Carnegie Mellon University
Information Networking Institute

PROJECT

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science in Information Networking

Title: MIGSOCK II - Migratable TCP Socket in Linux

Presented by: Kevin C. Lee

Accepted by the Information Networking Institute

Thesis Advisor(s): Katia P. Sycara
Print Name

Signature **Date**

Thesis Reader: Joseph A. Giampapa
Print Name

Signature **Date**

INI Director: Pradeep Khosla
Print Name

Signature **Date**

Project Presentation Date: 4/15/04

TR #__________
Office use only

MIGSOCK II - Migratable TCP Socket in Linux

Kevin Lee

May 12, 2004

Acknowledgments

I would like to thank my advisors Joseph A. Giampapa and Katia P. Sycara for being critical reviewers of my project. I would also like to thank God and all the members in the church for praying for me. I would also like to thank all the entire MS14 class for such an enriching experience. Finally, I would like to give thanks to my family for supporting me in every step of my thesis work.

Contents

List of Figures

Abstract

Network process checkpointing and restarting are the acts of saving a network process's states and resuming them successfully at some later time. Network process migration describes the entire procedure of checkpointing a network process and restarting it on a different host computer. A successful migration involves not only the states of a process and its socket be de-serialized but also appropriate changes of its socket be reflected based on the identity of the host.

There are different approaches to implementing network process migration ranging from the transport-layer ones all the way up to the application-layer ones. MIGSOCK, developed by CMU INI students, is one of the approaches implemented at the OS level with both a modified kernel and a kernel module. Motivations of MIGSOCK are user transparency, independence of a migrating process upon its host, and performance. User transparency is the idea of hiding migration details from user programs and thus removing user programs from the burden of taking care of the migration themselves. Independence of a migrating process upon its host is an effort to achieve true network process migration as opposed to host mobility, which is an act of migrating a host with all its processes, in some approaches. Ultimately, a process and its host should not be constrained to migrate as one entity.

This project seeks to address the deficiencies of the current MIGSOCK by implementing support for checkpointing multiple sockets for single-processed, multi-processed, and multi-threaded programs. The extended functionalities of MIGSOCK broaden the scope of network applications to include multi-processes and multi-threaded ones and eliminate the constraint on the number of application's connections. The implementation requires minimal changes to the existing MIGSOCK data structures and module. At the same time, it still provides the same user-level transparency as the existing MIGSOCK.

To provide the extra functionalities of MIGSOCK, the project extends CRAK, a public-domain process checkpointing utility, to support checkpointing and restarting of a multi-processed and multi-threaded program. By combining modified CRAK and additional features, the project has demonstrated the wide usage of MIGSOCK in checkpointing and restarting network applications. The project concludes with an evaluation to compare the performance overhead of MIGSOCK and Zap, an extension of CRAK that implements socket migration at the user level, via netfilter. Results of the experiments show that the MIGSOCK kernel-level approach is more suitable than the Zap user-level approach in supporting network process migration in typical single-service and multi-service scenarios.

Chapter 1

Introduction

Network process migration describes the entire procedure of checkpointing a network process, relocating it to another host computer, and restarting it. In detail, a migrating process needs to pause its execution and notify its communicating peer about migration. It then needs to relocate to a destination host and resume its execution. Finally, it re-establishes communication with its network peer without resetting or breaking the previous connection.

User-level socket translation and kernel-level socket migration are two approaches to network process migration. Both approaches need to take care of buffering incoming packets and stalling incoming connections while a network process is migrating. Upon migration, they also need to ensure re-assignment of the new IP address, and re-establish the previous communication. One difference between kernel-level approaches and user-level ones is that kernel-level approaches do not require the use of a proxy to cache incoming packets and intercept communications. Furthermore, TCP semantics such as TCP transition states are left untouched. Last but not least, kernel-level approaches only interrupt communications traffic between a migrating process and its remote peer once, whereas user-level approaches translate the IP address and port twice for every packet once a process has migrated, hence kernel-level approaches should be faster than user-level approaches.

MIGSOCK is an integrated kernel-level approach to network process migration. It is the most complete kernel-level approach by far as most of the approaches to network process migration are user-level ones [8][11][13][14][15]. Many kernel-level approaches are specialized in their own systems; hence, they do not provide interoperability with the other systems [7]. Steps in carrying out a network process migration in sequence are:

1. To temporarily suspend the process to migrate and migrate any active resource (e.g. I/O, libraries on disk, etc.) that it might be using, and save the suspended process state as a static image.

2. To notify any remote network peers to temporarily suspend sending

messages to the process that is about to migrate.

3. To temporarily suspend its network sockets and save that state as a static image.
4. To transfer that static images to a destination host,
5. To restore the process's execution state from the process static image.
6. To restore the network socket's execution state from the static image.
7. To reassign any active resources that are different on the new system, to reassign the process's new IP address and/or port number
8. To notify the remote network peers of the migrated process' new address.
9. To restart the migrated process, and to resume communications with the remote peers.

The system requires modifications of the TCP in sending and receiving to recognize MIGSOCK special messages special messages that are necessary for the MIGSOCK protocol to suspend network peers and to resume communications with them after updating them with their new IP and port address. To continue the work left off by Kuntz and Rajan [5], the project proposes extensions to MIGSOCK in a few areas. The extensions broaden the scope of network applications to include multi-processes and multi-threaded ones. Furthermore, they eliminate the constraint on the number of application's connections. In short, the extended MIGSOCK should be able to checkpoint and restart multiple sockets of a single-processed, multi-processed, and multi-threaded program with the help of modified CRAK in checkpointing and restarting these programs.

To confirm the claim that a kernel-level approach outperforms a user-level approach, the project also evaluates the performance of MIGSOCK against Zap. The decision to select Zap for comparison is because it is by far the most complete user-level approach at the time of the research. Various requirements such as preserving resource naming consistency, avoiding potential resource naming conflicts, and avoiding creating dependencies among components of the system [9] make it a promising candidate for comparison. From the description of ZAP, which is not available for testing, it appears to use CRAK for process checkpointing — same as MIGSOCK — and Netfilter to effect user-level IP and port translation and forwarding. Hence, MIGSOCK and Zap can be compared by evaluating the cost of checkpointing and restarting MIGSOCK sockets versus the overhead of Netfilter in translating and redirecting packets.

The remainder of this paper is organized as follows: Chapter 2 introduces process checkpointing and provides different approaches to it. It discusses briefly the current status of MIGSOCK. Chapter 3 explains socket programming from a brief introduction of its API to low-level system calls. Some important kernel structures such as `sock` are introduced. The way to access a socket's inode is revealed. Chapter 4 characterizes processes and threads. It shows Linux system calls to create multi-processed and multi-threaded programs. Outputs from these two programs are shown to illustrate differences between processes and threads. Chapter 5 discusses implementation for checkpointing and restarting sockets for single-processed, multi-processed, and multi-threaded programs. The chapter describes test programs used to test MIGSOCK's extended functionalities. Chapter 6 explains experiments conducted to evaluate the overhead of MIGSOCK and Netfilter. Overall findings conclude this chapter. Chapters 7 and 8 cover future and related works to network process checkpointing and restarting. Chapter 9 concludes the paper.

Chapter 2

Process Checkpointing

This chapter defines process checkpointing. It explains approaches to process migration and lacks across these approaches. By introducing such, it is hoped that readers will realize the purpose of MIGSOCK.

2.1 Overview of Process Checkpointing

Process checkpointing is defined as an act of saving a process's states and restart it at some later time. A successful process checkpointing mechanism requires the same process to be restarted at the point where it left off upon checkpointing.

There are numerous benefits to process checkpointing. For one, it promotes transaction rollback of a system. By checkpointing important processes of a server periodically, the same processes can be restarted upon the server's failure. Transaction rollback is automatically guaranteed since the last checkpointed image before the server's failure signifies the last successful transaction. By restarting this image, transaction is automatically rolled back. Another important benefit of process checkpointing is server-load balancing where a heavily loaded server can checkpoint CPU-intensive tasks and offload them to some less loaded servers based on some sort of load-balancing mechanism. Other benefits include increase of data accessibility and ease of system adminstration.

The current difficulty as discussed in [5] is that current process checkpointing mechanisms are local to particular systems. These mechanisms have some sort of dependence, e.g., system resources. In addition, current process checkpointing ignores socket checkpointing. Because of the wide diffusion of network applications, programs are no longer stand-alone entities. Many programs rely on communications with servers to keep themselves updated. The client/server model makes it imperative to extend current process checkpointing to sockets checkpointing. Many existing network process checkpointing mechanisms provide host mobility by doing some sort of address translation across the network stack layers. However, these approaches introduce the need of a proxy. Furthermore, host mobility inherently means tight dependence of processes on hosts. The project believes that an end-to-end approach argued in [12] removes problems posed by current network process checkpointing approaches.

2.2 OS and User-Level Approaches

Approaches to process checkpointing can be broken into two categories: OS and user-level approaches. OS-level approaches are either done within the OS core or built as an OS module. MOSIX [2] is one where process checkpointing is built into the operating system. A migrating process keeps part of itself on the Unique Home Node (UHN) where the process is first created. The other component of itself is migrated over to some other host. The process continues executing on the destination host. If there are resource dependencies, the *deputy* or the first component of the process on UHN resolves them by remote procedure call. An operational assumption is that process checkpointing only works on MOSIX operating systems.

`chpox` and `epckpt` are kernel modules that can be loaded to the Linux kernel to accomplish process checkpointing. The main construction of these kernel modules is that they capture the `task_struct` of a process and map its virtual memory to a file. User-level approaches include `ckpt`, `eksy`, `Dynamite checkpointer`, and various others. Most of them accomplish process checkpointing by the use of libraries. Process checkpointing requires re-linking of these libraries. Moreover, some user-level approaches require special code insert for checkpointing to be successful. In most cases, user-level approaches are not so ideal because of re-linking and additional code insert. Since checkpointing mechanism is done further away from the kernel, it is deemed to suffer bigger performance overhead.

2.3 Network to Application Layer Approaches to Network PS checkpointing

Network process checkpointing defines the act of saving a network process's states and restarting it successfully at some later time. Successful checkpointing requires the establishment of the previous connection and its states upon restart. There are many approaches to network process checkpointing that range from network layer all the way up to the application layer (See Chapter 8). What is common among these approaches is that they all require some sort of a third party to take care of the current connection during checkpointing and migration [10]. A proxy is placed as a care-of for the checkpointed process. Data on the current connection is buffered at the proxy. Some sort of mechanism to stall the connection to the remote process at the proxy prevents the remote process from disconnecting from the migrating process due to the long delay of receiving a response. Upon restart, the data buffered at the proxy is then forwarded to the migrated process. Existing connection is taken care of to reflect changes of the migrated process's physical address by some means.

2.4 What is MIGSOCK?

Migratable Socket (MIGSOCK) [5] is a network process checkpointing utility that is built as a kernel module. Unlike most of the network process checkpointing approaches, it does not require the presence of a proxy. Furthermore, since it uses CRAK [16] to accomplish process checkpointing, it removes the dependency of a network process to its host. To take care of data transfer during migration, the remote process is put to sleep. This mechanism is accomplished by modifying the kernel to look for MIGSOCK special migration messages. Since it is done purely within the kernel, it is assumed that hosts where source and remote processes reside must run on MIGSOCK kernel.

Currently, MIGSOCK only supports checkpointing of a single TCP socket of a process. This project seeks to extend MIGSOCK by adding functionalities of checkpointing multiple sockets of a process, a multi-threaded program, a multi-processed program, and multiple sockets of a multi-thread and multi-process program. Modifications of code include CRAK user-level controlling programs, MIGSOCK module, and MIGSOCK user-level controlling programs. Implementation details are documented in Chapter 5.

2.5 MIGSOCK Design Goals

MIGSOCK design goals are mentioned in [5] in detail. This section reviews and elaborates on of them.

Transparency The scope of transparency is limited to the remote process, with which the migrating process communicates, being unaware of the socket migration. Program transparencies are only limited to the client/server model, where one process is sending and the other is receiving, in MIGSOCK.

State Preservation TCP states in the establishment phase of a TCP connection [6] are preserved as a socket is migrated from one host to another. The migrated socket need not conduct with the remote host the three-way handshake all over again. Modifications of the IP addresses and port numbers on either end of a socket do not count as the TCP state.

Interoperability Interoperability is defined as the ability of a process on the MIGSOCK kernel to communicate with another process on a non-MIGSOCK kernel. Modifications should not preclude or isolate MIGSOCK hosts from making connections with hosts on other standard TCP Linux kernels.

Interoperability only extends to communications with a host that does not support MIGSOCK. Disconnection from the remote process on a non-MIGSOCK system would occur if the process on the MIGSOCK system decided to start migrating its socket. To ensure interoperability, both sides should notify each other of MIGSOCK support.

Performance Overhead in processing data packets is non-existent as regular packets go through the normal TCP state transitions. The only overhead in MIGSOCK stems from its system calls. The performance degradation of MIGSOCK is anything but significant. Comparison of its performance to Zap is verified and documented in Chapter 6.

Cross-Platform Portability Modularity accounts for the cross-platform portability. However, it is not a sufficient condition. On the other hand, since MIGSOCK requires little modifications to the TCP and TCP implementation is fairly generic across different operating systems, MIGSOCK is qualified for meeting this requirement.

Chapter 3

Background of Socket Programming

In order to understand how to migrate a socket along with a process, it is necessary to understand the control of a socket that makes the end-to-end communication possible. Furthermore, handover of processing a packet from one layer to the next in the OSI layer model should be examined so that a peer on the modified MIGSOCK kernel would suspend itself during remote socket migration. Knowing the flow of TCP communication from the user space to the kernel space allows one to understand where code insert should take place. Knowing some important kernel data structures related to sockets should also reveal states of a socket for checkpointing.

This chapter begins with a summary of the socket programming API and proceeds with an example of a typical TCP socket communication program. The life of a TCP packet is discussed by going over some controlling functions in the network stack layers. Important kernel data structures related to sockets and some tutorial in finding socket inode close this chapter. Documented functions that are mentioned again throughout the chapter are noted without their parameters for ease of reading and the sake of space. Throughout the rest of the thesis, `SOURCEHOME` is defined as the root of the kernel source tree. Pathnames are defined relative to it.

3.1 Socket Programming API Overview

In this section, an overview of the socket programming API is explained. The API are system calls encompassing the creation of sockets, the establishment of a connection, the transfer of data, and the connection tear-down in both the client and server sides.

There are some header files that need to be included for the API to work. They are `<sys/types.h>`, `<sys/socket.h>`, and `<netinet/in.h>`.

- `int socket(int family, int type, int protocol)`

Inputs

family A family of protocols to be used. Some known families of protocols are Internet, Unix internal, Xerox NS, and IMP. AF_INET defines the Internet protocols family. Only this will be the concern to this project.

type This refers to the type or the behavior of the socket communication. For example, `SOCK_STREAM` provides sequenced, reliable, two-way, connection-oriented byte streams. `SOCK_DGRAM` provides datagrams, where connectionless, unreliable messages are sent to sockets. `SOCK_STREAM` will be the concern.

protocol Protocol defines the communication protocol between the two ends. Some common protocols are UDP, TCP, and ICMP. 0 is input for this parameter to denote TCP.

Description The function creates a new socket, from which communication can proceed between two end points.

Return Value It returns the descriptor of the socket. -1 is returned if an error occurs.

- `int bind(int sockfd, struct sockaddr *my_addr, int addrlen)`

Inputs

sockfd The file descriptor of the socket created using `socket()`.

**my_addr* It points to a structure of type `sockaddr` that contains the host address and port the socket opens on. This makes connection to this host possible.

addrlen It is simply the length of sockaddr. One would normally pass sizeof(struct sockaddr).

Description The function binds the socket to the host machine which another host can connect to.

Return Value 0 is returned on success. On error, -1 is returned and *errno* is set appropriately.

- `int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)`

Inputs

sockfd The file descriptor of the socket created using `socket()`.

**serv_addr* It points to a structure of type `sockaddr` that contains the destination address and its port. `sockaddr` structure is an abstract structure that is implemented by another structure of the same size, `sockaddr_in`, which is defined in `SOURCEHOME/include/linux/in.h` directory.

addrlen It is simply the length of `sockaddr`. One would normally pass `sizeof(struct sockaddr)`.

Description The function connects to the remote host with the created socket file descriptor.

Return Value 0 is returned if the connection succeeds. On error, −1 is returned and *errno* is set appropriately.

