Carnegie Mellon University
Information Networking Institute

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science in Information Networking

MIGSOCK

Migratable TCP Socket in Linux

Bryan Kuntz and Karthik Rajan

Accepted by the Information Networking Institute

Thesis Advisor: Katia Sycara
Print Name

Signature **Date**

Thesis Reader: Joseph Giampapa
Print Name

Signature **Date**

Academic Advisor: Richard Stern
Print Name

Signature **Date**

Thesis Presentation Date: Thursday, the 21st of February, 2002

TR #__________
Office use only

Acknowledgment

We would like to thank everyone who directly or indirectly contributed to our thesis. Thanks to our advisors Dr. Katia Sycara and Joseph Giampapa. Thanks to all of the support we received from the Information Networking Institute: Susan Jones, Lisa Currin, Jasen Lentz, Joe Kern, and Dr. Richard Stern. Thanks to the members of the Intelligent Software Agents laboratory at the Robotics Institute. Thanks to the entire MS12 class for enriching our experience throughout the program. Finally, thanks to Nithya, Shannon, and our families for their support during the program and otherwise.

Contents

List of Figures

Abstract

Process migration is the act of transferring a process between two machines during its execution. Although it has not achieved widespread use, it is expanding in importance with the growth of distributed computing. There are already some important identified contributions that this capability can provide to the field of distributed systems, including dynamic load distribution and data access locality. But there are also some potential benefits of the development of this idea that will be realized as more advances continue to be made.

We have identified a shortcoming in many of the current process migration approaches: the lack of network socket migration support at the operating system level. A network socket is the interface provided by the operating system to a user program that abstracts-away the complexities involved in network communications. A socket is represented in a process by a socket descriptor. Comprehensive process migration systems should include this socket descriptor as part of the process that migrates or else you limit the candidates for process migration to the processes that do not have network connections.

We have focused our efforts exclusively on this problem of socket migration. We provide a Linux kernel module that re-implements TCP to make migration possible. This implementation requires minimal modifications (patches) to the kernel files and makes the migration option available transparently to user applications without them needing to be recoded. The remainder of the functionality exists in a kernel module that can be loaded on demand by the kernel.

We have demonstrated the utility of socket migration by migrating a process using an existing process migration system called CRAK, while migrating the socket using our implementation. We have also demonstrated an alternative use for migratable sockets, the handoff of a live socket from one process to another.

Chapter 1

Introduction

Process migration is the act of transferring a process between two machines during its execution. It involves capturing the state of a running process at a point in time, transferring this state to a new host, and recreating the process on this host from the point at which it halted. There are some important identified contributions that this capability can provide to the field of distributed systems, including dynamic load distribution, fault tolerance, and data access locality. Process migration has not achieved widespread use and we feel that one of the reasons for this is the lack of good network socket migration support in the existing process migration systems.

The standard definition for a socket is simply the combination of a machine's host address and port [12]. A connected socket then is the combination of the two communicating machines' hosts and ports – four values all-together. A network socket interface is provided by the operating system to a user program. It abstracts-away the complexities involved in network communications by providing an API or set of system calls. An application using a particular family of protocols, such as TCP/IP, can avoid all of the programming required to implement its side of the communication with a remote host by simply invoking this set of system calls. Migration of sockets during process migration is difficult because a socket can be in one of multiple states – unconnected, listening, connecting, and connected – and because a migration candidate may have multiple open sockets with processes on different hosts. More importantly, migrating an open socket involves cooperation from the remote communicating entity. Some current process migration systems choose to address socket migration without having to make modifications to the remote socket implementation [5] [15] [16] [17] [21] [22]. The systems that do choose to include the remote coordination accomplish it without providing kernel-level support, usually in the form of an application layer socket recreation [6]. Thus some of these systems ignore the socket migration component altogether, others code solutions that add bulk and complexity to the applications. We do not feel that these approaches provide the support required for a robust socket migration capability. We propose to focus our efforts exclusively on the problem of socket migration, by tackling it head-on with a solution at the operating system kernel level that involves modifying the TCP implementation.

Our solution is called MIGSOCK, for MIGratable SOCKet. It consists of a patch to and a module for the Linux kernel, providing a set of system calls that support socket migration. This implementation resides at the OS kernel level, and the TCP layer of the network stack (OSI layer 4). Thus the migration of sockets can be performed transparently on any application programs that are compiled to run on the Linux distribution that we port to. These programs do not need to be specially coded or configured for migration. MIGSOCK does not degrade the performance of TCP except for the two messages needed to initiate and conclude migration of the socket. Also, with one small exception, TCP continues to function correctly when communicating with a remote host that does not support socket migration. MIGSOCK provides a socket migration capability that can be used in two realms. It can be used to accompany a process migration system such as CRAK that does not support socket migration, and it can be used to enable the handoff of a live network socket connection between two separate processes. These two uses will be developed in the sections of this paper.

The remainder of this paper is organized as follows. Chapter 2 introduces process migration and provides a summary of the important characteristics and features of process migration systems. In chapter 3 we introduce socket migration and discuss varying socket migration techniques, highlighting the differences and improvements offered by a kernel level implementation versus a user level implementation. Chapter 4 of this paper provides a high level background of how UNIX in general and LINUX in particular implement socket communication. An understanding of this will be necessary before the discussion of our design can be presented. In chapter 5 we offer a high level depiction of our design, which will serve as an introduction to chapters 6 and 7 of this paper where the design and implementation are laid-out in full. Chapter 6 is the detailed design section, and chapter 7 reviews the implementation. Chapter 8 demonstrates the tests and results that were conducted to verify functionality. We explain how our network socket migration mechanism was easily integrated with an existing process migration scheme to prove the concept. In chapter 9 we cover the work that is related to process and socket migration. It includes a survey of the important modern process migration implementations, as well as applications or APIs that in some form involve the migration of a network socket.

Chapter 2

Process Migration Background

We consider socket migration a facilitator for process migration. To understand the impact socket migration has in this context, we first examine process migration.

2.1 What is Process Migration?

A process is an operating system representation of a program [25]. In the most general sense, process migration is the suspension of execution of a process on one machine and the resumption of that process on another. This migration requires that the state of the process be captured at a given moment and recorded or serialized in a persistent file or network message. This state is made up of the CPU register values, the process memory including both user and kernel space, open file descriptors including sockets, other device specific parameters, and any other kernel data structures associated with the process. Having collected a consistent description of the state of the process at the point of capture, the next step is to transfer this state information to the new host of the process. The final step is the recreation of the process on this host by building and populating the appropriate kernel data structures with this state information and scheduling the process [7].

Process migration systems are implemented in many different ways. However, there are some common characteristics that the varying approaches must share in order to support the process migration we are talking about [7].

- *Process execution point maintained*

  True process migration requires that the migrated process resume execution from the point at which it left off on the old host. Reinitializing the process in order to resume it is not an acceptable solution.

- *Export/Import interface.*

  The system must provide some type of export/import interfaces that allow state extraction from the source node and importation by the destination.

- *Post migration accessibility.*

The migrated process should be accessible by the same name and mechanisms after the migration as it was before. It should have access to the same communication channels, files, and devices.

- *Source host cleanup.*

  All of the resources that the source kernel administers to the exported process must be returned. Data structures must be freed, file or network pipes must be waited-on or aborted, and no outstanding system calls can be running on behalf of the process.

2.2 Process Migration Importance

There are four primary advantages of using process migration [7]:

Load balancing The process loads can be dynamically distributed among various nodes in a cluster by dynamically shifting processes from heavily loaded systems to systems that are relatively free. This is the most important advantage of the four; the popular network operating systems – including Mach [17], Mosix [15], and Sprite [16] – support process migration to achieve this end. In the Sprite Network Operating System for example, processes are migrated by invoking a system call on the process (from the shell for example). The kernel keeps track of which machines are idle, and chooses one of these hosts as the target destination for the process. The migration is transparent to both the process and the user other than the performance boost evident after relocation. The Sprite paper indicates that the process's environment, including file and device access, is preserved but network sockets are not mentioned.

Fault tolerance Processes can be migrated from failing nodes to other nodes in the system, thus improving the overall reliability of the system. Systems that employ periodic, transparent check-pointing of the process, such as Libckpt [29] and MIST/MPVM, use rollback recovery to achieve fault-tolerance. Such systems store the process state to disk during the frequent checkpoints. If the system crashes, the most recent copy of the process state can be retrieved by a neighboring node and resumed from that checkpoint [23].

Anytime system administration By migrating all important processes from a machine, the administrator can perform his system maintenance work on a node at any time and not be concerned with potentially killing tasks that are running there.

Data access locality By moving the code to the source of data rather than moving the data to the code, systems can be made more efficient in terms of network traffic and process throughput.

2.3 Process Migration Difficulties

Besides some commercial uses of Mosix, the systems described above have not been widely adopted into the marketplace for several reasons [7]:

Lack of support for heterogeneity The proliferation of the Internet over the past several decades has encouraged the development of systems and applications that are capable of running and communicating in a heterogeneous environment. Process migration requires the capture and transfer of state, which is difficult to accomplish if the participating hosts have a different architecture or OS.

Lack of applications Scientific or long running applications are the types that could benefit most from migration capabilities. These make up only a small percentage of today's applications, as the majority are standard PC desktop applications.

Lack of infrastructure Despite their apparent usefulness, distributed operating systems have not been widely transferred to the marketplace because none of the major Operating System vendors have taken the initiative to come out with explicit process migration support.

Socket Migration Inadequately Addressed As discussed in the previous subsection, most implementations of process migration have skimmed through socket migration due to the complexity of adding transparent socket migration to the systems.

2.4 General Process Migration Approaches

All modern systems that support process migration can be categorized into one of two groups based on the level at which process migration is implemented. These are user space and kernel space, as shown in Figure 2.1. Kernel space is made up of the OS Core, which performs the base kernel operations such as process scheduling and memory allocation. On top of this core, layers are built which provide more use-specific functionality such as the networking functionality or device drivers. All of the functionality provided by these layers and by the core is accessible to the user space through a kernel-provided system call interface. Programming languages in the user space write libraries that make use of these system calls. User programs are coded using these libraries.

2.4.1 OS Level support

In systems that support process migration at the OS level, the user code remains unchanged. This can be done in one of two ways:

- *Underlying OS*

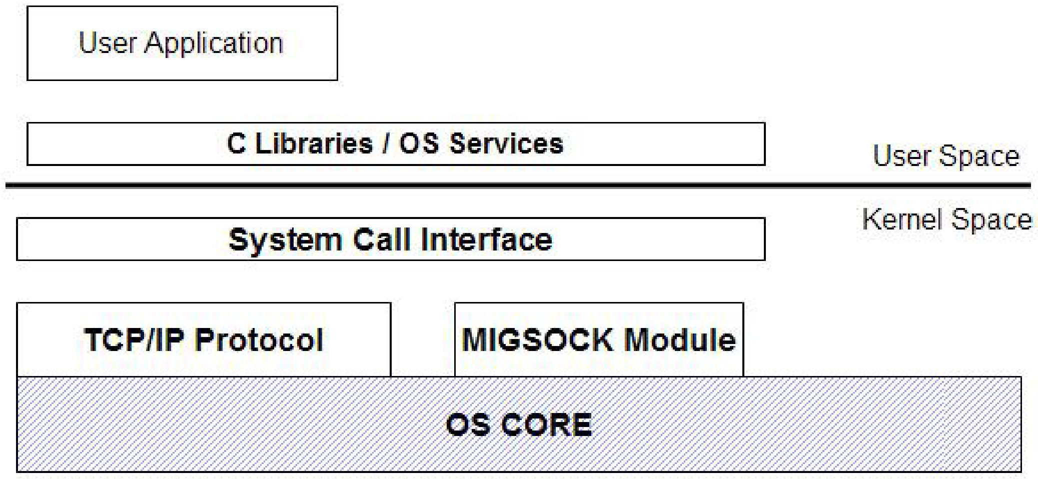

Figure 2.1: Linux System Architecture

Modifications can be made to the underlying OS to support process migration. Two good examples of this are MOSIX[15] and Sprite[16].

- *Microkernels*

  There are many distributed systems that support process migration through modifications to the base microkernel architecture. There are two mains systems of this kind – MACH[17] and AMOEBA[19].

2.4.2 User Level solutions

User level solutions can again be grouped into two sets – the first being support through application relinking /recompiling and the second through programming environment support

- *Application relink and recompile.*

  In this method the application program is linked / compiled with special migration libraries. These libraries provide routines for state capture and migration. The system is less transparent than OS level implementations, and requires that programs be aware of mobility. A good example of this is Condor[21].

- *Programming environment.*

  This involves support from the programming language and execution environment for process migration. Java is well suited for such applications and is one of the main platforms for migration. All Mobile agent systems would come under this category, including Agent-TCL [26], Aglets[27], MOCKETS [6], TACOMA and Telescript [1]. The system is portable across platforms / machines. However, transparency is limited since users must program their applications in their specific programming languages to take advantage of the provided mobility code. Also, the system performance is relatively slow when compared to the other systems.

Chapter 3

Socket Migration Discussion

The simplest definition of a network socket is the combination of a machine's host address and port [12]. A connected socket then is the combination of the 2 communicating machines' hosts and ports – local IP, local port, remote IP, and remote port. The OS provides user programs with a set of socket system calls or API. This API sits on top of layer 5 of the OSI model [11] (Figure 3.1). The complexities of network communication are abstracted by this API. The user program calls the socket system calls to initially establish communication, the result of which is a socket file descriptor. This socket file descriptor can then be used like a normal file descriptor to read and write data. Thus an application using a particular family of protocols, such as TCP/IP, can avoid all of the programming required to implement its side of the communication with a remote host by simply invoking this set of socket system calls.

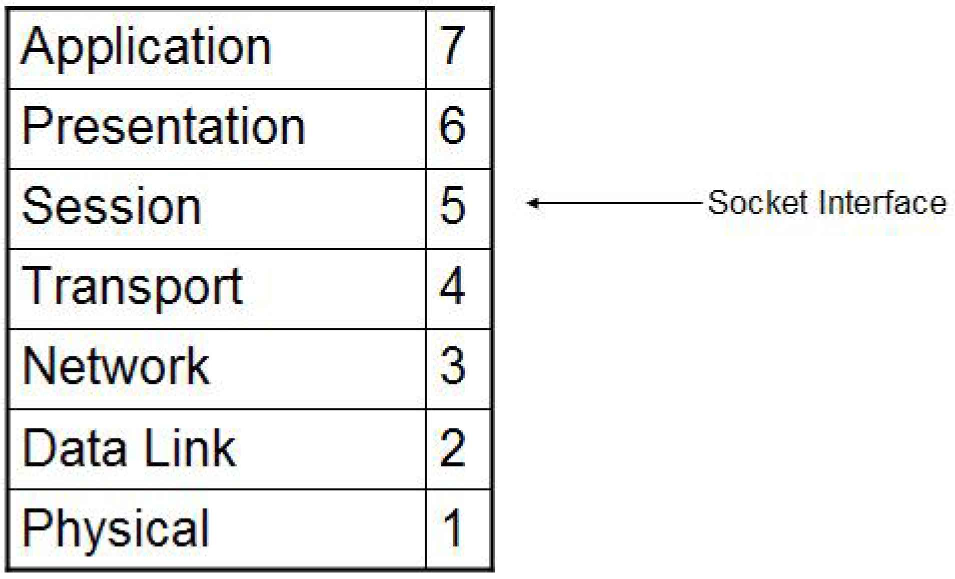

Figure 3.1: OSI Model

This section will discuss various approaches to solving the socket migration problem. The discussion will be general in that it will not mention existing socket migration solutions. The Related Work chapter near the end of this paper will review some of the actual implementations and discuss their merits and demerits.

3.1 What is Socket Migration?

Socket migration is the seamless transfer of one end of a live socket connection from one host to another, without loss or interruption to the packet flow. There are two important impacts of socket migration. One is that it enables process migration to include live network connections among the process's migrating state. The other is that it enables the handoff of a live socket connection. This handoff could take place between two separate processes potentially running on different hosts, or a process could handoff the socket to itself while it is still executing in the case of a network reconfiguration. Socket migration is difficult because the implementations are constrained by the semantics of TCP/IP and the requirements for interoperability with different TCP/IP stack implementations.

Socket migration for process migration A socket, represented in a process or program by a socket descriptor, is part of the process's state and should ideally be included when a process's state is transferred to a new host. This requires that the communication be suspended on both ends and that the state of the socket be identified and preserved with the rest of the process's state at checkpoint time. When the process state is serialized and transferred to the destination host, the socket descriptor is included. When the process is reconstructed on the destination host using the serialized state data, open sockets must be created so that the process can continue to access these socket descriptors. These sockets should be connected to the remote entity so that communication can proceed as it did before the migration began. Figure 3.2 illustrates the main steps involved in socket migration.

Socket handoff Socket migration in this context requires that only the state of the socket, not the parent process, be captured, serialized, and transferred. Figure 3.3 illustrates this scenario. It requires that the socket destination host be ready to accept the live connection by running an identical process that is somehow synchronized to the same state as the source process at the time of migration. In other words, the upper (application) layers should have access to a socket descriptor and be ready to use it in the connected state. An example use for this is allowing processes to maintain execution during a network reconfiguration. An ad-hoc private network might suddenly need to reassign its hosts' IP addresses to avoid a DoS attack, or when a new router goes online. A process's socket could be suspended, handed-off to itself, and resumed using socket migration while this

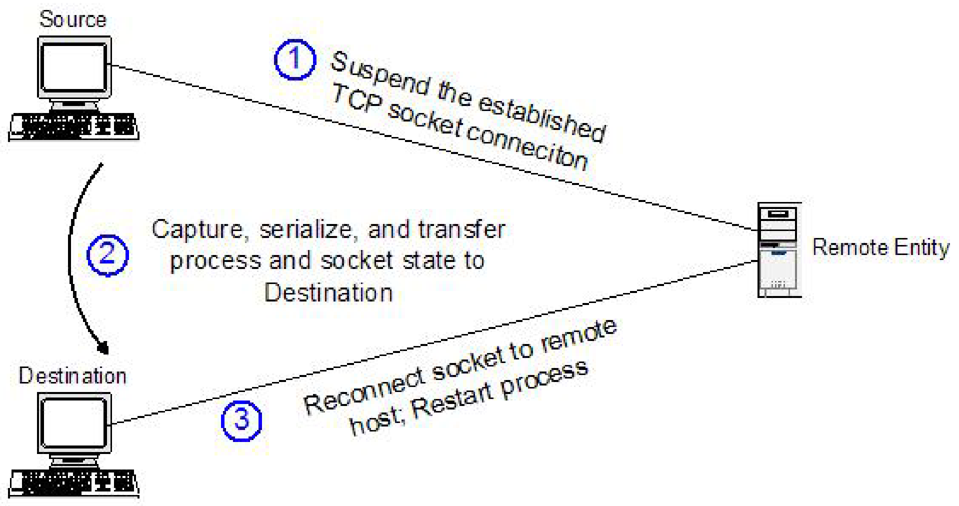
Source
1
Suspend the established TCP socket conneciton
2
Capture, serialize, and transfer process and socket state to Destination
Remote Entity
Destination
3
Reconnect socket to remote host; Restart process

Figure 3.2: Socket Migration

reconfiguration takes place.

