time, caused by the overhead of redirecting the window system communication through a proxy. Since the overhead is less than a millisecond, it should not ever be perceptible to users.

System	Latency (usec)
None	70
Guievict	107
Xmove	133

Figure 4.9: Average round trip time for a minimal X protocol request and reply.

4.4 Security

Guievict introduces three issues related to the security of X window applications and servers.

First, the owner of the application process must be able to control who is able to migrate or replicate its GUI. The policy we have implemented is to allow only the owner of a process to perform guievict operations on the process. This policy is enforced by two mechanisms. First, the standard operating system protection prevents one user from modifying a process of another user, which implies that a process can only be hijacked by its owner. Second, the guievict library authenticates the messages it receives from the guievictor. It uses the credential passing mechanism of Unix domain sockets to ensure that the sender of a message is the same user who owns the application process. These mechanisms suffice to protect an application process from ordinary users, but not, of course, from the superuser of the execution host.

Second, the guievict server extension should not weaken the security of the X window system, a goal we believe we have met. Since the GetResources request only returns information about the resources of the application that issues the request, it cannot be used to learn about the resources of another application. Although a man-in-the-middle attack could be staged to inject a GetResource request in another application's connection to the window system, the information that would be revealed could also be obtained from passive eavesdropping on the connection, an existing X windows vulnerability [24]. The defense, then and now, is to encrypt the connection.

Third, the owner of a desktop host must authorize guievict to re-attach a GUI to their display. Most X server access control mechanisms (such as MIT-MAGIC-COOKIE) require an authorized application to possess a server-generated capability that it can present to the server when it establishes a connection. This capability gives its possessor complete access to the X server. The desktop owner must have a secure way to transfer the capability to the guievict user, and they must trust the guievict user not to abuse the access to the server. Guievict does not provide capability transfer mechanisms and it does not change the access control policies of the X server. These issues are trivial in migration scenarios, where the desktop user and the guievict user are usually the same, since the user can transfer the capability when they log in to the execution host to run the guievictor. However, these issues must be revisited in any GUI replication system built over guievict's replication mechanism.

Chapter 5

File I/O

Ideally, a moving application should be able to access its files at any time during its execution. Sustaining this illusion requires various strategies for preserving access to files. Movement may force an application to change its method for accessing a file system. It may also make access to a file system impossible, unreliable, or slow, prompting users to copy their files to another file system. Even a simple mobility scenario may involve the use of local file systems, network file systems, remote I/O protocols, and file caches (see Figure 5.1).

We have designed a system that performs these adaptations automatically and transparently on behalf of an application. It presents an unchanging file name space to a moving application, while dynamically changing the location of files and binding of file names to fit the current context of the application. A major challenge in designing such a system is maintaining access to the various file systems used by the application and the policies for switching and copying among them as the application context changes.

Our system is centered around a new name space composition language, called *flac* (*file access*), that we designed to manage these issues. The goal of flac is to represent the dynamic

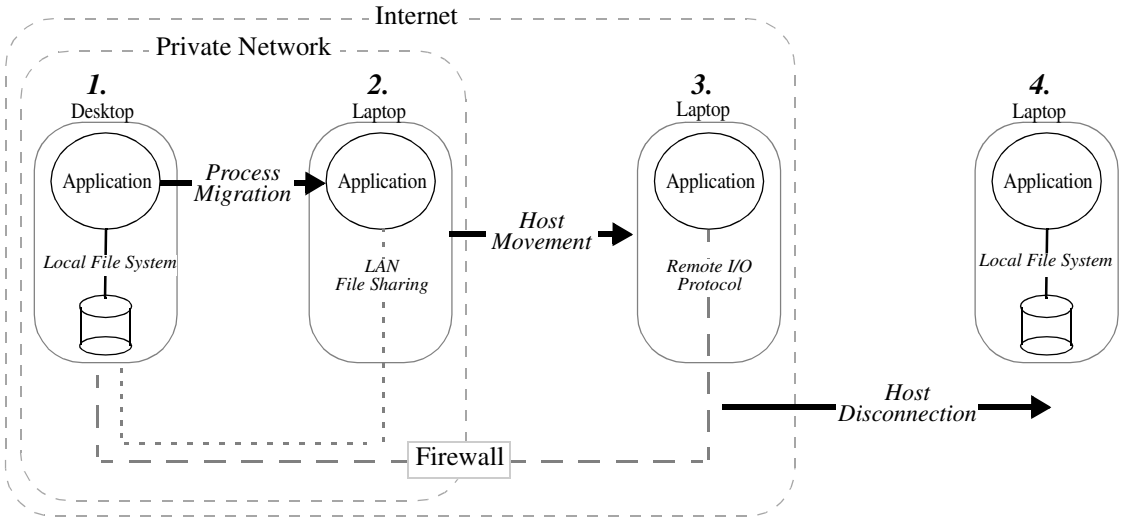

Figure 5.1: File access methods change with application context.
The application (1) starts on the host serving its files, and accesses them directly from the local disk, (2) migrates to a nearby host, and switches to local area network file system, (3) moves with its host outside the local network, and switches to a remote I/O protocol through a tunnel to the file server, and (4) is disconnected from the network, and switches to a local cache of recently accessed files.

behavior of an application's name space with simple, concise expressions. Flac has a descriptive role: it allows us to describe and compare different file name space semantics. We have identified a small set of basic composition operators that concisely capture the semantics of a broad variety of new and old file name space semantics. It also has a prescriptive role: it allows us to automatically generate implementations of new name space semantics via a system that translates flac specifications into code.

Flac is based on the following concepts (see Figure 5.2):

- File systems are abstractly represented as *services*. *Primitive service* types represent methods for accessing file systems, and are instantiated to represent a way to access a specific file system.
- Operators called *name space combinators* operate on services, yielding new ones. We define four types of combinators, each designed to capture an essential form of dynamic name space behavior. For example, the `transfer` operator expresses file copying between

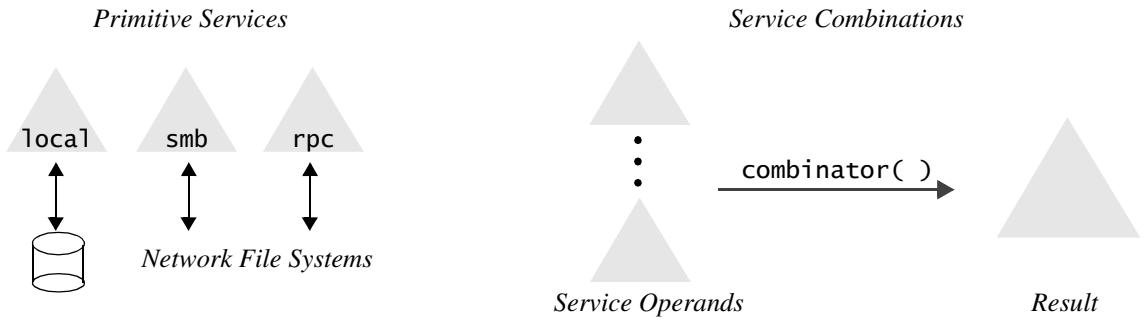

Figure 5.2: Flac abstractions.

two services, and the `select` operator expresses the dynamic selection of a service based on the context of the application.

- A particular set of file name space semantics is expressed in a *flac specification*, which has the form of an expression containing name space combinators and primitive service instances.

We have implemented a prototype *flac run-time* to gain experience with the language and to provide file access service to mobile applications. The job of a flac run-time is to implement the name space prescribed by a flac specification for an application. The run-time binds instances of primitive services to actual file systems in the user's environment, intercepts file I/O operations performed by the application, interprets them according to the flac specification, and passes the resulting operations to the real file systems identified by instances of primitive services in the specification. We use our prototype to provide file access for ordinary applications that is uninterrupted across host mobility, process movement, and disconnection. This is a more comprehensive range of mobility than has been shown by any previous system. Unlike previous systems, our prototype is implemented entirely at user level, transparent to application, and does not require privileged access or cooperation from the administrator of file systems or host machines.

The remainder of this chapter presents an overview of the flac language (Section 5.1), a series of examples demonstrating its descriptive power (Section 5.2), and a discussion (Section 5.3) and evaluation (Section 5.4) of the prototype flac run-time.

5.1 The Language

We describe, through a series of examples, services, specifications, and each of the flac combinators.

5.1.1 Services and Specification

A service represents a tree of file names, a mapping from these names to files, and a set of operations upon the files such as read, write, create, and remove. Perhaps the simplest example of a service is the local file system of the machine on which an application is running. This name space is described in flac as:

```
root local( )
```

The `root` keyword, which occurs exactly once in every flac specification, declares that the service expression to its right is the name space described by the specification. In this case, the service expression is an instance of the `local` service, a primitive service that we pre-defined to be the local file system of the current machine.