- `int listen(int sockfd, int backlog)`

  Inputs

  sockfd The file descriptor of the socket to listen to the connection on.

  backlog This specifies the number of connections on the incoming queue. Before a connection gets accepted, it will wait in this queue.

  Description This function causes a socket to listen to connections on this socket file descriptor.

  Return Value 0 is returned on success. On error, −1 is returned and *errno* is set appropriately.

- `int accept(int sockfd, void *addr, int *addrlen)`

  Inputs

  sockfd The file descriptor of the socket listening for connection.

  **addr* A pointer to `struct sockaddr` that is implemented by `struct sockaddr_in`.

  **addrlen* A pointer to an integer that is set to the `sizeof(struct sockaddr)` upon successful completion of `accept()` call.

  Description This function accepts the incoming connection from the wait queue where incoming connections are kept. A new socket file descriptor is returned. Furthermore, the `sockaddr` structure where `*addr` is pointing to will be populated with the incoming connection's address and port. Data communication from now on will use the new socket file descriptor.

  Return Value A non-negative integer that is the file descriptor for the socket is returned on success. On error, −1 is returned and *errno* is set appropriately.

- `ssize_t read(int fd, void *buf, size_t count)`

  Inputs

  fd The file descriptor from which data is read.

  **buf* A pointer to some type of data to be read to from the file descriptor.

  count Number of bytes in the data that `*buf` is pointing to.

  Description This function reads data from the fd to one pointed to by `*buf`.

Return Value On success, the number of bytes read is returned. On error, −1 is returned and *errno* is set appropriately.

Note A similar function `int recv(int sockfd, void *buf, int len, unsigned int flags)` does exactly what `read()` does except it is limited to socket file descriptor. `flags` is usually set to 0.

- `ssize_t write(int fd, const void *buf, size_t count)`

  Inputs

  fd The file descriptor from which data is written.

  **buf* A pointer to some type of data to be written to the file descriptor.

  count Number of bytes in the data that `*buf` is pointing to.

  Description This function writes data pointed to by `*buf` to the fd.

  Return Value On success, the number of bytes written is returned. On error, −1 is returned and *errno* is set appropriately.

  Note A similar function `int send(int sockfd, const void *msg, int len, int flags)` does exactly what `write()` does except it is limited to socket file descriptor. `flags` is usually set to 0.

- `int close(int fd)`

  Inputs

  fd The file descriptor to be closed.

  Description This function closes the connection on the socket.

  Return Value 0 is returned on success. −1 is returned when there is an error.

Figure 3.1 shows how the socket programming API is used. The API is a client/server model where the server takes the initiative and opens its socket and waits for the client to establish a connection. Upon completing the connection establishment, both server and client enter into the data transfer phase. When they finish communicating, both terminate by closing their own socket descriptors. The server then can continue and accept a new connection.

3.2 TCP Socket Communication

In this section, simple count server and client programs demonstrate the use of socket programming API. The programs are the basis of checkpointing and restarting. The client and server programs are also the basis of a set of test programs on which multiple socket single-processed, multi-processed, and multi-threaded migration are performed. The section starts by describing the programs in general terms and then going into details to describe

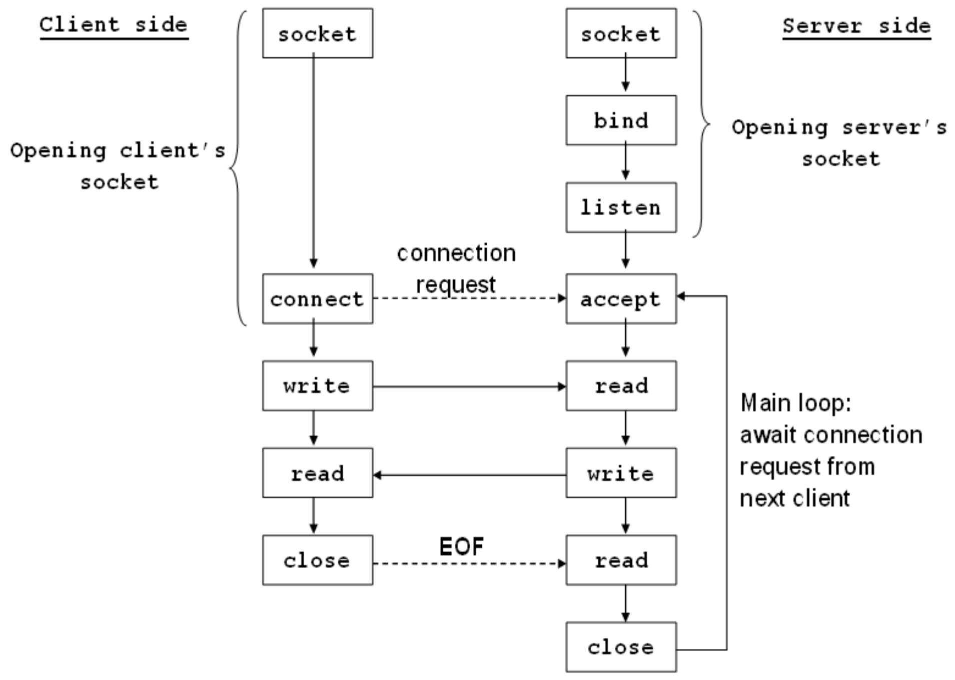

Figure 3.1: Socket Programming API Overview.

the use of helper functions.

The count server program listens to a port specified on the command prompt. When both sides finish connection establishment, the server will keep sending integers in ascending order starting from 1. The count client receives the number and outputs it on the command line. Following shows the code for both the count server and client.

Count Server:

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>

int main(int argc, char *argv[]) {
  int sd, csd;
  int val, retval, size;
  unsigned char inp[10];
  struct sockaddr_in srv, client;
```

```
if (argc != 2) {
  printf("Usage: %s <port number>\n", argv[0]);
  exit(-11);
}

// create a socket
if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  printf("Failed to create socket.\n");
  exit(-1);
}

// clear the sockaddr_in structure
memset(&srv, 0, sizeof(struct sockaddr_in));
srv.sin_family = AF_INET;
srv.sin_port = htons(atoi(argv[1]);
srv.sin_addr.s_addr = INADDR_ANY;

// bind the socket to the host machine
retval = bind(sd, (struct sockaddr*)&srv, sizeof(struct sockaddr));
if (retval < 0) {
  printf("Bind failed.\n");
  close(sd);
  exit(1);
}

// start listening for connection
listen(sd, 5);

// accept connection on this socket
csd = accept(sd, (struct sockaddr*)&client, &size);

// close sd because there will not be any connection
close(sd);

val = 0;

// now start sending val to the client
while(1) {
  printf("Sending %d\n", val);
  memset (inp, 0, 10);
  memcpy((void *)inp, (void *)&val, sizeof(int));
  if(write(csd, inp, sizeof(int)) < 0) {
    printf("Write failed.\n");
    exit(1);
  }
```

```
    val++;

    // sleep for a second to slow down the output on the client's
    // side
    sleep(1);
  }

  return 0;
}
```

`htons()` is a macro that converts host byte ordering to network byte ordering. It is imperative to use it since different operating systems represent numbers differently, whether in big endian or little endian. A conversion macro allows the client and server to have a consistent view of numbers despite different representations.

The count server program illustrates nicely the socket programming API. The only trick is to make sure that a port is specified so that the client knows which port to connect to on the server. Keep in mind that the port number should be fairly large (above 1028) to avoid clashing with services offered on lower ports.

Count Client:

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <netdb.h>

int main(int argc, char *argv[]) {
  int sd;
  int out, retval, det;
  unsigned char inp[10];
  char *server;
  struct sockaddr_in srv;
  struct hostent *hp;

  if (argc != 2) {
    printf("Usage: %s <hostname> <port>\n", argv[0]);
    exit(-1);
  }

  // create a socket
  if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    printf("Failed to create socket.\n");
    exit(-1);
  }

  hp = gethostbyname(argv[1]);
  serveraddr.sin_family = AF_INET;
  srv.sin_family = AF_INET;
  srv.sin_port = htons(atoi(argv[2]));
  bcopy((char *)hp->h_addr,
          (char *)&ser.sin_addr.s_addr, hp->h_length);

  // connect to server
  retval = connect(sd, (struct sockaddr*)&srv, sizeof(struct sockaddr));
  if (retval < 0) {
    printf("Connect failed.\n");
    return -1;
  }

  // now starting data transfer phase with the server
  while(1) {
    memset (inp, 0, 10);

    if(read(sd, inp, 10) > 0) {
      memcpy((void *)&out, (void *)inp, sizeof(int));
      printf("socket Receiving %d\n", out);
    }
  }

  return 0;
}
```

The count client program is straight-forward. Here a helper function `gethostbyname()` is introduced. It returns a pointer to `struct hostent`. `hp->h_addr` points to the first IP address listed in `hp->h_addr_list[0]`. `h_addr` cannot be found in `hostent` structure because it is defined as `#define h_addr-> h_addr_list[0]`. The function comes in handy when IP address of the machine to be connected is not known. To use the function, make sure to `#include <netdb.h>`. There is one other helper function of `gethostname(char *hostname, size_t size)` that is useful when one would like to find out the IP address of the host machine. Detailed usage can be looked up in the manual page in any Linux/Unix operating system.

3.3 Life of a TCP Packet

Understanding the life of a TCP packet completes the MIGSOCK paper by presenting why the modification occurs at the place they occur. The discussion covers the life of a TCP packet from the user-level `sendto()/recvfrom()` system call to kernel-level functions.

Before a network process can be migrated, it needs to inform the remote process about the migration. The remote process is then stopped by the kernel during migration. As the process that initiates the migration completes, it informs the remote process to wake up. The remote process will then be woken up and both parties resume their communication. Mechanism such as socket hijacking and socket state saving and resuming will be discussed in Chapter 5.

The main point to take away is that migration is carried out by special migration messages. The headers of these messages are encoded to contain information to be discerned by the remote process. Checking these messages happens at particular places of TCP implementation. With these messages, the kernel is then able to stop and start the remote process. Moreover, the kernel is able to discard these messages as opposed to sending them to the upper layer of the network stack.

The following shows what a TCP packet traverses as it goes down the network stack:

1. *sys_sendto(int fd, void *buff, size_t len, unsigned flags, struct sockaddr *addr, int addr_len)* in SOURCEHOME/net/socket.c:

   A packet with data pointed by `*buff` is fed to this function. `fd` is the socket file descriptor on which data is sent. `*addr` points to the address and port of the host where data is sent to.

2. *sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)* in SOURCEHOME/net/socket.c:

   This function is invoked within `sys_sendto()`. `*sock` is obtained by referring to the inode of the file belonging to the socket file descriptor. Detail of finding the file will be discussed in Section 3.5. `*msg` points to `struct msghdr` where `msg->msg_iov` contains the data pointed to by `*buff` in `sys_sendto()`.

3. *inet_sendmsg(struct socket *sock, struct msghdr *msg, int size, struct scm_cookie *scm)* in SOURCEHOME/net/ipv4/af_inet.c:

   This function is invoked by calling `sock->ops->sendmsg(sock, msg, size, &scm)` within `sock_sendmsg()`. `sock->ops` is a pointer to `struct proto_ops` , which is an abstract structure of a set of function pointers. Since `*sock` is of the Internet protocols, function pointer points to the implementation of `inet_sendmsg()`.

4. *tcp_sendmsg(struct sock *sk, struct msghdr *msg, int size)* in SOURCEHOME/net/ipv4/tcp.c:

This function is invoked by calling `sk->prot->sendmsg(sk, msg, size)` within `inet_sendmsg()`. `sk->prot` is a pointer to `struct proto`, which is an abstract structure of a set of function pointers. Since `*sk` is of the TCP protocol, function pointer points to the implementation of `tcp_sendmsg()`.

MIGSOCK code insert in `tcp_sendmsg()` checks for the special MIGSOCK message. If a special message is received, the process on this socket will be put to sleep. The process gets waken up by a signal. The signal is triggered by the MIGSOCK restart message in `tcp_v4_rcv()`. Since the kernel keeps track of open files and sockets, the special message will be received despite the process responsible for the socket is sleeping. It is up to the kernel to either pass the message up to the process or discard it altogether. In this case, the kernel will send a signal to wake up the process and discard the special message.

5. *tcp_send_skb(struct sock *sk, struct sk_buff *skb, int force_queue, unsigned cur_mss)* in SOURCEHOME/net/ipv4/tcp_output.c:

   Before a packet gets from the TCP layer to the IP layer, `tcp_send_skb()` is called. `*skb` is a pointer to `struct sk_buff`, that holds individual communication packets. Headers from various layers are encapsulated within this structure. Data is allocated in memory using `sk_buff` related API.

6. *tcp_transmit_skb(. . .)* in SOURCEHOME/net/ipv4/tcp_output.c → *ip_queue_xmit(. . .)* → *ip_route_output(. . .)* in SOURCEHOME/net/ipv4/ip_output.c → *NF_HOOK()*:

   The packet is routed from the TCP layer to the IP layer. Netfiltering, Linux network filter support, is then done at the IP layer to manipulate the packet and forward it either to the host machine or out to the wire.

   MIGSOCK code insert in `tcp_transmit_skb()` will check `migsock_ptr` flag in `struct sock`. If it is set, a special message will get generated to inform the remote host that migration will be underway.

Following shows what a TCP packet traverses as it goes up the network stack:

1. *ip_rcv(. . .)* in SOURCEHOME/net/ipv4/ip_input.c → *NF_HOOK(. . .)* → *ip_rcv_finish(. . .)* in SOURCEHOME/net/ipv4/ip_input.c → *ip_route_input(. . .)* in SOURCEHOME/net/ipv4/route.c → *ip_local_deliver(. . .)* in SOURCEHOME/net/ipv4/ip_input.c:

   Upon receiving the packet, it gets forwarded to `NS_HOOK()`. `NS_HOOK()` takes care of filtering the packet and determining whether the packet should be dropped, forwarded, or passed up. Assuming the packet is to be passed up to the higher layer, `ip_local_deliver()` will eventually be invoked.

2. *tcp_v4_rcv(struct sk_buff *skb)* in SOURCEHOME/net/ipv4/tcp_ipv4.c → *tcp_rcv_established(...)* → *tcp_event_data_recv(...)* in SOURCE-HOME/net/ipv4/tcp_input.c.

   As the packet arrives in the TCP layer, it enters into the `tcp_event_data_rcv()` function. `sk->data_ready()` will then be called. This function queues the data to the socket.

   `tcp_v4_rcv()` has MIGSOCK code that checks whether the special message is a restart message. If it is, the new host's IP address and port are applied to the socket at the remote site. Signaling the process on this socket to wake up for restart will take place here.

   Within `tcp_rcv_established()`, MIGSOCK code insert will check if the incoming packet is a special MIGSOCK message. If there is, it will be discarded.

3. *tcp_recvmsg(struct sock *sk, struct msghdr *msg, int len, int nonblock, int flags, int *addr_len)* in SOURCEHOME/net/ipv4/tcp.c:

   This function is invoked by calling `sk->prot->recvmsg(sk, msg, size)` within `inet_recvmsg()`. `sk->prot` is a pointer to `struct proto`, which is an abstract structure of a set of function pointers. Since `*sk` is of the TCP protocol, the function pointer points to the implementation of `tcp_recvmsg()`.

4. *inet_recvmsg(struct socket *sock, struct msghdr *msg, int size, struct scm_cookie *scm)* in SOURCEHOME/net/ipv4/af_inet.c:

   This function is invoked by calling `sock->ops->recvmsg(sock, msg, size, &scm)` within `sock_recvmsg()`. `sock->ops` is a pointer to `struct proto_ops`, which is an abstract structure of a set of function pointers. Since *sock is of the Internet protocols, function pointer points to the implementation of `inet_recvmsg()`.

5. *sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)* in SOURCEHOME/net/socket.c:

   This function is invoked within `sys_recvfrom()`. `*sock` is obtained by referring to the inode of the file belonging to the socket file descriptor. Detail of finding the file will be discussed in Section 3.5. `*msg` points to `struct msghdr` where `msg->msg_iov` contains the data pointed to by *buff in `sys_sendto()`.

6. *sys_recvfrom(int fd, void * ubuf, size_t size, unsigned flags, struct sock-addr *addr, int *addr_len)* in SOURCEHOME/net/socket.c:

The packet with data pointed by *ubuf arrives at this function. fd is the socket file descriptor on which data is waiting to be received. *addr points to the address and port of the host where data is obtained from.

