Reliable Network Connections

Victor C. Zandy and Barton P. Miller
Computer Sciences Department
University of Wisconsin - Madison
{zandy,bart}@cs.wisc.edu

Abstract

We present two systems, reliable sockets (rocks) and reliable packets (racks), that provide transparent network connection mobility using only user-level mechanisms. Each system can detect a connection failure within seconds of its occurrence, preserve the endpoint of a failed connection in a suspended state for an arbitrary period of time, and automatically reconnect, even when one end of the connection changes IP address, with correct recovery of in-flight data. To allow rocks and racks to interoperate with ordinary clients and servers, we introduce a general user-level Enhancement Detection Protocol that enables the remote detection of rocks and racks, or any other socket enhancement system, but does not affect applications that use ordinary sockets. Rocks and racks provide the same functionality but have different implementation models: rocks intercept and modify the behavior of the sockets API by using an interposed library, while racks uses a packet filter to intercept and modify the packets exchanged over a connection. Racks and rocks introduce small throughput and latency overheads that we deem acceptable for the level of mobility and reliability they provide.

1 INTRODUCTION

We present two new systems, *reliable sockets* (rocks) and *reliable packets* (racks), that each provides transparent network connection mobility to ordinary applications using only user-level mechanisms. These systems have several major features in common. 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. They automatically recover broken connections without loss of in-flight data as connectivity is restored. When an application using either of these systems 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. Finally, designed to be ready-to-use by ordinary users, 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.

The failure detection and recovery mechanisms are reliability features intended to enable applications to transparently endure the travails of mobile computing, such as moving to another network, unexpected link failure (e.g., modem failure), and laptop suspension. Rocks and racks can automatically resume a broken connection 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. This interface is one of several value-added services provided to rocks- or racks-aware applications by the *rocks-expanded API* (RE-API).

The remote detection of sockets enhancements is accomplished by a new user-level *Enhancement Detection Protocol* (EDP). This protocol achieves a tricky task: it enables user-level socket enhancements to test for compatible functionality at the other end of a new connection without affecting unenhanced applications. 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 [5] or firewalls [6]. The protocol is general purpose: it can be used by other mobility systems such as MSOCKS [10] or mobile TCP sockets [19,20], and it can support systems that enhance sockets with other functionality such as compression, encryption, or quality-of-service. For example, the EDP could end the common practice of reserving two ports, one encrypted and one non-encrypted, for network services such as IMAP [4].

The major 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, while extending the behavior of these functions to mask connection failures from the application. In contrast, racks are based on a separate user-level daemon that uses an advanced kernel-level packet filter to redirect the flow of selected packets to the daemon [15]; packet filters with such functionality currently include the Linux netfilter and the FreeBSD divert sockets. Racks do not require any code to be interposed in processes that use them. The two approaches each have distinct advantages and drawbacks and both implement transparent enhancements of network communication functionality in user-space without requiring kernel modifications. Both implementations are complete; we distribute and maintain them to be used with real applications.

Avoiding kernel modifications has two main benefits. First, it facilitates deployment. Kernel modifications must be maintained with each kernel version and are inherently unportable. Furthermore, system administrators are rightly paranoid to apply kernel patches that do not provide functionality that is absolutely necessary.

Second, user-level mobility is much easier to combine with process checkpointing mechanisms. We have used rocks to expand the scope of Condor [9] process migration to include distributed jobs that communicate over sockets, and to check-

MOBICOM'02, September 23-26, 2002, Atlanta, Georgia, USA.

point parallel programs based on MPI. Rocks are an ideal abstraction for such functionality: as a user-level abstraction, they are portable and can be compatible with both user-level and kernel-level checkpointing mechanisms, while as a replacement to the sockets API, they can save communication state without being aware of application-specific details such as message formats and buffering. Rocks and racks are a key component in our system for *roaming applications*, whose goal is to enable the migration to a new host of the entire context of a desktop application, including its network, I/O, and user interface state, without introducing any modifications to the application, operating systems, or network infrastructure.