Flac can be extended to support different types of primitive file services. For our examples we use two other primitive services types: the `smb` service, which provides access to the Windows local network file system [45], and the `rpc` service, a remote I/O service similar to the Condor remote I/O protocol that provides remote access to a file system over the Internet [40].

Primitive services are instantiated by listing a set of attribute-value pairs that describe the attributes of the service instance. Some services, such as `local`, are instantiated with no attributes. The `smb` service has one attribute, `share`, that names a file server on the local network and one of its exported file systems. The `rpc` service has one attribute, `host`, that names the host name of the machine serving a file system accessible by the rpc protocol. The number and names of attributes and the syntax of their values are determined by the implementor of each primitive service type.

The following specification describes a name space comprised of the Windows share named `root` served by the host `desktop`. It could be used to describe the name space of the application in Step 2 of Figure 5.1.

```
L = smb(share="\\desktop\root")
root L
```

Flac supports the binding of identifiers to service expressions to make flac specifications easier to read and write. In the example above, the identifier `L` is bound to the `smb` instance that follows it. The reference to `L` has the same meaning as substituting the `smb` instance expression in the first line.

5.1.2 Path operations

Flac uses Unix-like path syntax in which path components are separated by slashes. Three flac operators combine a service and a path into a new service (see Figure 5.3). The `subtree` operator yields a service representing the sub-tree of its service operand rooted at the path. The `cuttree` operator complements `subtree` by yielding its service operand minus the sub-tree selected by `subtree`. The `extendtree` operator prefixes each name in a service with the path operand. The following sections contain examples of the use of these operators.

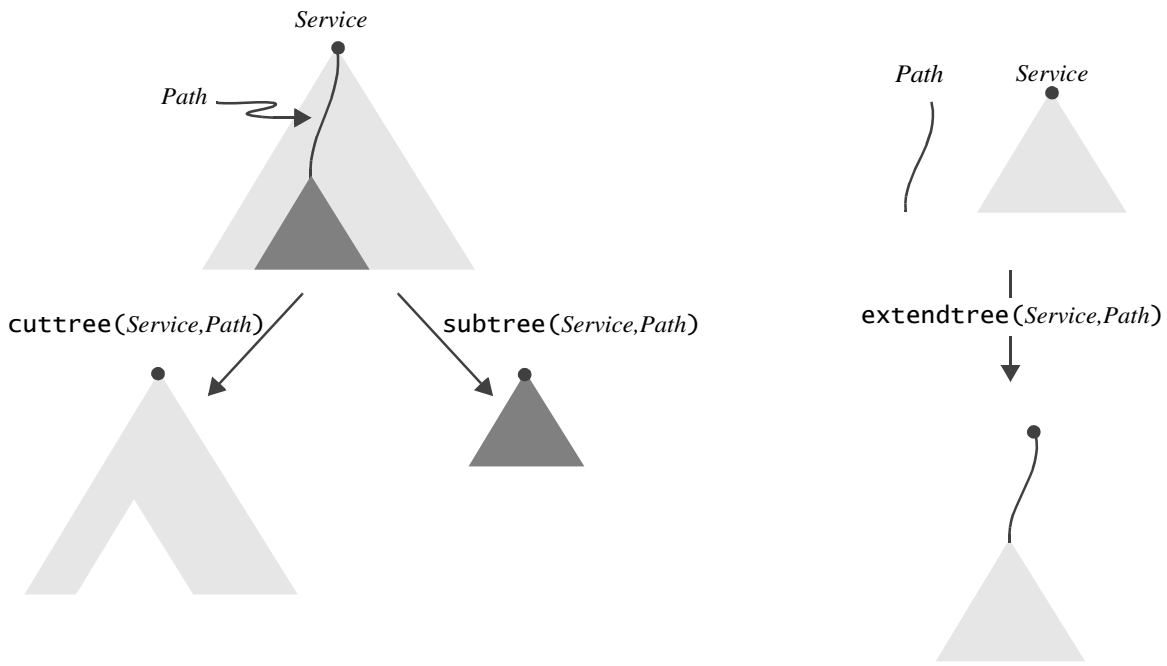

Figure 5.3: Path operators.

5.1.3 Overlay combinator

The `overlay` combinator combines two services into a single service in which names from both services are visible. Conceptually, the first service is layered over the second service. Names that occur in one service but not the other are visible in the resulting service. If a name occurs in both services, the occurrence in the first occuldes the occurrence in the second.

The `overlay` combinator completes the functionality of the path operators by establishing the identity relationship shown in Figure 5.4. One feature of this relationship is that the overlay and path operators together form a basis for describing conventional operating system name space operators. For example, the *mount* operator in Unix-based operating systems attaches a new file service at the end of a path, called a *mount point*, in the current file name space. The operator can be expressed in flac as:

```
T = overlay(extendtree(N,P),cuttree(S,P))
```

where `S` is the name space before the mount, `P` is the path to the mount point, `N` is the name space to be mounted, and `T` is the resulting name space.

Several systems, including some variants of BSD Unix [53] and the 3-D file system [38], have a *union* mount operator, a variant of the mount operator that lays the mounted name space over (or under) the name space at the mount point, instead of replacing it. This operator can be expressed as a mount operator that does not cut out the existing part of the tree at the mount point. This expression lays the new service over the existing name space

```
T = overlay(extendtree(N,P),S)
```

while, with the operands transposed, this expression lays the new service *under* the existing name space

```
T = overlay(S,extendtree(N,P)).
```

These union operator semantics provide a *deep* union, in which name spaces are overlayed at all levels from their root to leaves. In contrast, Plan 9 provides a union operator that has *shallow* union semantics, in which only first level names below the root are overlayed. It is not possible to emulate these semantics generally in flac, although they can be reproduced for specific paths by applying the `cuttree` operator. Although we would like to be able to describe the semantics of all existing name space operators, Plan 9 architects have commented that a mount and deep union operator should be sufficient, and that there is no application that depends on shallow semantics [62].

5.1.4 Select Combinator

The `select` combinator combines two or more services into a single service in which operations are forwarded to one selected service based on the context of the application at the