3.4 Important Kernel Data Structures

There are some important kernel data structures that need to be explained to tie together the discussion in Section 3.3. The overview of these data structures not only complements Section 3.3, but it also helps one to know how socket API is referenced in the kernel space. The complete listing of these data structures are in Appendix A.

```
struct socket {
  socket_state            state;
  unsigned long           flags;
  struct proto_ops        *ops;
  struct inode            *inode;

  /* Asynchronous wake up list    */
  struct fasync_struct    *fasync_list;

  /* File back pointer for gc     */
  struct file             *file;
  struct sock             *sk;
  wait_queue_head_t       wait;

  short                   type;
  unsigned char           passcred;
};
```

struct socket, defined in SOURCEHOME/include/linux/net.h, forms the basis for the BSD socket interface. The state field indicates whether the socket is connected or unconnected. *ops field points to the operations either in inet_stream_ops or inet_dgram_ops. *inode points to the inode that this socket belongs to. A type field identifies the type of socket to either stream or datagram. *sk points to the substructure of the socket. It contains INET socket-specific information. struct sock is described as follows. Since only a few fields of struct sock concern us, partial definition of struct sock will be listed here. The full details of struct sock can be found in SOURCEHOME/include/net/sock.h.

```
struct sock {
  /* Socket demultiplex comparisons on incoming packets. */
  __u32                   daddr;          /* Foreign IPv4 addr      */
  __u32                   rcv_saddr;      /* Bound local IPv4 addr  */
  __u16                   dport;          /* Destination port       */
  unsigned short          num;            /* Local port             */

...
  wait_queue_head_t       *sleep;         /* Sock wait queue        */
...
  struct sk_buff_head     receive_queue;  /* Incoming packets       */
  struct sk_buff_head     write_queue;    /* Packet sending queue   */
...
  struct proto            *prot;
...
  /* Identd and reporting IO signals */
  struct socket           *socket;
...
  /* Callbacks */
  void                    (*state_change)(struct sock *sk);
  void                    (*data_ready)(struct sock *sk,int bytes);
  void                    (*write_space)(struct sock *sk);
  void                    (*error_report)(struct sock *sk);
  int                     (*backlog_rcv) (struct sock *sk,
                                          struct sk_buff *skb);
  void                    (*destruct)(struct sock *sk);
};
```

The first four fields contain information about the IP address and port of the host and the remote system. `*prot` field points to the operations either in TCP or UDP protocol (See Appendix A). An Internet protocol is specified during the creation of a socket using `socket()`. To send a TCP message, for example, the TCP operation can be accessed by doing `sk->prot->tcp_sendmsg(...)`. `*sleep` points to the head of a wait queue where processes blocked on this socket are kept (list wait queue structure). In MIGSOCK, the remote process is put into this wait queue. The state of this process is changed from `TASK_RUNNING` to `TASK_INTERRUPTIBLE`. Kernel scheduler is then invoked to remove the process from its running queue. When the kernel on behalf of the remote process receives another special message from the migrating process, the software interrupt puts the process back into the running queue. The process is removed from the wait queue and the communication resumes.

`receive_queue` and `write_queue` are where data to be received and sent are referenced. Data are stored in allocated memory regions, which need not be contiguous. Both `receive_queue` and `write_queue` point to the head of their respective queues, which are implemented as doubly-linked lists. `*socket` points back to the associated socket. The call back functions are triggered when appropriate actions take place. For example, `data_ready()` is triggered when data has been received. These callback functions are implemented in `SOURCEHOME/net/core/sock.c`.

When migration request is issued, the process is paused. The `receive_queue` might continue to grow during the interval of time in which the remote process has not received a MIGSOCK sleep signal. In this case, those packets in the `receive_queue` are lost. Packets can be recovered as some buffer can be created in `struct sock` and be serialized along with the rest of `struct sock` variables. The discussion of the loss of packets is further elaborated on in Chapter 7.

```
struct sk_buff {
  struct sk_buff  * next;         /* Next buffer in list */
  struct sk_buff  * prev;         /* Previous buffer in list */

  struct sk_buff_head * list;     /* List we are on */
  struct sock     *sk;            /* Socket we are owned by */
  ...
  /* Transport layer header */
  union
  {
    struct tcphdr   *th;
    struct udphdr   *uh;
    struct icmphdr  *icmph;
    struct igmphdr  *igmph;
    struct iphdr    *ipiph;
    struct spxhdr   *spxh;
    unsigned char   *raw;
  } h;

  /* Network layer header */
  union
  {
    struct iphdr    *iph;
    struct ipv6hdr  *ipv6h;
    struct arphdr   *arph;
    struct ipxhdr   *ipxh;
    unsigned char   *raw;
  } nh;

  /* Link layer header */
  union
  {
    struct ethhdr   *ethernet;
    unsigned char   *raw;
  } mac;
  ...
};
```

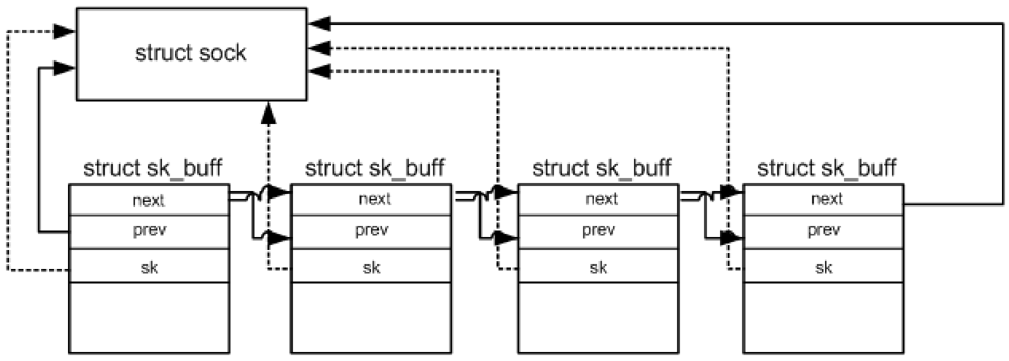

Figure 3.2: Relationship between `struct sock` and `struct sk_buff`.

The `sk_buff` structure, defined in `SOURCEHOME/include/linux/skbuff.h`, holds the communication packets belonging to a socket. Within the `struct sk_buff`, a pointer to the actual data in memory can be found to access the data. Since buffers holding packets of this type are arranged in a doubly-linked list, managed by `struct sk_buff_head`, `*next` and `*prev` fields are used to refer to the next `sk_buff` and previous `sk_buff` structure. `*list` points to the entry point of `sk_buff`. `*sk` points to the `struct sock` that it belongs to. Figure 3.2 [3] demonstrates the relationship between `struct sock` and `struct sk_buff`. The `h`, `nh`, and `mac` unions describe header information as it gets populated going down the network stack. `sk_buff` structure variable will be freed as the data gets sent out the wire.

3.5 Finding a Socket File Descriptor and Its Inode

The most important thing to checkpointing a socket is to be able to find the socket file descriptor associated with the socket in the user-level environment. Upon obtaining the file descriptor, the checkpointing module also needs to be able to obtain the socket inode in order to save it to a file. inode is the real description of the socket. In a regular case, inode references to data on disk. In this section, both finding a socket file descriptor and its inode will be described.

To find the socket file descriptor, one can simply look at the file descriptor table associated to a process. Socket file descriptors will be listed with "socket" noted. To look at the process file descriptor table, one simply issues `ll /proc/[PID]/fd`, where [PID] is the process ID obtained by issuing `ps -a`.

Following is the process descriptor table for process ID 27473.

```
[root@orpheus migsock_module]# ll /proc/27473/fd
```

```
total 0
lrwx------    1 root     root            64 Mar 31 14:05 0 -> /dev/pts/4
lrwx------    1 root     root            64 Mar 31 14:05 1 -> /dev/pts/4
lrwx------    1 root     root            64 Mar 31 14:05 2 -> /dev/pts/4
lrwx------    1 root     root            64 Mar 31 14:05 3 -> socket:[56123]
lrwx------    1 root     root            64 Mar 31 14:05 4 -> socket:[56124]
```

From the output, one can see that process 27473 owns socket file descriptors 3 and 4.

To understand how a socket's inode, struct sock, is obtained, it is important to know the relationship among a process, its file tables, file's structures, and inodes. Figure 3.3 demonstrates their relationship.

A process is encapsulated in a `task_struct` structure. `task_struct` keeps information about the process ID, its parent's ID, its current state, and various other information. Within `task_struct`, `*files` points to a structure called `files_struct`, which is the process's file descriptor table. The table has a double pointer `**fd` that points to a set of open files. A file is represented with the file structure. It contains a list of operations that can be done on it. For example, they are open, write, read, seek, etc. Beside that, the file structure contains `*f_dentry` that points to an entry of dentry in the directory entry. If it belonged to a socket, its member variable `i_sock` would not be NULL. inode's `u.socket_i` can then be accessed. In short, one obtains the socket inode of type `struct socket` by the following:

```
struct files_struct *files;
struct file * file = NULL;
struct inode *inode;
struct socket *sock;

files = proc->files;
file = files->fd[3];  // the file descriptor for the socket obtain from
                      // the process table

inode = file->f_dentry->d_inode;
if (!inode->i_sock || !(sock = &(inode->u.socket_i)))
  {
      retval = -ENOTSOCK;
      printk("MIGSOCK bug: Checking for socket failed.\n");
      goto done;
    }
```

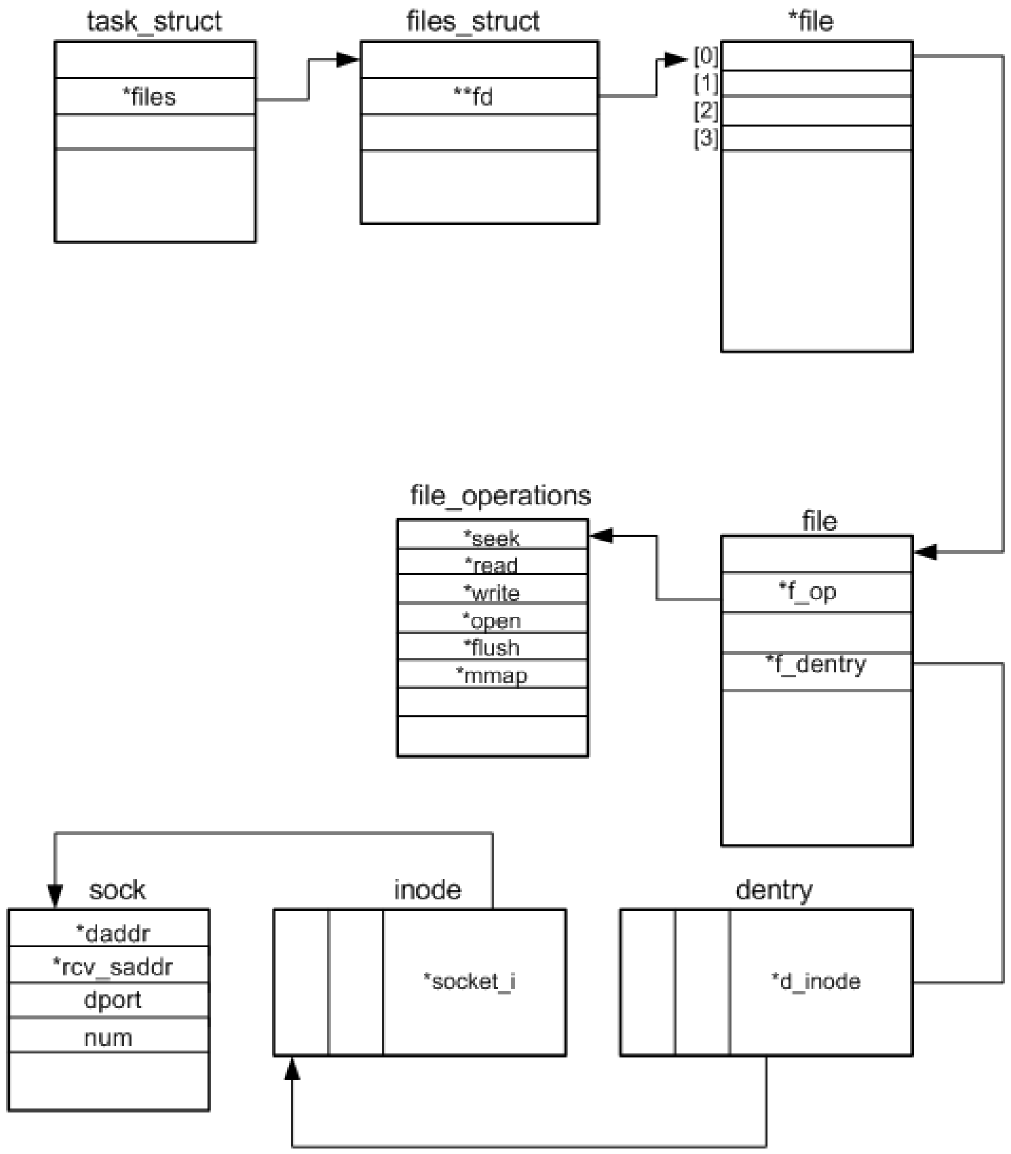

Figure 3.3: Relationship between Process, Socket File Descriptors, and Socket Inode.

Chapter 4

Background of Processes and Threads

Processes and threads are important as they are the building blocks of any operating systems. Processes are programs in action. They carry out operations and tasks of an operating system. In the Linux OS, as the boot procedure completes, the system loads and executes the kernel image. The kernel then creates the `init` process, which then spawns the other initialization programs defined in `/etc/init`, `/bin/init`, or `/sbin/init`. To improve the efficiency of performing a task, a process usually divides up its task and assigns the parts to threads. The fact that threads are created with the same memory space greatly save system programmers from taking care of inter-process communication. Furthermore, system overhead is a lot lower with the creation of threads than with the creation of processes. Despite such benefits, threads are not so easy to manage as their execution is often unpredictable. Moreover, lock and release mechanisms need to be in place to avoid overwriting shared memory. A system programmer would need a lot of expertise before diving into thread programming.

In this chapter, characteristics of processes and threads are discussed in detail. Knowing them assists one in understanding what needs to be checkpointed for a process and a thread migration. The chapter closes with a discussion on creating a multi-processed and a multi-threaded program to demonstrate differences between a process and a thread.

4.1 Characteristics of a Process

The description of a process is encapsulated within `task_struct` structure defined in `SOURCEHOME/include/linux/sched.h`. The `task_sruct` contains a process's state, scheduling information, (user and group) identifiers, inter-process communication, links, times and timers (of process creation time), file system, virtual memory, and processor specific context. When the kernel schedules a task, the `task_struct` of a process is loaded into a special part of the kernel's memory. Based on the context of the current process, the kernel knows where it should start. When a process is suspended, all CPU specific context such as the processor's registers and stacks related to that process will be saved. All relevant settings, values, and parameters of a process are saved as well, to enable a *context switch*, where one process is replaced with another one within the kernel memory.

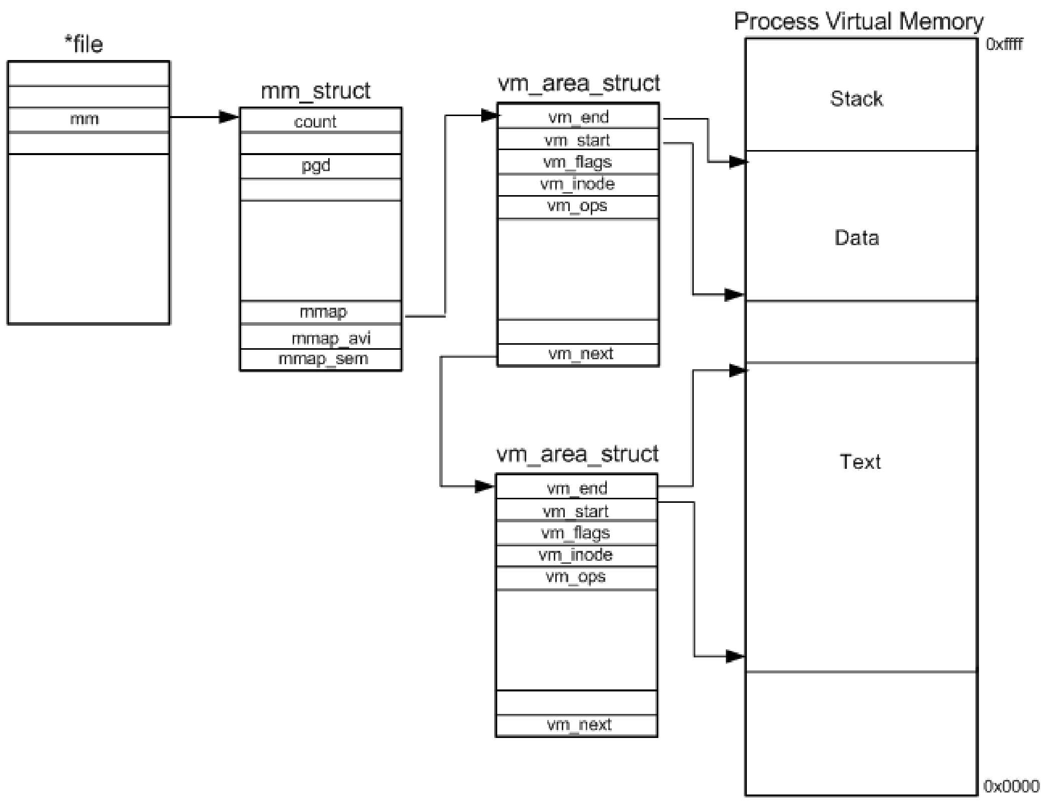

Figure 4.1: A Process's Virtual Memory.