3.2 General Socket Migration Approaches

There are three categories of approaches that we can think of for solving the problem of socket migration: proxy-based forwarding, packet spoofing, and host-to-host support. In each of the three approach scenarios, we call the migrating process the "local" or "source entity". It migrates from the source host to the destination host and becomes known as the "destination entity". The source/destination entity communicates with a "remote entity", which resides on a remote host.

3.2.1 Proxy Based Forwarding

Proxy based forwarding is the most noninvasive approach because it does not involve modifications to the underlying protocols. In one implementation, all packets to a socket on a source host are directed there using a single proxy host. The proxy is then responsible for directing the incoming packets to the destination host on the network where the destination process resides. Figure 3.4 illustrates this. In another approach, a proxy process can be instantiated on each host to replace the communicating process whenever its socket migrates to a new destination host. As shown in Figure 3.5, This sets up a chain of proxies between the source and the destination hosts until the process ultimately finishes executing.

In both cases, the proxy does not actually execute the server process – it just forwards packets to its destination address. Each process that opens a socket

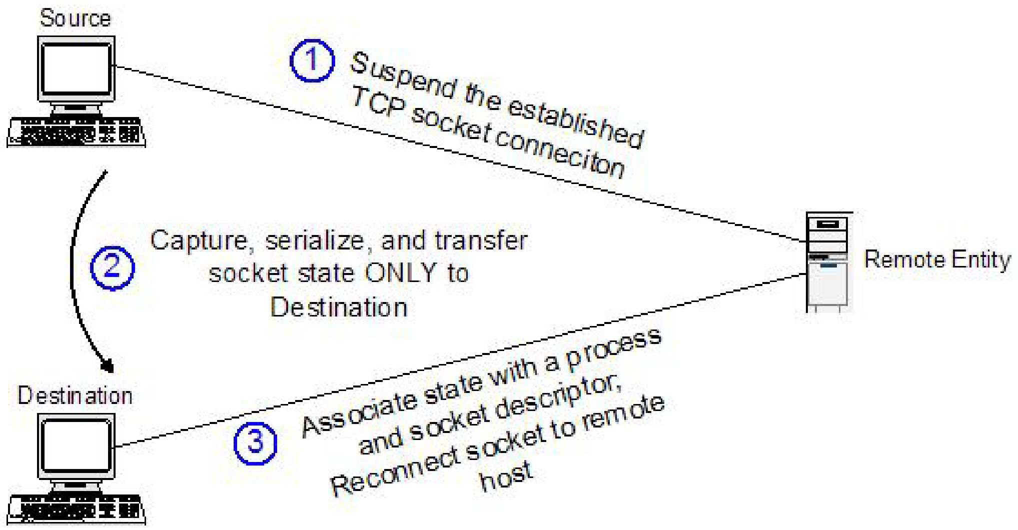
Source
1
Suspend the established TCP socket conneciton
2
Capture, serialize, and transfer socket state ONLY to Destination
Remote Entity
Destination
3
Associate state with a process and socket descriptor, Reconnect socket to remote host

Figure 3.3: Socket Handoff

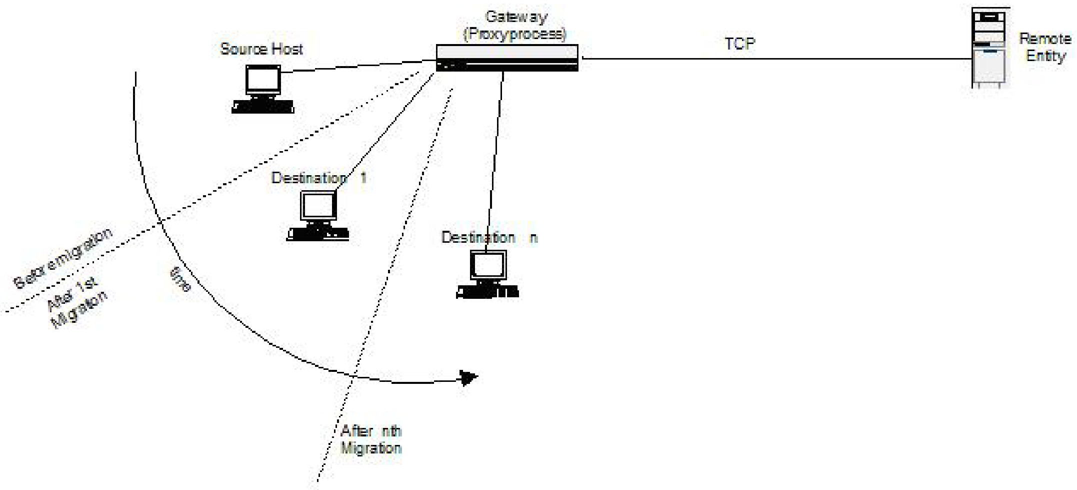

Figure 3.4: Single Proxy on the Gateway

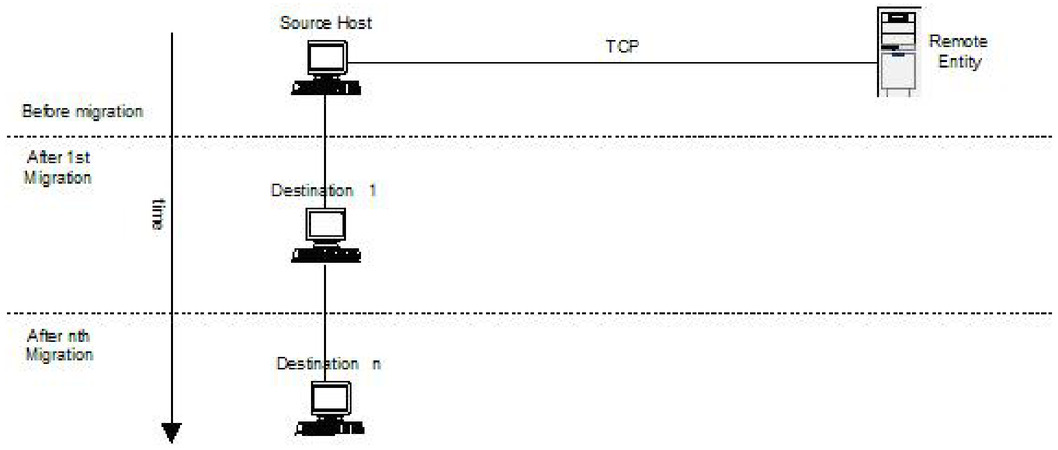

Figure 3.5: Chain of Proxies Created as a process migrates

has an entry in the proxy routing table. The table contains a unique key for the socket-process pair. Thus, all packets to this socket-process pair can be identified. With this key is an associated IP address and port. Thus, all packets destined for this socket-process pair is forwarded to the respective host IP and port.

When a process moves from one host to another, it updates the data on the proxy machine. Thus, the next packet will be forwarded to the new host IP / port.

These two methods have some advantages and disadvantages. The implementation is relatively simple, intuitive and uncomplicated. The proxy could be implemented on the gateway to the network if you can restrict the process migration to within the physical subnet. If you do not want to impose this restriction, then the second method can be used where a proxy process that forwards packets to the next known destination is instantiated to replace the source process on each machine that hosted a migrated process. The primary disadvantage to this is that all packets traveling between the two communicating entities are forwarded through the proxy chain, introducing a bottleneck and multiple points of failure. This generates a lot of extra network traffic and also uses up resources on each machine that is running one or more proxy processes.

3.2.2 Packet Spoofing

Packet spoofing relies on the fact that a host on a subnet can receive all packets passing through the subnet. For example, placing your network interface card in promiscuous mode allows it to catch every packet in the subnet regardless of whom it is destined for. When migrating a socket from one host to another, the destination process initiates a promiscuous mode capture of all packets on the destination host. Thus, packets intended for the source process on the source host are captured by the new destination host. When sending packets back to the remote host, the destination process replaces its local sender address and port information with the values from the source host. In other words, it "spoofs" the origin of the TCP/IP packet data. The remote entity will not be able to tell the difference. This relies on the fact that TCP/IP will not complain when sending packets with an address other than the real origin. Figure 3.6 illustrates a typical packet spoofing scenario.

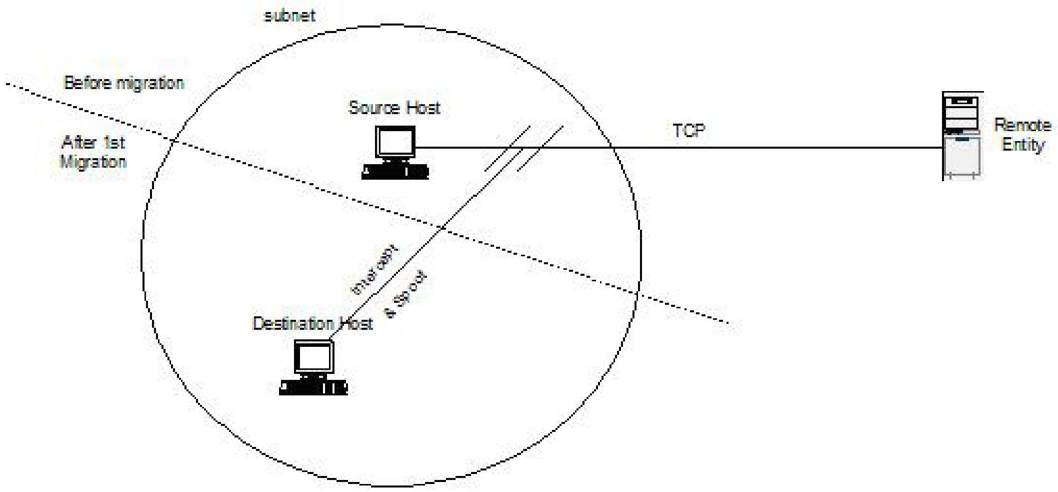

Figure 3.6: Packet Spoofing

In order to work, the following assumptions must be made:

- That promiscuous mode processing is possible. Not all network cards support promiscuous mode, on which this entire method is based.
- That the source host and the destination host must be on the same subnet in order for the source hosts' packets to be intercepted by the destination hosts' network card in promiscuous mode. This also implies that the two hosts are not part of a switched network.
- That there is some mechanism on the source host to prevent it from receiving the packets and sending a TCP reset signal to the remote entity. It would normally do this if it thinks the socket on its end is closed. This would cause the remote entity to close its half of the socket as well, thus defeating the whole purpose of the migration.
- Extra processing is involved in analyzing each TCP packet to determine which socket it belongs to.

The implementation is also fairly complex and inefficient. The processing and number of resources required to receive each packet, determine if it is destined for one of the migrated processes, and spoof addresses are substantial.

A variation of this spoofing mechanism is used for implementing TCP handoffs in load-balanced web server clusters [10] [20].

3.2.3 Host-to-Host Migration Support

Host-to-host migration is when a process moves from one host to another and notifies its remote counterpart of the change of address. Thus, the next packet from the remote host will be sent to the new host. This is shown in Figure 3.7. Such an implementation requires support from both sides of the peer-to-peer communication. For example, TCP sockets would require that the local and remote implementations both support this control message passing structure and should be capable of modifying the remote host address.

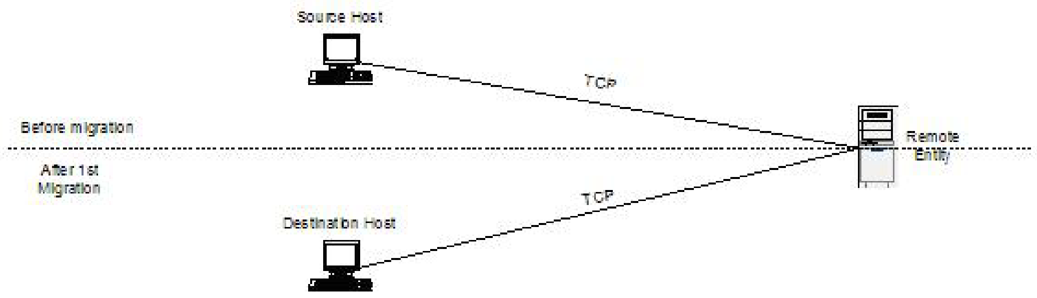

Figure 3.7: Host to Host Migration Support

Host-to-host migration support can be provided in user space or kernel space:

User space means that the socket migration capability is provided as an application layer API (layer 7 of the OSI model). This API can be built on the existing socket library implementation, and the user applications must use this API in place of conventional socket programming if they would like the socket migration option. Advantages of this are that it is simple to implement because it does not require modifications to the operating system. This further implies migration support across heterogeneous systems. That is, provide the API on two different operating systems and you can enable socket migration between the two. Disadvantages of offering user level support are that every application wishing to use the API needs to be recoded. Therefore it is not a transparent solution. Also the API makes the migration process slower than it has to be because it closes and recreates the sockets in order to simulate migration.

Kernel space socket migration support is provided in the operating system kernel (layer 4 and below in the OSI model). It requires direct modifications to the TCP and socket implementation in the operating system. The advantages of this approach are efficiency and performance. It is faster than the user space API because sockets are not closed and recreated; instead the socket state is frozen and transferred to a new host. Also this method is potentially transparent to the applications that run on the system if the socket migration control (initiation and conclusion) is separate from the processes that run. Disadvantages of this approach are that it requires all participating hosts support the migration capability, and that it is more complex to implement.

We chose to implement our design using the kernel space approach because the transparency and performance advantages are best suited for process migration.

Chapter 4

Background of Socket Implementation

This section is an in-depth discussion of the way sockets and TCP are implemented in any NET 3 or NET 4 [1] distribution based on the BSD networking core(the first and the most popular). It is necessary to spend time understanding the mechanisms and structures employed by the operating system in its implementation of network communication to lend context to the ensuing discussion of our socket migration design.

4.1 Overview of Network Implementation in BSD

The communications between applications or user programs on distributed hosts are enabled by the kernel in the form of system calls. The system calls provide the programmer with a uniform set of simple functions that abstract away the complexities of coordinating the remote correspondence. Figure 4.1 illustrates the way in which the kernel approaches the implementation of the most widely used combination, sockets with TCP/IP as the underlying protocol.

The TCP/IP layer in Figure 4.1 is the layer at which the kernel handles the intricacies of the transport and network communications. The Ethernet layer is the device driver for the network card. It is responsible for interfacing the network card while following one of the data link standards such as Ethernet or FDDI. The network card and the interface it provides is out of the realm of kernel space, and is hardware dependent. The layers of the OSI model are shown to the right in the diagram. The physical layer corresponds to the network hardware. The data link layer is the network card device driver. The network and transport layers are IP and TCP (UDP). The other OSI layers lie in user space, out of the realm of the kernel.

The system calls that the kernel provides to the user are at the socket layer. They include socket(), bind(), listen(), connect(), accept(), send(), receive(), as

[1]NET3 and NET4 refer to various versions of Linux network stack implementations

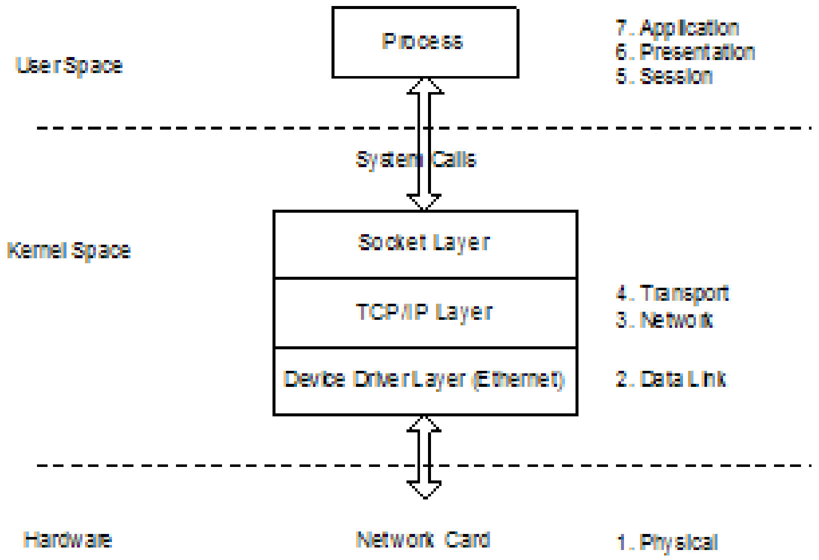

Figure 4.1: Network Stack Implementation

well as file functions such as read() and write() (since a socket descriptor is treated as a file descriptor) . These functions will be discussed in the next section from the user's perspective. In general, the kernel implements this layer by checking the input arguments for validity and for options/instructions pertaining to the lower layers.

4.2 Socket API – The User Perspective

In this section we provide a brief functional description of each of the TCP socket related system calls. The socket API is a client/server architecture and the discussions below will indicate whether the particular system call is a client or server function.

Socket()[2] is used by both the client and server to initialize the kernel structures for a new socket and return a socket descriptor. The socket descriptor is used in all future socket system calls, and differs only slightly from a file descriptor both in the way the kernel implements it and the way a user reads and writes to it.