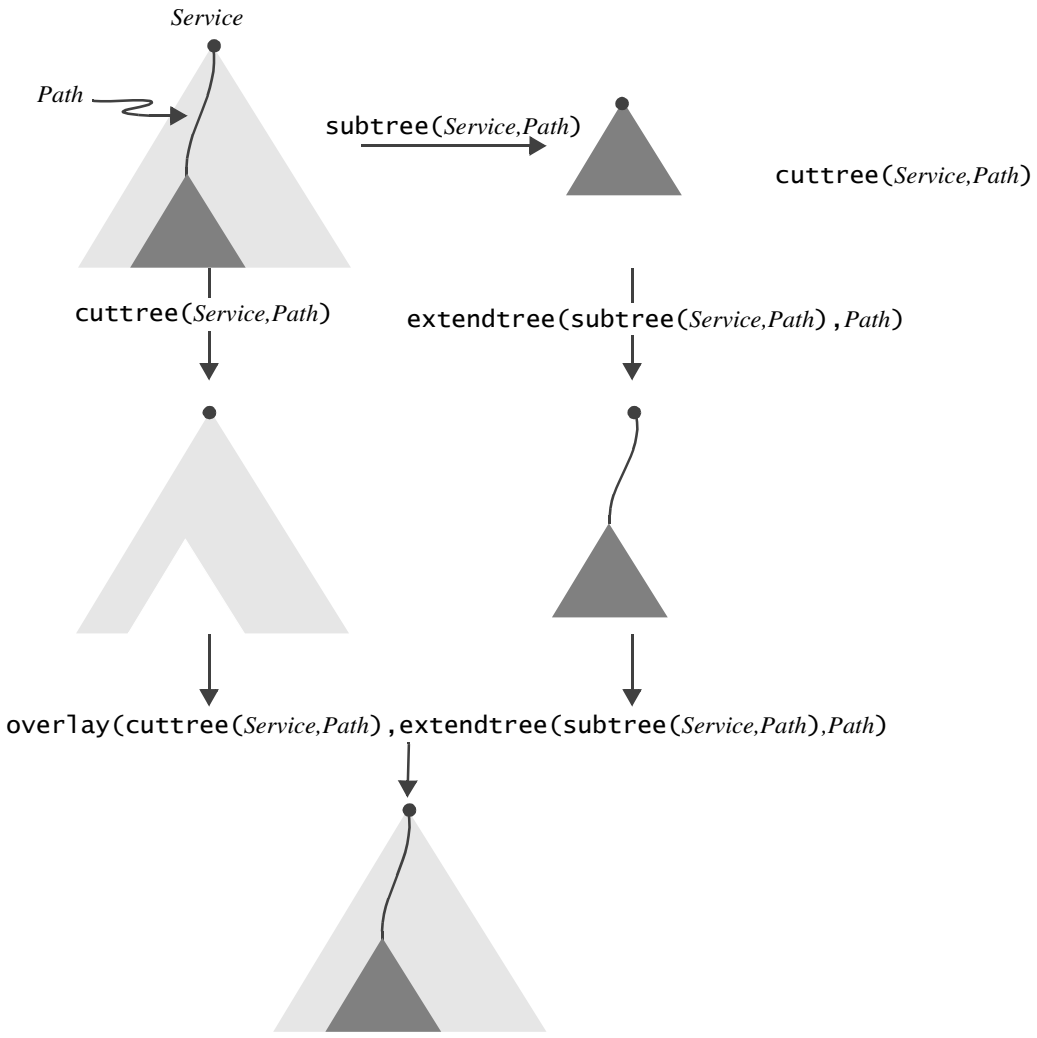

Figure 5.4: Identity relationship of overlay and path operators.
The order of the operands give to `overlay` *does not matter, since the operands do not have any names in common.*

time it issues the operation. For example, this specification supports the movement of the application in Figure 5.1 back and forth between Step 1, where is uses the `local` service, and Step 2, where it uses the `smb` service:

```
L = local( )
P = smb(share="\\desktop\root"))
root select(location,desktop:L,else:P)
```

The first operand to `select` identifies the type of context information that we use to select. We define three types: `location`, which looks at the current location of the application process; `operation`, which looks at the operation that the application process is currently performing; and `env`, which looks at values of the environment variables in the application process. Selection on a combination of context types is expressed by nested `select` expressions.

The remaining operands are two or more *pattern*`:`*service* pairs. Starting with the first and proceeding in order, each pattern in this list is compared to the context of the application. The operation is forwarded to the first service whose pattern matches. If no patterns match, the operation fails. A special optional pattern, `else`, matches when no other pattern matches.

The syntax and meaning of patterns depends on the context type. Patterns for the `location` type are strings that are compared with the name(s) of the machine on which the application is running. They can be DNS host names or textual IP addresses and can contain wildcards to match a set of host names (such as `*.cs.wisc.edu` to match any host in our department) or set of IP addresses (such as `192.168.0.0/16` to match any IP address beginning with `192.168`). For example, the following specification extends the previous example to support the application's movement to and from Step 3 of Figure 5.1 (where it uses the `rpc` service):

```
L = local( )
P = smb(share="\\desktop\root"))
R = rpc(host="desktop.my.net")
root select(location,desktop:L,192.168.0.0/16:P,else:R)
```

Patterns for the `operation` type match abstract descriptions of the file operation that the application is currently performing: `read` matches when the application is opening a file for

reading; `write` matches opens for writing; `readwrite` matches opens for reading and writing; `create` matches file creation; and `remove` matches file removal.

Patterns for the `env` type have the form `VAR=VAL`, and match when the environment variable named `VAR` has the value `VAL` in the application context.

5.1.5 Transfer Combinators

Strategies for maintaining access to remote files during periods of disconnection and for improving access performance over slow links often involve copying files to a closer filesystem. The `transfer` combinator is used to describe file transfer semantics between two services: a reliable one called the *primary* (such as the local file system), and a possibly unreliable one called the *secondary* (such as a remote file system).

The transfer combinator ties open or close operations on files in the primary service to file transfer operations from or to the secondary service. It supports copying in both directions. In `pull` mode, when the application opens a file, the secondary service is checked for a more recent copy of the file and, if it is or if the primary service does not have the file, the file is copied to the primary service. In `push` mode, when the application closes a file, the file is copied to the secondary service if it does not exist on the secondary or if it has been updated. In both modes, directories to contain the copied file are created on the destination service as necessary.

We want the application to be able to perform operations on files on the primary service even when the secondary service is inaccessible. We define three modifiers to the `push` and `pull` modes: `FAIL` causes the operation to fail if the secondary service is not available, `BLOCK` blocks the operation until the secondary service is available; and `IGNORE` causes the operation

to proceed. In `pull` mode, `IGNORE` allows the copy of the file in the primary to be opened without checking for a more recent version in the secondary service. In `push` mode, the file close operation completes with any updates written to the primary service, and the transfer of any updates is deferred. There is no guarantee when or if the updates will be transferred.

The `IGNORE` modifier creates the possibility of an application seeing inconsistent views of a file that it is exclusively updating. For example, an application running on a disconnected laptop may update a local copy of a file provided by the `transfer` combinator in `IGNORE` mode, and then be checkpointed in anticipation of being moved. If it next accesses the file directly from the secondary service, it is not guaranteed to see its previous updates. We solve this problem in our flac run-time implementation by enforcing an ordering property on process movement: a process may not be moved to a new context until its deferred updates are processed.

The primary and secondary services can also become inconsistent when the primary or secondary service is accessed by another program. We do not provide additional consistency semantics in the language to resolve this issue. Such features would require services to provide mechanisms for consistency such as locks or callbacks that we cannot assume are universally available. Specification writers can add stronger consistency to their specification by adding instances of services that provide consistency functionality. We have not experimented with these strategies, but two models are feasible. They can use existing file systems, such as Coda [37] or Ficus [51], that manage consistency between distributed caches or replicas of a file system. In this case, a single primitive service representing these file system replaces the use of the transfer combinator. Alternately, they can interpose a service between the `transfer` combinator and either or both the primary and secondary services that

intercepts file operations and triggers a separate consistency mechanism (such as the Unison file synchronizer [5]).

The syntax of the transfer combinator is

`transfer(`*primary*`,`*secondary*`,`*mode*`,`*pullmod*`,`*pushmod*`)`

where *primary* and *secondary* are the services, *mode* is `push`, `pull`, or `pushpull` (to specify both modes), *pullmod* is the modifier for the `pull` mode or `NONE` if the pull mode is not specified, and *pushmod* is the same for the push mode.

The transfer combinator can be used to maintain an off-line cache of file contents similar to that collected in Coda hoards [37] or the off-line file mechanisms provided by OS X [3] and MS Windows [45]. For example, the following specification extends our specification of Figure 5.1 to support the disconnected operation of Step 4.

```
SRC = select(location,
             192.168.0.0/16:smb(share="\\desktop\root"),else:rpc(host="desktop"))
CACHE = subtree(local( ),/cache)
XFER = transfer(CACHE,SRC,pushpull,IGNORE,IGNORE)
root select(location,desktop:local( ),else:XFER)
```

When the application is running on the laptop, copies of the files it accesses are stored in a directory on the local file system, and any updates the application makes to these files are copied back to the file system on the desktop host. The application can continue to access these files if the laptop becomes disconnected, since `IGNORE` permits read and write access to the local copy during disconnection.

5.2 Additional Examples

The previous section showed that flac can be used to specify functionality that is similar to mount and union name space operators in existing operating systems, remote file access strategies such as those used in Condor [40], and off-line file access provided by research

systems such as Coda [37] and commodity operating systems such as MS Windows [45] and OS X [3]. We give additional examples of flac descriptions of file access functionality. These examples demonstrate that flac can concisely describe, in a standard form, a variety of file name space semantics.

5.2.1 Replica selection

Several systems (such as Ficus [51], Fluid Replication [36], and Bayou [56]) provide file access for mobile clients using distributed file system replicas. Clients of these systems access files from the replica server closest to their current location. The system manages consistency among replicas on behalf of applications.

A flac specification for such location-based for replica selection can be written as:

```
R1 = rpc(host="replica1")
R2 = rpc(host="replica2")
R3 = rpc(host="replica3")
root select(location,(L1:R1 L2:R2 L3:R3))
```

where `L1,L2,L3` represent different locations and `R1,R2,R3` represent different replica servers. Structurally, this specification is no different from the ones we use in Section 5.1.4 for choosing among different methods for accessing the same file system. The difference is that the services here represent different replicas of a file system rather than different ways to access the same file system. When consistency is maintained by the file system, the difference is irrelevant.