Virtual memory is one of the most important components of a process. Virtual memory contains a program's image, which includes the header, executable code, and data. The header describes the contents of the image file; that is, how the file is to be interpreted. Furthermore, the header contains locations of the executable code and data. Executable code, the *text segment* of the program image, is a set of execution instructions. Data, the *data segment*, contains variables, structures, arrays of a program. There is also a *stack segment*, that holds information of functions the program is calling. Upon receiving a return value of a function, the information of the called function such as local variables, arguments to the function, returned address, and the old stack pointer are popped out of the stack. Figure 4.1 describes the relationship between `task_struck` of a process and its virtual memory.

Virtual memory also contains mapping to physical memory storing contents of files for reading. This newly allocated, virtual memory needs to be linked into the process's existing virtual memory so that it can be used. Last but not least, virtual memory contains links to libraries of commonly used code, for example file handling routines. It does not make sense that each process has its own copy of the library. Linux uses shared libraries that can be used by several running processes at the same time. To utilize them, the code and the data from these shared libraries must be linked into the process's virtual address space.

The second most important component of a process is its file system. A process contains open files that can be referenced by file descriptors. Open files include named files, pipes, and sockets. By referring to these descriptors, a process can indirectly access the inode of each open file with or without the operations provided for it.

In summary, a process is the execution of a program in action. It is represented with the `task_struct` structure. By taking care of the structure of a process, more specifically its virtual memory, file system, and processor's context, one can practically checkpoint a process and resume it on the same system. Next, we consider threads. Discussion of threads is focused mainly on their differences with processes.

4.2 Characteristics of Threads

A thread is simply a *lightweight* process in many aspects. One aspect is that the overhead of creating a thread is not as high as the overhead of creating a process. When a thread is created, the kernel does not need to make a new independent copy of the process address space and file descriptors. Besides its state, context, and its execution stack, a thread shares everything with its parent. Process address space, where executable code and data are kept, is visible to spawning threads. File descriptors from the parent address are freely accessible to child threads as well. Given that a thread is instantiated with less requirements, it implies that thread termination does not require as much effort as process termination.

The second aspect is concerned with the fact that data access among threads is like global variable access among a program's many functions. Data are shared among threads. Threads need not pass data to one another through inter-process communication like a file descriptor or shared memory space. The time that it saves to access process's resources and data makes a multi-threading model a better alternative in a typical application where concurrency requirements are top priority. Matrix multiplication is a good example of using threads. The outcome of each entry in a matrix can be computed separately and independently from outcomes of the rest of the entries. Since these concurrent operations may overlap in rows or columns of input matrices, threads are a way to go because the input matrices are shared among them.

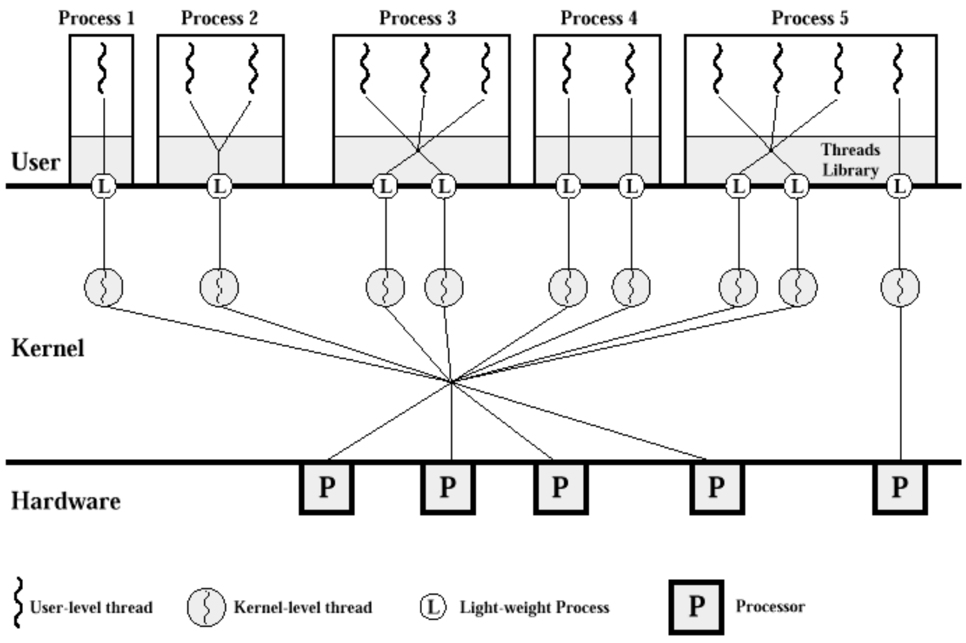

Figure 4.2: Solaris Thread Implementation.

There are three categories of thread implementation: user-level, kernel-level, and the combination of the two. The user-level approach is done by the use of thread libraries. Thread activity is invisible to the kernel. Because of that, when a thread is making a system call, all the rest of the threads will get blocked. Despite the operation dependence among threads, user-level thread implementation is flexible in adapting the best application-specific scheduling and synchronization algorithm. In contrast, kernel-level approach is by system calls. Operation independence of threads are traded off with user-level flexibility and cross-platform portability. The hybrid thread implementation combines the advantages of both user-level and kernel-level schemes by keeping scheduling algorithms in the user space and allowing operation independence and synchronization among threads. The latter is accomplished by letting programmers adjust the number of kernel threads to be created. Sun Solaris is an example of the hybrid thread implementation. Figure 4.2 shows the Solaris thread implementation.

4.3 Creating a Process

Process creation is done by the `fork()` system call. When one process calls the `fork()` command, an exact copy of itself and its state are created in memory. The newly created virtual memory points to the same physical memory as the parent's. The child process's `task_struct` structure will almost be identical as the parent process except process ID and the return value of `fork()`. From the parent's context, `fork()` returns child's process ID. From the child's context, `fork()` returns 0. Following is a sample code on process creation:

```
#include <linux/unistd.h>
#include <unistd.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <unistd.h>

int main (int argc, char**argv) {
  int pid;
  FILE *p;
  pid = fork();
  if (pid > 0) {
    // this must be the parent
    p = fopen("abc", "a");

    while(1) {
      printf("I'm parent, pid %d, child id %d\n", getpid(), pid);
    }
  }
  else {
    // this must be the child
    p = fopen("cde", "a");

    while(1) {
      printf("I'm child, pid %d\n", getpid());
    }
  }
  return 0;
}
```

Output shows:

```
I'm parent, pid 11686, child id 11687
```

```
I'm parent, pid 11686, child id 11687
I'm parent, pid 11686, child id 11687
I'm parent, pid 11686, child id 11687
I'm parent, pid 11686, child id 11687
I'm parent, pid 11686, child id 11687
I'm child, pid 11687
I'm child, pid 11687
I'm child, pid 11687
I'm child, pid 11687 ...
```

In this program, the parent and the child open file "abc" and "cde" for writing, respectively. As one can see, both the parent and child show up according to some kernel scheduling scheme. Besides their difference in PID, where parent is 11686 and child is 11687, both processes do not share file descriptors to files. The file descriptor table for each process demonstrates this:

Parent's descriptor table:

```
total 0
lrwx------    1 root     root     64 Apr  4 18:41 0 -> /dev/pts/0
lrwx------    1 root     root     64 Apr  4 18:41 1 -> /dev/pts/0
lrwx------    1 root     root     64 Apr  4 18:41 2 -> /dev/pts/0
l-wx------    1 root     root     64 Apr  4 18:41 3 -> /usr0/thesis/abc
```

Child's descriptor table:

```
total 0
lrwx------    1 root     root     64 Apr  4 18:47 0 -> /dev/pts/0
lrwx------    1 root     root     64 Apr  4 18:47 1 -> /dev/pts/0
lrwx------    1 root     root     64 Apr  4 18:47 2 -> /dev/pts/0
l-wx------    1 root     root     64 Apr  4 18:47 3 -> /usr0/thesis/cde
```

For the parent, its file descriptor 3 is pointing to file "abc". For the child, its file descriptor 3 is pointing to file "cde". Neither one of them shares the file descriptor to each other's file. Later on, we will see one example of multi-threaded program where threads share their file descriptors or access to files.

4.4 Thread API

In this section, the kernel-level `POSIX` thread API is discussed. In addition to knowing what the thread API has to offer, it helps in understanding the thread example in Section 4.5. The API is not intended to be exhaustive. But the discussion of the subset API is sufficient for the purposes of this thesis. One needs to `#include <pthread.h>` for the API to work. In addition, linking with the thread library `lpthread` is necessary during compilation. For example, `gcc -o thread thread.c -lpthread`.

- `int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void*(*start_routine)(void *), void *arg)`

  Inputs

  **tid* A pointer to `pthread_t` that holds the thread ID of the created thread.

  **attr* A pointer to `pthread_attr_t` which specifies the attributes of a thread. `NULL` sets it to default thread attributes. One of the default attributes is that the thread is non-detached.

  **start_routine* A function pointer that starts the execution of the thread. The thread exits with the exit status set to the value by the routine.

  Description The function creates a thread.

  Return Value 0 is returned on success. On error, a non-zero code is returned.

- `int pthread_join(pthread_t tid, void **status)`

  Inputs

  tid The ID of the thread that is to be joined.

  ***status* It points to the exit status of the thread with ID `tid`. Exit status can be set by `pthread_exit()`.

  Description The function blocks the calling thread until the thread with `tid` terminates. Thread with `tid` must not be in a detached state in order for the calling thread to block. The function is helpful when one wants to find out the status of a defunct thread.

  Return Value 0 is returned on success. On error, a non-zero code is returned.

- `int pthread_detach(thread_t tid)`

  Inputs

  tid The ID of the thread that is to be detached.

  Description The function detaches a thread that has `tid`. It guarantees that memory resources of the thread will be freed after thread termination. If the thread with `tid` is detached, calling `pthread_join` on such a thread will fail. If some thread joins this thread with `tid`, calling `pthread_detach` will fail.

  Return Value 0 is returned on success. On error, a non-zero code is returned.

- `pthread_t pthread_self(void)`

  Description The function returns ID of the thread. It is called within the executing routine of the thread.

Return Value A non-negative number is returned.

- `void pthread_exit(void *retval)`

  Inputs

  **retval* A pointer to the returned value of the calling thread.

  Description The function exits the calling thread with all thread-specific data bindings released. It is not clear from the documentation if memory resources will be freed without explicit declaration of `pthread_detach`.

4.5 Creating a Thread

A simple multi-threaded program is listed in this section to demonstrate the use of the thread API. By the end of this section, one should realize obvious differences between threads and processes.

```
#include <pthread.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

// function pointer declaration
void *openFile(void *arg);

int main() {
  pthread_t tid;
  int i;
  char filename[5] = "a1";
  char filename2[5] = "a2";

  // create two threads, each accesses a different file, "a1", "a2",
  // respectively
  pthread_create(&tid, NULL, openFile, filename);
  pthread_create(&tid, NULL, openFile, filename2);

  // must be inserted to keep the program running
  while(1) {
    printf("Parent pid %d\n", getpid());
  }
  return 1;
}

void *openFile(void *arg) {
  pthread_t tid;
  FILE *p;
  pthread_detach(tid);
  tid = pthread_self();

  // open file for writing
  p = fopen((char *) arg, "a");

  while(1) {
    printf("My thread ID is %d; my pid is %d\n", tid, getpid());
  }
  pthread_exit(0);
}
```

In this program, two threads are created. Each opens a file for writing. They will be in the infinite loop as the parent. Note that the `main` function needs a while loop to keep the entire program running. If the main function ends, the threads will be forced to terminate regardless of their states. A typical output looks like the following:

```
Parent pid 11936
Parent pid 11936
Parent pid 11936
Parent pid 11936
Parent pid 11936
My thread ID is 1026; my pid 11938
My thread ID is 1026; my pid 11938
My thread ID is 1026; my pid 11938
My thread ID is 1026; my pid 11938
...
My thread ID is 2051; my pid 11939
My thread ID is 2051; my pid 11939
My thread ID is 2051; my pid 11939
My thread ID is 2051; my pid 11939
...
Parent pid 11936
Parent pid 11936
Parent pid 11936
...
```

As the program runs, the three processes switch control with one another. The process table shows 4 PIDs related to this program. PID 11936 is the ID of the main function. 11937 is the controlling thread, which oversees two threads created by `pthread_create`. PID 11938 and 11989 are two threads running `openFile` routine.

```
total 0
lrwx------    1 root     root     64 Apr  4 22:22 0 -> /dev/pts/0
```

```
lrwx------    1 root     root      64 Apr  4 22:22 1 -> /dev/pts/0
lrwx------    1 root     root      64 Apr  4 22:22 2 -> /dev/pts/0
lr-x------    1 root     root      64 Apr  4 22:22 3 -> pipe:[14532]
l-wx------    1 root     root      64 Apr  4 22:22 4 -> pipe:[14532]
l-wx------    1 root     root      64 Apr  4 22:22 5 -> /usr0/thesis/a1
l-wx------    1 root     root      64 Apr  4 22:22 6 -> /usr0/thesis/a2
```

The above shows the file descriptor table for process with PID 11936. In fact, all processes from PID 11937 to 11939 all have the same file descriptor table. This indicates that all the threads have access to the same files. Moreover, read and write pipes are available among threads for inter-communication.

Chapter 5

Implementation of Network Process Migration

This chapter assumes that readers have read Chapters 3 and 4 and know various data structures related to sockets and characteristics of processes and threads. First, it discusses the design of socket and process migration, specifically, the relationship among the kernel, kernel module, and the user programs. Next, it explains MIGSOCK data structures for the support of multiple socket migration. Modifications of the user program to invoke socket migration are then discussed. CRAK is briefly introduced to set the stage for its modification for support of multi-threaded and multi-processed programs. Finally, the chapter ends with the implementation of migration for multi-threaded and multi-processed programs with sockets.

5.1 Design Overview

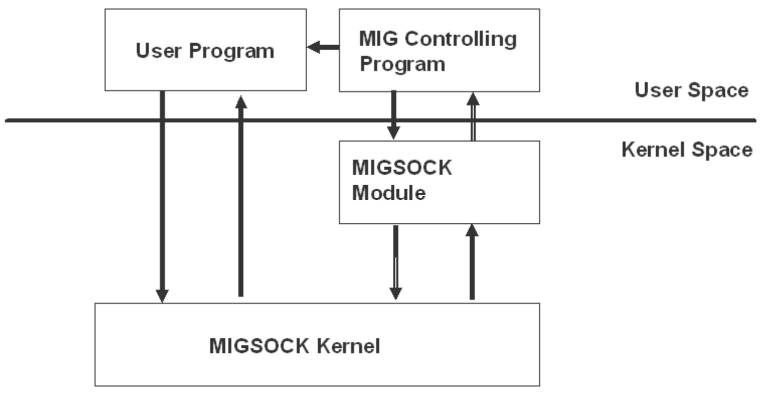

Figure 5.1: Layers of Socket Migration Design.