Bind() is used by the server to specify its TCP host address and port. It binds this information to the socket descriptor. Bind() essentially creates a half-socket with this assignment because a full-socket would include the address/port pair for both the client and the server. Bind() specifies only the server pair and is followed by a call to listen() to begin accepting incoming client requests.

[2]The actual system call

Listen() is a server function that attaches a half-socket to the port and begins listening for client request attempts. Listen() is typically followed in the server code by calls to accept().

Accept() is the last TCP call that is exclusively a server function. It is a blocking call that waits until an incoming request is received from the client. This request will contain the host address / port information for the client socket. Accept() will attach this information to a new socket descriptor, along with the server address and port, and return this new socket descriptor to the server code. The socket descriptor can now be read and written like a typical file descriptor.

Connect() is the client function that initiates an active TCP open connection request. It will assign an arbitrary port to the client and send the port / address duo to the server address and port specified in the arguments to connect(). It will block until the connection is established or return an error if the connection fails for some reason.

Read(), Write(), and Close() are used in the same way that they would be with a file descriptor, but with a socket descriptor as the first argument instead. Read() will block until an EOF is sent by the other end. Write() will block until the buffer that is passed to the call has been completely copied into the kernel buffer. Close() will initiate the active / passive close operation for TCP.

4.3 Internal Representation and Data Structures

The main data structures used to support the socket API in a BSD style NET 3 implementation are itemized below in order of use. Please refer to Figure 4.2 (figure from [11]) for a graphical representation of the data structures.

proc data structure The proc structure is central to the kernel. Every process has a proc structure associated with it. This structure contains all information about the process, like its process id, process group id, parent process id etc. Among these things is a link to the file descriptor table. Often times this is referred to as the Process Control Block. In Linux this is called *struct task_struct*.

file descriptor table The file descriptor table has pointers to two structures. One is a pointer to the open file table. The other is an array of flags used to indicate if a particular file is open or not. Every process has a file descriptor table.

open file table The open file table is an array, with each element pointing to a device/ file / socket. The index into an array is a file descriptor / socket descriptor that is returned by an open or socket call respectively. Each element is a pointer to the file structure described next.

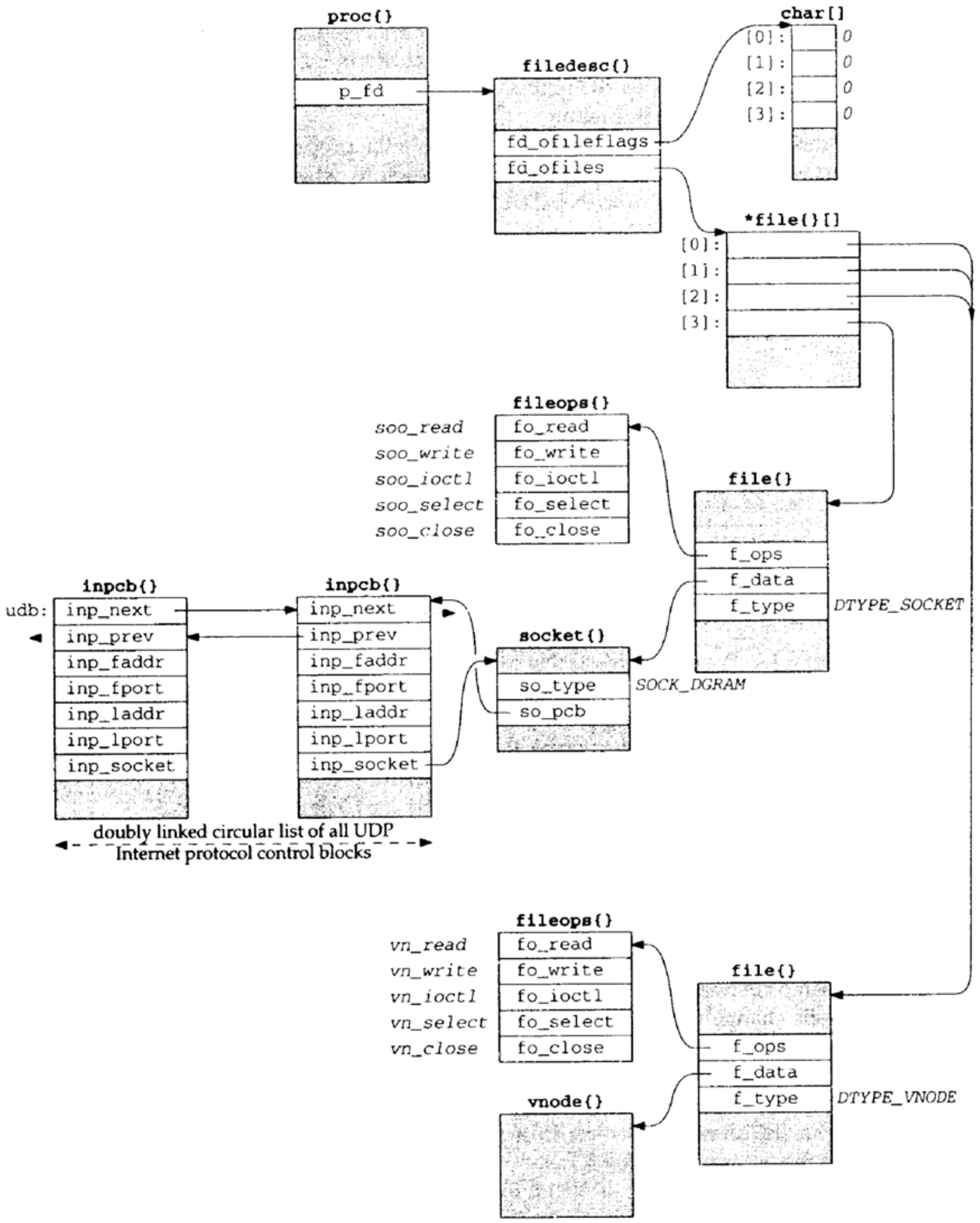

Figure 4.2: Internal Data Structures

file structure This file structure is a structure similar to an inode. It contains three elements. The first is a pointer to the file operations structure described below. The other is a pointer to the actual data block storing more information on the file / socket. The third element is a type field, which specifies the type of file being referenced – that is a file or a socket or a pipe etc. In Linux it is called *struct file*.

file operations structure The file operations structure contains pointers to functions that implement the main operations possible. For example, the elements include pointers to functions that implement the open, close, read and write functions. Thus, when processing, the kernel can directly access the necessary function to call instead of processing the type of file structure and then branching.

socket structure This structure contains information about the type of socket – whether a UDP or TCP socket. The structure also contains a pointer to the inetpcb structure. In Linux this is represented by *struct socket* and *struct sock*, which is contained within *socket*.

inetpcb structure This structure contains information on the destination IP and port, the host IP and port. Each structure element has front and back pointers to other inetpcb structures in the system. Thus, they form a doubly linked list. The structure also contains a pointer back to the socket structure. This pointer is used while processing incoming messages. In Linux this is called *struct tcp_opt*.

4.4 Creating a new socket

Creating a new socket from the user's perspective has been explained in the previous section. This section deals with the actual kernel level data structures and procedures that occur during the creation of a socket.

When a socket system call is made by a user application, the C library function that implements the system call in the user level copies the parameters into the stack and suitably triggers the software interrupt for that system call. Processing now shifts to the kernel mode. The kernel first checks to see if the parameters passed to the function are valid. It then references the *proc* structure followed by the file descriptor table to locate the open file table. It then locates the lowest unused entry in the open file table and maps it to a new *file* data structure. This index is also returned as the result of the socket system call to the application. It's function is similar to that of a file descriptor. The file type in this structure is set to DTYPE_SOCKET – to indicate a socket connection. The appropriate file operation structure is linked to this file structure. The file data pointer is linked to a newly created *socket* structure that holds the socket type and a pointer to the process control block for the socket. As the variables for this socket have not been initialized yet (no connect and bind calls have been made yet), this pointer is left NULL.

When a connect / bind / sendto (for UDP) call is made, the internet control block is initialized with the respective fields containing the data passed through the *sockaddr_t* structure. This includes the destination host / port and the local host / port. The connection is then added to the list of active ports (the doubly linked list).

4.5 Sending Data

Before going into the actual process of sending data, a passing reference has to be made about the kernel *mbuf* structure. The kernel deals with messages in fixed blocks of 128bytes. These blocks have a fixed format and can be used for creating linked lists. Data is passed between the various kernel functions using this structure.

When an application wants to send data through a socket, it invokes a system call. For example, for UDP the call is sendto(). The structure of a simple send function for UDP is given below:

*sendto (sockfd, buff, BSIZE, flag, (struct sockaddr *) &serv, sizeof(serv))*

Explanation of the various variables:

- **sockfd** The socket file descriptor obtained from a socket call
- **buff** The buffer containing the data to be transmitted
- **BSIZE** The size of the buffer
- **Flag** The flags
- **Serv** A structure of type sockaddr containing the remote host and port nos.

The following steps are involved in sending the message through the socket connection:

1. The kernel checks to see if the data reference passed to the function has valid memory addresses.
2. It then creates an mbuf structure / mbuf linked list for the data that needs to be sent.
3. The kernel then creates a special mbuf structure containing the destination host and port information. This mbuf structure is made the head of the linked list containing the data. The information is obtained by traversing through the kernel data structures explained in the previous section.

4. The new linked list is then passed to the IP layer for processing. The IP layer adds its headers to the link list head (the linked list head mbuf is so placed and formatted that it resembles the header of a datagram packet that progresses thorough the protocol hierarchy).
5. The mbuf list is then passed on to the link layer. The link layer adds its own headers to the mbuf list head (now the format resembles a link layer datagram).
6. The device is then invoked to transmit the data on the physical medium.
7. The mbuf structures are then released and added to the kernel's pool.

4.6 Receiving Data

When an application wishes to receive data, it generally calls a receive function with the necessary parameters. The user process is then blocked waiting till data comes in / a timeout occurs. The format of a simple UDP receive call is shown below:

```
recvfrom(sockfd, buff, BSIZE, 0, (struct sockaddr *) &raddr, (int *)rsize)
```

Buff and BSIZE refer to the memory region where the incoming data has to be stored, and the size of the same. Raddr and Rsize refer to the remote host address details (IP + port) and the size of the structure respectively.

When a user application makes this call, the following things happen:

1. The user application is blocked waiting on the socket.
2. When data comes in thorough the link, the hardware causes a *splimp* interrupt.
3. This interrupt executes the link layer that takes the data from memory and creates the mbuf structures. It also removes the link level headers. It then adds this memory to the IP queue.
4. After adding it to the IP queue, the *splnet* interrupt is raised.
5. The interrupt causes the execution of the IP layer code. The IP layer code removes the IP headers. Based on the higher-level protocol (TCP / UDP), the mbuf structure is added to the corresponding protocol input queue.
6. The TCP / UDP level code then processes the data in queue. It goes through the doubly linked list and locates the internet control block corresponding to the particular port. Once this port is determined, the reverse pointer is then used to determine the process.

7. Once the process is determined, it is sent a SIGIO, which makes the process runable.

Chapter 5

High Level Design

The design proposed in this project is based on the host-to-host paradigm presented in chapter 3. This chapter first goes into the design goals set forth for the proposed socket migration system. Then the assumptions and constraints are discussed. The next section will then discuss the actual high-level design of the migration system and the overall steps involved. The last section will sketch out the differences involved in server socket migration.

5.1 Design Goals

The following design goals guided the design of the MIGSOCK migratable socket system:

Transparency The whole socket migration process should be transparent to the applications on either side of the socket connection. There should be no change in application code, and it should support most current programs.[1]

Kernel Level Support The migration should be implemented at the kernel level. The implementation should be at the transport layer(OSI Level 4) as opposed to implementations that are wrappers to the standard sockets library (in layer 5). Kernel level support results in overall faster migration as there are no user overheads.

Compliance with TCP Semantics The implementation should preserve the TCP end-to-end semantics. Additionally, there should be no loss of data when sockets are migrated from one host to another. This makes the MIGSOCK implementation compatible with other normal TCP stacks, and enables the use of standard TCP parameters like windows, sequence numbers etc.

State Preservation The implementation should preserve the state of the source socket and transfer it to the destination host. The socket should not be re-established by going through the three-way handshake (SYN - SYN ACK - ACK). This supports faster migration by minimizing messaging overheads.

[1]Some programs like FTP store the IP address of the other end of the socket connection. Thus these programs cannot be migrated without application support.

Interoperability The implementation should be able to interoperate with other TCP implementations. In other words, the implementation should be able to communicate with other TCP stacks that do not have these special modifications. A host that has socket migration support should be able to communicate with a host that does not support it. Of course migration will not be possible unless both support it; but standard TCP communication should work correctly.

Performance There should be no significant slowdown in the network subsystem with this implementation. The implementation should be scalable and should not cause adverse delays in processing data packets.

Cross-platform portability The implementation of MIGSOCK, with little or no modification should be portable across as many operating systems as possible, just as TCP is, so as to facilitate socket migration across these OSs.

Modular Design Migration support should be implemented as a kernel module to support dynamic loading and the ability to disable support without rebooting a new kernel.

Ease of Integration into existing process migration environments The implementation of MIGSOCK should be easy to integrate with both new and current process migration systems.

One note on interoperability: Our implementation does function normally with other TCP stacks when there's no migration. However, when migration is initiated, it does not check to see if the remote host supports migration. This results in certain unpredictable behavior on the remote side not supporting socket migration. This can be easily rectified by using a handshake protocol to determine the compatibility of the remote host.

5.2 Assumptions and Limitations

The following are the assumptions made about the operating context necessary to support socket migration.

1. It is assumed that to support migration, both ends of a socket connection must support MIGSOCK. In other words, both the end points must enable the MIGSOCK kernel option.

2. A process migration system must be in place for socket migration to accompany it. We do not implement a process migration system; rather we use CRAK in our tests and demonstrations.

3. A socket handoff requires that the destination process be in a position to use a socket descriptor in a connected state.

4. MIGSOCK is currently available only for the 2.4.6 Linux kernel. Other versions of Linux and variants could be supported once the design is implemented and tested.

5. The security aspects of MIGSOCK are not considered. For example, MIGSOCK is prone to a "man-in-the-middle" attack. Security issues are further discussed in Section 6.4.

6. There will be an unavoidable latency when a process migrates from one host to another. This delay is caused due to the serialization of the process state, transfer of process state from one host to another, and the re-setup of the process state and data structures in the kernel. The timeouts involved in the system should be large enough to account for this delay. Alternatively, mechanisms should be present that effectively ignore this lag.

7. There are some services that copy the IP address and port information of a remote socket into user space for future use. Even after migration, the service might try to connect to the old host, though the kernel has the right information associated with the socket. Thus, any changes made to the kernel data structures after a migration should be propagated to the application. This problem would not occur if the application would refer to the kernel data structures whenever it needed to determine the host and port information of a remote socket interface.

8. We assume that the transfer of serialized socket state from source to destination is handled by the application calling the MIGSOCK module. Thus, the kernel module does not transfer the state from source to destination. We will transfer the state to and from the user application layer via a data buffer. The user is responsible for writing this buffer to a file (or otherwise), transferring it to the destination, and passing it back to the system call via a data buffer.

5.3 High Level Design

Although the design of MIGSOCK is scalable, the design algorithm is given for hosts communicating over a single socket connection. This is done for the sake of simplicity. The algorithm is illustrated in figure 5.1.

1. One of the processes decides (or the system decides for it) that it will migrate to a new host. The process initiating the migration is referred to as the control process. The migration mechanism recognizes that there are one or more open sockets for this process.

2. For each open socket, the process on the source host needs to inform the remote process that it intends to move to a destination host. The generation of new TCP packets should cease from both ends, and existing un-transferred packets should remain in the buffers. Any socket calls by the remote process should block appropriately as they would for any delay in the transfer of the

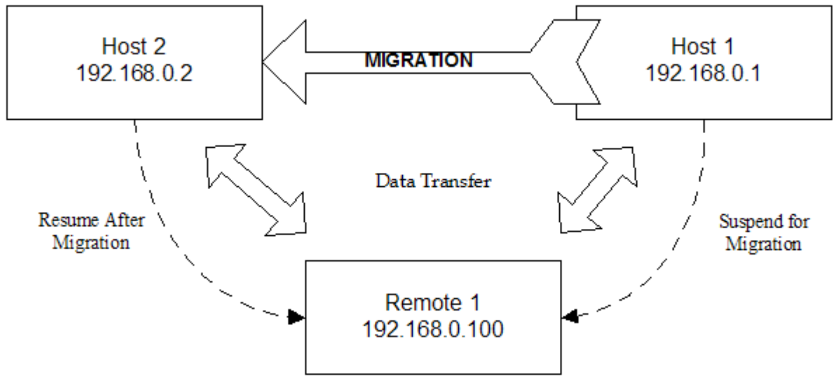

Figure 5.1: Overview of Socket Migration

TCP data. Once this is achieved for all open sockets, then communications can be considered suspended. When resuming communications, the socket descriptor at the remote process should be updated with the new host and port information of the migrating process.

One important thing to note at this step is that the remote process might attempt to create a new socket connection to the migrating process at its source host and port. This connection attempt will obviously fail since the process is no longer running on that host. The remote process programmer must be cognizant of the fact that the source socket might have relocated, and must code the application to use the existing socket descriptor to obtain the destination host and port information so that he can be assured of continued contact with the migrated process.

3. While the relocating process executes its migration, it must serialize all of the kernel data structures that make up its communication state and send this to its new host along with the rest of its execution state. Once all of the state has been received on the destination machine, these structures can be returned to the kernel as they would if the process had terminated normally. In other words, the kernel must clean up and free all of the memory associated with that process.

4. When the migrating socket arrives at its destination host, it must de-serialize this data and recreate new data structures to represent the state of its existing sockets.

5. When the relocating process is reactivated on the new host, it will send a TCP message to the server informing it that it is up and running on host X and port Y, and TCP communications can proceed as normal. Communications are resumed. The TCP buffers will empty as packets pass each other on the wire, and the socket system calls on the stationary host will return (wake up) allowing that application to continue sending and receiving data with the relocated process.