Some users maintain consistency of replicated file systems manually. For example, a user may keep copies of music files on their home and office computers for fast access from either location. As the user adds files from each location but forgets to copy them, each replica may have copies of files that are not in the other replica. Here is a flac specification describing an

alternative in which a flac run-time automatically manages the separate name spaces for the user by merging the file systems into a single name space:

```
HERE = subtree(local( ),/music)
THERE = subtree(select(location,office:rpc(host="home")
                               home:rpc(host="office"))
                /music)
UNIFIED = transfer(HERE,THERE,pull,IGNORE,NONE)
root overlay(extendtree(UNIFIED,/music),local( ))
```

The user can add new music files to either their home or office computer. These files will be transparently copied to the other computer the first time they are accessed from it.

5.2.2 Importing Environments

Name space operators have other uses beside mobility. One typical use of the union operator by Plan 9 users is to substitute different versions of libraries or compilers when building software [57]. The operator changes the objects to which names are mapped, not the names, so no other modifications (such as changing pathnames in build scripts) are needed to build the software with the alternate files. This functionality can be described with specifications such as

```
BIN = extendtree(overlay(subtree(local( ),/home/user/bin),subtree(local( ),/bin)),
                 /bin)
LIB = extendtree(overlay(subtree(local( ),/home/user/lib),subtree(local( ),/lib)),
                 /lib)
root overlay(BIN,overlay(LIB,local( )))
```

In a similar problem, different vendors of the nominally same operating system sometimes release different versions of compilers and libraries (this is common among Linux distributions, for example). Developers should test their software on each vendor's version to detect potential version-dependent bugs. This process can be time-consuming and tedious: developers must maintain separate machines for each operating system version, and arrange

for the software source tree to be copied to each machine and for modifications to be merged back into the master copy.

```
      Test Machine Name Space
RH = extendtree(subtree(rpc(host="A"),
                        /home)),
                /home
root overlay(RH,local( ))
```

```
 Developer Machine Name Space
LH = extendtree(subtree(local( ),
                        /home)),
                /home)
RM = rpc(host="B")
root overlay(LH,RM)
```

Figure 5.5: Two ways to test software.
A is the developer's machine; B is a host running version B of the operating system. On the left, developer home directories, where software to be tested is stored, are imported. On the right, the home directories are layered over B's imported file system.

Flac could help by importing the source tree on each test machine from a remote server (see Figure 5.5, left), an arrangement that entails less configuration and maintenance than conventional alternatives such as a distributed file system on each machine or version control software. However, we suggest the *inverse* approach of importing the local file system of a test machine to the developer's workstation (see Figure 5.5, right). One advantage of this arrangement is that developers can build and run tests on their local machines, not test machines; this enables older, slower machines to be recycled into test platforms.

Two extensions to the semantics in Figure 5.5 can provide other advantages. First, we can extend it to support any number of test platforms. A single script can control the testing on multiple platforms from the same machine. Selection of the test machine can be implemented by the `environ` context type (see Figure 5.6).

```
LH = extendtree(subtree(local( ),
                        /home)),
                /home)
root overlay(LH, select(env,VER=B:rpc(host=B),
                            VER=C:rpc(host=C),
                               VER=D:rpc(host=D)))
```

```
VERSIONS = B C D
foreach v in $VERSIONS
    setenv VER $v
    echo 'version: ' $v
    make all test clean
endfor
```

Figure 5.6: Testing on several operating system versions.
The flac specification on the left selects a version based on the value of the environment variable VER. The shell script on the right, when in the name space of the specification, builds and tests the software on each version.

```
LH = extendtree(subtree(local( ),/home)),/home)
Z = subtree(local( ),/os-menagerie)
RM = select(env,VER=B:transfer(subtree(Z,/B),rpc(host="B"),pull,IGNORE,NONE)
                VER=C:transfer(subtree(Z,/C),rpc(host="C"),pull,IGNORE,NONE)
                VER=D:transfer(subtree(Z,/D),rpc(host="D"),pull,IGNORE,NONE))
root overlay(LH,RM)
```

Figure 5.7: Collecting operating system versions.
A local collection of the files needed to build the software on each version of the operating system is automatically maintained in `/os-menagerie`.

Second, the application of transfer combinators may reduce dependence on test machines. The specification in Figure 5.7 generates a local copy of precisely the slice of the file system of the test machine that is needed for testing. Assuming that future versions of the software being tested do not introduce new dependencies, the test machine is no longer needed. In cases where changes create new dependencies, the use of `IGNORE` will cause the operations to fail, and the developer can bring a new test environment and copy the missing files. This use of copying also reduces the overhead of remote access to frequently accessed files such as header files and compiler executables.

5.2.3 Copy-on-write unions

The union operators in the 3-D filesystem [38] and BSD Unix [53] have additional *copy-on-write* semantics that we did not describe in Section 5.1.3. This feature is used to provide the illusion of having write access to a read-only sub-tree of the name space. A writable file system is union mounted over the read-only sub-tree. Applications can open a file in this union for writing, but instead of modifying the original version of the file, the file is copied to the writable file system and the update is applied to that copy. Subsequent opens of the same file return the updated version, since the writable file system is layered over the read-only sub-tree. A typical application of this functionality is to experiment with modifications to a source code tree without making permanent changes to it.

We can describe this type of union mount in flac as follows.

```
R = subtree(S,P)
U = select(operation,write:W,
                      readwrite:transfer(W,R,pull,BLOCK,NONE),
                        else:overlay(W,R))
T = overlay(extendtree(U,P),S)
```

where `S` is the name space before the mount, `P` is the path to the read-only sub-tree, W is the writable name space to overlay, and `T` is the resulting name space.

5.3 Flac Run-Time Implementation

We give an overview of the architecture of our prototype for implementing flac, then discuss the major issues in its design and implementation. These issues include context tracking, responses to location changes, implementation of transfer combinators, process migration, and interception of application operations.

The prototype, written for Linux, has three components (see Figure 5.8). An application process is linked with the *flac run-time library*, which traps the application's I/O operations and forwards them to the *flac daemon*, a separate process running on the same host as the application process. The flac daemon implements the flac specification. It sends file operations to primitive *service processes,* which communicate with the file systems represented by service instances. One service process exists for each primitive service instance in the flac specification. It also receives updates about the location of the host on which the application is currently running from the *location monitor*, a separate daemon that runs on every host that the application visits.

The prototype is started for an application program by invoking a command launcher called `flac` that executes the application program re-linked with the run-time library. During its initialization, the run-time library opens the file containing the flac specification (named in

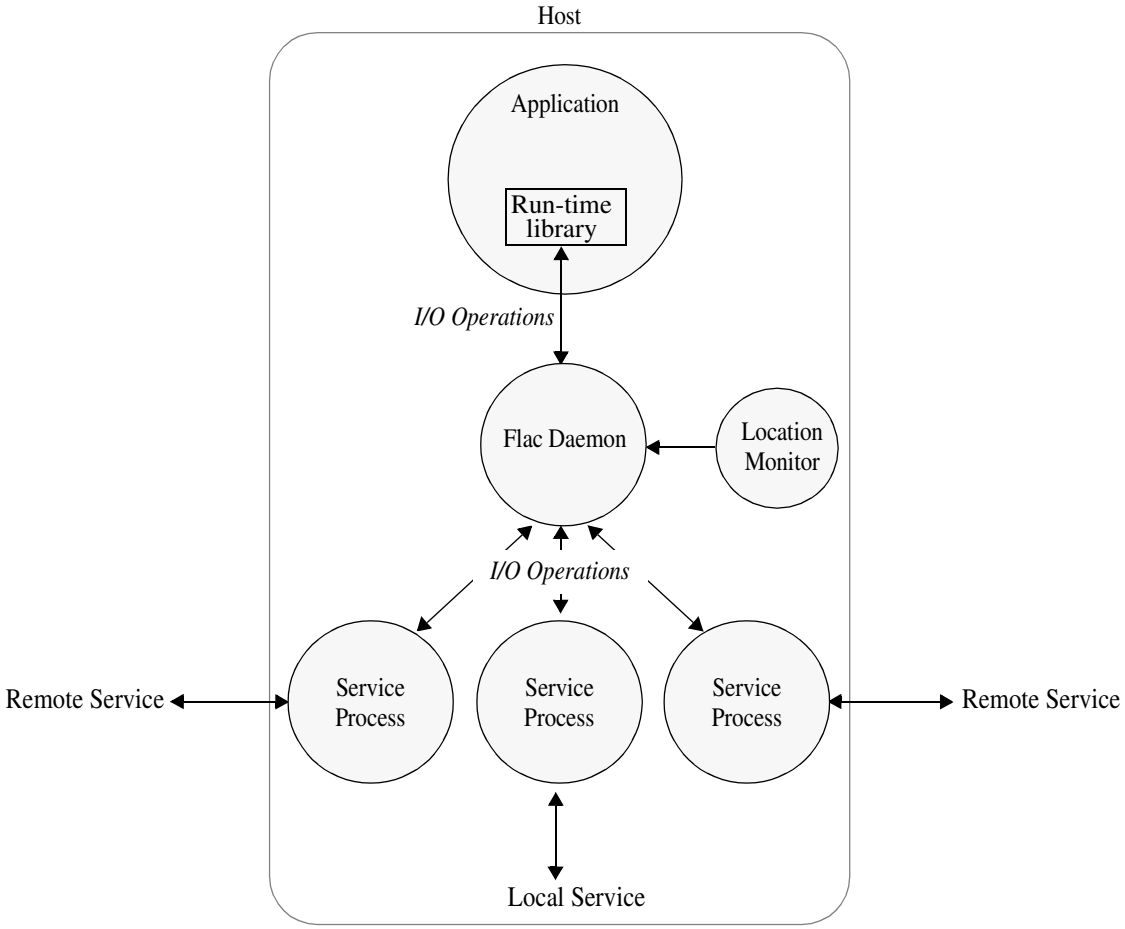

Figure 5.8: Architecture of prototype flac run-time.