Our systems complement the functionality of existing mobile networking systems. For example, they can be layered over network stacks that support Mobile IP [17] or the TCP migrate option [21] to recover connections that are not preserved by these systems, such as those that become disconnected for periods longer than the TCP retransmission timeout. On the other hand, for users who cannot install the kernel mechanisms required by these network stacks or the network infrastructure required by Mobile IP, rocks and racks replace their functionality with user-level mechanisms. Furthermore, rocks and racks support a mobility model more fine-grained than that of Mobile IP: while Mobile IP requires all applications at a given IP address to move together, rocks and racks enable independent mobility of individual connections. This freedom enables individual connections to be easily migrated to new hosts.

To summarize, the contributions of this paper are:

- Two new user-level techniques for transparent network connection mobility, including mechanisms for failure detection, connection suspension, and connection recovery that preserves in-flight data.
- A comparison of the implementation techniques of user-level packet filtering and interposition as techniques for transparently enhancing the functionality of a network connection. We look at issues of transparency and performance.
- A new user-level EDP for remotely determining whether enhanced socket functionality, such as rocks or racks, is present at the other end of a TCP connection.
- An analysis of the role of unreliable, connection-less protocols such as UDP in the presence of network connection reliability and mobility.

The remaining sections of the paper are as follows. Section 2 discusses the TCP failure model that motivated this work. Section 3 presents the enhanced socket detection protocol. Section 4 presents the architecture and functionality of rocks and discuss the issues we have had with its implementation model. Section 5 presents racks. Section 6 discusses the security issues of rocks and racks. Section 7 discusses our use of rocks and racks in process checkpointing and migration. Section 8 discusses our approach to the reliability and mobility of UDP. Section 9 evaluates the performance of rocks and racks. Section 10 discusses related work.

2 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 and relate its failure modes to mobility events.

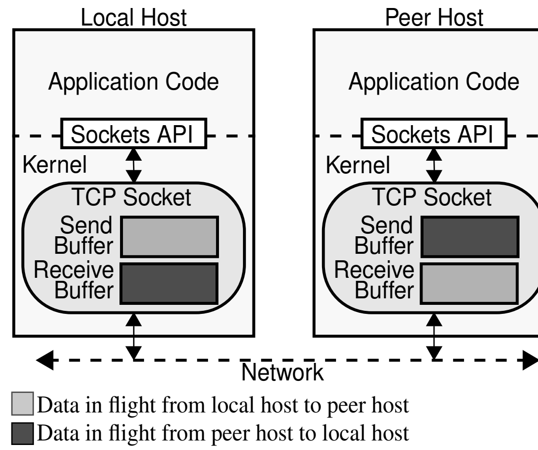

Figure 1: An established TCP connection.

2.1 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 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 [1]. Once an abort has occurred, the socket becomes invalid to the application.

2.2 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.
- Change of IP address: A host might move to a new physical subnet, requiring a new IP address, or a suspended host might lose its IP address and DHCP might subsequently provide a different 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.
- Change of physical address: Through process migration, applications in execution may move from one computer to another. Process migration is an important mobility feature for people who use multiple computers, such as a laptop for travel and separate desktop computers at home and work, because it frees users from having to restart their applications when they move. 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. This characteristic of sockets has long been an obstacle to migration of applications using sockets.
- Host crashes: 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. We do not explore host crashes in this paper because they entail application recovery, while we consider only connection recovery.

3 Detecting Socket Enhancements

When establishing a new connection, our systems use a novel protocol, the Enhancement Detection Protocol, to detect whether the remote socket also supports our systems (either one) and, if it does not, to trigger the fall back to ordinary socket behavior on that connection. With rare exceptions, this protocol is transparent to the application code at each end. Besides network connection mobility, the protocol is a general-purpose approach to safe, portable, and user-level remote detection of any type of socket enhancement. Servers can freely use the protocol since it imposes a trivial performance penalty when accepting connections from unenhanced clients. All significant costs of the protocol are incurred by clients that use socket enhancements. We have verified that the protocol works with many standard services, including ssh, telnet, ftp, and X windows.

It is tricky to remotely distinguish a 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 [19,20], since an unenhanced process is likely to misinterpret it. 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 although there are TCP and socket options, TCP options cannot be read or written from user space without emulating all of TCP over a (usually privileged) raw socket, and it is not possible to use socket options to construct a distinct socket configuration that could be remotely sensed.