These five steps constitute the major sections of work involved for successful transfer of the network socket along with the process. They highlight the important pieces of the problem and ignore the intricacies that will be dealt with in the ensuing sections of this paper where the implementation is laid out in detail.

5.4 Considerations for Server Socket Migration

Server sockets do not necessarily require special consideration while designing a socket migration system. Server sockets are different from client sockets in the way that they are set up. A server program listens for the incoming connection requests instead of initiating them. Once a socket descriptor is part of a live connection however, there is no difference between the server socket and the client socket. The reason for this is that the four components that uniquely identify a socket – *(hostIP, hostPort, remoteIP, remotePort)* – are the same for both client sockets and server sockets.

Socket descriptors that are used during the listen() system call (setup using the bind() system call) are a different story. Such socket descriptors are not *connected* because they only consist of a local IP address and port. Migration of this type of socket descriptor is probably not of practical use, as *listening* sockets are mainly used to provide service to clients that know their IP address and port[2].

Server sockets are generally bound to a port where they listen for incoming connections. These ports that they bind to are generally well known and publicized. For example, the HTTP server generally listens on port 80, the FTP server on 21, SSH on 22, POP3 on 110, SMTP on 25, telnet on 23 etc. Thus, when these server processes move from one host to another, finding the server process becomes a problem for clients. For example, a client might want to reach the FTP server on 128.2.213.151. When this server process shifts to 128.2.64.158, the port on the first machine is closed and thus the client will not be able to connect to it.

Another problem arises with port sharing and conflicts. A port that a migrated process normally listens to could be in use at the remote host. Thus, port sharing is not possible, but socket connections that were formed before the server process migrated would still be valid. The problem with changing IP address and port numbers could be solved by a discovery service. For these reasons, we will deal only with *connected* sockets.

[2]An argument can be made that clients can use directory services to locate a process, and thus, such processes could benefit from server socket migration

Chapter 6

Detailed Design

The purpose of the detailed design chapter is to formalize and elaborate on the high level design previously presented and to provide a blueprint for implementation. We will start by introducing the API that will be provided to user level programs. We will show how these user level programs should utilize the interface to achieve socket handoff or migration. Next we will give guidelines for how the two communicating endpoints should behave at the onset, during, and at the conclusion of socket migration. Finally we will touch on some considerations for security.

6.1 MIGSOCK Application Programming Interface

The following system calls comprise the socket migration API. In the next section we will show that these can be coded as indirect system calls via the ioctl() call, but are essentially no different than a typical kernel interrupt.

1. *Send_Migration_Request(pid, sockfd)*

   Input Arguments
   pid Process ID of running process that is to be migrated.
   sockfd Socket file descriptor number for the socket that is to be migrated.

   Description
   This is the first system call invoked by the user program from the source host to initiate a socket migration. It sets a local flag in the kernel socket structure to indicate that socket migration is underway. It then constructs and sends a message to the remote socket indicating its intention to migrate its half of the socket connection to a new host.

   Return Value
   integer Returns 0 on success or a negative error value on failure.

2. *Serialize_Socket_Data (pid, sockfd, buf)*

Input Arguments
pid Process ID of running process that is to be migrated.
sockfd Socket file descriptor number for the socket that is to be migrated.
buf A byte array that is large enough to hold all of the serialized data that describes a local socket. This buffer is passed to and populated by the kernel.

Description
This function is also called from the source host and should follow the Send_Migration_Request call. It captures all of the pertinent state data for the socket indicated by the sockfd argument. This data is organized into the byte array buf that is passed from user space. When the function returns, this buffer will contain a serialized representation of the socket state. It can be written to a file or otherwise passed to the destination host where the socket can be reconstructed. This buffer should not be modified before it is passed to a deserialize function; otherwise it could become unreadable. The function also cleans up the now unused socket resources; the socket structures are removed or returned to the kernel.

Return Value
integer On success, returns the number of bytes written to the buffer. Returns a negative error value on failure.

3. *Deserialize_Socket_Data (pid, sockfd, buf)*

   Input Arguments
   pid Process ID of running process that is to be migrated.
   sockfd New socket file descriptor identifying the socket on the remote host.
   buf A byte array that is large enough to hold all of the serialized data that describes a local socket. This buffer is passed to and used by the kernel to populate the socket data structures. It should be the same data that was returned to the user in the buf argument of the Serialize_Socket_Data function.

   Description
   This function is invoked when the destination process is ready to resume the connection with the remote host. It is almost identical to Deserialize_And_Restart except that it assumes the destination process is running. It reconstructs the socket state using the serialized data from buf, allocating a new local TCP port in the process. It assumes that the socket file descriptor that is passed in as an argument is a valid socket that is not currently connected (ie - it is the result of a call to socket(), but not connect()). When this call returns, the sockfd that was passed in is now connected to the remote process so that when it wakes up, communication can proceed as normal.

   Return Value

integer Returns the new TCP port number that was generated for this socket.

4. *Deserialize_And_Restart(pid, sockfd, buf)*

   Input Arguments
   pid Process ID of running process that is to be migrated.
   sockfd New socket file descriptor identifying the socket on the remote host.
   buf A byte array that is large enough to hold all of the serialized data that describes a local socket. This buffer is passed to and used by the kernel to populate the socket data structures. It should be the same data that was returned to the user in the buf argument of the Serialize_Socket_Data function.

   Description
   This function is invoked when the destination process is ready to resume the connection with the remote host. It is almost identical to Deserialize_Socket_Data except that it assumes the destination process is currently stopped. It reconstructs the socket state using the serialized data from buf, allocating a new local TCP port in the process. It assumes the socket file descriptor that is passed in as an argument is a valid socket that is not currently connected (ie - it is the result of a call to socket(), but not connect()). When this call returns, the local process is added back to the running queue and the sockfd that was passed in is now connected to the remote process. When the remote process wakes up, communication can proceed as normal.

   Return Value
   integer Returns the new TCP port number that was generated for this socket.

5. *Send_Remote_Restart (pid, sockfd, msg_contents)*

   Input Arguments
   pid Process ID of running process that is to be migrated.
   sockfd New socket file descriptor identifying the socket on the remote host.
   msg_contents Contains new host and port information, along with old host and port information so the remote process knows which of its sockets is now resuming communication.

   Description
   This function is the destination host counterpart of Send_Migration_Request (which is called from the source host). It sets a local flag in the kernel socket structure indicating that the socket migration is complete. It then constructs and sends a message to the remote socket indicating that it should wake up any waiting threads and resume normal communication.

Return Value
integer - Returns 0 on success or a negative error value on failure.

6. *Get_Local_Host (pid, sockfd)*

   Input Arguments
   pid Process ID of running process
   sockfd Socket file descriptor number

   Description
   This function returns the local IP address of the host that this process runs on. It requires the pid and sockfd inputs because it extracts the address value from the socket data structure to ensure that the address is in the correct format when passed to the remote host.

   Return Value
   integer Returns the IP address of the local host in a format appropriate for passage to the remote host.

6.2 Interface Usage

6.2.1 Source Side

Initiating a socket migration involves two major steps on the source side. First, notify the remote party that this socket will migrate. Second, capture and serialize the socket state. An algorithm for using our API is presented here:

```
// Obtain the pid and socket fd for the migrating socket.
Send_Migration_Request(pid, fd)
num_bytes = Serialize_Socket_Data (pid, sockfd, buf)
write(somefile, buf, num_bytes);
// Transfer somefile to destination host
```

6.2.2 Destination Side

Finishing a socket migration involves three major steps on the destination side. First, create a socket on or steal a socket from the destination process. Second, replace this socket's state with the serialized data stored in the file. Finally, notify the remote process that migration is complete and that communication can resume. An algorithm for using our API is presented here:

```
// Create a socket using the socket() call for the process
// Obtain the pid and socket fd for the process and newly
// created socket.
read(somefile, buf);
```

```
newport = Deserialize_And_Restart(pid, sockfd, buf)
newhost = Get_Local_Host(pid, sockfd)
// Add the newport and newhost to msg_contents
// Get oldhost and oldport from buf and add this to msg_contents
Send_Remote_Restart(pid, sockfd, msg_contents)
```

6.3 Process or Thread Behavior

Recall that socket migration is supported only for sockets that are in the connected state. Refer to the message timeline in Figure 6.1 for establishing a socket connection . Note that once the TCP connection is *established*, socket migration can proceed. When socket migration finishes, the TCP link is once again in the *established* state.

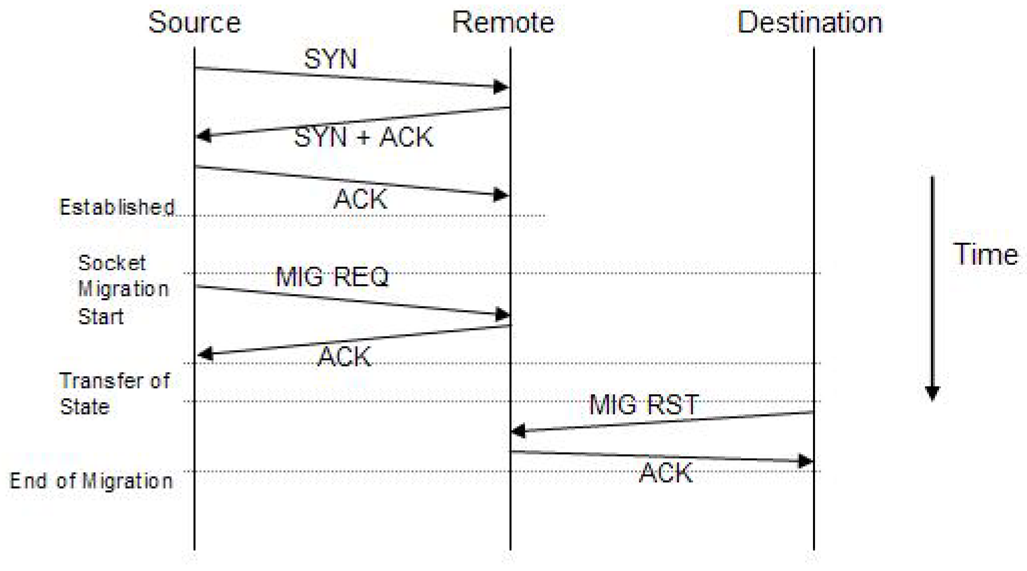

Figure 6.1: TCP Message Timeline

To address the behavior characteristics of the local process and the remote entity during socket migration, we'll split time into three periods: migration initiation, state relocation, and migration conclusion. The initiation period includes the capture and serialization of the socket state. Once this has completed, the state must be transferred to its destination host where migration will conclude. When it arrives, the recreation of the sockets constitutes the final stage of the migration. For the following discussion, the migrating socket starts on the *source* host and process and winds up on the *destination* host and process. The other half of the socket connection is called the *remote* entity.

6.3.1 Stage 1 – Migration Initiation

Migrating Process As noted in the previous section, migration of a local socket begins with the Send_Migration_Request() system call. Once underway, no data should be written to that socket by the local host. The user should suspend the process if necessary before calling Send_Migration_Request() to ensure that no more packet data is generated locally. Send_Migration_Request() sets a migration flag in the local sock structure and then generates a special TCP message to the remote process.

The packet transmission function checks the migration flag before every send, and sets a bit in the outgoing packet's header to reflect this flag. Therefore once the flag is set, all outgoing packets in this socket's queue will be modified. The special TCP message is generated by Send_Migration_Request() for the case where there are no messages in the outgoing queue. In other words, its only purpose is to make sure the remote party gets the message immediately if there are no other news bearers around. The receiver function at the remote host will watch out for this special message and drop this packet so that the data contained within is not accidentally passed up to the application layer. The remote party will send a TCP ACK before the packet is dropped.

Remote Process In addition to dropping the packet, the receiver function sets its own local socket flag to indicate that a migration is under way for its communicating entity. This flag will be checked each time before a send() function is called on this socket. If it is set, then the send() will be added to a wait queue and put to sleep. This will put the thread to sleep in the case of a multithreaded application, or the entire process otherwise. Of course this sleep merely prevents additional packets from joining the outgoing queue. Any packets already in the socket queue, including the ACK to the special TCP messenger packet, will be sent by the kernel. Note that if this remote process or thread makes no attempt to send data over the socket throughout the entire migration, then it will continue to run unimpeded. It will only block when attempting to send data to the socket.

Migrating Process Again Before proceeding, the local process should sleep for a second or two to ensure that all of the packets in the queue of the remote entity have been received. It should receive ACKs for any TCP messages it sent, and should send an ACK for any further messages it receives. Only when the queues are empty and the TCP guarantees are met should the socket migration proceed. Once this is the case, the Serialize_Socket_Data() call can be made on the local socket. Since the local socket state will be static (no more traffic), this call can safely copy the important socket data structures into user space. The last step is for the local socket to be destroyed, returning the resources to the kernel.

This involves freeing any memory associated with the socket, releasing the port, and other related items.

6.3.2 Stage 2 – State Relocation

Migrating Process In the case of a socket handoff, this process should continue to run as normal; the only difference being that it has one less socket now. Any calls to that socket should fail. In the case of a process/socket migration, the local process will be killed anyway so no specifications are required for its behavior during this stage. The state is transferred from the source to the destination in the form of the buffer that was returned from the Serialize_Socket_Data() call. This buffer can be written to a file, passed over the network, or transferred in any other creative way. But the bytes within the buffer should not be modified because the deserialize functions expect to be able to parse the buffer in the same form that it is created.

Remote Process To reiterate what was said above, the remote process can continue to execute and accomplish tasks during this stage of the migration unless it attempts to write data to the socket. Recall that a local flag remains set throughout the duration of the migration so that if such an attempt is made, the process will sleep indefinitely until socket migration has finished.

6.3.3 Stage 3 – Migration Conclusion

The behavior of the local process in this stage is slightly different depending on whether a socket handoff or process/socket migration took place. This is why two different deserialize functions exist. For a socket handoff, the destination process already exists and is running. For a socket migration, the destination process is being created along with the new socket, so it is in the suspended state. Both scenarios are presented here.

Socket Handoff

Migrating Process The first step in completing a socket handoff is to identify a local socket descriptor that will now be connected to the remote host. Of course, this socket should be running the same upper layer protocols as the source and remote processes. The TCP and lower layers will work fine regardless of which application layer protocols are running, but Telnet can never talk to FTP for example. Once an appropriate socket is identified, it will be reconnected to the remote party by passing the state buffer to the Deserialize_Socket_Data() function. This will update all of the addresses, sequence numbers, and other aspects of the socket with the values from the source process. This function will return the local port number of this socket. The user process should then collect the local host IP address, and extract the old source port and IP address values from the state buffer. It should pass these four values (old port, old IP, new port, new IP) to the

Send_Remote_Restart() function which will generate a second special TCP message to be sent to the remote host. This is the wake-up call to the remote send().

Remote Process The remote receive function is always checking for the arrival of the special TCP message. When it detects it, it first changes the migration flag to indicate that socket migration is finished for this socket. The receive function then dissects the contents to retrieve the old and new values for the host and port. It needs the old values so that it can look up the socket descriptor in its hash tables, and send a signal to the socket wait queue in the case of a blocking send(). Once it has retrieved the socket, it replaces the host and port values in the data structures with the new values, rehashes the socket, and then signals the wait queue. If the process is blocking, it will wake up at this point and resume the send(). If the process was not waiting on a send(), then it has been executing all along and has no idea that the socket has changed at all.

Socket Migration

Migrating Process The first step in completing a socket migration is to create a new local socket descriptor that will now be connected to the remote host. This is accomplished using the standard socket() call. This socket should have the same descriptor number as it did on the source host for the application layer to continue to refer to the correct socket. Once an appropriate socket is created, it will be reconnected to the remote party by passing the state buffer to the Deserialize_And_Restart() function. This will update all of the addresses, sequence numbers, and other aspects of the socket with the values from the source process. Finally it returns the local port number of this socket. The user process should then collect the local host IP address, and extract the old source port and IP address values from the state buffer. It should pass these four values (old port, old IP, new port, new IP) to the Send_Remote_Restart() function which will generate a second special TCP message to be sent to the remote host. This is the wake-up call to the remote send(). At this point the state of the migrated process is changed from *suspended* to *running*.

Remote Process The behavior of the remote process is the same as for a socket handoff.

6.3.4 Summary of Steps for Socket Handoff

In the following algorithm, we assume that the socket is being handed off from the Source computer to the Destination computer, with the other end-point of the connection being the Remote computer.

1. The process holding the socket is first suspended.

2. The socket file descriptor is then determined.

3. A message is then sent to the Remote computer indicating a socket migration. Any writes on the remote host would then be suspended till migration is complete.

4. The state of the socket is captured and written to a file.

5. The state is then transferred to the Destination host using a shared file system.

6. The process id and socket file descriptor of the process receiving the handoff socket is determined.

7. The socket currently held by the process on the Destination is destroyed.

8. The socket state is replaced by the transferred state.

9. A message is sent to the Remote host indicating the new IP address and Port number of the socket, causing an update of the remote host socket structures.

10. Normal communication is resumed after this point and the socket behaved like a normal ESTABLISHED socket.

6.3.5 Summary of Steps for Socket Migration

1. The user program is first suspended.

2. A message is sent to the Remote host indicating a start of migration.

3. The state of the open socket is serialized to file.

4. The socket data structures are then destroyed, so that any calls to close() from CRAK would not result in the closing of the TCP connection.

5. The CRAK checkpoint function is called and the process state is serialized to file.

6. The process is KILLED by CRAK

7. On the remote host, the CRAK restart function is called to restore the checkpointed process.

8. The process is put to sleep.

9. The state of the open socket is restored from file by the MIGSOCK restart routines.

10. A message is sent to the remote host indicating that migration is complete.

11. The sleeping process is then woken up, restoring normal operation.

6.4 Considerations for Security

Though we do not directly deal with any security issues with the proposed Migratable Sockets implementation, we do realize the vulnerability of the proposed implementation to spoof attacks. In its current form, any host monitoring the data connection can step in after the initial migration request. As shown in Figure 6.2, a malicious host can take over a live socket connection by sending a false restart message after the migration request message.