an environment variable or command-line argument to `flac`), starts the flac daemon, and passes to it the flac specification and a description of the initial context of the application process. The flac daemon parses the specification, locates a shared library for each primitive service type that occurs in the specification, and calls, for each instance of the service, the library's *start* function, passing the attribute-value pairs of the instance. The start function creates the service process, which connects to the service.

The run-time library intercepts calls by application code to functions that implement I/O operations (system call stubs in the C library), and forwards them as I/O request messages to the daemon. Inside the daemon, each request is interpreted by traversing the relevant nodes in

the combinator expression tree. At each combinator node a message dispatch function implements the semantics of the combinator, typically by re-writing the message or generating new messages. At the primitive instance nodes, the request is forwarded to the associated service processes. The service processes invokes the corresponding I/O operations on the file system they represent, converts the results to a response message, and returns the response to the daemon. The daemon collects responses along the reverse path taken by the original request, and sends a response to the run-time library, where it is converted to an I/O operation return value or error that is returned to the application code.

Most of the message dispatch functions in the daemon have straightforward implementations. For example, the dispatch function for the `subtree` operator looks for requests that involve file names (such as open). It re-writes these messages by prepending its path operand to the file name argument, and passes them to the underlying service. Other messages are passed unmodified.

5.3.1 Tracking Context

Each context type (`location`, `env`, and `operation`) is tracked by a separate mechanism. For `location`, we identify changes to network interfaces as the application's location changes. The location monitor on each host monitors the status of network interfaces on the host and asynchronously notifies the flac daemon when a network device is added, removed, or re-configured (such as when a wireless device comes in or out of range of a network). When the application moves to a different host, it creates a new flac daemon (initialized to restore the previous I/O state, as described in Section 5.3.4), and contacts the location daemon on the host.

The application's environment is sent to the daemon during initialization of the run-time library. Updates to the environment could be tracked by interposing code on the function (`setenv`) that the application calls to change its environment, but we have not needed this functionality.

For the `operation` type, the flac daemon infers the operation that the application is performing from the last I/O request it received from the run-time library.

5.3.2 Responding to Location Changes

Location changes may provoke three types of responses: service re-starting, open file re-opening, and deferred update flushing.

First, services need to be re-started when the application moves to a new host or when movement enables access to a previously unavailable service. Each primitive service's shared library can register a callback to be notified of location changes; some primitive services, such as `local`, use this callback to determine when to start a new instance process. Primitive services for remote TCP-based services (such as `rpc`) can use rocks to avoid special restart procedures.

Second, files need to be re-opened whenever the service for that file changes or is re-started. Each time a location change occurs, the daemon examines each open file, determines whether the change affects the file and, if so, invokes the re-open operation. The re-open operation regenerates the state of the file from a record that the daemon maintains for each open file. For all types of files, this record contains the name of the file and the I/O mode for which it was opened. The daemon re-opens the file by opening the same file name in the (possibly different) service in the same mode. If the file cannot be re-opened on the new

service, the daemon does not raise an error until the application attempts to access it. The error reporting is deferred because the location may change favorably before the next access.

The record of an open file contains additional information that depends on the type of file. For ordinary files, it contains the current file pointer. During re-opening, the file pointer is set to the old position. For device files (such as audio devices and terminals), the daemon maintains a log of all file control operations (such as `ioctl` or `fcntl`) that the application performs on the file. They are re-played when the file is re-opened to restore the state of the device (such as volume level or sample rate) to its previous setting. These logs tend to be short, comprising device initialization operations that were performed when the file was opened., However, if they become too long we could apply optimizations, similar to the log optimizations performed by Coda [37], to discard redundant or superseded operands. Other types of open files, such as named pipes, are not supported in the prototype.

Third, location changes that occur while there are pending `transfer` combinator updates are a hint that it may be possible to flush the updates. Each location change events triggers the flac daemon to identify and flush pending updates that can be sent from the new location.

5.3.3 Transfer Combinator

The message dispatch function for the `transfer` combinator requires two mechanisms. First, it requires a way to determine whether the secondary service is available. Each primitive service shared library provides an *isavailable* function that the transfer combinator can call to determine whether the service is currently available.

Second, it requires a way to compare file versions to decide when to copy a file from the secondary service instead of using a copy in the primary service. We use the file's last

modification time attribute as a version number, which is simple and file-system independent but forces us to assume that the clocks on the file servers are closely synchronized.

5.3.4 Process Migration

The decision to use a separate process for the flac daemon, rather than putting its functionality in the run-time library, was motivated by the complexities we faced with the design of the rocks library (see Section 3.3.2). In particular, a separate process simplifies the implementation of the daemon's asynchronous operations. Migration of the application process, however, is more complicated with this design, because the state of an additional process must be captured, moved, and regenerated.

The flac daemon exports an *I/O session*, analogous to a window session (see Chapter 4), that is a representation of the name space of the application and the names and state of files that the application has opened. Before being checkpointed for migration, the run-time library (in a callback invoked by our checkpoint library) requests the I/O session from the flac daemon, and then causes the flac daemon to exit. When restarting from its checkpoint on a new host, the run-time library (in another callback) starts a new flac daemon and passes to it the saved I/O session. The new daemon initializes itself to serve the name space and open files represented in the I/O session. Compared to the window sessions we capture in Chapter 4, I/O sessions are much simpler since we designed the flac daemon to export precisely what we want.

To preserve the consistency of transfer combinator services (exclusively) accessed by the application, the run-time library takes responsibility for flushing pending updates before migration, as discussed in Section 5.1.5. In the callback for process checkpointing, the run-

time library tells the flac daemon to flush its pending updates, and blocks the checkpoint operation until the daemon replies that it has successfully flushed.

5.3.5 Intercepting I/O operations

The run-time library intercepts I/O operations at library calls, using the same user-level, application-transparent interposition mechanism as the rocks library (see Section 3.3). New issues that are specific to implementing flac include `exec` calls, `mmap` calls, and multiplexed I/O.

5.3.5.1 Calls to `exec`

The `exec` system call replaces the address space of the calling process with the contents of an executable file. Under flac, this file may be located on a remote file system, but of course the `exec` call cannot be remotely executed in the flac daemon or a service process, since the purpose is to change the calling process. When the run-time library intercepts an `exec` call, it opens the requested executable through the flac daemon, copies its contents to a temporary file in the local file system, and then invokes the `exec` system call, substituting the name of the local copy for the original argument.

5.3.5.2 Calls to `mmap`

The `mmap` system call maps a part of the contents of an open file to the address space of the calling process. The operating system does not provide a mechanism to map virtual memory to remote files, so `mmap` requires special handling. Our approach is similar to our handling of `exec` calls: when the run-time library intercepts an `mmap` call, it copies the portion of the file to be mapped to a temporary file in local file system, opens the file, and invokes the `mmap` system call on the resulting file descriptor.

There are two issues. First, although `mmap` is typically used for read-only access to a file (such as shared library loading), it can optionally copy writes to the mapped memory back to the file. The prototype does not support this option, but it could be easily implemented as a deferred write-back of the temporary file that is triggered when the application unmaps the memory, invokes the `msync` system call to flush updates, or exits. Since the POSIX definition of `mmap` only guarantees that writes are copied back to the file when one of these events occurs [23], the consistency semantics of this implementation model should not violate those expected by any application that uses `mmap` to write files.