The design of socket migration can be broken down into two levels: the kernel level and the user level. Figure 5.1 shows the layered design of socket migration. In the kernel level, it includes modifications of the kernel and kernel module. Kernel modification includes encoding a TCP packet with a special MIGSOCK message upon departure and handling a TCP packet with a special MIGSOCK message upon arrival. Encoding a special MIGSOCK message occurs in `tcp_transmit_skb(...)` in `SOURCEHOME/net/ipv4/tcp_output.c` when the packet is transmitted out on the wire. On the receiver's end, checking a special MIGSOCK message happens in `tcp_v4_rcv(...)` and `tcp_rcv_established(...)` in `SOURCEHOME/net/ipv4/tcp_input.c`. Responding to the special MIGSOCK message by putting the remote process to sleep occurs in `tcp_sendmsg(...)` in `SOURCEHOME/net/ipv4/tcp.c`. Users are recommended to read Chapter 3.3 on *the Life of a TCP Packet* and MIGSOCK paper [5] for implementation details. The existing kernel code does not need modifications.

Kernel module modifications are mainly involved with MIGSOCK data structures in order to support multiple sockets (See Section 5.2). By changing this particular kernel MIGSOCK data structure, it opens the door for the module to access the socket inode for each socket file descriptor of a process/thread. Functions that need to accommodate such a change include `sys_migsock_req(...)` for requesting for migration, `sys_migsock_rst(...)` for resuming communication, `sys_migsock_tofile(...)` for serializing socket states, `sys_migsock_fromfile(...)` for de-serializing socket states, and `sys_migsock_restart(...)` for initializing a socket with de-serialized socket states.

User level design can further be categorized into two parts. One is the user-level controlling program that is responsible for taking care of the actual socket migration by interacting with the kernel module. The other is the user-level program which is a normal socket program utilizes the socket API. To migrate a socket, the controlling program first stops the migrating program. It then hijacks the socket and sends a migration request to the migrating program's remote peer. To resume communication, the controlling program issues kernel module system calls to reinstate the socket. It then hijacks the socket and sends a restart request to restart communication with the migrating program's remote peer. The controlling program terminates itself once the remote peer of the migrating program has received request for migration or request for restart message. No explicit acknowledgement triggers the termination. Because of reliability guaranteed by TCP, controlling programs terminate upon execution of MIGSOCK commands.

Figure 5.2 demonstrates the steps of the controlling program's hijacking of the socket. The MIGSOCK messages are guaranteed to arrive at the remote peer due to the reliability provided by TCP. The dotted vertical lines represent the user-level controlling programs that hijack the communication with the remote process to perform socket checkpointing and restarting. Note that it is possible to either serialize the socket and send MIG REQ simultaneously or deserialize the socket and send MIG RST simultaneously

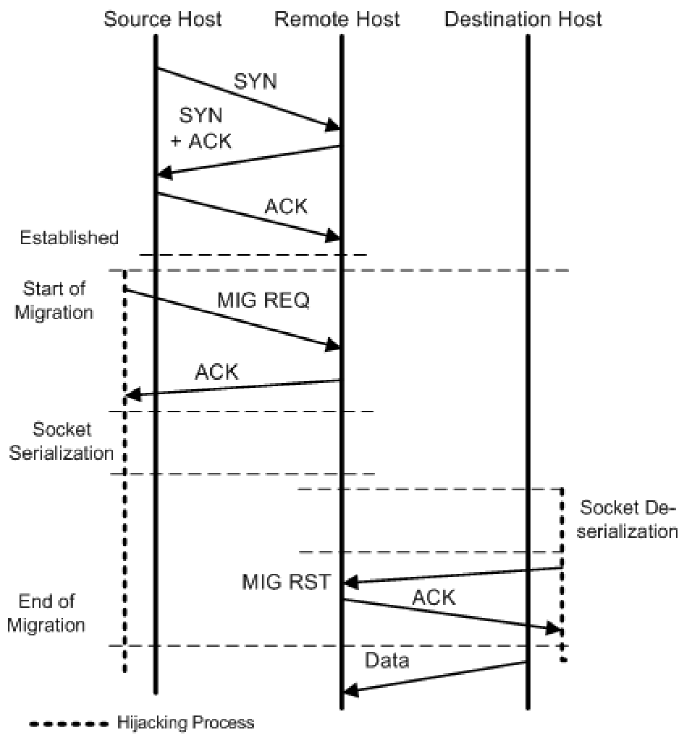

Figure 5.2: Message Timeline Showing When Hijacking Controlling Programs Are Active.

in a pipelining fashion for the sake of efficiency.

Process migration is performed at the user level. CRAK Module remains unmodified. What needs to be modified is the user-level controlling program for support of multi-threaded and multi-processed program migration. Implementations of both checkpointing and restarting are discussed in Section 5.4.1 and 5.4.2.

5.2 MIGSOCK Data Structures Modifications

There are two main MIGSOCK data structures. Both of them are parameters that facilitate communication between the user-level controlling program and kernel module. Information passed from the user space to the kernel space include the process ID and socket file descriptors. Information passed from the kernel space to the user space include the entire states of a socket. The states will be de-serialized later when the process finishes migration.

Readers are recommended to look up those two data structures, `struct migsock_params` and `struct migsock_config_data`, in MIGSOCK[5]. `struct migsock_params` structure is the one that needs modification, which is shown in the following:

```
/* Structure used to transfer data between user space and kernel module */
struct migsock_params {
  // This must be first and of sufficient size
  // to store the serialized socket data
  char data[MIGSOCK_MAX_BUFF];
  pid_t pid;

  /**************************************
     Modified Code Section Begins
  **************************************/
  // Stores the socket file descriptors
  int sockfd[5];
  // This is to indicate to the module which socket we are referring to
  int flag;
};
```

Here, `sockfd` is changed from `int sockfd` to `int sockfd[5]` to support migration of up to 5 sockets. `flag` is added to indicate to the kernel module on which socket that migration request should be sent. In addition, it indicates to the kernel module states of which socket need to be saved and restarted. The modification of `struct migsock_prarams` structure applies to socket, process, and network process migration.

5.3 Single Process with Multiple Sockets

Kernel module and user-level controlling program need to be modified to conform to the modified MIGSOCK data structure. Following two subsections describe each in detail.

5.3.1 Kernel Module Modification

The modification applies to `migsock_mod.c`. For all occurrences of

```
if (usrbuf->sockfd < files->max_fds)
    file = files->fd[sockfd];
```

replace it with

```
if (usrbuf->sockfd[usrbuf->flag] < files->max_fds)
    file = files->fd[usrbuf->sockfd[usrbuf->flag]];
```

where `files` is of type `struct files_struct` and `file` is of type `file`. `usrbuf` is of type `struct migsock_params`. It it passed as one of the parameters to MIGSOCK module system calls. As mentioned in Section 5.2, `flag` lets the kernel module know which of the socket in `sockfd[]` that the user-level controlling program is referring to. Once the file is known, socket inode can be obtained as described in Section 3.5.

5.3.2 User-Level Program Modification

The user-level program modification involves populating `sockfd[]` within `struct migsock_params` structure as they become available. Issuing user-level program on the command line is limited to 5 sockets per program. What the controlling program then does is for each socket on the command line, it issues the same migration request and either serializes socket states to a file or deserialize it to MIGSOCK. Since each socket is separate from the others in the sense that each has its own file descriptor and is on a unique communication channel, this scheme works. As for the remote peer that hosts sockets, it puts itself into the sleep queue for each socket. In other words, each socket contains the same process. When a wake-up message is received, the process is interrupted. It then removes itself from the corresponding socket sleep queue and starts communicating on that particular socket.

Following shows the partial code of socket migration. The complete code of socket migration and socket restart can be found in Appendix B.

```
int main() {
  ...
  num_sockets = 5;
```

```
// Take socket file descriptors from the command line
// command line follows: user_prog <pid> <sock1> <sock2> <sock3>
// <sock4> <sock5>
for(i = 0; i < num_sockets; ++i) {
  arg->pid = atoi(argv[1]);
  arg->sockfd[0] = atoi(argv[2]);
  arg->sockfd[1] = atoi(argv[3]);
  arg->sockfd[2] = atoi(argv[4]);
  arg->sockfd[3] = atoi(argv[5]);
  arg->sockfd[4] = atoi(argv[6]);
}

for(i = 0; i < num_sockets; ++i) {
  ...
  // Tell MIGSOCK kernel module which sockfd[] it should be
  // looking for
  arg->flag = i;

  // 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 %d.\n", ret);
    goto done;
  }

  // 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");
    goto done;
  }

  // Write the serialized data to file
  sprintf(filename, "fdata%d.bin", i+1);
  fout = open(filename, O_WRONLY | O_CREAT);
  if (fout == -1) {
    printf("%dth socket: Failed to create %s.\n", i + 1, filename);
    goto done;
  }

  // Write the serialized data in arg->data to MIGSOCK_FILENAME
  ret = write(fout, (void *)(arg->data), MIGSOCK_SERIAL_DATA_SIZE);
  if (ret == -1) {
    printf("%dth socket: Failed to write to %s.\n", i + 1, filename);
    ret = -1;
    close(fout);
```

```
      goto done;
    }

    printf("%d bytes of serialized data written to %s\n", ret, filename);

    close(fout);
  }
  ...
  return ret;
}
```

At the end, five files are created with names `fdata1.bin`, `fdata2.bin`, `fdata3.bin`, `fdata5.bin`, and `fdata5.bin`, respectively. Each contains serialized socket states to restart the socket.

This is the simplest modification of the user-level controlling program. However, the modification is not the cleanest and the most efficient. From the trace of MIGSOCK checkpointing five sockets, the first request message for migration will put the remote process to sleep. Since the process is sleeping, packets sent to the other four sockets on this process will be queued up. In other words, there is no need to send a migration request message for each of the four remaining sockets. The kernel buffers those request messages. When a restart message is sent on the first socket, the process removes itself from the sock's sleep queue. It will go to sleep again because of the request message for migration from the second socket. It then wakes up by the restart message from the second socket. This procedure goes on until the last restart message is consumed.

A more efficient but rather complex algorithm would be to send only one message each for request for migration and restart. In this approach, the restart message would need to have a large enough data segment to contain information about the new host where the migrating process resides on. The data segment needs to be as large as the number of data for each socket multiplied by the number of checkpointed sockets since each socket has different communication information (e.g., port).

5.4 Multi-Processed and Multi-Threaded Programs

The implementation of checkpointing and restarting for multi-processed and multi-threaded programs follows naturally from the discussion of their characteristics in Chapter 4. By knowing their anatomy, i.e., what structure that represents them, one should understand the flow of multi-processed and multi-threaded checkpointing and restarting. Rather than presenting the code to illustrate the implementation details, each of the two sub-sections presents readers procedures in checkpointing and restarting multi-processed and multi-threaded programs. Both sub-sections concentrate on CRAK user-level controlling programs, `ck.C` and `restart.C`. These two are C++ programs that utilize the CRAK checkpointing/restarting module.

5.4.1 CRAK Checkpointing

Steps to checkpoint a multi-processed and multi-threaded programs are as follows:

1. Determine from the command line if one wants to checkpoint a process, multi-processed program, or multi-threaded program.
2. If it is a multi-processed or multi-threaded program, get the children PIDs of the process and put it in a list called `pid_list` in the order that they were spawned by the parent. Else, i.e., a single-processed program, skip this step.
3. Put the parent at the *end* of the `pid_list` to compose a full process list.
4. For each process in the `pid_list`, call the `checkpoint` system call in the CRAK module and supply it with the file name for serialized data, process id, and other relevant checkpointing options.

The `checkpoint` system call is not part of the modification so the discussion of it is skipped here. The main idea is that the call gathers and saves the virtual memory, file system, and signal handlers by the `task_struct` structure of a process. Readers are recommended to read Section 4.1 for details of `task_struct`.

Since the parent and its children process have no dependency on one another (they all have their own PIDs, virtual memory, and file systems), the last step of the checkpointing procedure is valid. One might argue that if a family of processes have their virtual memory pointing to the same physical memory, the destruction of a process in the family can cause the kernel to free up the physical memory. However, it is not the case. Linux kernel is smart enough to only garbage-collect a piece of physical memory when it becomes available, i.e., all processes pointing to it are destroyed.

Checkpointing threads uses the same procedure as checkpointing processes. Threads are *lightweight* processes. They behave the same as processes except their file descriptors and pipes are copied among the family of threads . File descriptors are created for threads that do not own them when other threads that own them are started. These file descriptors belonging to different threads point to the same file. Destroying one thread in a family of threads does not prevent the other thread from accessing the file that is opened by the destroyed thread. Write and read pipes are created when a thread is created. They are extra functionalities that allow threads to communicate with one another. Similarly, destroying one thread in the family does not prevent the other thread from accessing its own pipes.

Despite their independence from one another, each thread's operation depends on the parent thread. If the parent thread terminates, the child threads will terminate as well. However, the reverse is not true; i.e., the termination of child threads will not affect the parent thread. Hence, it is important to checkpoint child threads before the parent thread. To make sure that checkpointing the parent does not abort its children, parent thread is put in the very end of the family of processes to be checkpointed as mentioned in Step 3.

5.4.2 CRAK Restarting

Steps to restart a multi-processed and multi-threaded programs are as follows:

1. Determine from the command line if one wants to restart a process, multi-processed program, or multi-threaded program.

2. If it is a single-processed program, gather the process's open files and other relevant information from the serialized data file.

   (a) Build file descriptors corresponding to files.
   (b) Call the `restart` system call in the CRAK module.

3. If it is a multi-processed or multi-threaded program, iterate through the serialized files in the order in which they are created, create a process list that contains the parent, each child, and their open files and other relevant information. The list is sorted in the order of process/thread being created.

   (a) Fork the restart user-level controlling program.
   (b) For each child, fork another process.
   (c) Build pipes if there are any in a process or thread. Assign pipes to read and write pipes, respectively.
   (d) Build file descriptors corresponding to files.
   (e) Call the `restart` system call in the CRAK module on each child.
   (f) Repeat 3b until reaching the end of the list.
   (g) Call the `restart` system call in the CRAK module on the parent.

Since the `restart` system call remains unmodified. it will not be discussed in detail. In a nutshell, it undoes what `checkpoint` system call does. It reads and deserializes the process data. It then populates the data to appropriate elements of the current process's `task_struct`. Readers are recommended to read Section 4.1 for details of `task_struct`. CRAK `ckpt.c` shows the details of checkpointing and restarting for those who are intended.

Files are first created according to the checkpointed files. Each checkpointed file is associated with a file descriptor. If the open file descriptor does not match with the checkpointed one, it is duplicated to create a consistent view of the process image.

Pipes are created by using the `dup2` system call to duplicate the checkpointed pipes in the family to ones in a process. First a file system of a process is checked to see if it is a pipe inode. If it is, its type, whether read or write, is determined. `dup2` then duplicates the existing pipes built by the `pipes` system call. This reinstates the full status of an open pipe.

For a multi-processed or a multi-threaded program, Step 3 creates $n + 1$ processes where n is the number processes in the program. The additional process, responsible for resurrecting the family, is the user-level controlling program and the grandfather of all these n processes.

5.5 Multi-Threaded and Multi-processed Programs with Sockets

Having talked about multiple-socket migration, checkpointing, and restarting for multi-processed and multi-threaded programs, it is logical to move on to ask whether multi-threaded and multi-processed programs with sockets can be checkpointed and restarted. Combining the two requires joint efforts from CRAK and MIGSOCK. CRAK is used to checkpoint and restart a program except its sockets. MIGSOCK is used to checkpoint and restart sockets of the program.

Before looking at the recipe for checkpointing a multi-threaded program, one should keep two assumptions in mind. First, threads are only allowed to talk on sockets that they create. Although threads share the set of sockets, most of the sockets that are inherited by a thread are not used to send packets. Second, the remote host must be a single-processed program with multiple sockets. The goal is to demonstrate the ability to checkpoint and restart sockets for a multi-threaded/processed program. The form of the remote host should be kept as simple as possible to minimize faults of checkpointing and restarting, if they were to happen, to the migrating program. The recipes for checkpointing and restarting multiple sockets for a multi-processed and multi-threaded program are illustrated below:

Checkpointing a multi-processed program with sockets:

1. Obtain the process list that contains a family of PIDs with the given PID as the parent in the multi-processed program.

2. For each item in the list, suspend it by issuing a `SIGSTOP` signal.
3. For each item in the list except the parent, call the MIGSOCK checkpointing system call to suspend the remote process and save socket's states.
4. For each item in the list, call the CRAK checkpointing system call to save the process's states and kill the process. The parent process is checkpointed last to avoid premature termination.