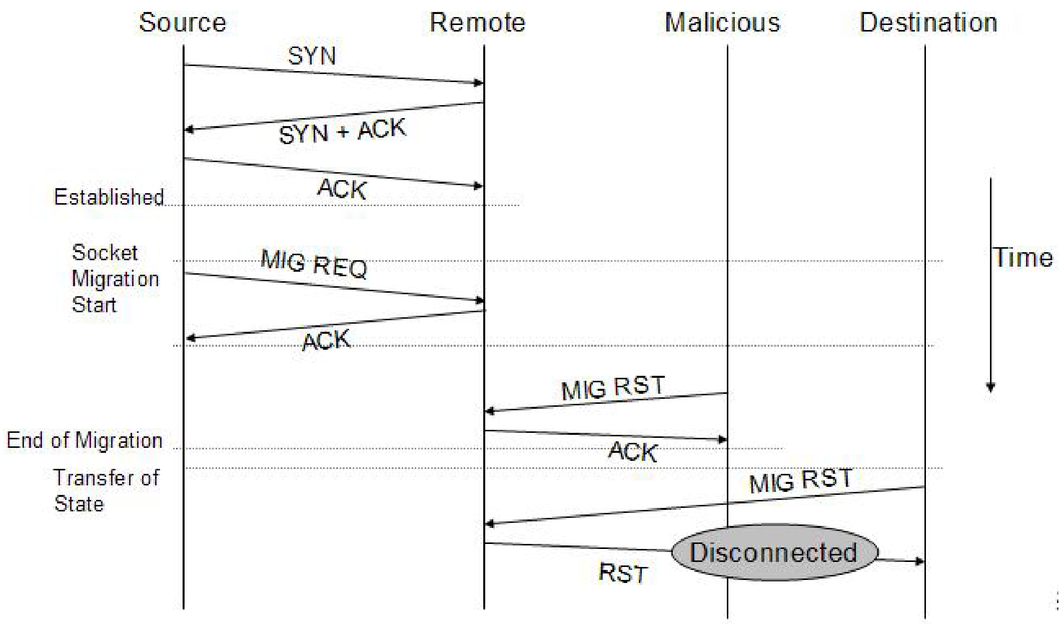

Figure 6.2: Possible Security Attack

The above security issue can be easily solved by using a Public Key Infrastructure or some form of Digital Signature. As shown in Figure 7.2, the message header includes a *code* field that can be used for a unique signature known only to the two communicating hosts. In the current implementation, this code is fixed at the time when the kernel and its supporting user programs are compiled. However, this can be easily modified, and the code can be changed every time a message is sent.

Chapter 7

Implementation

In this section we will show where and how we carried out the detailed design. We will give some general information about the implementation, followed by the actual file and function names for the module and kernel. A minimal amount of code will be included where it is necessary to illustrate a point.

7.1 General Points

To choose an operating system for implementing this project, it is necessary of course to have the complete source code tree. This narrowed the field to two options, Free BSD and Linux. Although the network stack implementation for Free BSD is more carefully coded and commented we chose Linux for two reasons. First of all, because of the commercial presence of Linux, it is more likely for instances of practical application of our solution than there would be for Free BSD. Secondly, there is already at least one process migration solution for Linux – CRAK – that provides us a means to integrate and test our project realistically.

We implemented MIGSOCK on Linux Kernel version 2.4.6. Modularity was achieved by using the Linux device driver architecture. For more information about this architecture, please refer to Linux Device Drivers [24]. The kernel module was implemented in C and was tied to the device file */dev/migsock*. Any applications wishing to employ our module functions must first open this device and get a file descriptor for it. Then they must pass this file descriptor as the first argument to an ioctl() call.

The API functions listed in the detailed design chapter are coded within a Linux kernel module that can be dynamically loaded and unloaded from the kernel. This makes the design less invasive than if it were all done within the kernel itself. However, parts of the design require modifications to the packet send and receive functions in the TCP layer. The TCP code exists within the kernel, making modifications unavoidable in this case.

To understand the nature of the changes we made to existing TCP files within the Linux kernel, please refer to Figure 7.1. We think of the implementation of TCP as being split into two halves, a top-half and a bottom-half. This figure shows the handshake between the two halves for a socket read(). A write() would be similar except that data would move in the opposite direction.

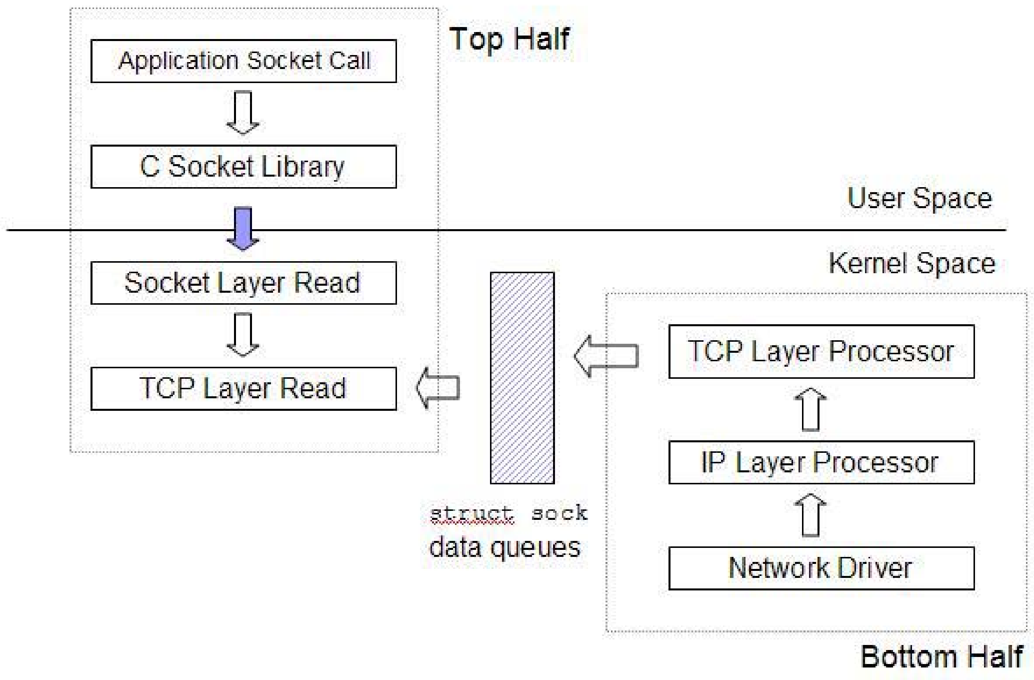

Figure 7.1: Interaction between top-half and bottom-half of TCP implementation

Bottom Half When a packet comes in on the network device driver, it is passed up to the IP layer and then to the TCP layer. The packet does not yet belong to a particular socket. The TCP layer determines the socket that this packet belongs to from the IP and port combination. It then places the packet into the read queue of this socket.

Top Half A user space application initiates a read() on the socket descriptor. This read call propagates to the TCP layer tcp_recvmsg() function. This function coordinates the removal of the packets from the read queue that belong to this particular socket. It buffers the data and passes it back up through the layers to the user calling function.

All of the MIGSOCK functionality in the module file discussed below can be thought of as part of the top half, in the kernel space above the TCP layer. The changes made to the kernel files affect the bottom-half of TCP, except for the changes to tcp.c, which affect the top-half of TCP.

In the sections that follow, we will need to specify our location within the kernel source directories. We will identify the root of the source tree install point as SOURCEHOME and give relative pathnames from there. The location of the MIGSOCK module implementation is SOURCEHOME/net/ipv4/migsock.

7.2 Header Files

It was mentioned in the previous chapter that a field would be added to one of the socket data structures in the kernel enabling the use of the migration flag.

First of all, there are two kernel header files that include modifications to support socket migration:

```
SOURCEHOME/include/linux/sched.h
SOURCEHOME/include/net/sock.h
```

All changes made to the kernel are controlled by a compiler flag. *CONFIG_MIGSOCK* is defined whenever socket migration in selected in the kernel configuration scripts. The definition of this variable controls the inclusion of the MIGSOCK code into the kernel by enclosing it between #ifdef and #endif compiler directive pairs.

The change to sched.h is an additional member added to *struct task_struct*. It is of type wait_queue_t and is used in the Deserialize_And_Restart() function implementation to add the migrated process to the wait queue.

The change to sock.h is an additional member to *struct sock* called *migsock_ptr*. It is of type *struct migsock_info* which is defined in migsock.h. This pointer holds the flags used on both the migrating side and remote side of the socket. This is the flag that facilitates the communication between the module functions and the TCP functions.

There are two primary MIGSOCK header files supporting migratable sockets. migsock.h is appropriate for inclusion in kernel and module files as it contains all of the macros and definitions employed in the implementation. migsock_user.h is only appropriate for inclusion in the user level applications that will invoke the migsock functionality. It contains the macros for the interface specification, and some structures, but excludes structures that use data types available only to the kernel. Both of these header files are found in the SOURCEHOME/net/ipv4/migsock directory. Any other kernel directory should make a soft link to migsock.h to include it. Any user applications should make a soft link to migsock_user.h in their local directory. Some points to note about each file follow:

- *migsock_user.h*
  A user program desiring to use the MIGSOCK API will need to specify the particular ioctl() call it is making by referring to it with the macro given in this header file. The macros are:

```
MIGSOCK\_IOCTL\_REQ
MIGSOCK\_IOCTL\_RST
MIGSOCK\_IOCTL\_GETHOST
MIGSOCK\_IOCTL\_TOFILE
MIGSOCK\_IOCTL\_FROMFILE
MIGSOCK\_IOCTL\_RESTART
```

  An actual ioctl() call looks like:

```
retval = ioctl(fd, cmd, (unsigned long)arg);
```

  where
  fd the file descriptor of the open device file (/dev/migsock)
  arg a structure of type struct migsock_params. It must be allocated by the user program.
  cmd one of the macros listed above.
  Next the header file includes the definition for the migsock_params structure. As the name implies, it facilitates the passage of parameters to the ioctl() call:

```
struct migsock_params {
   /* This must be first, and of sufficient
    * size to store serialized socket data */
   char data[MIGSOCK_MAX_BUFF];

   pid_t pid;
   int sockfd;
};
```

  In the message passing ioctl() calls – IOCTL_REQ and IOCTL_RST – the data value of this structure variable will be populated with *struct migsock_config_data*:

```
struct migsock_config_data {

  /* This must be first.  It will hold the message used by the
   * receive side to determine that this is a special TCP packet.
   */
```

```
    char id[MIGSOCK_ID_CODE_SIZE];

    unsigned char opcode;           /* Type of operation */
    __u32 ohost;                    /* Old Source Host */
    __u16 oport;                    /* Old Source Port */
    __u32 nhost;                    /* New Source Host */
    __u16 nport;                    /* New Source Port */
};
```

The *id* member of the migsock_config_data structure will be populated with the string in MIGSOCK_ID_CODE. This string is used by the source, destination, and remote hosts to signify special migration messages. See section 6.3. The migsock_config_data structure represents the message format shown in Figure 7.2.

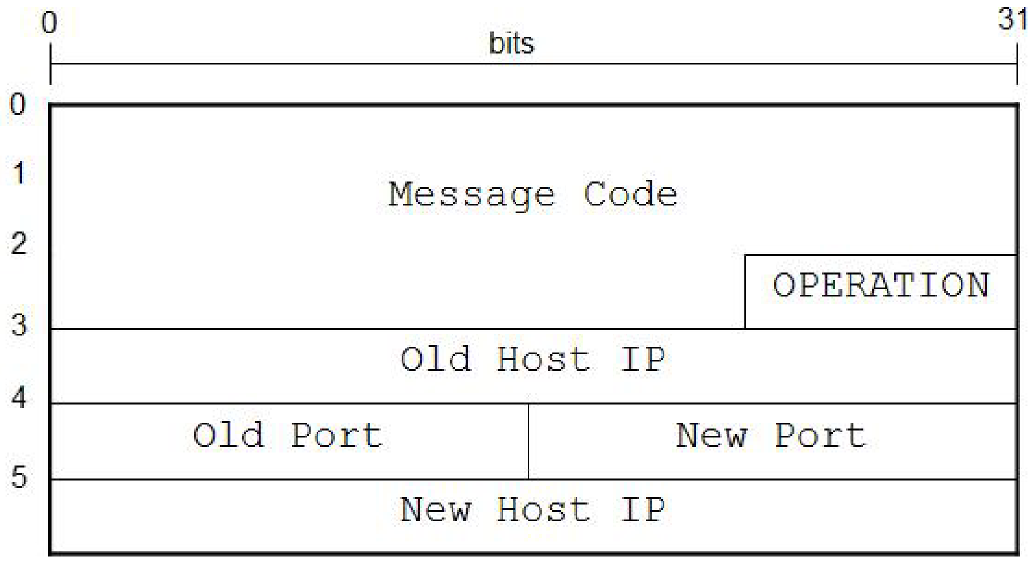

Figure 7.2: Socket Migration Message Format

- *migsock.h*

  This file includes migsock_user.h and therefore contains everything in it. In addition, it declares the functions that are defined in the migsock_mod.c module file and declares two more structures. The first structure is:

```
struct migsock_info {

    /* set to SEND_REQ after request by migrator */
    short req_flag;
```

```
      /* set to RECV_REQ after receiving request by remote */
      short rcv_flag;
   };
```

As mentioned above, the *migsock_ptr* member of *struct sock* is of this data type. The req_flag is set on for the migrating sock structure to indicate that migration is underway. The rcv_flag is set on the remote sock structure to indicate the same thing.

The other structure in this file is the migsock_serialized_data structure. It is too large to list here, but in summary it has members that will hold important values from *struct socket*, *struct sock*, and *struct file* that are necessary for state representation. It also holds the entire tcp_opt structure.

7.3 Module File

Each of the functions referred to in the detailed design chapter can be coded as part of the module. The modules interact with the TCP kernel functions via the migration flags. Recall that during the initiation stage, there is a flag that is set by the Send_Migration_Request() module function to indicate that migration is underway. This flag is set within the sock structure representing this socket, making it readable by the TCP layer as well. When the TCP packet transmit function is called, it checks this flag for every socket and updates the header of outgoing packets accordingly if it is set. When this flag is reset following the migration, the TCP transmit function treats packets from this socket no differently than any other socket in the kernel.

On the remote host, there are no modules that need to be loaded because all of the migration support is supplied within the kernel TCP implementation.

Every function takes a pid and sockfd as input arguments. The reason is that the first step in each of these functions is to retrieve the corresponding kernel data structures belonging to this process and socket before proceeding.

There is a single file that implements all of the module functions that are declared in migsock.h: *SOURCEHOME/net/ipv4/migsock/migsock_mod.c*.

The first step in implementing the module functions is to define the file_operations structure for /dev/migsock:

```
struct file_operations ops = {
   owner:         THIS_MODULE,
   ioctl:         migsock_ioctl,
   open:          migsock_open,
   release:       migsock_release,
};
```

Then migsock_open() and migsock_release() are implemented within the module file to add functionality to the standard file open and file release functions. They simply invoke the MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT macros respectively. The migsock_ioctl() implementation extends the standard ioctl() call by performing a switch on the cmd argument – one of the IOCTL macro – and invoking the corresponding module function. The function names, with the corresponding IOCTL macros, relate to the API functions listed in the design section as follows:

Macro	function definition	API function
MIGSOCK_IOCTL_REQ	sys_migsock_req	Send_Migration_Request
MIGSOCK_IOCTL_RST	sys_migsock_rst	Send_Remote_Restart
MIGSOCK_IOCTL_GETHOST	sys_migsock_gethost	Get_Local_Host
MIGSOCK_IOCTL_TOFILE	sys_migsock_tofile	Serialize_Socket_Data
MIGSOCK_IOCTL_FROMFILE	sys_migsock_fromfile	Deserialize_Socket_Data
MIGSOCK_IOCTL_RESTART	sys_migsock_restart	Deserialize_And_Restart

7.4 Kernel Source Files

There are four bits within the TCP header that are reserved for future use. We appropriate the least significant of these bits, res1, for our purposes. On the send side, the TCP function responsible for transmitting a single packet must be modified to check the socket structure containing the migration flag to see if it is set. If so, it sets this bit in the header of outgoing packets in this socket. When the receiver catches a packet with this bit set, it knows to check the contents of the packet to see if it is the special message. The contents of this message include an identifying byte sequence, an opcode, and host and port information. A special migration message is identified by comparing the byte sequence to the expected value. A match tells the receiver to ACK and discard this packet. The opcode identifies the packet as a migration initiation or migration completion message. In the case of a migration completion message, the receive side will extract the host and port information before dropping the packet.

In chapter 6 we explained how on the migrating hosts the module functions communicate with the TCP functions using the migration flag. On the remote host, there are no module functions. However, the migration flag in the sock structure is still used by the TCP code to facilitate communication between the packet receive function and the send() function. The packet-receive function runs within the network daemon and is therefore independent of any application layer processes. The send() function is invoked by application processes to pass data to the socket. The packet receive function sets the migration flag for a particular socket whenever a migration initiation request comes in. The send() function checks this flag each time before sending a packet out on the socket. If the flag is set, it will block. When the packet receive function gets the message indicating migration is finished, it unsets the migration flag and signals the wait queue where the send() function is blocking.

We have added somewhere on the order of 200 lines of code to the files in the SOURCEHOME/net/ipv4 directory of the kernel. This is a minimal and requisite amount of modification to the kernel implementation of TCP. The four files affected in this directory are tcp.c, tcp_input.c, tcp_output.c, and tcp_ipv4.c. Any changes to these files are enclosed within #ifdef/#endif pairs.

The header file migsock.h would need to be included in each of these kernel files because the modifications use the newly defined macros and structures.

The changes to the files are outlined here.

Bottom Half

- *tcp_output.c*
  In tcp_transmit_skb(): After the TCP header has been built, check the *migsock_ptr* flag in *struct sock*. If it is set, then set the res1 bit in the header to indicate to the remote host that migration is underway.