The protocol is as follows (see Figure 2):

1. *Client Probe*: The client and server perform a four step protocol to establish to the client that the server is enhanced:
   1a. The client opens a connection to the server.
   1b. The client closes its end of the connection for writing, using the sockets API function shutdown.
   1c. The server detects the end-of-file and announces in response that it is enhanced, then closes the connection.
   1d. 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 announcement that it is enhanced. 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 that is exchanged must be a pattern that is extremely unlikely to be produced by unenhanced clients or servers. The server generates a long (1024 byte) random byte array as its announcement, and the client returns this array as its enhancement announcement. The announcement pattern may be shared among different enhanced server processes.

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, the server 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, if 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.

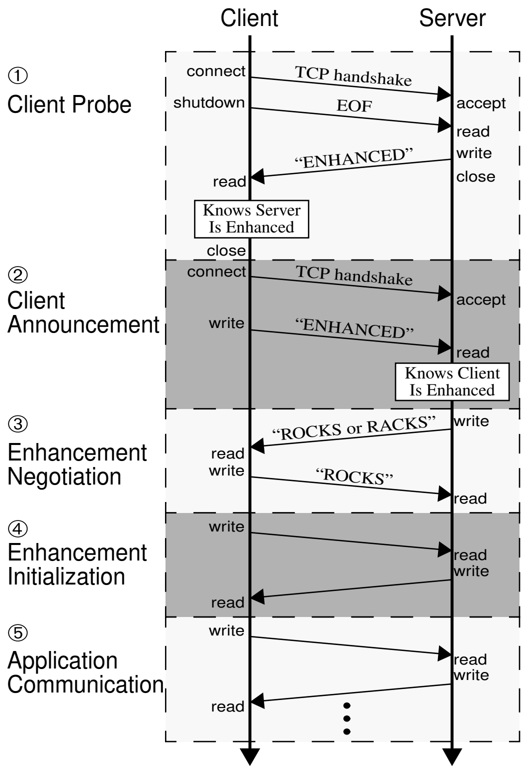

Figure 2: The socket enhancement detection protocol.

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, 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 we consider to be a problem with deployment, not the protocol.

4 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). The library exports the sockets API to the application code to enable it to be transparently dropped in 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.

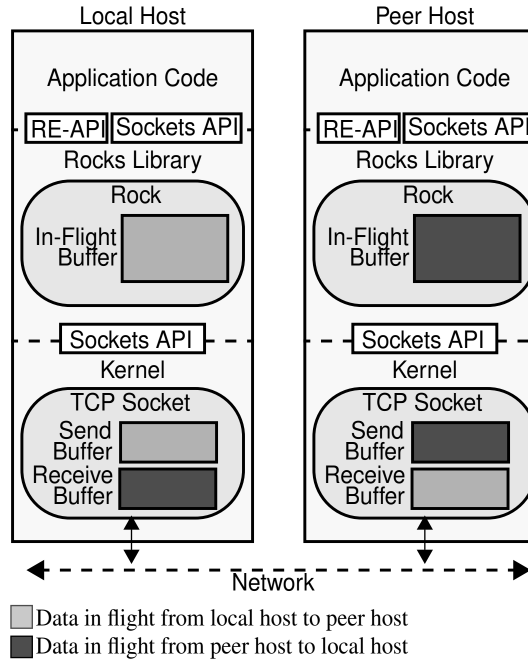

Figure 3: The reliable sockets architecture.

4.1 Rocks Overview

The operation of a reliable socket can be summarized by the state diagram shown in Figure 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 process, not the internal TCP socket state maintained by the kernel. A reliable socket begins in the CLOSED state.

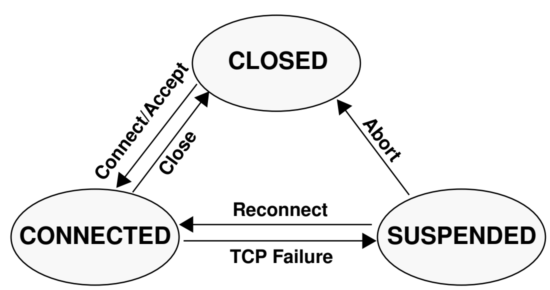

Figure 4: The reliable socket state diagram

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 [12] 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 in-flight buffer is sized to guarantee 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 we cannot guarantee that a heartbeat is consumed before the next one arrives, we cannot prevent this corruption. Third, on some operating systems, 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 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.