Second, sometimes a process calls `mmap` to map into its address space the data represented by a device file, such as the kernel memory device (`/dev/kmem`), which cannot be copied like ordinary files. Mapping devices is an uncommon operation, so the flac prototype does not support it but for one exception: `/dev/zero`, the device that many programs map to allocate zero-filed pages. The run-time library directs `mmap` calls on this device to the local `/dev/zero`; since zeroes are zeroes, remotely accessing this device would be senseless. Support for the mapping of other devices files would require a mechanism to map virtual memory to remote files, which we believe is difficult to implement efficiently from user space. We will revisit this issue if we find examples of applications that would benefit from its resolution.

5.3.5.3 Multiplexed I/O

Multiplexed I/O occurs when an application wants to perform I/O on two or more files. It is supported by system calls such as `poll` and `select` that return an indication of which members of a set of files can be read or written without blocking. The run-time library cannot

directly invoke these calls, since open files are managed by service processes. When the library intercepts a call to `poll` or `select`, it sends a series of *isready* messages to the flac daemon, one for each open file in the call's arguments, and then blocks waiting for a response. The daemon batches and forwards these messages to the corresponding service processes for the open files. The service processes respond when files are ready. The daemon passes the first response it gets to the run-time library, which returns to the application code the indication that the corresponding open file is ready.

5.4 Evaluation

The flac prototype is sufficiently operational to demonstrate the degree of mobility that the flac language can enable. We have used it to provide an unmodified text editor (GNU Emacs [73]) with uninterrupted access to the file systems of our home and office workstations. The editor can migrate across hosts, move with a host across networks, and operate while disconnected. Its file access mechanisms run as unprivileged user software that requires no unusual support from the operating system (such as a special file system driver) or cooperation from administrators. Most importantly, its operational behavior is derived from a concise description of name space semantics, which users can extend to accommodate new file system and execution locations.

Our environment (with names changed for clarity) is illustrated in Figure 5.9. In our home network, the workstation `home` serves its file system to its private local network as an `smb` service and across a NAT device (at public address `my.host.net`) to the internet as an `rpc` service. At work, our `office` workstation serves its file system to the internet as an `rpc` service. In addition, of course, the file systems are `local`ly accessible to programs running on

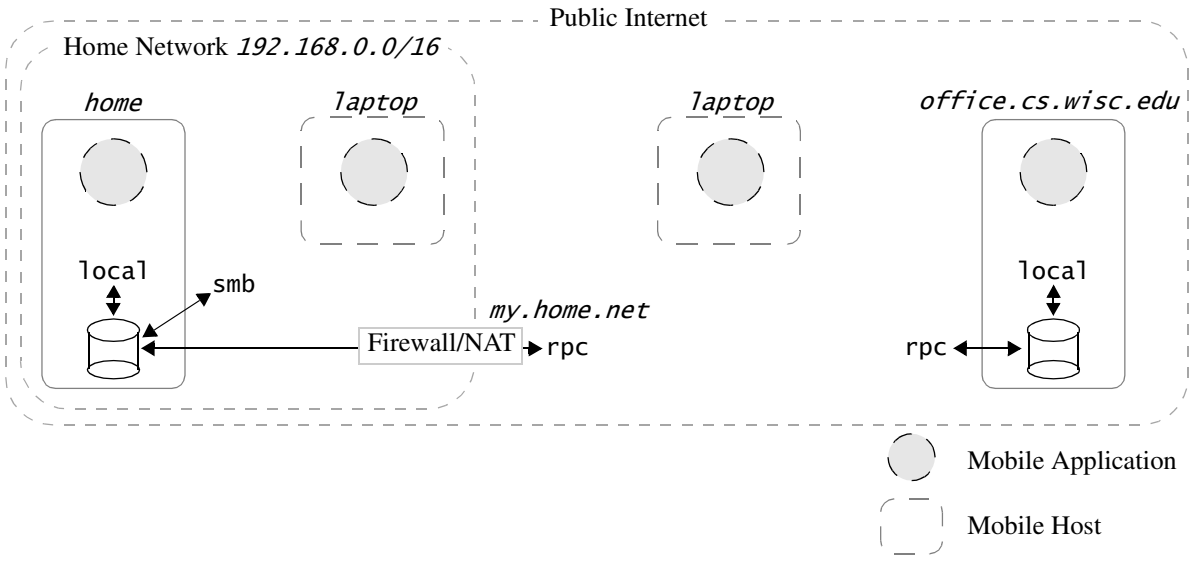

Figure 5.9: Environment of our flac experience.
The mobile application moves among the `home`*,* `office`*, and* `laptop` *hosts.* `Laptop` *itself moves between the home network and public internet, and sometimes operates disconnected.*

these hosts. We also have a `laptop` that moves between the home network and public internet. The `laptop` sometimes operates disconnected. Our editor can perform any sequence of host and process movements around this environment. We use guievict, as described in Chapter 4, to move its graphical user interface among hosts.

The flac specification in Figure 5.10 completely describes the semantics of the name space for our mobile editor. The name space, for all locations, looks like the `office` file system with the `home` file system mounted at `/n/home`. The most interesting aspects of the specification are the two uses of the `transfer` combinator. First, when the application runs on `laptop`, the files it accesses are copied from the home and office file systems to its local file system. Like the `transfer` combinator example in Section 5.1.5, the push and pull modifiers are set to `IGNORE` so that files are accessible while `laptop` is disconnected. Second, when running on `office`, files from the home file system are cached in the local file system, because data transfers in the upload direction from the home network are slow. However it

```
# access from home workstation
HOME-FROM-HOME = local( )
OFFICE-FROM-HOME = rpc(host="cumin.cs.wisc.edu")

# access from office workstation
OCACHE = subtree(local( ),/u/z/a/zandy/cache)
HOME-FROM-OFFICE = transfer(subtree(OCACHE,/home),
                            rpc(host="my.home.net"),
                            pushpull, BLOCK, BLOCK)
OFFICE-FROM-OFFICE = local( )

# access from laptop
LCACHE = subtree(local( ),/cache)
HOME-FROM-LAPTOP = transfer(subtree(LCACHE,/home),
                            select(location,
                                   192.168.0.0/16:smb(share="\\home\root"),
                                   else:rpc(host="my.home.net")),
                            pushpull, IGNORE, IGNORE)
OFFICE-FROM-LAPTOP = transfer(subtree(LCACHE,/office),
                              rpc(host="cumin.cs.wisc.edu"),
                              pushpull, IGNORE, IGNORE)

# root composition:
# the application sees the office file system
# with the home file system mounted on /n/home
HOME = select(location,
              home:HOME-FROM-HOME,
              office:HOME-FROM-OFFICE,
              laptop:HOME-FROM-LAPTOP)
OFFICE = select(location,
                home:OFFICE-FROM-HOME,
                office:OFFICE-FROM-OFFICE,
                laptop:OFFICE-FROM-LAPTOP)
root overlay(extendtree(HOME,/n/home), OFFICE)
```

Figure 5.10: Flac specification for Emacs.

uses more strict push and pull modifiers than those used on `laptop`, forcing I/O operations to block during rare periods of disconnection with the home network to avoid inconsistencies.

Overall, the system delivers the functionality we envisioned: an application follows our movement, enjoying transparent uninterrupted access to its files without modification or special infrastructure. The main weakness is that we have no consistency guarantees for files that, as discussed in Section 5.1.5, we update with Emacs running on the disconnected `laptop` while they are simultaneously updated by other programs on `home` or `office`. Since we are usually the only user of the files that we access with Emacs and we only update them with the

same instance of Emacs, this weakness is not a problem for this application. However, we will need to investigate the consistency options we suggested in Section 5.1.5 to support other types of file access patterns.

Another minor weakness is that this specification suffers (like other examples in this chapter) from redundant subexpressions that reduce its readability. These redundancies could be eliminated with a language feature for defining a macro or function; we expect to experiment with such a feature as we gain further experience with flac.

Chapter 6

Conclusion

Our goal has been to enable running applications to seamlessly follow their users. We have identified the barriers to application mobility that are unaddressed by previous work in process migration and mobile computing, and we have developed new techniques that overcome these barriers without requiring users to modify their applications or environments. This final chapter reviews our contributions and offers perspectives on the results.

6.1 Contributions

Our research enables running applications to follow their users as they move from machine to machine and as they move with their machines to different locations. We have shown that functionality for application mobility can satisfy three major technical goals:

- **Transparent to applications:** ordinary applications can be made mobile without re-programming the application source code or even re-linking the application binaries.
- **Independent movement:** users can move only the applications that they can and want to move.
- **No modification to infrastructure:** application mobility can be deployed entirely at the convenience of its users.

Achieving these goals allows us to provide application mobility service that is flexible and convenient for users of existing applications who work in environments with limited administrative support for mobility.