- *tcp_ipv4.c*
  In tcp_v4_rcv(): This function is changed in 2 major places: before the *struct sock* is retrieved and after.
  Before
  1. Check the res1 bit of the incoming packet.
  2. If set, retrieve the message contents and determine the opcode.
  3. If the opcode indicates this packet is a *restart* message, then grab the old host, old port, new host, and new port information.
  4. Use the old host and port information to look up the *struct sock*.
  5. Set the ts_recent member of the tcp_opt member of *struct sock* to 1.
  6. Apply the new destination host and port values to the socket and rehash it.
  7. Make the destination cache structure member obsolete so that it uses these new address values.

  After
  1. Check the res1 bit of the incoming packet, and check the header value to make sure this is the special message.
  2. If the bit is set and the message code indicates this is a special migration message, then check the opcode.

3. If opcode indicates a migration request, then make sure a migration is not already underway and, if not, set the migration flag in *struct sock* to indicate as much.
4. If opcde indicates a restart message, then make sure that a migration is underway and, if so, reset the migration flag in *struct sock* to 0. Then signal the sleep queue of the socket to awaken any send() attempts.

- *tcp_input.c*
  In tcp_rcv_established(): Check the res1 bit in the packet. If it is set, and if this is a special migration message, then discard it.

Top Half

- *tcp.c*
  New function wait_for_migration_to_finish(): Check the migration flag in *struct sock*. If set, then put the calling thread or process to sleep.

  In tcp_sendmsg(): Near the top of the function, call wait_for_migration_to_finish() to see if it is necessary for the calling thread to wait on a migration in process.

Chapter 8

Integration and Testing

The migratable socket implementation provides many generic API calls that can be easily tailored to fit the needs of an application. To test the API, we came up with two different test scenarios. The first was to test the system with a simple case of socket handoff. The next test was to integrate socket migration with an existing process migration system and to demonstrate the ease of integration.

8.1 Socket Handoff

Socket handoff refers to transfer of one endpoint of a live socket connection to another host. The socket structure is migrated from one host to another, but the process is not. This is often used in a clustered server environment where load balancing needs to be achieved [9] [10].

To demonstrate socket handoff, we migrate one end-point of a live telnet connection on machine A to a telnet process on Machine B. The application process receiving the socket on the destination host must be in a state to communicate on a live connected socket. This is generally achieved through specially coded applications. However, most common programs follow the typical progression of socket calls – bind(), listen(), accept(). The program is ready to receive data on a CONNECTED socket (which is what we handoff) only after the above sequence of steps. Thus, to use the program for socket handoff, it becomes necessary to perform *Socket Stealing.*

To steal a socket, we first establish a live CONNECTED socket. The application is then ready to accept data on this socket. We then overwrite this connected socket with the socket being handed off. Thus, the application will now communicate on the socket being handed off and the old one is closed. Thus, the old socket is stolen to get the application ready for communication over a handoff socket.

To test socket handoff, we created test programs that utilize the MIGSOCK module to achieve the steps previously described to achieve socket handoff. There are two programs that achieve the overall process of socket handoff - *start_handoff* and *finish_handoff*. The source code for both programs is given in the appendix. Below, we provide snippets from the source code and explain the steps achieved.

Code Flow

The handoff is started by calling start_handoff with the process ID of the process whose socket is being handed off, and the socket file descriptor of the socket. This can be obtained by looking into the *fd* directory of the process's /proc directory. For example, for process 3354, the socket file descriptor can be obtained by looking in */proc/3354/fd/*.

Start_handoff does the following on the source host:

1. It first opens the MIGSOCK device file to make further calls using the ioctl() interface.

   ```
   fd = open("/dev/migsock", O_RDONLY);
   ```

2. A migration request is sent to the remote host through the correct socket. For this, the process ID and socket file descriptor are passed to the MGISOCK module through the *arg* structure shown below:

   ```
   arg->pid = atoi(argv[1]);
   arg->sockfd = atoi(argv[2]);
   strcpy (data->id, MIGSOCK_ID_CODE);
   data->opcode = MIGSOCK_SEND_REQ;
   data->ohost = data->oport = data->nhost = data->nport = 0;
   memcpy ((void *)arg, (void *)data,
                      sizeof (struct migsock_config_data));

   /* Send migration request to remote process.
    *  Puts him to sleep. */
   ret = ioctl(fd, MIGSOCK_IOCTL_REQ, (unsigned long)arg);
   ```

3. After this, the state of the socket is serialized and obtained in a buffer by calling the serialize routines.

   ```
   ret = ioctl(fd, MIGSOCK_IOCTL_TOFILE, (unsigned long)arg);
   ```

4. The serialized data is then written to file in MIGSOCK_FILENAME.

   ```
   fout = open(MIGSOCK_FILENAME, O_WRONLY | O_CREAT);
   /* Write this serialized junk in arg->data to file */
   ret = write(fout, (void *)(arg->data),
                               MIGSOCK_SERIAL_DATA_SIZE);
   ```

On the client side, the finish_handoff routine completes the handoff process on the destination host. This function takes the same arguments as start_handoff, except that the process ID passed in is the process ID of the process receiving the handoff socket, and the socket file descriptor the one that is being replaced by the new handoff socket.

Finish_handoff does the following:

1. The MIGSOCK device is again opened as before.
2. The serialized data is then stored in a structure that will be passed on to the MIGSOCK module.
3. The FROMFILE process is then called that initializes the socket file descriptor provided with the data from the socket being handed off. This also creates a new port number for the socket.

   ```
   ret = ioctl(fd, MIGSOCK_IOCTL_FROMFILE, (unsigned long)arg);
   ```

4. The new host and port information is then communicated to the remote host using the RST ioctl() call.

   ```
   ret = ioctl (fd, MIGSOCK_IOCTL_RST, (unsigned long)arg);
   ```

5. Once this is done, normal communication is restored and socket handoff is complete.

8.2 Socket Migration

In order to demonstrate our socket migration functionality, we integrated our MIGSOCK migration mechanism with CRAK, a checkpoint / restart system for Linux. Process migration in CRAK is achieved by checkpointing a process and capturing its state to a file, transferring its state using a shared medium to another computer, and restarting the process there.

To successfully integrate CRAK with MIGSOCK, we needed to make modifications to CRAK. CRAK does not support processes with sockets. When it detects that a file descriptor is a socket, it simply aborts the checkpoint process.

For the initiating process, we modified CRAK's file handling functions to include sockets among the file descriptors it transfers, and prevent it from aborting when it encounters sockets.

For the restart process, we modified CRAK to setup the base socket structures (file descriptor to inode to socket mapping). This was done by calling the socket system call and duplicating the file descriptor to initialize the correct socket file descriptor.

In addition to this, we needed to make some changes to the scheduling of processes. The CRAK restart process works by ultimately replacing itself with the process that was checkpointed. Thus, the process begins executing immediately – even before the socket data structures are initialized by the MIGSOCK socket migration functions. The checkpointed application would then restart and find that the socket was closed, resulting in an abort of socket communication.

Thus, we made changes to CRAK and put it to sleep in a wait queue declared within its Process Control Block (task_struct). The MIGSOCK restart process then restores the state of the open socket from a file, and wakes up the process put to sleep by CRAK.

In terms of code functionality, there are four programs that are used to migrate a process along with its socket from a source host to destination host. The four programs, in the order in which they are run are as follows:

start_migrate Start migrate first serializes the state of a process's socket and writes that to a file. It then modifies the internal kernel structures for the socket such that the TCP socket is not closed when the program is terminated. It is almost identical to start_handoff.

ckpt This is a CRAK program that calls the CRAK module. It serializes the state of the process and writes it to a file. All associated file descriptors are closed during this process. However, socket connects are not closed as the start_migrate has already set the internal kernel variables to indicate a closed socket.

restart This program is run on the destination host to restore the process from the serialized file. It is a part of the CRAK program suite. When this program is successful, it is effectively replaced with the process that was serialized. The basic socket structures are set up, but no sockets are created or connected.

finish_migrate This program completes the whole process migration process. The serialized state of the socket is restored from file. Then the connection is re-established.

The algorithm for finish_migrate is given below. The source code is available in the appendix.

1. The MIGSOCK device is again opened as before.
2. The serialized data is then stored in a structure that will be passed on to the MIGSOCK module.

3. The RESTART process is then called that initializes the socket file descriptor provided with the data from the socket being handed off. This also creates a new port number for the socket.

   ```
   ret = ioctl(fd, MIGSOCK_IOCTL_RESTART, (unsigned long)arg);
   ```

4. The new host and port information is then communicated to the remote host using the restart ioctl() call.

   ```
   ret = ioctl (fd, MIGSOCK_IOCTL_RST, (unsigned long)arg);
   ```

5. Once this is done, normal communication is restored and socket handoff is complete.

A typical sequence of calls to the user programs would look as follows:

```
SOURCE HOST:
# start_migrate <process_id> <sock_fd>
# ck process_id <filename>

DESTINATION HOST:
# restart <filename>
# finish_migrate <process_id> <sock_fd>
```

Chapter 9

Related Work

9.1 Process Migration Systems

9.1.1 MOSIX

MOSIX [15] is a scalable computing cluster operating system developed by the Hebrew University of Jerusalem. Specifically, MOSIX consists of software that modifies the host UNIX operating system with cluster computing capabilities. These cluster computing abilities include the process migration capabilities, load balancing as well as adaptive resource sharing among the nodes in a cluster. There have been 8 versions of MOSIX to date, the most recent being developed for LINUX.

Packaging

MOSIX consists of two packages. The first is a kernel patch that is applied to the Linux source tree. This source patch implements preemptive process migration capabilities at the kernel level. The kernel patch monitors node performance, and if necessary migrates processes from one node to another.

The second is a user-level package that implements load sharing and monitoring. Its primary use is to deliver balanced work distributions between the various nodes in the cluster, without preemptive process migration. This package is useful when a job's tasks are short enough to not warrant migration, but many in number to distribute across the cluster.

Architecture

MOSIX implements two key components as kernel modules – a preemptive process migration mechanism (PPM) and a set of algorithms for adaptive resource sharing. The PPM module can migrate any process, at any time, to any node. Migration is controlled by the outcome of the above-mentioned algorithms. Manual process migration can also be triggered.

Each process has a unique node, called Unique Home Node (UHN), where it was created. I/O and network sockets are bound to this node. Thus, processes can migrate to remote nodes, but would ultimately return to the UHN for user, I/O and network socket communication.

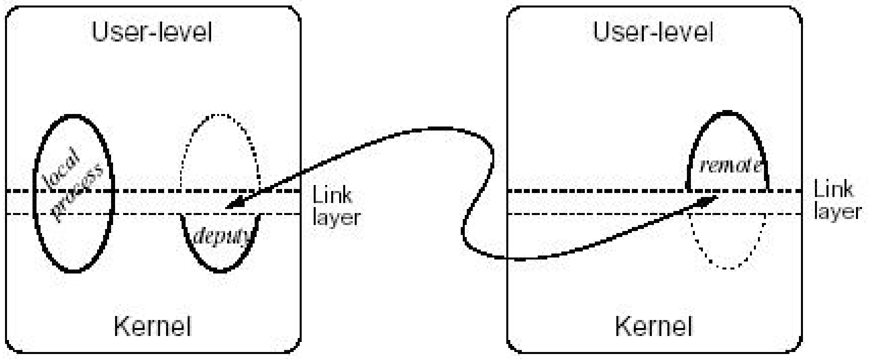

Figure 9.1: Mosix Kernel Architecture

Processes are internally split into two components by the MOSIX kernel. The first component called the *deputy* is never migrated, and always resides at the UHN. The other entity is the *remote*, which is basically the user context [1] The *remote* can be migrated from one node to another. This migration is performed by the *deputy*. The deputy also keeps track of the resources used by the user process (remote). The *remote* calls upon the deputy's services from other nodes for things like user input and I/O.

The way MOSIX handles system calls is very interesting. If a system call is independent of the node, the system call is made locally. Otherwise the call is sent back to the *deputy* at the UHN. The *deputy* then executes the system call on behalf of the *remote*.

Effectively, the user-level / kernel-level interaction is abstracted across the network in MOSIX. This is done by introducing a LINK Level between the user level and kernel level. A communication link is established between the *deputy* and *remote* which is used for the asynchronous transmission of events, such as signals and wake-up events.

MOSIX supports socket migration indirectly by forwarding through the *deputy*. A process establishes a socket only at the UHN and nowhere else. All communication on that socket has to go through the UHN irrespective of the current location of the process.

[1] Every process has two contexts – a user context that consists of program code, stack, data, memory maps and process registers; and a kernel context which contains the internal kernel execution state for the process, the kernel stack, system call information, signals pending etc.

The overall system architecture introduces severe penalties for system calls and other kernel accesses, network latency being the primary problem. To ameliorate this, MOSIX uses kernel/user page caches to improve the common case kernel memory access routines like copy_from_user() and copy_to_user().

9.1.2 CRAK

CRAK [22] is a transparent checkpoint / restart package for LINUX based on Epckpt *epckpt* and Libckpt [29]. CRAK provides a mechanism to migrate Linux applications without modifying or recompiling them. CRAK is implemented as a kernel module. CRAK claims to support network socket migration, but no implementation source code is available.

CRAK provides a loadable kernel module that is used to checkpoint and restart a process. User programs are provided that checkpoint and restart processes, using the kernel module. Checkpointing captures the process's code segments, data, registers, user and kernel stacks and kernel context and serializes it to a file. Open files are closed. Resources held by the process are freed and returned to the kernel. The running process is then killed.

The restart user utility restarts the checkpointed process. The current process (the restart user utility process) is converted to the chekpointed process by remapping its memory maps, re-opening files and restoring the registers. If the process was blocked in a system call during checkpointing, the system call is restarted.

A solution to network socket migration is proposed in the recently revised version of the original CRAK paper. However, no implementation is available. The approach to socket migration discussed in the paper is again at the module level, with no kernel modifications. CRAK makes use of user level socket calls to recreate the socket, and then uses a kernel module to modify the TCP state to reflect the previously open socket. The new host IP / port changes are propagated to the other end of the socket using rsh / ssh or alike.

CRAK does have limitations. It assumes that the migrating hosts are homogeneous – run the same Linux kernel version, same architecture and even have the same library versions. It also assumes the presence of a global file system such as NFS or AFS, and that the open files are stored in such a global file system. The socket migration mechanism proposed also has extra overheads like the use of rsh / ssh.

9.1.3 CONDOR

Condor [21] is a high throughput distributed computing environment designed to harness the unused CPU cycles of a large collection of workstations. CONDOR strives to effectively utilize the wasted computing cycles of workstations to achieve high throughput over prolonged periods of time. It's an on-going project at the

University of Wisconsin at Madison, and was started in 1988.

CONDOR is a user space migration system. It relies on specialized libraries for achieving migration. In this process, transparency is lost, and applications need to be recompiled and linked for use. CONDOR has been ported to various architectures and Unix variants. Efforts are also under way to port CONDOR to Windows NT.

CONDOR relies on advertisements from workstations to identify free computing power. Advertisements include the computing power available and the type of host CPU. Jobs are matched against the advertisements and are assigned to various workstations.

Processes are linked against the CONDOR libraries, as opposed to the normal C libraries. Processes thus linked can be migrated using a simple checkpoint and restart mechanism. The libraries also introduce remote system call capabilities. System calls are redirected to a shadow process on the source machine for execution.

The CONDOR libraries keep a record of all the system calls being called by a process, along with a list of resources being used. The library checkpoints a process and commits it to stable storage (NFS / AFS etc). This serialized process state is then restarted at another workstation to achieve process migration. CONDOR closes file descriptors before migrating and reopens them on the restarted side.

CONDOR has many limitations which makes it useful only for scientific computational purposes. It does not support processes that call fork() or exec(). It does not support any form of IPC; it does not support processes that communicate to others via **sockets**, pipes, open files, signals or other means.

9.1.4 SPRITE

The Sprite Network Operating System was developed at Berkeley between 1984 and 1994, with a goal to achieve time-shared system performance from a network of workstations, with the performance guarantees of the latter. Sprite is a new kernel in its own right – it is derived by heavily patching the BSD kernel.

At the heart of Sprite is a shared network file system, with a single image and a fully consistent cache. Sprite relies heavily on this file system to implement most of the System V functionality. For example, user level RPC, shared memory, virtual memory and even TCP connections are implemented using the file system. Thus the file system servers become the single main bottleneck in Sprite.

Similar to MOSIX, migration is based on the concept of a home machine and a foreign process. Every process has a shadow process associated with it on the home machine. All system calls are executed by this shadow process. Communication takes place either using the file system or through kernel-kernel RPC.

Sprite does support rudimentary socket migration in the sense that data is delivered to the process wherever it is in the cluster. This is done by splicing the link into two, and through the use of the file system communication architecture between the foreign process and its shadow process on the home system.

9.2 Socket Migration Solutions

9.2.1 Mockets – Transparent Redirection of Network Sockets

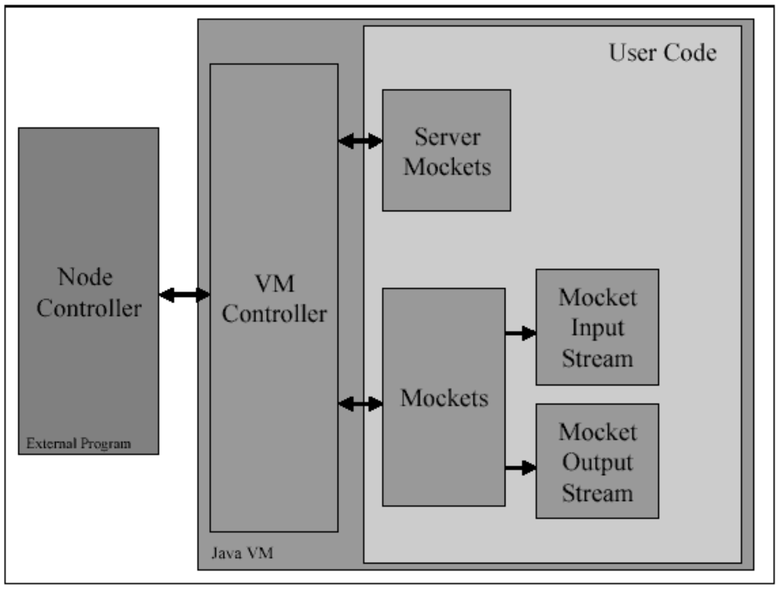

Figure 9.2: Mocket Components

The idea for mobile sockets, or Mockets[6], came from the need to support process migration in the NOMADS system from the University of West Florida.