We provide 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 server that redirects its client's connection to another server over an ordinary TCP connection. Rock-enabled clients can effectively establish reliable socket connections with ordinary servers by running rockd on the same host as the server. Although the connection between the rockd and the server is not reliable, it is immune to TCP failures, including server host IP address changes, since it is local to the host. To simplify the use of rockd, rock detects the use of some common client applications, and automatically attempts to start rockd on the server host if necessary.

4.2 Experience

We have been using rocks on a daily basis for over a year, primarily with interactive network clients such as ssh, ftp, 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 rock; on other server hosts, we establish rocks connections through rockd. Not having to restart applications is addictive, and using rocks widely generally works well since the EDP switches to ordinary sockets when necessary. However, we sometimes run into trouble when trying to use rocks with a new application. The main problem 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 interposition to maintain application transparency.

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 steps around this problem by patching the C library with corrected calls at run time, but this requires knowledge of the problematic functions, which may change with new library versions (though we are developing a tool to automate the search). 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 [24], Condor [9], or those created by the Bypass toolkit [25]. 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. To handle this case, whenever a server rock accepts a reconnection attempt it locates, by indexing a shared table with the identifier established during initialization, 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, then during its initialization the library transfers the state of the rocks from the temporary process. If the call succeeds but, because the preloading does not work in the new binary, the rocks library is not loaded, 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. They introduce additional operating system dependencies that must ported; since they are mostly Unix oriented, they will comprise a significant part of the effort to port to Windows.

5 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 [21] and the TCP splice of the MSOCKS proxy [10]. 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 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 [15]; racks follows this tradition. However, as the primary use of packet filters turned to network monitoring [11,14], 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 recent Linux netfilter technology, but it also could use FreeBSD's divert sockets.

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 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.

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.

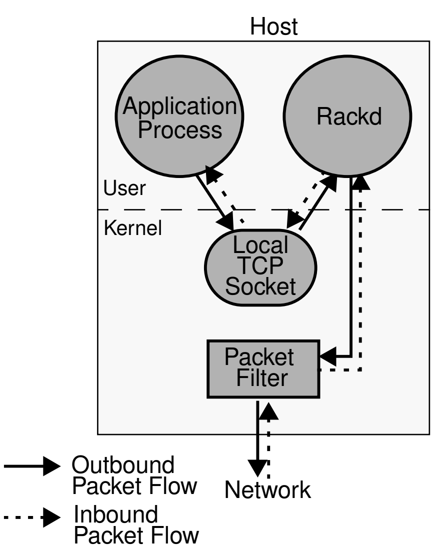

Figure 5: The reliable packets architecture.

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 our 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 from the EDP probe it determined 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 a TCP option 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 option, 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 [1] 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 [10].

6 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 [12].

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. We could easily 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 protocols for encryption and authentication, such as SSH and IPsec (encrypted connections are the common case in our daily use).

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.

7 Process Checkpointing

Rocks and racks can extend process checkpointing mechanisms to handle parallel programs, such as those that use MPI [13] or PVM [7] in direct message routing mode. 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 [22,23] and MIST [2], are explicitly aware of the communication library used by the application.

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 added 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 network proxy we are developing for roaming applications enables any subset of the processes to be migrated at the same time, and more generally, the RE-API can be used to link an arbitrary mechanism for locating migrated process with the rocks library.

Racks and rocks can also be used to obtain a global checkpoint of a parallel application from which the application can be restarted after a hardware failure. Care must be taken to ensure that the checkpoint 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 [3].

We have used racks and rocks to checkpoint and migrate the processes of an ordinary MPI application running under MPICH [8]. Our application runs on a cluster of workstations using the MPICH p4 device for clusters. Once the application is started, each process can be signalled to checkpoint and then terminate or stop. Using this technique, we plan to extend Condor support for MPI applications [26] to include checkpointing and migration. To obtain a globally consistent checkpoint, each process will stop itself after it checkpoints and Condor will be responsible for restarting them.

8 UDP

Rocks or racks are not an obvious fit with UDP-based applications, however the mobility features of rocks and racks can be a clear benefit to UDP applications, enabling a program to continue its communication following a change of address or an extended period of disconnection. For example, they 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.