Restarting a multi-processed program with sockets:

1. For each serialized process image:
   (a) Fork a process and call the CRAK `restart` system call to populate memory with the serialized process's states.
   (b) If the process is not the parent, put it to sleep.
2. For each restarted process except the parent and the controlling thread, call the MIGSOCK restarting system call to deserialize socket states and to wake up the remote process.

Checkpointing a multi-threaded program with sockets:

1. Obtain the process list that contains a family of PIDs with the given PID as the parent in the multi-threaded program.
2. For each item in the list, suspend it by issuing a `SIGSTOP` signal.
3. For each item in the list except the parent and the controlling thread, call the MIGSOCK checkpointing system call to suspend the remote process and save socket states.
4. For each item in the list, call the CRAK checkpointing system call to save the process's states and to kill the process. The parent thread is checkpointed last to avoid premature termination.

Restarting a multi-threaded program with sockets:

1. For each serialized process image:
   (a) Fork a process and call CRAK restarting system call to populate it with serialized process's states.
   (b) If the process is not the parent or the controlling thread, put it to sleep.
2. For each restarted process except the parent and the controlling thread, call the MIGSOCK restarting system call to de-serialize socket states and to wake up the remote process.

The only difference between a multi-processed and multi-threaded program with sockets in checkpointing is in Step 3. Since a multi-processed program does not have a controlling thread as a multi-threaded program, only the parent is excluded from MIGSOCK system calls. Similarly in restarting, a multi-processed program does not have a controlling thread as a multi-threaded program, only the parent is excluded from being put to sleep in Step 1b and from MIGSOCK system calls in Step 2.

The modification of the user-level controlling program requires that the checkpointed program be stopped first to keep it from sending any packets. Since the parent and the controlling thread do not hold sockets on which communication with the remote host takes place, only sockets belonging to processes that are neither the parent nor the controlling thread need to be checkpointed and restarted. Thus when restarting the sockets, the controlling program does not reinstate socket states for the parent and the controlling thread. It does not hurt if one would checkpoint and restart the socket owned by the parent to demonstrate completeness and correctness. However, the above recipes prove to be sufficient to carry out checkpointing and restarting operations for multi-processed/threaded programs.

5.6 Test Programs

Several test programs are developed to test these modifications to MIGSOCK. They are a type of client/server application. In summary, the server sends a number which starts from 0 and gets incremented by 1 to the client. The client receives the number and displays it on the console. The description of these programs are as follows:

`count_client.c` & `count_server.c`: They are used to test that MIGSOCK checkpoints and restarts multiple sockets for a single-processed program. Both the client and the server have two sockets. Sockets on the server are bound to the server's IP address. The sockets of the client are connected to the server's. Upon connection, the incremental number is sent to two sockets on the server and received on two sockets on the client's side.

`count_client2.c` & `count_server2.c`: They are used to test that MIGSOCK checkpoints and restarts a *single* socket for a single-processed program. Both the client and the server have one socket. The socket on the server are bound to the server's IP address. The socket of the client is connected to the server's. Upon connection, the incremental number is sent to the socket on the server and received on the socket on the client's side.

`client3.c` & `server3.c`: They are simply echo programs that are used to evaluate Netfilter performance. The client will send some data to the server where the server echoes it back. Data need not be numbers. To evaluate the performance of Netfilter with respect to data size and packet number, the `client.c` program allows user to specify the number of packets and the size of the packet to be sent over to the server at runtime. The server allows the user to specify the size of the packet as well. Detailed usage is documented in Section 6.2.2.

`proc.c`: This is used to test modified CRAK to checkpoint and restart a multi-processed program. Two processes are forked within `my_fork.c`. Each child process opens a unique file. One needs to make sure that processes are restarted correctly with their own file descriptors intact.

`thread.c`: This is used to test modified CRAK to checkpoint and restart a multi-threaded program. Two threads are spawned within `my_thread.c`. Each child thread opens a unique file. One needs to make sure that processes are restarted correctly with their own file descriptors intact *and* shared among threads.

`proc_client.c` & `proc_server.c`: They are used to test MIGSOCK checkpointing and restarting multiple sockets for a multi-processed program. The client creates 5 sockets that are connected to the listening socket on the server. For each connection that is accepted, the server forks a process and sends the incremental number on the socket file descriptor returned by `accept()` system call. Five children processes are forked to support five connections from the client.

`thread_client.c` & `thread_server.c`: They are used to test MIGSOCK checkpointing and restarting multiple sockets for a multi-threaded program. The client, the same as `proc_client.c` creates 5 sockets that are connected to the listening socket on the server. For each connection that is accepted, the server spawns a thread and sends the incremental number on the socket file descriptor returned by `accept()` system call. Five children threads are spawned to support five connections from the client.

Chapter 6

Evaluation

MIGSOCK is a kernel-level approach to implement network process migration. Kernel is being modified to make migrations possible. It is expected that its performance should be better than that of an user-level implementation. This chapter presents experiments that attempt to validate that claim.

Zap [9] was selected because it was by far the most similar user-level approach to MIGSOCK that one could find in this field. Zap provides a virtualization layer on top of which a *Process Domain* (pod) exits. A pod is a group of processes that have the same virtualized view of the system. By utilizing Zap, each process within a pod is able to perform its service without problems of dependencies and conflicts of the underlying system.

Zap migrates network processes by first installing a proxy process that will stall current connections. Zap then suspends the pod by stopping all processes in it, saving virtualization mappings and process states. Upon restart, Zap restores processes in the pod by populating their process states. It then informs its remote host and resumes the communication. The pod and its remote host still keep their own addresses. The virtual to physical address mapping is taken care of by Zap to ensure that both ends do not need to worry about the connection virtualization and translation. Zap uses Netfilter to accomplish address translation between the pod and its communication partner. Address translation is done twice as a packet goes out and arrives at either end. The packet's destination address first gets translated to the physical address before being sent off the wire. Once it arrives at the other end, it gets translated again to the virtual address before entering into a pod.

Since both MIGSOCK and Zap use CRAK as a basis for process migration, this chapter will focus on the overhead of MIGSOCK in checkpointing and restarting sockets and the overhead of Netfilter in translating and directing packets. Setup, procedures, and results of MIGSOCK and Netfilter are described in Section 6.1 and Section 6.2, respectively. Overall findings concludes the evaluation by summarizing and giving justification as to which approach is better suited for network process migration.

6.1 MIGSOCK Checkpointing and Restarting Overhead

The overhead of MIGSOCK checkpointing and restarting derives mainly from the latency involved in sending and receiving special messages. As discussed before, these messages are MIGSOCK-specific TCP packets that notify the remote host to block packet transmission on the particular socket and to wake the process up upon completion of socket migration. To capture this latency, a time stamp is recorded before and after related MIGSOCK system calls. The difference in time should give a fairly accurate approximation of the overhead. The following code segment records the time difference in the user-level controlling program:

```
#include <sys/time.h>

// declare two variables of type struct timeval
struct timeval start_time, stop_time;

// start timing
gettimeofday(&start_time, NULL);
...

// time of a system call one wants to capture
...

// stop timing
gettimeofday(&stop_time, NULL);


// print out difference in time in milliseconds
printf("start_migrate: Total time in millisecond %ld to send req\n",
    stop_time.tv_usec/1000 + stop_time.tv_sec * 1000
    - start_time.tv_usec/1000 - start_time.tv_sec * 1000);
```

`gettimeofday(...)` gets the current time of the system and writes the value to the memory pointed by a pointer of type `struct timeval`, where it is defined as:

```
struct timeval {
  long tv_sec;        /* seconds */
  long tv_usec;       /* microseconds */
};
```

The system calls that need to be timed are `sys_migsock_req(...)` for checkpointing a program and `sys_migsock_restart(...)` for restarting a program.

6.1.1 Setup and Procedures

Since measuring the overhead of checkpointing and restarting a socket is the focus of this evaluation, the user-level socket API programs will concentrate on the single-socket communication. In other words, the server program only accepts and service one incoming connection from the client. Both the server and client are single-processed programs. They are documented in Section 3.2.

The server and client are located close to each other, less than 1 meter apart, to eliminate overhead due to propagation delay. The server will first initiate itself for the client to establish a connection. The client will be migrated to the same machine to isolate additional overhead introduced by a different distance. In summary, the procedures of this part of the experiment are:

1. The server initiates itself on Host A.
2. The client on Host B connects to the server on Host A.
3. (a) The MIGSOCK system call is called to stop the client and put the server to sleep.
   (b) The MIGSOCK system call is called to serialize the states of the socket with which the client establishes the connection to the server.
4. The CRAK checkpointing program is called to save the client process's states and kills it.
5. The CRAK restarting program is called to create a client process on Host B and populate it with the serialized image of the process.
6. The MIGSOCK restarting program is called to recover the socket from the serialized image of the socket and wake up the server.
   (a) The MIGSOCK system call is called to de-serialize the states of the socket with which the client established the connection to the server.
   (b) The MIGSOCK system call is called to start the server and resume the connection

Only the times in steps 3 and 6 are measured since, they are the only MIGSOCK-related system calls. The time is measured by inserting a timestamp before and after the call. The difference indicates the overhead of each of the system calls.

6.1.2 Results

Figure 6.1 shows the time it takes to checkpoint and restart a socket. The experiment is conducted over 10 trials. The average, variance, and standard deviation were calculated. Assuming the data is normally distributed, one expects 99.7% of the checkpointing data to lie between 0.57ms and 2.10ms. One would also expect 99.7% of the restarting data to lie between 0.03ms and 5.55ms. We take the worst time in the ranges of checkpointing and restarting and use the sum to compare MIGSOCK's performance to Zap. In short, in the worst-case scenario, it takes MIGSOCK 2.10ms to checkpoint and 5.55ms to restart a socket. The total time it takes for MIGSOCK is 7.65ms.

Figure 6.1: Table of Times to Checkpoint and Restart.

# Time (in ms)	Checkpointing	Restarting
Avg.	1.53	2.79
Variance	0.04	0.84
Std. Dev.	0.19	0.92

6.2 Netfilter Translating and Directing Overhead

"Netfilter is a set of hooks (or chains) inside the the Linux kernel that allows kernel modules to register callback functions with the network stack. A registered callback function is then called back for every packet that traverses the respective hook within the network stack."[1] There are five default hooks placed evenly across the kernel routing to ensure packets are properly processed before being forwarded. The five default chains `PREROUTING`, `INPUT`, `FORWARD`, `POSTROUTING`, and `OUTPUT` are shown in Figure 6.2 [4]. Each chain has a set of rules that determine the fate of packets.

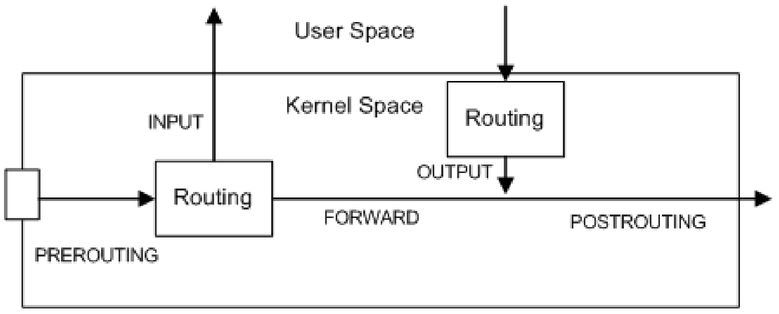

Figure 6.2: Netfilter Chains.

As a packet comes in, it enters the `PREROUTING` chain where it usually gets mangled. Information in the packet's header such as Source Address, Destination Address, or TOS will be modified. This alters the routing decision as it passes the next stage. If the packet is destined for the host, it passes to the `INPUT` chain before it gets passed further up to processes waiting for it. Otherwise, if the kernel does not know what to do with the packet or it does not have forwarding option enabled, the packet gets dropped. But if forwarding is enabled, it gets passed right along to the `FORWARD` chain. For the packets from the host to the outside world, it enters the routing decision before leaving for the `OUTPUT` chain, where it can get mangled before being sent out on the wire. Finally, all packets that are either forwarded through or from this host pass the last chain of `POSTROUTING` before leaving the host. One can see that packets entering a Linux box will have to pass at least three of these chains (`PREROUTING`, `FORWARD`, and `POSTROUTING`). This ensures some sort of security to the Linux system as a packet gets identified and appropriate action is taken.

Within a rule, a target or an action is defined. Default actions provided in Netfilter are `ACCEPT`, `DROP`, `QUEUE`, and `STEEL`. When a packet matches a rule in a chain, its fate will be determined by the action associated with the rule. If a packet matches none of the rules in a chain, the default action will be taken in the chain. Users can define their own actions which can either be a chain referenced by the default chains in Netfilter or an extension to the default action.

`iptable` is a user-level program that allows users to manipulate packets with `iptable`'s services. Each service is associated with a table. Some notable services are NAT, filter, and mangle. These tables contain chains in Netfilter. By specifying rules within the chains, `iptable` provides an easy way for users to control packets in and out of a system. Readers are recommended to look at the manual page for `iptables` for more details.

This section takes a look at the overhead of Netfilter in two areas. It seeks to demonstrate some relationship between its overhead and characteristics of packets. In addition, it seeks to demonstrate some relationship between rules in a chain and its overhead.

6.2.1 Packet Size and Number of Packets

This section wishes to find out the effect of Netfilter, more specifically, the overhead that is involved to transmit a packet passing through Netfilter if there is any. It is hypothesized that the overhead of Netfilter is proportional to the number of packets it needs to process. Furthermore, it is also hypothesized that this overhead increases as the packet size increases.

Setup and Procedure

The experiment requires a client to pass packets to a server and the server to pass the same packets back. There are two scenarios: regular and Netfilter-enabled. In a regular scenario, a client will connect to the servers public address and port directly. In a Netfilter-enabled scenario, a client connects to the servers public address and port. Packets are routed by Netfilter to a virtualized IP address where the server is listening on connections and to which its port is bound. Figure 6.11 illustrates these two scenarios.

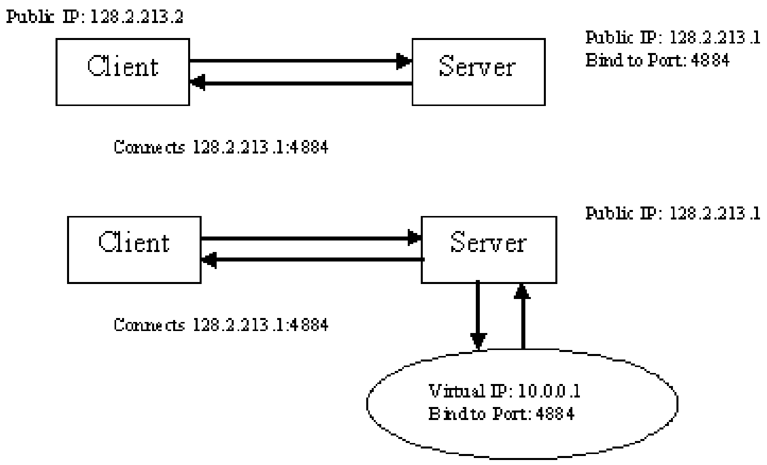

Figure 6.3: Experiment 1 Setup

To measure the Netfilter overhead, the time it takes for packets that left the client to arrive back at the client is recorded. Although the propagation time is included between the client and server in the measurement, it is negligible due to distance. The client and server machines are set up in such a way that their distance is about 1 meter apart. Furthermore, since the same setup is used for both regular and Netfilter-enabled scenarios, variation in the measurement would have to be due to Netfilter.

To start the regular scenario of this experiment, one would start the server first:

```
./server 128.2.213.1 4884 200
```

Where ”128.2.213.1” and ”4884” are the IP and port that the server is listening on. ”200” is the number of bytes per packet the server is expected to receive. Following is the server usage:

```
./ server <ipaddr> <port_num> <bytes per packet>
```

To start the client in the regular scenario of this experiment, one would issue:

```
./client 128.2.213.1 4884 1000 200
```

Where "128.2.213.1" and "4884" are the IP and port that the client is connected to. "1000" indicates the number of packets it is sending. And "200" is the number of bytes per packet. Following is the client usage:

```
./client <ipaddr> <port_num> <num_packets> <bytes>
```

The Netfilter-enabled scenario contains 3 steps: creating a virtualized IP, adding a rule to invoke Netfilter, and starting up the server program.