A Mocket is a Java class that allows agents to open network socket connections to other systems and then to be able to move between hosts running the same modified Java Virtual Machine without having to interrupt, disconnect, and reconnect any of these sockets. A Java programmer wishing to employ this capability would use the Mockets class in place of the Java socket API.

The Mockets provide a Java stream interface to the programmer. These buffers enable the Mocket to be suspended prior to migration, and resumed after relocation in a manner transparent to the user. A Mocket can be used in programs just like a java socket. Migration is only possible if both ends of the connection

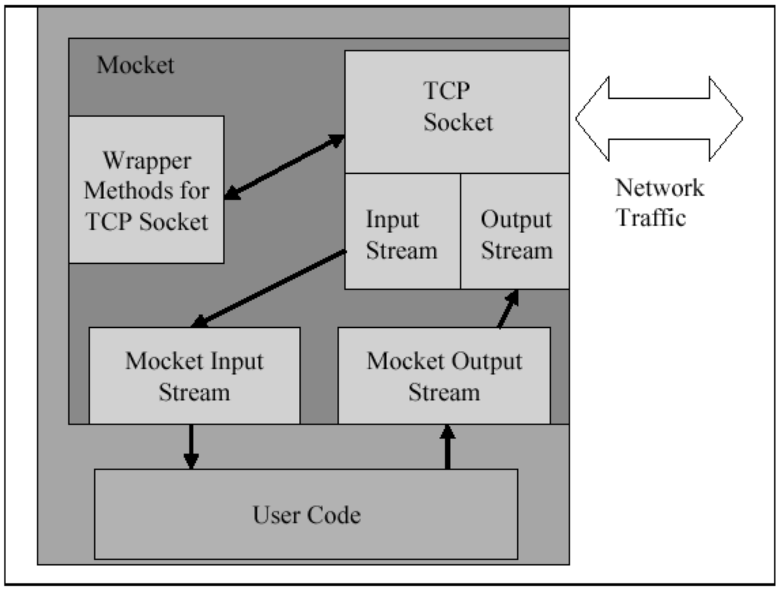

Figure 9.3: Mocket Structure

are using Mockets. The future work section suggests that a Mocket could be migrated while connected to a normal TCP socket by employing a proxy between the mobile agent (using Mockets) and a remote host (using a normal socket). The proxy would use normal TCP sockets to communicate with the remote host, and use MOCKETS to communicate with the mobile agent.

The Mockets design is fairly hefty (figure 9.2) – requiring each participating node (host) to run a NodeController program and each virtual machine (VM) to run a VMController. The NodeController acts as a repository for all MOCKETS on that particular host. Any node hosting a Mocket must run a NodeController because it acts as the single point of entry for all control and query messages to and from all Mockets - both from the network and from the application layer. It determines the destination VM and passes these messages along. The VMController is in charge of passing messages to and from the Mockets that are open in a particular Java program, as well as coordinating the suspend and resume operations.

The third component is the Mocket API, which is just a wrapper around a normal TCP socket (figure 9.3). A TCP socket is established between the two communicating Mocket layers whenever a Mocket is initialized or resumed, and is closed whenever a Mocket is closed or suspended. Thus Mockets are essentially an additional layer of abstraction between the user and the system. That is, they do not solve the migration problem by addressing it at the socket implementation level; instead they avoid it and create additional bulk with the Node and VM controllers.

9.2.2 Modular TCP Handoff

TCP Handoff, as discussed in [20] refers to the dynamic handoff of an active TCP connection from one host to another. Though it does not address process migration in particular, it is relevant to our work, as our migratable socket implementation can be used to provide a similar solution.

TCP Handoff has been presented as a technique to improve current load balancing architectures for web servers. The front-end receives an HTTP (or other) request from a client and opens a socket connection with that client. It then chooses a back-end server from a pool and assigns it the responsibility of servicing the request. In the process, it hands off the active TCP request to the back-end, thus creating a communication loop between the client, the front-end, and the back-end. The back-end effectively spoofs the IP of the front-end. This loop allows the client to remain under the impression that it is communicating exclusively with the front-end, while in fact its messages are being forwarded by the front-end to the back-end, which replies to its request (see Figure 9.4).

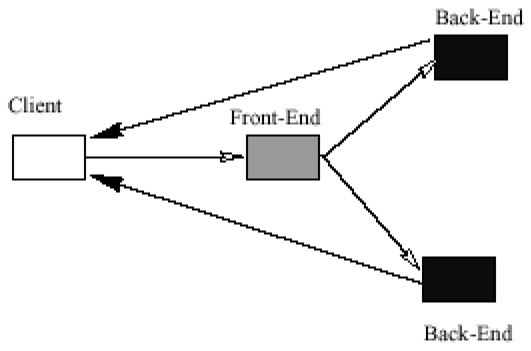

Figure 9.4: TCP Handoff

To achieve the loop communication the designers must make modifications to the implementation of TCP – called the TCP handoff protocol. These modifications come in the way of kernel modules that insert layers of abstraction between the IP, TCP, and socket layers (see figure 9.5). The lower layer is called BottomTCP (BTCP) and lies between the IP and TCP layer. The upper layer is called UpperTCP (UTCP) and lies between the TCP and socket layer. By introducing these two layers they have provided a wrapper around the TCP layer that completely encapsulates the loop. For example, the BottomTCP layer on the back-end can replace its local host IP address with the front-end IP address before passing it down to the IP layer. The UpperTCP layer will intercept messages passed up to the application that might confuse it (since the forwarding is transparent to it).

Because the front-end has to act as a go-between for the client and the back-

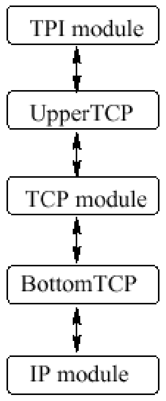

Figure 9.5: Plug-in Modules for TCP Handoff

end, it is not a true handoff of the TCP connection in that sense.

9.2.3 Mobile TCP Socket

The Mobile TCP Socket Implementation discussed in this section is presented in a technical report from The Australian National University [8]. The report talks about the implementation of a mobile socket layer below the normal TCP socket as part of a solution for the Mobile IP problem. In mobile IP, laptops move from one cell to another, changing IP addresses and other parameters in the process. Normally, any active network connections will be lost in the process. There have been many solutions proposed to implement persistency of network connections. The paper suggests that a mobile socket will help in faster re-establishment of network connections, with little or no loss of connectivity.

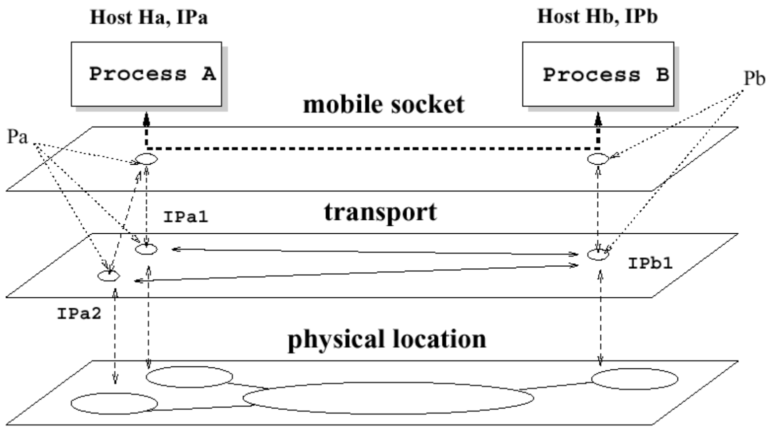

Figure 9.6: The Mobile Socket Layer

The report suggest the implementation of a mobile socket layer (MSL) (see figure 9.6). The MSL implements an abstraction called Virtual Socket. The virtual socket forms the bridge between the actual socket call and the lower IP level.

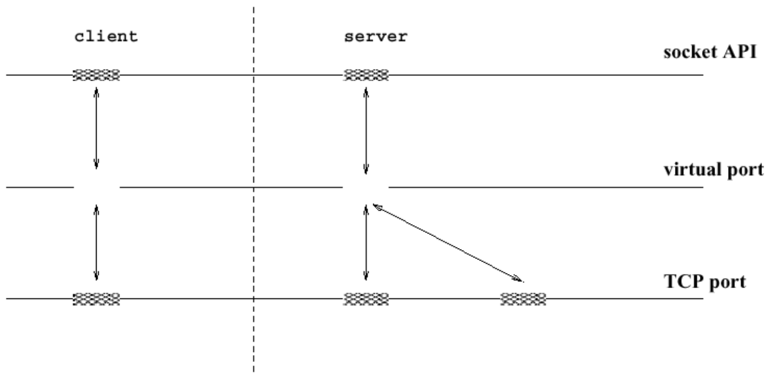

Figure 9.7: The Virtual Socket

Each machine on the network can have two IP's during migration – a home IP that never changes and a current IP that varies depending on the cell in which it is located. Socket persistence is a problem when they are established with the tuple (current IP, currentPort, remoteIP, remotePort). The virtual socket (see figure 9.7) however is based on a virtual port ID. Thus, a socket using this abstraction is distinguished by (virtualID, homeIP, homePort, remoteIP, remotePort). The virtualID is assigned by the server socket when a client socket connects to it. When a client machine moves, this mapping is changed to (virtualID, currentIP, homePort, remoteIP, remotePort). This change is achieved thorough control messages passed between the two virtual socket layers. Thus, this tuple remains unique on all machines. Persistence of state is achieved through buffering in the virtual socket. It also allows for the sharing of ports. However, a lot of control messages are introduced in the process.

The virtual socket – TCP socket – application socket associations are as follows:

- Each physical TCP port is associated with a virtual socket (one to one mapping).
- Each socket (software abstraction used by programs) is associated with a virtual socket (many to one mapping).

Thus the socket forwards incoming packets to the appropriate stream based on the parameters mentioned.

The virtual socket abstraction provides a good solution for the mobile IP problem. However, it assumes that the machine remains the same – which is not really useful for a process migration situation. The solution does preserve TCP end-to-end semantics. However, in reality, the lower level connection is re-established each time the machines shifts domains (rather than preserving the connection).

The above can be thought of as a lower level implementation of the Mockets idea – the abstraction is brought below the socket layer rather than above it. Though the solution proposed is not directly applicable to the socket migration problem, the kernel level implementation offers cues to a suitable socket solution.

Another fact to consider is that the proposal's current form is just a report. The system has not been implemented yet, and thus there are no papers discussing its performance.

9.2.4 Migratory TCP (M-TCP)

Migratory TCP [31] [32] [33] is a migration oriented transport protocol developed by the Distributed Computing Lab at Rutgers University. M-TCP is targeted specifically at the clustered server environment.

The primary intention of M-TCP is to decouple a service from the fixed identity of its service provider. In M-TCP, a client host can initiate migration of the remote endpoint of a live connection. Migration can take place transparent to the client application, and may occur under various conditions like server overload, network congestion, degradation in performance perceived by client etc. The origin and destination server hosts cooperate by transferring supporting state in order to accommodate the migrating connection. The client initiates migration with a SYN to a destination server, which then fetches the supporting state and completes the handshake for the migrated connection.

A client is supplied with the addresses of cooperating servers when it connects to its primary service provider. It can then migrate the connection to one of these alternate servers.

M-TCP assumes that each server can uniquely identify and isolate the state associated with each service session. Transfer of this state from one server to another allows the servers to serve the client request without any interruptions.

M-TCP has been implemented for the BSD kernel, but has not been ported to Linux yet. Additionally, a modified version of PostgreSQL database server is available for testing.

9.2.5 TCP Splicing and MSOCKS

Projects such as MSOCKS [35] support service connectivity by using proxies. A mobile client establishes a connection to a proxy, which then connects to the remote server. The proxy thus splices the two links – mobile client to proxy and proxy to remote server. The proxy maintains a record of all connections and

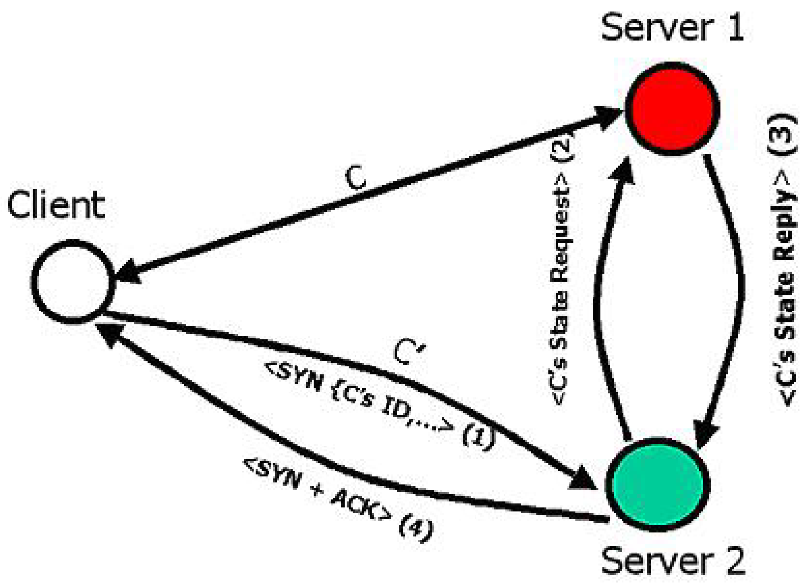

Figure 9.8: M-TCP Message Exchange

forwards incoming packets to the current location of the mobile client. When the location or network address of a mobile client changes, it updates the proxy, which then forwards all incoming packets to the new address. Though the project helps in maintaining session connectivity, it does not address the issue of state migration.

9.2.6 Unverified Solutions

There are two ongoing projects that are allegedly taking strides toward providing kernel level support for network socket migration. The MOSIX project that was discussed in section 9.1.2 contends on their website that they are currently in the process of building this capability into their system. However, an email response posted in April 2001 indicated that the current status of socket migration support is "Ongoing R&D." No other details can be found anywhere about this topic. The CRAK project that was also mentioned in section 9.1.2 claims to support socket migration as well but it is not currently included in any of the freely available versions. Furthermore, we were denied access to any details of CRAK's approach to socket migration directly by the lead developer of the system. Thus we can only conclude that kernel support for socket migration in each of these two systems remains an unsolved problem.

Chapter 10

Future Work

In section 6.4 of this paper we noted that security was not a priority for phase one of work on MIGSOCK. We did, however, discuss a way in which a Public Key Infrastructure could be incorporated into MIGSOCK using the Message Code field in the Message Header. Thus a future phase of the project could involve the incorporation of a security structure to prevent spoof attacks.

Figure 6.1 showed where socket migration fits into the TCP message timeline. In particular, the communicating TCP sockets must be in the *Established* state in order for our implementation to work. This automatically excludes the migration of listening sockets. We felt that for a first phase of work on socket migration it was more important to handle connected sockets. The usefulness of migrating listening sockets is debatable because if there is no connection, then there is no connection state and therefore nothing to migrate. However, it should be possible to migrate a process that has listening sockets. One approach could be to simply close such sockets in a process on the source host, and recreate them on the destination host after process migration.

We mentioned in chapter 7 that MIGSOCK was developed on Linux version 2.4.6, and is not supported on other versions or operating systems. Another phase of the project could include the porting of our code to other versions of Linux or other operating systems.

The current MIGSOCK implementation makes the assumption that socket migration will not be attempted on a connection unless both participating hosts support it. Sending a migration request to a remote host without MIGSOCK support could corrupt the communication protocol and terminate the socket connection. As a result, the user must be cognizant of which machines in a network support migration, and which processes have sockets that are connected to machines that don't support it, thus excluding these processes as candidates for migration. A better approach can be developed in a future phase of work that improves this compatibility with non-migration-supporting hosts.

In the current implementation, we rely on the presence of a shared file system or a messaging medium to transfer process and socket state from one machine to another. A state-transportation mechanism can be built that automatically transfers state from one host to another for automated migration without user intervention.

Chapter 11

Conclusion

In chapter 5 we described the goals that guided our design of socket migration in MIGSOCK. These were transparency, kernel level implementation, compliance with TCP semantics, state preservation, interoperability, performance, cross-platform portability, modular design, and ease of integration into existing process migration environments. Most of these objectives were met successfully with a few qualifications.

The implementation was indeed done at the kernel level, and it was a modular design in two ways. First, the MIGSOCK system calls were coded in a Linux kernel module that can be dynamically loaded and unloaded into the kernel at runtime. Second, the steps required to complete migration successfully were each split into a separate system call function, making the design modular in a more conventional sense of the word.

Cross-platform portability is hard to prove at this point since we did not port our design to any systems besides Linux 2.4.6. However, because of the modular nature of the coding, we believe MIGSOCK will be easily ported to other systems as long as they support TCP. MIGSOCK demonstrates ease of integration into existing process migration environments in that we coupled it to CRAK with very little modifications to the CRAK source code.

State preservation was achieved since our implementation did not require the re-instantiation of sockets after migration. Furthermore, the design ensures that no data is lost during the migration, thus preserving the state of the application layer protocols.

MIGSOCK does not violate the semantics of TCP. It essentially sits on top of TCP and employs its mechanisms during a migration. The changes to TCP primarily involved the sending, processing, and catching of the special migration messages so that they did not interfere with the behavior of TCP. In terms of TCP interoperability, a host that supports MIGSOCK can communicate perfectly with a host that does not. However, if the former attempts to initiate a migration of its socket, it may corrupt the application layer communication protocol on the remote host if that host does not support MIGSOCK.

Performance is difficult to characterize for socket migration because there are no benchmarks to compare it with. Process migration takes time, and socket migration invariably adds overhead to this. However, we showed in chapter 6 that the messaging semantics were kept to a minimum.