For mobile-aware applications, we provide callbacks in the RE-API through which they 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.

9 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.

9.1 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 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 10usec) 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.

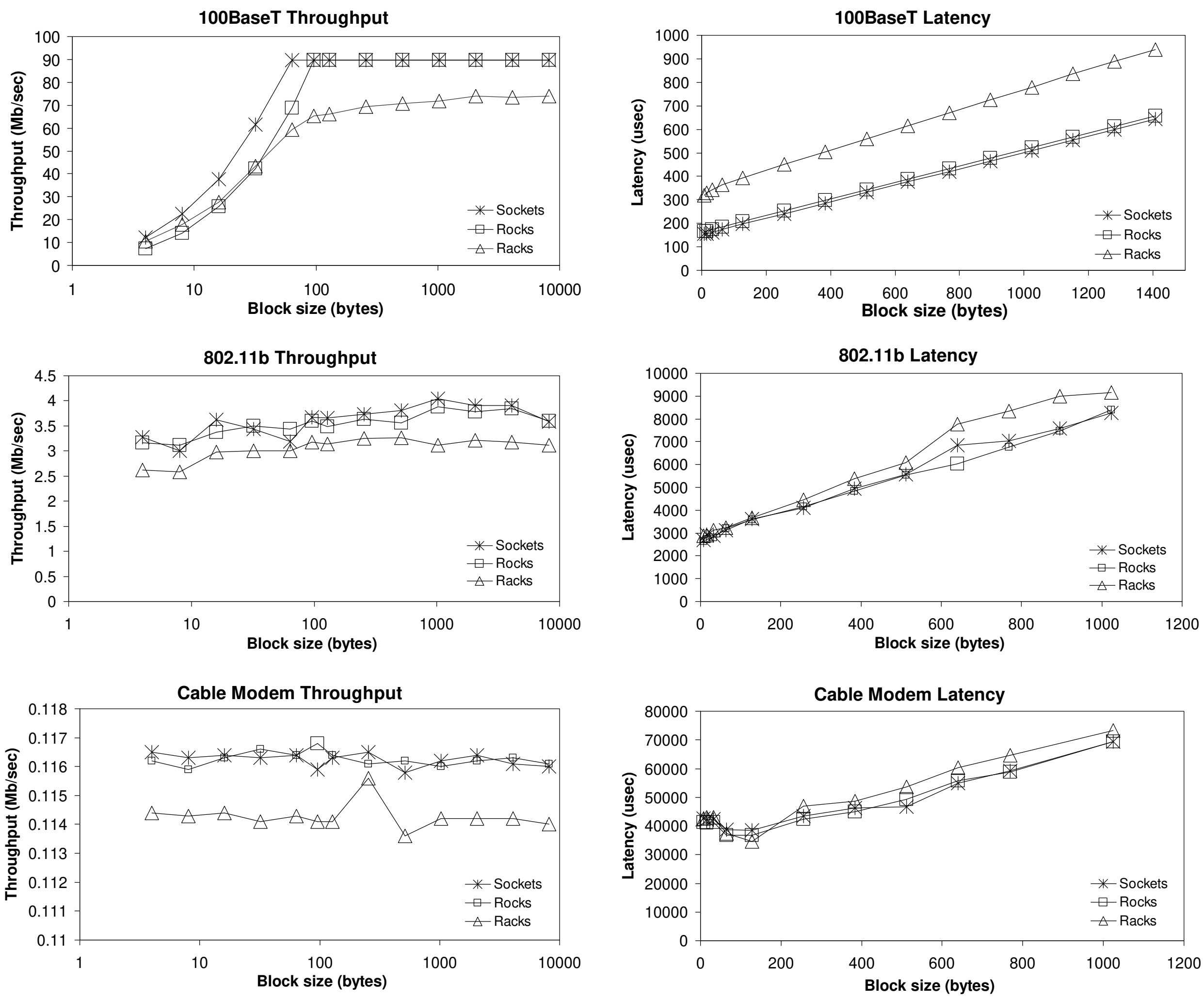

Figure 6: Average rocks and racks throughput and latency over 100BaseT, 802.11b, and cable modem links.