Our challenge was to overcome the absence of user-level process migration techniques that can handle the requirements of application mobility. These requirements include support for movement of resources external to the process abstraction — network connections, graphical user interfaces, and access to files — and tolerance of the conditions of their movement, such as moving to different administrative domains and being disconnected from remote resources for extended periods of time.

We addressed this challenge by defining three abstractions that, when taken together, complete the process migration functionality necessary for application mobility:

- **Reliable Network Connections:** Network connections that automatically detect failures caused by movement, and that recover from these failures transparently to the applications that use them. Our new enhancement detection protocol enables the use of this abstraction to spread incrementally in the Internet without affecting compatibility with applications that do not yet support it. This protocol is a general-purpose solution to the problem of safely detecting, at user-level, the presence in a remote support for any type of network communication enhancement such as compression or encryption.
- **Window Sessions:** A transportable representation of the state of an application's graphical user interface that allows the user interface of a running application to be moved, either with or independently of the application process, from one display to another. This abstraction is sufficiently general to support the additional functionality of replicating user interfaces across multiple displays.
- **Mobile File Access:** Applications access files through a name space that appears static, but which changes its file access strategies as a process moves. Our flac language is a descriptive and prescriptive language for managing such strategies. Flac enables the

concise description of file access strategies in terms of the various services that provide access to name spaces, and name space composition operators that capture mobile file access semantics such as context-dependent selection of a file service. These same descriptions can be used to prescribe the run-time behavior of an application's file name space by handing them to a system, such as our prototype flac run-time, that implements the semantics of flac.

We have implemented systems supporting these abstractions. Our implementations demonstrate that these abstractions can be built entirely from user-level code with no application modifications and with generally imperceptible overhead. We have shown that these abstractions support the mobility of ordinary programs across administrative boundaries and over extended periods of disconnection.

6.2 Perspectives

We see several directions for further inquiry and development from our work. These directions include eliminating the remaining premeditative steps from our techniques, introducing judiciously selected support for mobility into existing infrastructure, and designing user interfaces for controlling application mobility functionality.

First, in each of our systems we attempted to avoid the need for premeditative user action related to mobility. We were most successful with GUI mobility: guievict users take no explicit actions until the moment they decide to move their interface to a new display. We previously achieved similar success with process hijacking [88], allowing users to checkpoint and migrate running programs (only those that do not use network connections or graphical user interfaces, and that have restricted types of file access) with no premeditative steps. However, our systems for both reliable network connections and mobile file access are designed to be started when the application process is started, and cannot be initialized once

the application has started to access network or file resources. We would like to investigate if this limitation could be overcome. The result would be an application mobility service that could be operated without requiring users to start their applications in special ways.

Second, one advantage of knowing how to solve application mobility problems at user level is that it provides insight into what essential functionality future operating systems, libraries, and window systems could provide to support this functionality. With our new insight, we can look for opportunities to add minimal, general-purpose functionality to our infrastructure that would reduce or eliminate the parts of user-level application mobility systems that are difficult to implement correctly, hard to maintain, or redundant. Our experience reinforces the arguments for some previously proposed functionality, including the ability to read the kernel-level buffers used by network protocols [4,52,84,89], and systems for code interposition [79] that hide the unportable aspects of interposition and that allow multiple interposed libraries to operate independently in the same application process.

An example of possible new functionality is the ability to inventory the instances of each type of system resource that have been allocated for an application. Such an interface should provide the identifier for each instance and a description of the state of the instance. The description of the state should be sufficiently rich to allow the creation of a copy of the instance that is indistinguishable, from the application's perspective, from the original.

Third, we have developed the essential mechanisms for application mobility, but we do not have an entirely frictionless system. The major missing piece is a better interface for users to select the applications, or the parts of applications, they want to move and to specify their destination. Today, users of our system face command-line tools whose arguments include process identifiers to identify programs and IP or DNS addresses to identify locations; while

these are natural for technical-minded users, they are probably too arcane for other users. A related interface problem is conveying the status of mobile applications that are blocked when, because of movement, needed resources are unavailable. Although our systems safely prevent applications from failing in these conditions, they do not offer simple explanations about the current problem or its possible remedies. It is not clear how easy it is to translate the technical conditions into simple explanations.

References

[1] H.M. Abdel-Wahab and M.A. Feit. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. *IEEE TriCom '91: Communications for Distributed Applications and Systems*. Chapel Hill, NC, USA, April 1991, pp. 159-167.

[2] H. Abdel-Wahab and K. Jeffay. Issues, Problems and Solutions in Sharing X Clients on Multiple Displays. *Internetworking – Research and Practice* **5**, 1, March 1994, pp. 1-15.

[3] Apple Computer, Inc. OS X Manual. February 2003.

[4] A. Bakre and B.R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. *15th International Conference on Distributed Computing Systems (ICDCS '95)*, Vancouver, British Columbia, Canada, May 1995.

[5] S. Balasubramaniam and B. C. Pierce. What is a File Synchronizer? *4th ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, Dallas, TX, October 1998.

[6] J.E. Baldeschwieler, T. Gutekunst, B. Plattner. A Survey of X Protocol Multiplexors. *ACM SIGCOMM Computer Communication Review* **23**, 2, April 1993.

[7] D.A. Bandel. A NATural Progression. *Linux Journal* **98**, June 2002.

[8] J. Bazik. XMX – An X Protocol Multiplexor. http://www.cs.brown.edu/software/xmx.

[9] C. Bormann and G. Hoffmann. Xmc and Xy – Scalable Window Sharing and Mobility. *8th Annual X Technical Conference*, January 1994.

[10] R.T. Braden. Requirements for Internet Hosts – Applications and Support. *Internet Request for Comments* RFC 1122, October 1989.

[11] D.R. Brownbridge, L.F. Marshall, and B. Randell. The Newcastle Connection, or, UNIXes of the World Unite! *Software Practice and Experience* **12**, 12, December 1982, pp. 1147-1162.

[12] B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications* **14**, 4 , Winter 2000, pp. 317-329.

[13] J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R.M. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. *Computing Systems* **8**, 2, Spring 1995.

[14] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global State of Distributed Systems. *ACM Transactions on Computer Systems* **3**, 1, February 1985.

[15] Y. Chen, J.S. Plank, and K. Li. CLIP: A Checkpointing Library for Intel Paragon. *SuperComputing* '97, San Jose, CA, 1997.

[16] P. Chen and B. Noble. When Virtual is Better Than Real. *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, May 2001.

[17] A Cobbs. Divert(4). *FreeBSD 4.7 Kernel Interfaces Manual*, June 1996.

[18] M. Crispin. Internet Message Access Protocol: Version 4rev1. *Internet Request for Comments* RFC 2060, December 1996.

[19] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience* **21**, 8, August 1991, pp. 757-785.

[20] R. Droms. Dynamic Host Configuration Protocol. *Internet Request for Comments* RFC 2131, March 1997.

[21] K. Egevang and P. Francis. The IP Network Address Translator (NAT). *Internet Request for Comments* RFC 1631, May 1994.

[22] P. Ferguson and D. Senie. Network Ingress Filtering. *Internet Request for Comments* RFC 2267, May 2000.

[23] B.O. Gallmeister. **POSIX.4: Programming for the Real World**. O'Reilly and Associates, Sebastopol, CA 1995.

[24] S. Garfinkel and G. Spafford. **Practical UNIX & Internet Security**, 2nd Edition. O'Reilly and Associates, Sebastopol, CA, April 1996.

[25] D. Garfinkel, B.C. Welti, T.W. Yip. HP SharedX: A Tool for Real-Time Collaboration. *Hewlett-Packard Journal* **45**, 2, April 1994, pp. 23-36.

[26] A. Geitz, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts, 1994.

[27] J. Gettys. *Private communication*. June 2003.

[28] J. Gettys. The Future is Coming: Where the X Window System Should Go. *2002 Usenix Annual Technical Conference (Freenix Track)*, Monterey, CA, June 2002, pp. 63-69.

[29] J. Gettys and K. Packard. The X Resize and Rotate Extension —RandR. *2001 Usenix Annual Technical Conference (Freenix Track)*, Boston, MA, June 2001.

[30] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing* **22**, 6, September 1996.

[31] T. Gutekunst, D. Bauer, G. Caronni, Hasan, and B. Plattner. A Distributed and Policy-Free General-Purpose Shared Window System. *IEEE/ACM Transactions on Networking* **3**, 1, Februrary 1995.

[32] D. Hendricks. A Filesystem For Software Development. *1990 Summer Usenix Technical Conference*, Anaheim, CA, June 1990, pp. 333-340.