Transparency was achieved at several levels. First, no changes are required for any of the user level applications or processes that wish to migrate TCP sockets. Thus no recoding is required to make an application migratable. Second, since socket migration is supported at the system level, control is contained within the user program that invokes these calls. Thus, the running processes that are migration candidates can be effectively unaware that socket migration is being enacted upon them. Furthermore, the remote user and process will be totally unaware that his counterpart is relocating. He will only detect a temporary block in communication while the migration takes place.

Appendix A

User Program Source Code

A.1 Start_Handoff.c

```
/* start_handoff.c
 *
 * This user program is run on a process and a socket that is
 * connected to a remote host.  It will suspend the socket by
 * putting the remote socket to sleep.  That remote socket will
 * wake up when finish_handoff is called from the new host and
 * process.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#include "migsock_user.h"

int
main (int argc, char *argv[])
{
    int fd; /* file descriptor for /dev/migsock */
    int ret; /* return val */
    struct migsock_params *arg =
(struct migsock_params *)
malloc (sizeof (struct migsock_params));
    struct migsock_config_data *data =
(struct migsock_config_data *)
malloc (sizeof (struct migsock_config_data));
    int fout;
    if (argc < 3)
      {
  printf ("Usage : user <pid> <sockfd>\n");
  return -1;
      }

    arg->pid = atoi (argv[1]);
    arg->sockfd = atoi (argv[2]);

    strcpy (data->id, MIGSOCK_ID_CODE);
    data->opcode = MIGSOCK_SEND_REQ;
    data->ohost = data->oport = data->nhost = data->nport =
0;
    memcpy ((void *) arg, (void *) data,
    sizeof (struct migsock_config_data));

    fd = open ("/dev/migsock", O_RDONLY);
    printf ("FD = %d\n", fd);
    if (fd < 0)
      {
  printf ("Bad FD.\n");
  return -1;
      }
    /* Send migration request to remote process.  Puts him to sleep. */
    ret =
ioctl (fd, MIGSOCK_IOCTL_REQ, (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_REQ system call failed.\n");
  return -1;
      }
    /* Allow enough time for ack to come back before proceeding */
    sleep (2);

    /* Serialize data and store the result in arg->data */
    ret =
ioctl (fd, MIGSOCK_IOCTL_TOFILE,
       (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_TOFILE system call failed.\n");
  return -1;
      }
    fout = open (MIGSOCK_FILENAME, O_WRONLY | O_CREAT);
    if (fout == -1)
      {
  printf
      ("Failed to create %s.  Migration aborted\n",
       MIGSOCK_FILENAME);
  return -1;
      }
    /* Write this serialized junk in arg->data to MIGSOCK_FILENAME */
    ret =
write (fout, (void *) (arg->data),
       MIGSOCK_SERIAL_DATA_SIZE);
    if (ret == -1)
      {
  printf
      ("Failed to write to %s.  Migration aborted\n",
       MIGSOCK_FILENAME);
  return -1;
      }

    printf ("Migration initiated.\n");
    printf ("%d bytes of serialized data written to %s\n",
    ret, MIGSOCK_FILENAME);
    close (fout);
    close (fd);
    return ret;
}
```

A.2 Finish_Handoff.c

```
/* finish_handoff.c
 *
 * This user program is run on a process and a socket that can be
 * migrated (stolen) from communicating with some arbitrary host
 * and reconnected to the remote host that was talking to the
 * start_handoff side.  It will send a message to this remote host
 * saying two things: 1) wake up.  2) point your socket to my
 * address and port.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
```

```
#include <sys/ioctl.h>
#include <fcntl.h>

#include "migsock_user.h"

int
main (int argc, char *argv[])
{
    int fd;
    int ret;
    struct migsock_params *arg =
(struct migsock_params *)
malloc (sizeof (struct migsock_params));
    struct migsock_config_data *data =
(struct migsock_config_data *)
malloc (sizeof (struct migsock_config_data));
    char fdata[MIGSOCK_SERIAL_DATA_SIZE];
    int fin;

    if (argc < 3)
      {
  printf ("Usage : user2 <pid> <sockfd>\n");
  return -1;
      }

    arg->pid = atoi (argv[1]);
    arg->sockfd = atoi (argv[2]);
    fd = open ("/dev/migsock", O_RDONLY);
    if (fd < 0)
      {
  printf ("Bad FD.\n");
  return -1;
      }

    fin = open (MIGSOCK_FILENAME, O_RDONLY);
    if (fin == -1)
      {
  printf
      ("Failed to open %s.  Migration aborted.\n",
       MIGSOCK_FILENAME);
  return -1;
      }
    /* read in file contents */
    ret = read (fin, fdata, MIGSOCK_SERIAL_DATA_SIZE);
    if (ret != MIGSOCK_SERIAL_DATA_SIZE)
      {
```

```
  printf
      ("Read failed. Data file size maybe incorrect. Abort.\n");
  close (fin);
  return -1;
      }

    strcpy (data->id, MIGSOCK_ID_CODE);
    data->oport = *((__u16 *) fdata);
    data->ohost = *((__u32 *) (fdata + 4));
    /* Get your IP address */
    ret =
ioctl (fd, MIGSOCK_IOCTL_GETHOST,
       (unsigned long) arg);
    if (ret == 0)
      {
  printf
      ("MIGSOCK_IOCTL_GETHOST system call failed.\n");
  return -1;
      }
    data->nhost = (__u32) ret;

    memcpy ((void *) arg, (void *) fdata,
    MIGSOCK_SERIAL_DATA_SIZE);
    /* Now update structures with serizlied data */
    ret =
ioctl (fd, MIGSOCK_IOCTL_FROMFILE,
       (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_FROMFILE system call failed.\n");
  return -1;
      }
    data->nport = (__u16) ret;

    printf ("Socket Migrated to Local Port %d\n",
    ntohs (data->nport));
    printf ("from host %d . %d . %d . %d ",
    *((unsigned char *) &(data->ohost)),
    *(((unsigned char *) (&(data->ohost)) + 1)),
    *(((unsigned char *) (&(data->ohost)) + 2)),
    *(((unsigned char *) (&(data->ohost)) + 3)));
    printf ("on port %d\n", ntohs (data->oport));
    /* Now send the restart message */
    data->opcode = MIGSOCK_SEND_RST;
    memcpy ((char *) arg, (char *) data,
    sizeof (struct migsock_config_data));
    ret =
ioctl (fd, MIGSOCK_IOCTL_RST, (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_RST system call failed.\n");
  return -1;
      }

    return ret;
}
```

A.3 Start_Migrate.c

```
/* start_migrate.c
 *
 * This user program is run on a process and a socket that is
 * connected to a remote host before crak is called on that host.
 * It will suspend the socket by putting the remote socket to
 * sleep.  That remote socket will wake up when finish_migrate.c
 * is called from the new host and process, after the process
 * has been migrated using crak.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <signal.h>

#include "migsock_user.h"

int
main (int argc, char *argv[])
{
    int fd; /* file descriptor for /dev/migsock */
    int ret; /* return val */
    struct migsock_params *arg =
(struct migsock_params *)
malloc (sizeof (struct migsock_params));
    struct migsock_config_data *data =
(struct migsock_config_data *)
```

```
malloc (sizeof (struct migsock_config_data));
    int fout;

    if (argc < 3)
      {
  printf ("Usage : user <pid> <sockfd>\n");
  return -1;
      }

    arg->pid = atoi (argv[1]);
    arg->sockfd = atoi (argv[2]);

    if (kill (arg->pid, SIGSTOP) == -1)
      {
  printf
      ("Couldn't Suspend Process %d.  Aborting migration.\n",
       arg->pid);
  return -1;
      }

    strcpy (data->id, MIGSOCK_ID_CODE);
    data->opcode = MIGSOCK_SEND_REQ;
    data->ohost = data->oport = data->nhost = data->nport =
0;
    memcpy ((void *) arg, (void *) data,
    sizeof (struct migsock_config_data));

    fd = open ("/dev/migsock", O_RDONLY);
    printf ("FD = %d\n", fd);
    if (fd < 0)
      {
  printf ("Bad FD.\n");
  return -1;
      }

    /* Send migration request to remote process.  Puts him to sleep. */
    ret =
ioctl (fd, MIGSOCK_IOCTL_REQ, (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_REQ system call failed.\n");
  return -1;
      }
    /* Allow enough time for ack to come back before proceeding */
    sleep (2);
    /* Serialize data and store the result in arg->data */
    ret =
ioctl (fd, MIGSOCK_IOCTL_TOFILE,
       (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_TOFILE system call failed.\n");
  return -1;
      }

    fout = open (MIGSOCK_FILENAME, O_WRONLY | O_CREAT);
    if (fout == -1)
      {
  printf
      ("Failed to create %s.  Migration aborted\n",
       MIGSOCK_FILENAME);
  return -1;
      }
    /* Write this serialized junk in arg->data to MIGSOCK_FILENAME */
    ret =
write (fout, (void *) (arg->data),
       MIGSOCK_SERIAL_DATA_SIZE);
    if (ret == -1)
      {
  printf
      ("Failed to write to %s.  Migration aborted\n",
       MIGSOCK_FILENAME);
  return -1;
      }

    printf ("Migration initiated.\n");
    printf ("%d bytes of serialized data written to %s\n",
    ret, MIGSOCK_FILENAME);
    close (fout);
    close (fd);
    return ret;
}
```

A.4 Finish_Migrate.c

```
/* finish_migrate.c
 *
 * This user program is run on a process and a socket that has been
 * migrated from some host and restarted on this host using crak.
 * It will reestabilsh connection to the remote host that was
 * talking to the start_migrate side. First it wakes the process
 * that crak restore just created.  It will then  send a message to
 * the remote host saying two things: 1) wake up.  2) point
 * your socket to my address and port.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <fcntl.h>

#include "migsock_user.h"

int
main (int argc, char *argv[])
{
    int fd;
    int ret;
    struct migsock_params *arg =
(struct migsock_params *)
malloc (sizeof (struct migsock_params));
    struct migsock_config_data *data =
(struct migsock_config_data *)
malloc (sizeof (struct migsock_config_data));
    char fdata[MIGSOCK_SERIAL_DATA_SIZE];
    int fin;

    if (argc < 3)
      {
  printf ("Usage : user2 <pid> <sockfd>\n");
  return -1;
      }

    arg->pid = atoi (argv[1]);
    arg->sockfd = atoi (argv[2]);
    fd = open ("/dev/migsock", O_RDONLY);
    if (fd < 0)
      {
  printf ("Bad FD.\n");
  return -1;
      }
    fin = open (MIGSOCK_FILENAME, O_RDONLY);
    if (fin == -1)
      {
  printf ("Failed to open %s.  Migration aborted\n",
  MIGSOCK_FILENAME);
  return -1;
      }
    /* read in file contents */
    ret = read (fin, fdata, MIGSOCK_SERIAL_DATA_SIZE);
    if (ret != MIGSOCK_SERIAL_DATA_SIZE)
      {
  printf
      ("Read failed.  Data file size maybe incorrect. Abort.\n");
  close (fin);
  return -1;
      }
    close (fin);

    strcpy (data->id, MIGSOCK_ID_CODE);
    data->oport = *((__u16 *) fdata);
    data->ohost = *((__u32 *) (fdata + 4));

    memcpy ((void *) arg, (void *) fdata,
    MIGSOCK_SERIAL_DATA_SIZE);
    /* Update structures with serialized sock, socket, and file data */
    /* Also wake up suspended crak process */
    ret =
ioctl (fd, MIGSOCK_IOCTL_RESTART,
       (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_RESTART system call failed.\n");
  return -1;
      }
    data->nport = (__u16) ret;

    printf ("Socket Migrated to Local Port %d\n",
    ntohs (data->nport));

    /* Get your IP address */
    ret =
ioctl (fd, MIGSOCK_IOCTL_GETHOST,
       (unsigned long) arg);
    if (ret == 0)
      {
```

```
  printf
      ("MIGSOCK_IOCTL_GETHOST system call failed.\n");
  return -1;
      }
    data->nhost = (__u32) ret;

    printf ("From host %d . %d . %d . %d  ",
    *((unsigned char *) &(data->ohost)),
    *(((unsigned char *) (&(data->ohost)) + 1)),
    *(((unsigned char *) (&(data->ohost)) + 2)),
    *(((unsigned char *) (&(data->ohost)) + 3)));
    printf ("and port %d\n", ntohs (data->oport));

    /* Now send the restart message */
    data->opcode = MIGSOCK_SEND_RST;
    memcpy ((char *) arg, (char *) data,
    sizeof (struct migsock_config_data));
    ret =
ioctl (fd, MIGSOCK_IOCTL_RST, (unsigned long) arg);
    if (ret < 0)
      {
  printf
      ("MIGSOCK_IOCTL_RST system call failed.\n");
  return -1;
      }

    close (fd);
    return ret;
}
```

Bibliography

[1] Joseph Kiniry, Daniel Zimmerman. *A hands-on look at Java mobile agents.* IEEE Internet Computing, VOlume1, No. 4, 1997

[2] Ophir Holder, Israel Ben-Shaul, Hovav Gazit. *System Support for Dynamic Layout of Distributed Applications.*In Proceedings of International Conference on Distributed Computing Systems (ICDES'99)

[3] Ophir Holder, Israel Ben-Shaul, Hovav Gazit. *Dynamic Layout of Distributed Applications in FarGo.* In Proceedings of 1999 international conference on Software Engineering

[4] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, Erich Nahum. *Locality-Aware Request Distribution in Cluster-based Network Servers.* Department of Electrical and Computer Engineering, Rice University.

[5] Lior Amar, Amnon Barak , Ariel Eizenberg, Amnon Shiloh.*The MOSIX Scalable Cluster File Systems for LINUX.* Papers on MOSIX available at www.mosix.org

[6] Timothy S. Mitrovich, Kenneth M. Ford, and Niranjan Suri. *Transparent Redirection of Network Sockets.* In proceedings of OOPSLA Workshop on Experiences with Autonomous Mobile Objects and Agent Based Systems.

[7] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, Songnian Zhou. *Process Migration.* In ACM Computing Surveys, Volume 32, No. 3, September 2000

[8] Xun Qu, Jeffrey Xu Yu and Richard P Brent. *A Mobile TCP Socket.* TR-CS-97-08, Joint Computer Science Technical Report Series, The Australian National University

[9] Mohit Aron, Darren Sanders, Peter Druschel and Willy Zwaenepoel. *Scalable Content-aware Request Distribution in Cluster-based Network Servers.* Proceedings of the 2000 Annual Usenix Technical Coference.

[10] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. *Efficient Support for P-HTTP in Cluster-Based Web Servers.* Proceedings of the USENIX Annual Technical Conference, 1999

[11] W. Richard Stevens. *TCP/IP Illustrated Volume 1.* Addison-Wesley Professional Computing Series.

[12] W. Richard Stevens. *TCP/IP Illustrated Volume 2.* Addison-Wesley Professional Computing Series.

[13] W. Richard Stevens. *TCP/IP Illustrated Volume 3.* Addison-Wesley Professional Computing Series, 1999

[14] W. Richard Stevens. *Unix Network programming Volume 1, Second Edition.* Prentice Hall International PTR, 1998

[15] Amnon Barak and Oren La'adan. *The MOSIX Multicomputer Operating System for High Performance Cluster Computing.* From the MOSIX site at www.mosix.org.

[16] John K Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N Nelson, Brent B Welch. *The Sprite Network Operating System.* Computer, Volume 21, No. 2, Feb 1988.

[17] School of Computer Science, Carnegie Mellon University. *The MACH Operating System.* At http://www.cs.cmu.edu /afs/cs.cmu.edu/project/mach /public/www/mach.html. 1985 - 1994

[18] Amnon Barak and Oren La'adan, Amnon Shiloh. *Scalable Cluster Computing with MOSIX for LINUX.* Available at www.mosix.org.

[19] Computer Science Department, Virje University. *The AMOEBA Distributed Operating System.* http://www.cs.vu.nl/ pub/amoeba/amoeba.html.

[20] Wenting Tang, Ludmila Cherkasova, Lance Russell, Matt W. Mutka. *Modular TCP Handoff Design in STREAMS-Based TCP/IP Implementation.* Hewlett-Packard Labs, Department of Computer Science and Engineering, Michigan State University.

[21] Michael Litzkow, Miron Livny, and Matt Mutka, *Condor - A Hunter of Idle Workstations*, Proceedings of the 8th International Conference of Distributed Computing Systems, pages 104-111, June, 1988.

[22] Hua Zhong and Jason Nieh, *CRAK: Linux Checkpoint / Restart As a Kernel Module*, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001

[23] James S. Plank, *An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance*, University of Tennessee Technical Report CS-97-372, July, 1997

[24] Alessandro Rubini, Jonathan Corbet , *Linux Device Drivers, 2nd Edition*, OReilly & Associates Inc., June 2001.

[25] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, *Operating System Concepts, Sixth Edition* , John Wiley & Sons, Inc., June 15, 2001.

[26] Robert S. Gray, *Agent Tcl: A transportable agent system*, In Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland, December, 1995.

[27] IBM Research, *IBM Aglets Website*, http://www.trl.ibm.com/aglets/

[28] Eduardo Pinheiro , *EPCKPT Checkpoint Project*, Homepage at http://www.cs.rutgers.edu/ edpin/epckpt/

[29] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, *Libckpt: Transparent Checkpointing under Unix*, Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995, pp. 213–223.

[30] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*, OReilly & Associates Inc., January 2001.

[31] Rutgers University, *M-TCP*, http://discolab.rutgers.edu/mtcp/

[32] Florin Sultan, Kiran Srinivasan, Deepa Iyer and Liviu Iftode, *Highly Available Internet Services Using Connection Migration*, Rutgers University Technical Report DCS-TR-462, December 2001

[33] Kiran Srinivasan, *MTCP: Transport Layer Support for Highly Available Network Services*, Master of Science Thesis. Rutgers University Department of Computer Science Technical Report DCS-TR-459, October 2001

[34] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, Timothy S. Mitrovich, *An Overview of the NOMADS Mobile Agent System*, University of West Florida

[35] D. A. Maltz, P. Bhagwat. *MSOCKS: an architecture for transport layer mobility* Proc. IEEE 17th InfoCom'98, 1037-1045, 1998