To create a virtualized IP, one does:

```
ifconfig eth0:1 10.0.0.1 netmask 255.255.255.0
```

Where "eth0" is the primary Ethernet interface and "10.0.0.1" is the virtualized IP address.

To add a rule to Netfilter, one does:

```
iptables -t nat -D PREROUTING -p tcp -I eth0 -j DNAT -to 10.0.0.1
```

This tells Netfilter that arrival TCP packets with destination address will be routed or "NATed" to 10.0.0.1. One can verify that such a rule is added by issuing:

```
iptables -t nat -n -L
```

A sample output looks like the following:

```
Chain PREROUTING (policy ACCEPT) target     prot opt source
destination DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0
to:10.0.0.1

Chain POSTROUTING (policy ACCEPT) target     prot opt source
destination

Chain OUTPUT (policy ACCEPT) target     prot opt source
destination
```

Results

The times for regular and Netfilter-enabled scenarios for each packet size are shown in both graph and table forms for purposes of clarity. Difference in times is in the order of 3.5 seconds for 1,000,000 800-byte packets. It indicates that Netfilter is scalable.

Figure 6.10 shows the difference in time vs. the number of packets of 3 sizes. The difference gets larger as the number of packets grows. However, the size of packets does not influence the time. The three lines are all within small time differences. Although it contradicts to one of the hypotheses mentioned earlier in this section, it does make sense. As a packet passes through Netfiler, only the header of the IP packet is looked at regardless its size. Since the header of an IP packet is constant, the time it takes for Netfilter to filter the packet is also constant.

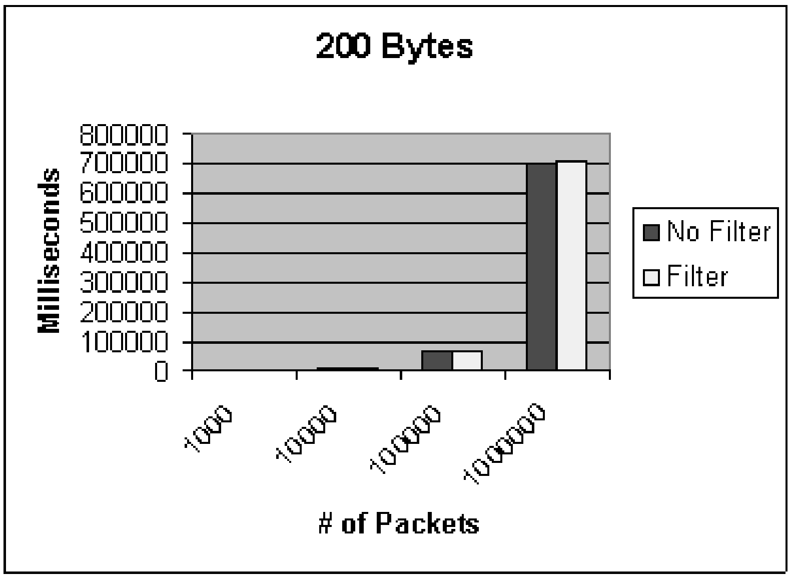

Figure 6.4: # of 200-byte Packets vs. Time.

# of Packets	1,000	10,000	100,000	1,000,000
(reg) Milliseconds	702	7,047	70,050.5	703,517
(netfilter) Milliseconds	704	7,062	70,646	706,345

Figure 6.5: Table of Times as # of 200-byte Packets Grows.

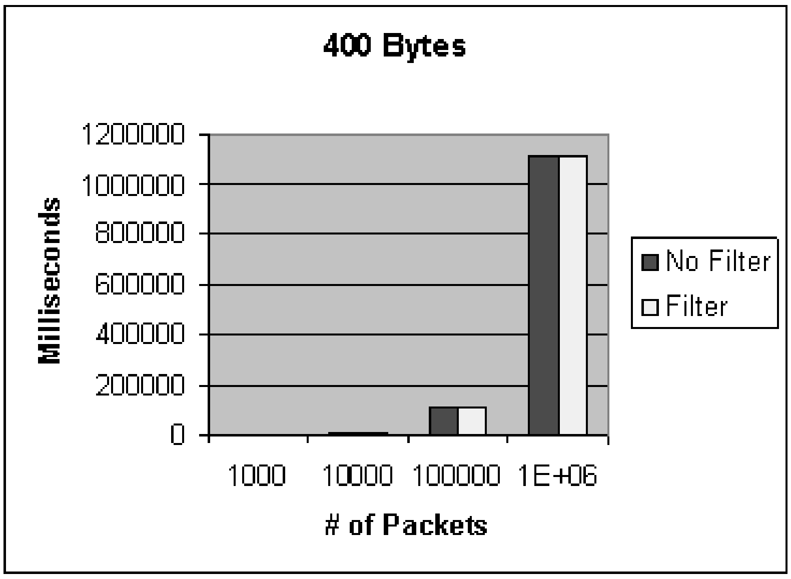

Figure 6.6: # of 400-byte Packets vs. Time.

# of Packets	1,000	10,000	100,000	1,000,000
(reg) Milliseconds	1,111	11,102	111,109	1,110,822
(netfilter) Milliseconds	1,114	11,139	111,335	1,113,109

Figure 6.7: Table of Times as # of 400-byte Packets Grows.

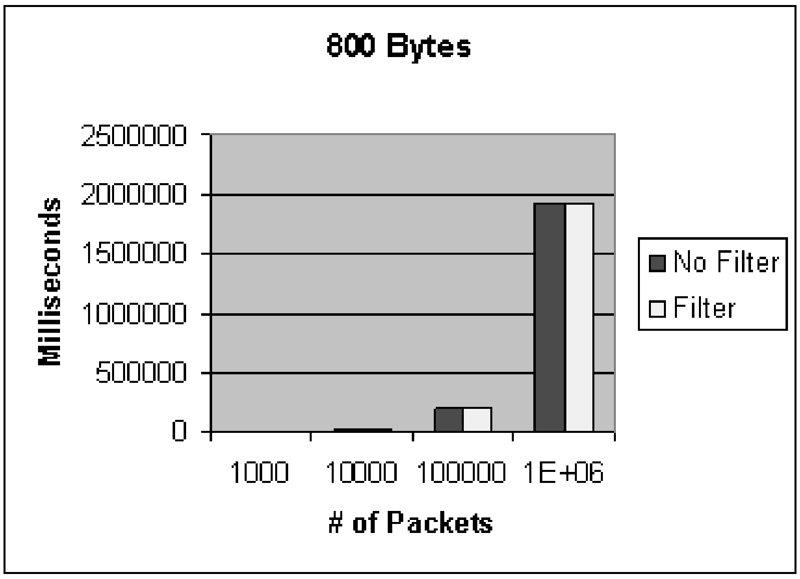

Figure 6.8: # of 800-byte Packets vs. Time.

# of Packets	1,000	10,000	100,000	1,000,000
(reg) Milliseconds	1,926	19,246	192,459	1,924,456
(netfilter) Milliseconds	1,926	19,276	192,797	1,927,430

Figure 6.9: Table of Times as # of 800-byte Packets Grows.

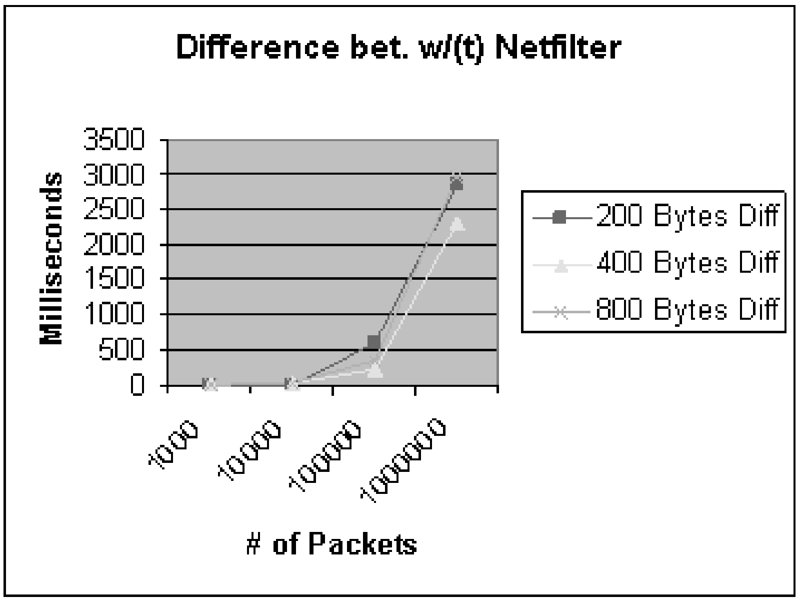

Figure 6.10: Difference in Times between Reg and Netfilter vs.# of Packets of 3 sizes

6.2.2 Rules in the NAT Table

This section investigates the effect of Netfilter as its rule table grows. It is hypothesized that as the number of rules in the NAT table increases, the time it takes to find the rule that matches with some header information of a TCP packet also increases. More specifically, if there is one rule in the Netfilter table, and the time it takes for Netfilter to route a packet that matches with the rule is t, then the time it takes to match that rule in a table of N rules will be $O(N * t)$.

Setup and Procedure

The experiment setup is as before, in which a client passes packets to a server and the server passes the same packets back. In a Netfilter-enabled scenario, a client connects to the server's public address and port. Packets are routed by Netfilter to a virtualized IP address where the server is listening on connections and to which its port is bound. Figure 6.11 demonstrates the scenario.

To measure the Netfilter overhead, the time it takes for packets that have left the client to arrive back at the client is recorded. Although the propagation time is included between the client and server in the measurement, it is negligible due to distance. The client and server machines, both use 3 COM 3c900 10/100 MBPS Ethernet cards, are set up in such a way that their distance was about 0.60 m apart. The switch connecting both the

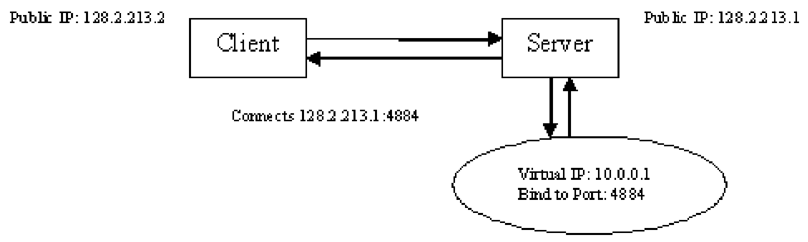

Figure 6.11: Experiment 2 Setup.

client and server is a 10/100 MBPS switch.

The experiment contains 3 steps: creating a virtualized IP, adding a rule to "invoke" Netfilter, and starting up the server and client program.

To create a virtualized IP, one does:

```
ifconfig eth0:1 10.0.0.1 netmask 255.255.255.0
```

Where "eth0" is the primary Ethernet interface and "10.0.0.1" is the virtualized IP address.

To add a rule to Netfilter, one does:

```
iptables -t nat -D PREROUTING -p tcp -I eth0 -j DNAT -to 10.0.0.1
```

This tells Netfilter that arrival TCP packets with destination address will be routed or "NATed" to 10.0.0.1. One can verify that such a rule is added by issuing:

```
iptables -t nat -n -L
```

Where "-t nat" specifies the NAT table that iptables command should operate on . "-n" prints IP addresses and ports in numeric format. "-L" lists all rules in a selected chains. If no chain is specified, all chains are listed. In this case, all rules of all chains of NAT table were listed. A sample output looks like the following:

```
Chain PREROUTING (policy ACCEPT) target     prot opt source
destination DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0
to:10.0.0.1

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

50 rules are added to the NAT table in Netfilter. Out of the 50 rules added to the table, only one rule is matched with incoming packets. Mismatched rules fall into 3 categories. Rules in the first category direct an incoming port to an outgoing port. They specify how an incoming port should be mapped to to an outgoing port. In this category, rules are intentionally made with ports that do not match port 4884. Rules in the second category specify a destination address that does not match with the address of any incoming packet. Rules in this category are intentionally made with addresses that are incorrect so as to disqualify them. Rules in the third category change the source address of a packet to an address of different port as packets go out to the wire. The third category affects Netfilter in a different chain from the first and second categories. Rules in this category should be ignored since they only affect packets that go out of the receiving host.

The following code shows a sample of all 3 categories:

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source          destination
[first category]
DNAT       tcp  --  0.0.0.0/0       0.0.0.0/0     tcp dpt:2000 to:10.0.0.1:2000
DNAT       tcp  --  0.0.0.0/0       0.0.0.0/0     tcp dpt:3000 to:10.0.0.1:3000
...
[second category]
DNAT       tcp  --  0.0.0.0/0       128.2.213.2   to:10.0.0.1:2000
DNAT       tcp  --  0.0.0.0/0       128.2.213.2   to:10.0.0.1:3000
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source          destination
[third category]
SNAT       tcp  --  0.0.0.0/0       0.0.0.0/0     to:10.0.0.2:2000
SNAT       tcp  --  0.0.0.0/0       0.0.0.0/0     to:10.0.0.2:3000
..
```

Example commands to issue those 3 types of rules, in addition to the correct rule are listed as follows:

```
First Type:
iptables -t nat -A PREROUTING -p tcp --dport 45000 -i eth0 -j DNAT
--to 10.0.0.1:25000

Second Type: iptables -t nat -A PREROUTING -p tcp -d 128.2.213.2
-i eth0 -j DNAT --to 10.0.0.1:26000

Third Type: iptables -t nat -A POSTROUTING -p tcp -o eth0 -j SNAT
--to 10.0.0.2:7000
```

```
Correct Type: iptables -t nat -A PREROUTING -p tcp -i eth0 -j DNAT
--to 10.0.0.1:1000
```

The last step of the experiment is to fire off server and client count programs. The server must be started by:

```
./server 10.0.0.1 1000 200
```

Where ”10.0.0.1” and ”1000” are the IP and port that the server is listening on. ”200” is the number of bytes the server is expected to receive. Following is the server usage:

```
./server <ipaddr> <port_num> <bytes>
```

To start the client in the regular scenario of this experiment, one would issue:

```
./client 128.2.213.1 1000 1000 200
```

Where ”128.2.213.1” and ”1000” are the IP and port that the client is connected to. The second ”1000” indicates the number of packets it is sending. And ”200” is the number of bytes per packet. Following is the client usage:

```
./client <ipaddr> <port_num> <num_packets> <bytes>
```

Results

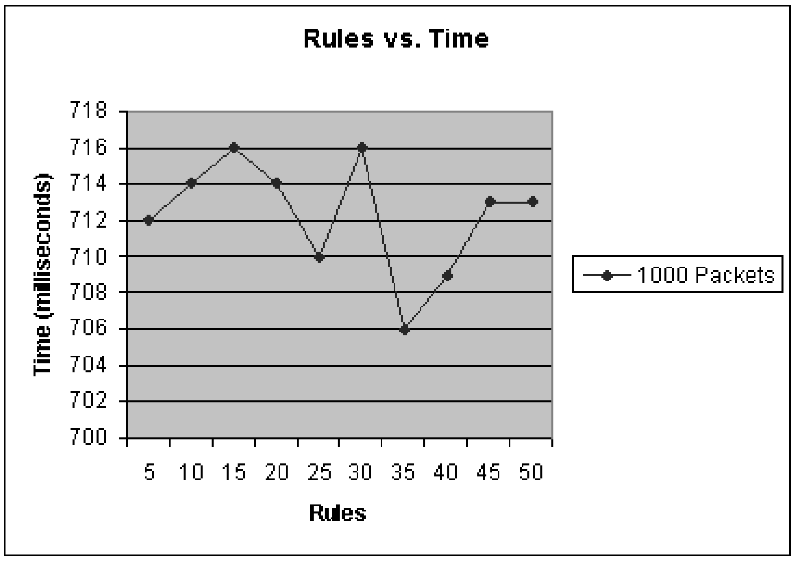

Figure 6.12: Rules vs. Time for 1,000 Packets.