The performance effects of racks and rocks are less easily discerned on the slower links. While we have exclusive access to the 100BaseT switch, our measurements on the 802.11b and cable modem networks are 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.

9.2 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 Table 1. 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 2ms. Although these times are high, connection times are still about 4ms, which we deem an acceptable cost for the added reliability and mobility.

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

Table 1: Average TCP connection establishment time.

9.3 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 our 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.

10 Related Work

Many techniques for network connection mobility have been proposed. Unlike these systems, racks and rocks emphasize reliability over mobility, viewing mobility as just another cause of network connection failure. They provide reliability by integrating mechanisms for rapid failure detection and unassisted reconnection with mechanisms for preserving connection state. The other distinguishing features of our systems are that they are implemented entirely outside of the kernel and they enlist a new user-level protocol to interoperate safely with ordinary sockets.

Mobile TCP sockets [19,20] and Persistent Connections [27] interpose, like rocks, a library between the application code and the sockets API that preserves the illusion of a single unbroken connection over successive connection instances. Between connections, Mobile TCP sockets preserve in-flight data by using an unspecified kernel interface to the contents of the TCP send buffer (such interfaces are not common), while Persistent Connections makes no attempt to preserve in-flight data. Mobile sockets cannot handle TCP failures that result in the abort of the TCP socket, since that action destroys the contents of the socket send buffer. Both of these techniques depend on external support to re-establish contact with a disconnected peer, and neither interoperates safely with ordinary applications. MobileSocket [16] provides to Java programs the same in-flight data reliability of rocks and racks using an in-flight data buffer similar to that of rocks, but it lacks automatic failure detection, uses a more complicated in-flight data buffering scheme that restricts application data flow, and lacks an interoperability feature like the EDP, so it can only operate with other applications that use MobileSocket.

The TCP Migrate option [21] 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 technique is similar to racks in that it manipulates connection state at the packet level. However, it is based on a modification to the kernel implementation of the transport protocol, not manipulation of the packets. In addition, it lacks automatic failure detection and automatic reconnection and it does not support extended periods of disconnection.

MSOCKS [10] has architectural similarities to both rocks and racks. MSOCKS is a proxy-based system that enables a client application process to establish a mobile connection with an ordinary server. The proxy uses 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. The TCP splice maps the state of the original connection held open by the server with the state of the current connection held by the client. The TCP splice could be reimplemented at user level with the rackd packet filter technique to map packets between local and remote socket state. In addition, MSOCKS interposes a library between client application code and the operating system that redirect socket API calls to the MSOCKS proxy, and it uses a uses a mechanism similar to the rocks in-flight buffer to preserve data sent from the client to the server. MSOCKS lacks mechanisms for automatic failure detection and reconnection and its TCP splice is implemented in the kernel.

An alternative to TCP-specific techniques, Mobile IP [17] routes all IP packets, including those used 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. Except for masking IP address changes from TCP sockets, Mobile IP does not handle failures to TCP connections. It depends on external mechanisms for detecting disconnection and initiating reconnection.

11 Conclusion

Rocks and racks transparently protect ordinary applications from network connection failures, including those caused by a change of IP address, link failure, and extended period of disconnection. Besides being an unavoidable part of life with mobile computers, these failures can also occur unexpectedly during non-mobile communication, such as when modems fail or dynamic IP address leases expire. Rocks and racks do not require modifications to kernels or network infrastructure, work transparently with ordinary application binaries, and using our new Enhancement Detection Protocol transparently revert to ordinary socket behavior when communicating with ordinary peers. We routinely use these systems sockets for end-user interactive applications, such as remote shells and remote GUI-based applications, and for checkpointing and migrating parallel programs.

As part of our ongoing work on roaming application, we are developing a network proxy for more general network connection mobility. This proxy will provide support for simultaneous movement of both ends of a connection and support the rocks- and racks-based connections with ordinary peers that do not support either system.

Acknowledgements

Somesh Jha offered several insightful observations about rocks and racks security and made helpful suggestions on the overall paper. Phil Roth made many useful comments throughout the development of the paper. Joel Summers and Marios Skounakis convinced us to use a UDP-based control socket. A reviewer's observation led us to a more robust choice of connection identifier.

References

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

[2] 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.

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.