[33] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems* **6**, 1, February 1988, pp. 51-81.

[34] O. Jones. Multidisplay Software in X: A Survey of Architectures. *The X Resource*, Issue 6, O'Reilly & Associates, Jan 1993, pp. 97-113.

[35] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *Internet Request for Comments* RFC 2401, November 1998.

[36] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. *USENIX Conference on File and Storage Technologies (FAST '02)*, January 2002, Monterey, CA.

[37] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems* **10**, 1, February 1992, pp. 3-25.

[38] D.G. Korn and E. Krell. A New Dimension for the Unix File System. *Software Practice and Experience* **20**, S1, June 1990, pp. 19-34.

[39] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*, Callicoon, NY, June 2002, pp. 40-46.

[40] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report #1346, Computer Sciences Department, University of Wisconsin, April 1997.

[41] D.A. Maltz and P. Bhagwat. MSOCKS: An Architecture for Transport Layer Mobility. *INFOCOM '98*, San Francisco, CA, April 1998.

[42] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. *1993 Winter Usenix Conference*, San Diego, CA, 1993.

[43] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone (Editor). **Handbook of Applied Cryptography**. CRC Press, 1996.

[44] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. May, 1994.

[45] Microsoft Windows Team. **Microsoft Windows XP Professional Resource Kit**. 2nd Edition. Microsoft Press, 2003.

[46] D.S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys* **32**, 3, September 2000, pp. 241-299.

[47] J.C. Mogul. Efficient Use of Workstations for Passive Monitoring of Local Area Networks. *ACM Symposium on Communications Architectures and Protocols* (*SIGCOMM '90)*, Philadelphia, PA, 1990.

[48] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. *11th Symposium on Operating System Principles (SOSP '87).* Austin, TX, November 1987.

[49] T. Okoshi, M. Mochizuki, Y. Tobe, and H. Tokuda. MobileSocket: Toward Continuous Operation for Java Applications. *IEEE International Conference on Computer Communications and Networks (IC3N'99)*, Boston, MA, October 1999.

[50] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[51] T.W. Page, R.G. Guy, J.S. Heidemann, D.H. Ratner, P.L. Reiher, A. Goel, G.H. Kuenning, G.J. Popek. Perspectives on Optimistically Replicated, Peer-to-Peer Filing. *Software Practice and Experience* **28**, 2, February 1998, pp. 155-180.

[52] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluter-based Network Servers. *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, USA, October 1998.

[53] J. Pendry and M.K. McKusick. Union Mounts in 4.4BSD-Lite. *1995 Usenix Annual Technical Conference.* New Orleans, LA, USA, January 1995.

[54] H. Pennington. **GTK+/Gnome Application Development**. Que, 1999.

[55] C. Perkins. IP Mobility Support. *Internet Request for Comments* RFC 2002, October 1996.

[56] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. *16th ACM Symposium on Operating Systems Principles (SOSP-16)*, Saint Malo, France, October 5-8, 1997, pp. 288-301.

[57] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems* **8**, 3, Summer 1995, pp. 221-254.

[58] J.S. Plank, M. Beck, G. Kingsley. Libckpt: Transparent Checkpointing under Unix. *USENIX Winter 1995 Technical Conference*, New Orleans, LA, January 1995.

[59] J. Postel. Internet Protocol. *Internet Request for Comments* RFC 791, September 1981.

[60] J. Postel. Transmission Control Protocol. *Internet Request for Comments* RFC 793, September 1981.

[61] M.L. Powell and B.P. Miller. Process Migration in DEMOS/MP. *9th ACM Symposium on Operating System Principles*, October 1983.

[62] D. Presotto. Private communication. August 2003.

[63] X. Qu, J.X. Yu, and R.P. Brent. A Mobile TCP Socket. Technical Report TR-CS-97-08, Computer Sciences Laboratory, RSISE, The Australian National University, Canberra, Australia, April 1997.

[64] X. Qu, J.X. Yu, and R.P. Brent. A Mobile TCP Socket. *International Conference on Software Engineering (SE '97)*, San Francisco, CA, USA, November 1997.

[65] T. Richardson, F. Bennett, G. Mapp, and A. Hopper. Teleporting in an X Window System Environment. *The X Resource*, Issue 13, O'Reilly & Associates, Jan 1995, pp. 133-140.

[66] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing* **2**, 1, January/February 1998, pp. 33-38.

[67] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics* **5**, 2, April 1986, pp. 79-109.

[68] B.K. Schmidt, M.S. Lam, J.D. Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-client Architecture. *17th ACM Symposium on Operating Systems Principles (SOSP '99)*. Kiawah Island, South Carolina, December 1999.

[69] A.C. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. *6th IEEE/ACM International Conference on Mobile Computing and Networking (Mobicom '00)*. Boston, MA, August 2000.

[70] A.C. Snoeren, D.G. Anderson, and H. Balakrishnan. Fine-Grained Failover Using Connection Migration. *3rd Usenix Symposium on Internet Technologies and Systems (3rd USITS)*. San Francisco, CA, March 2001.

[71] E. Solomita. Xmove Version 2.0 Beta 2. ftp://ftp.cs.columbia.edu/pub/xmove, November, 1997.

[72] E. Solomita, J. Kempf and D. Duchamp. Xmove: A Pseudoserver for X Window Movement. *The X Resource*, Issue 11, July 1994, pp. 143-170.

[73] R.M. Stallman. **GNU Emacs Manual, Fifteenth Edition**. Free Software Foundation, 2002.

[74] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. *10th International Parallel Processing Symposium*, Honolulu, HI, 1996.

[75] G. Stellner and J. Pruyne. Resource Management and Checkpointing for PVM. *2nd European PVM User Group Meeting*, Lyon, France, 1995.

[76] W.R. Stevens. **UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI**. Prentice Hall, 1998.

[77] D. Sweet. **KDE 2.0 Development**. SAMS, 2000.

[78] D. Thain, J. Basney, S. Son, and M. Livny. The Kangaroo Approach to Data Movement on the Grid. *10th IEEE Symposium on High Performance Distributed Computing (HPDC '01)*, San Francisco, California, August 2001.

[79] D. Thain and M. Livny. Bypass: A Tool for Building Split Execution Systems. *9th IEEE Symposium on High Performance Distributed Computing (HPDC '00)*, Pittsburgh, PA, August 2000.

[80] M.M. Theimer, K.A. Lantaz, and D.R. Cheriton. Preemptable Remote Execution Facilities for the V-System. *10th ACM Symposium on Operating System Principles,* Orcas Island, WA, December 1985.

[81] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. **Distributed Computing Systems: Concepts and Structures**, IEEE Computer Society Press, 1992, pp. 145-164.

[82] K. Wood, T. Richardson, F. Bennett, A. Harter, and A. Hopper. Global Teleoporting with Java: Towards Ubiquitous Personalized Computing. *Nomadics '96*, San Jose, March 1996.

[83] D. Wright. Cheap Cycles from the Desktop to the Dedicated Cluster: Combining Opportunistic and Dedicated Scheduling with Condor. *Linux Clusters: The HPC Revolution*, Champaign-Urbana, IL, USA, June 2001.

[84] D.K.Y. Yau and S.S. Lam. Migrating Sockets -- End System Support for Networking with Quality of Service Guarantees. *IEEE/ACM Transactions on Networking*, **6**, 6, December 1998, pp. 700-716.

[85] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH Protocol Architecture. *Internet Engineering Task Force Internet-Draft* draft-ietf-secsh-architecture-13, September 2002.

[86] F. Yuan. **Windows Graphics Programming**. Prentice Hall, 2001.

[87] E. Zadok and D. Duchamp. Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing. *1993 Summer Usenix Technical Conference*, Cincinnati, OH, June, 1993, pp. 69-85.

[88] V.C. Zandy, B.P. Miller, and M. Livny. Process Hijacking. *Eighth International Symposium on High Performance Distributed Computing (HPDC '99)*, Redondo Beach, CA, August 1999, pp. 177-184.

[89] B. Zenel. A Proxy Based Filtering Mechanism for the Mobile Environment. *Ph.D. Dissertation*, Computer Science Department, Columbia University, December 1998.

[90] Y. Zhang and S. Dao. A "Persistent Connection" Model for Mobile and Distributed Systems. *4th International Conference on Computer Communications and Networks (ICCCN)*. Las Vegas, NV, September 1995.

[91] E.D. Zwicky, S. Cooper, and D.B. Chapman. **Building Internet Firewalls**, 2nd Edition. O'Reilly and Associates, Sebastopol, CA, June 2000.