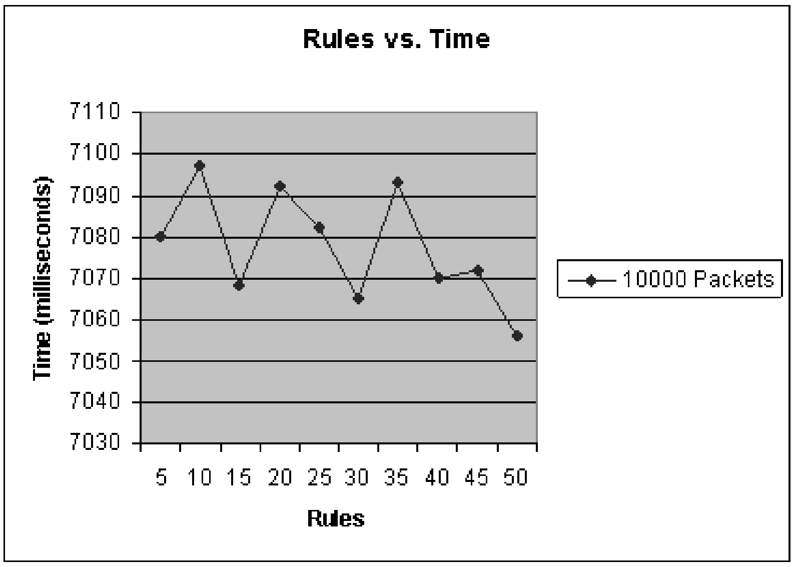

Figure 6.13: Rules vs. Time for 10,000 Packets.

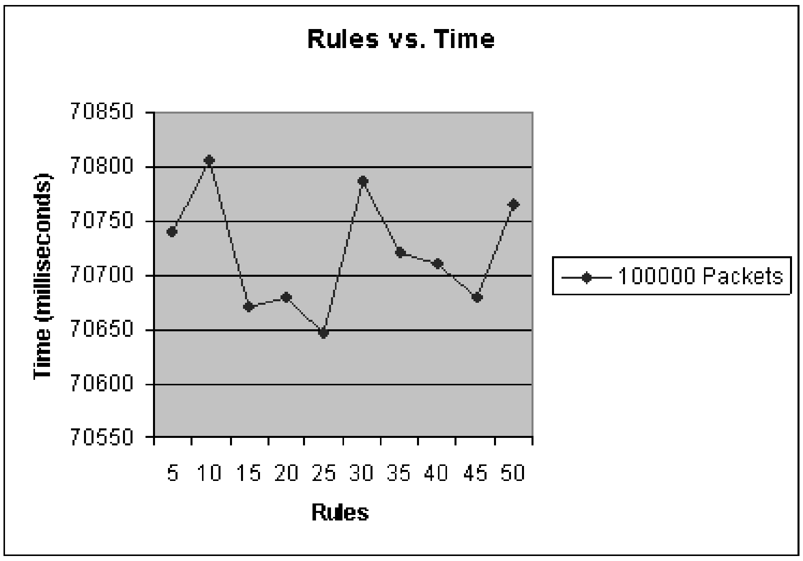

Figure 6.14: Rules vs. Time for 100,000 Packets.

Results were collected from sending 1,000, 10,000, 100,000 packets of size 200 bytes with rules varying from 1 to 50, with incremental of 5. Figures 6.12, 6.13, and 6.14 are graphs that show rules against time for 1,000, 10,000, and 100,000 packets, respectively.

Looking at these three graphs, one can see that time tends to fluctuate as the number of rules increases. In some instance, the increase in the number of rules actually corresponds to a decrease in time. However, from Figures 6.12 and 6.14, an increase in time does appear if one only looks at the beginning (5 rules) and ending points (50 rules). According to the documentation of Netfilter, rules are looked up sequentially until the right match is found. This leads to the conclusion that if there is any increase in time, the rule space must be fairly diverse and large. This experiment shows that Netfilter with 50 rules spanning two chains is resilient to stress of at least 100,000 200-byte packets. Although the experiment does not completely confirm the aforementioned hypotheses, it does bring out a practical value of Netfilter.

6.3 Overall Findings

Based on the two experiments conducted in Section 6.2, it is evident that Netfilter performs better than expected. Although the empirical data shows some overhead, it is minimal for large quantity of packets regardless of their sizes[1]. The overhead encountered for sending 100,000 packets is about 3 seconds. The number is obtained by subtracting 1,924,456ms from 1,927,430ms in Figure 6.9[2]. In a typical scenario where a web client accesses a web page from a web server, the server will fulfill the client's request in less than 100,000 packets. It means that the overhead of less than 3 seconds is anything but significant.

It is fairly scalable under data-intensive applications as well. Assuming a 5GB video streaming is delivered using UDP protocol to a process within a pod, it will take about 17 minutes to ensure the video stream be delivered safely and viewed properly[3]. Depending on the need of end-to-end applications, translation using Netfilter might be suitable.

MIGSOCK takes 7.65ms to perform a socket migration, which involves two messages. On average, Netfilter takes 0.0031 ms per packet[4]. In other

[1]It is because Netfilter only looks at the header of a packet

[2]Since the overhead of Netfilter is independent of packet sizes, one can obtain the overhead by looking at Figures 6.5 and 6.7 as well.

[3]$(5 * 10^9/1500/100,000 * 3/60)$ where 1500 is the size of an ethernet packet, 3 is the time it takes to process 100,000 packets, and 60 converts seconds into minutes.

[4](19276-19245.5)/10000 where 19,276 and 19,245.5 are obtained from Figure 6.9 and 10000 is the total number of packets

words, it takes 2,500 packets for MIGSOCK to break even with Zap[5]. In a typical scenario, one can readily agree that communication between a pod and its remote peer may require more than 2,500 packets. In those circumstances, it can be concluded that MIGSOCK is a better choice to socket migration.

Serving different types of services, represented by different rules by which each one is responsible for directing traffic to a pod in Zap, Netfilter has also shown its scalability. 50 rules are created and looked up by Netfilter in a relatively fast speed. However, one can get the lower number of 7.65ms using MIGSOCK than 712ms, according to Figure 6.12, using Zap. MIGSOCK can model a pod as a multi-threaded program. Each thread represents a process and shares the same virtual IP address and port with the rest of the threads. By iteratively checkpointing and restarting all multi-threaded programs, each pod can be migrated and resumed at a desirable host. Therefore, it is efficient to use MIGSOCK providing translation of packets to different pods as opposed to Netfilter. Assuming the time to deliver a packet to a pod is constant with respect to the number of pods, it takes about 19pods for Netfilter to break even with MIGSOCK[6]. Therefore, MIGSOCK wins on the front of serving multiple pods.

[5] $7.65/0.0031 \approx 2500$

[6] $712\text{ms}/5/7.65\text{ms} \approx 19$ where 712ms/5 is the time it takes to direct packets to a pod and 10ms is the time it takes to checkpoint and restart a pod in MIGSOCK.

Chapter 7

Future Works

There are several areas upon which MIGSOCK can improve. First, MIGSOCK is limited in providing complete user transparency. Since peer-to-peer communications are characterized by a socket acting as both a client and a server on the same connected socket, it is necessary to examine more closely the impact of migrating a socket as a process is writing to it. Another possibility for error that has not yet been experienced but should be examined more closely is the potential to lose data at the migrating socket. The current implementation puts the migrating process to sleep at the same time that the migrating socket sends out its ”start migrate” packet. It is possible that the non-migrating remote peer can send data to the migrating socket before the migrating socket is suspended but after the migrating process is suspended. At the TCP transport level, the transmission would appear to be successful, but in reality, the socket may never pass that data to the parent process. The future work is to investigate if packets during migration can be captured by perhaps creating buffers in `struct sock` structure and serialize it along with the rest of the `struct sock` variables.

Second, MIGSOCK is limited in its security features. The remote host is not aware of the identity of the hijacking user-level controlling program, which is responsible for putting the remote process to sleep. This means that anyone can spoof the identity of the controlling program and hijacks the connection. A malicious user can then signal the remote process to wake up and establish a connection. Any previous communication between the remote host and the migrating process can be deciphered by the malicious user. This kind of attack can be avoided by establishing an authentication protocol in which the controlling program has to authenticate itself to the remote host before the remote host can put the remote process to sleep or to signal it to wake up.

The third area is concerned with MIGSOCK's incompatibility with the other systems. For sockets to be successfully checkpointed and restarted, the source, destination, and remote host have to to run on the MIGSOCK kernel. It would be better if MIGSOCK can extend the capability of checkpointing and restarting of sockets to hosts that do not run on MIGSOCK kernel. There are two solutions to this. The first solution is to push the responsibility of the kernel to the the MIGSOCK module. Consequently, the module will be responsible for checking special MIGSOCK messages and signaling the remote process. This approach, however, requires both ends to run the controlling programs. It is necessary for the remote host to run the controlling program because it has to monitor the incoming special messages. The other disadvantage of this solution is that this inevitably means a performance hit for sockets' checkpointing and restarting. Rather than having the kernel process special messages, kernel module is given the responsibility to do so. The extra level of indirection makes MIGSOCK less efficient.

The other solution is to make modification of the TCP protocol a standard. The fields in a TCP packet would be reserved for MIGSOCK-related messages. Within the implementation of TCP in the kernel, special messages are checked and appropriate handling routines are added. Changes will be included in the official kernel. Although it does not have disadvantages of the first solution, this approach is not so well embraced by the Linux developers. As IPv6 standard emerges, it is believed that some mechanism is going to be developed to support socket checkpointing and restarting.

One of the limitations of MIGSOCK is that it only supports checkpointing and restarting of TCP sockets. The same recipe would apply to UDP sockets as TCP sockets hence it was left as a future work. Finally, MIGSOCK does not support checkpointing and restarting of listening sockets as documented in the original MIGSOCK report [5]. The reason behind it is that a listening socket does not yet have connections with a remote socket. States of such a socket can be trivially verified to ensure what is de-serialized is what is checkpointed. The project considers checkpointing a connected socket to be a more challenging problem as a successful communication with another host is re-established. However, future work covering checkpointing of a listening socket remains valid.

In testing the overhead of Netfilter with respect to rules, the hypothesis that the overhead increases as the number of rules increases is not successfully verified. The overhead fluctuates with the increased number of rules. Furthermore, the average overhead of one single rule is 0.142ms[1]. This is remarkably bigger than the Netfilter overhead of 0.0031ms/packet in the first experiment in 6.2.1. It leads one to believe that there are hidden variables when more than one rule is added to the Netfilter table. Future work in this area includes finding out the hidden variables that explain the sharp discrepancies. Furthermore, the experiment should be conducted with rules in the order of hundreds, spanning across more than two chains. Rules should

[1]712ms/1000/5 = 0.142 ms, where 1000 is number of packets and 5 is the number of rules in Figure 6.12.

also have greater diversities and complexities to reflect significant overhead from the Netfilter.

Chapter 8

Related Works

There are many approaches to network process checkpointing and restarting. This section describes some of the salient ones since the work done by the previous thesis work of Kuntz and Rajan [5].

8.1 Transport/Application Layer Approach: TCP Migration Options

An end-to-end approach to host mobility [14] argues that although mobility support at lower layer makes application programming easier, it comes at significant cost, complexity, and performance degradation. Furthermore, mobility modes such as TCP or UDP are constrained by lower-layer approaches. The paper [13] introduces the end-to-end approach to host mobility by utilizing DNS and developing TCP migration options with existing TCP protocol.

The DNS is needed to look up the address of a mobile host as it moves regularly. It is necessary since a correspondent host will be in the waiting state while the mobile host is migrating. The role of the correspondent host is not to actively seek for the mobile host; rather, it waits until the mobile finishes migration. Upon completion of migration, the mobile host will inform the correspondent host of its address. Therefore, DNS is not necessary though it is always good to know where the mobile host is. The paper talks about setting the time-to-live (TTL) field for the name mapping to zero to prevent stale mobile host name mapping entry.

TCP options are simply signals imbedded in TCP protocol to signal to end hosts whether a migration is underway. First, a TCP Migrate-Permitted option is sent along with TCP SYN segment to the correspondent host upon connection establishment. The mobile host information such as a verifiable token is saved for reconnection later. As the mobile host finishes migration, a TCP Migrate option is then sent to the correspondent host that will verify mobile host's identity, the last sequence number acked and resume the connection with the mobile host. During migration, the correspondent host will be in Migrate_Wait state in the new TCP protocol. So any future packets from the server will get dropped. The wait is 2MSL, which is assumed to be enough time for mobile host to finish migration.

The scheme proposed by this paper is similar to MIGSOCK in the sense that they are all end-to-end approaches. Furthermore, all make modifications on the TCP protocol. While implementation of synchronizing and resuming network states are the same, MIGSOCK go further to decouple the relationship between processes and mobile hosts. In other words, processes now can move among mobile hosts, which can move among different domains.

8.2 Application Layer Approach: Zap

"Zap is a virtualization layer on top of the operating system that introduces pod. A pod is a group of processes with a private namespace that presents the process group with the same virtualized view of the system."[9] Zap is responsible for translating system calls between a pod and its host operating system thus avoiding resource inconsistency, resource conflicts, and resource dependencies. For example, within a pod, a process has a virtual process ID. When such a process is checkpointed and restarted, the process has the same process ID since it lives within the pod. If there were no pod virtualization, the restarted process would not be able to be migrated over with the same process ID as it used to have. The PID that it had before might be occupied by some other process in the migrated host. The virtualization is important as it presents a true process migration method.

Network migration by Zap is clean in the sense that it differs from end-to-end, network-layer, transport layer, proxy-based split TCP connection, and socket library wrapper approaches seen so far. There is no modification of the end systems, the kernel, TCP, or transport layer protocol. Because of the fact everything is virtual, all processes within a pod communicate by a virtual IP address. The virtual IP address is then translated to an external IP address by Zap for communication between process in/outside of the pod. It preserves the established connection without perturbing semantics of application-layer, network-layer or below. To keep the connection alive during migration, a proxy is placed to stall current connection. However, the proxy is not involved as in the split TCP connection approach to synchronize and resume network states.

Implementation of translation between virtual and external IP addresses in Zap is done using Netfilter. It is a viable and by far the most promising one among all network migration approaches. In addition, the abstraction provided by pod decouples the dependency of processes on a particular host.

This is never addressed by all the other papers. They all assume that a process will stay on a mobile host.

Chapter 9

Conclusion

In conclusion, the thesis has clearly documented MIGSOCK's extended support for checkpointing and restarting multiple sockets for single-processed, multi-processed, and multi-threaded programs. The thesis has also detailed modifications of CRAK to support checkpointing and restarting of multi-processed and multi-threaded programs.

In evaluation, the paper demonstrates that MIGSOCK compares relatively well to Zap. For providing a single service, MIGSOCK breaks even with Zap beyond 2,500 messages. It is very likely this low threshold is reached given that a pod has many processes, and each of them is communicating with a remote peer. For providing multiple services where each service resides in a different pod, it takes about 19 pods for Zap to break even with MIGSOCK assuming that Zap's overhead in providing forwarding service to a pod is constant to the number of pods. The bottleneck of Zap happens at translating packets and directing them to a pod. This overhead is inherited and unavoidable regardless what underlying translation mechanism Zap adopts. For Zap to break this barrier, it means shifting the responsibility of translation to each pod. Although transparency and virtualization requirements are violated, they are not completely broken. Given the complexity of implementing a kernel-level solution, a hybrid system of MIGSOCK and Zap suggests a viable approach to network process checkpointing and restarting.

Appendix A

Kernel Data Structures

A.1 inet_stream_ops

```
struct proto_ops inet_stream_ops = {
  family:         PF_INET,

  release:        inet_release,
  bind:           inet_bind,
  connect:        inet_stream_connect,
  socketpair:     sock_no_socketpair,
  accept:         inet_accept,
  getname:        inet_getname,
  poll:           tcp_poll,
  ioctl:          inet_ioctl,
  listen:         inet_listen,
  shutdown:       inet_shutdown,
  setsockopt:     inet_setsockopt,
  getsockopt:     inet_getsockopt,
  sendmsg:        inet_sendmsg,
  recvmsg:        inet_recvmsg,
  mmap:           sock_no_mmap,
  sendpage:       tcp_sendpage
};
```

A.2 inet_dgram_ops

```
struct proto_ops inet_dgram_ops = {
  family:         PF_INET,

  release:        inet_release,
  bind:           inet_bind,
  connect:        inet_dgram_connect,
  socketpair:     sock_no_socketpair,
  accept:         sock_no_accept,
  getname:        inet_getname,
  poll:           datagram_poll,
  ioctl:          inet_ioctl,
  listen:         sock_no_listen,
  shutdown:       inet_shutdown,
  setsockopt:     inet_setsockopt,
  getsockopt:     inet_getsockopt,
  sendmsg:        inet_sendmsg,
  recvmsg:        inet_recvmsg,
  mmap:           sock_no_mmap,
  sendpage:       sock_no_sendpage,
};
```

A.3 tcp_prot

```
struct proto tcp_prot = {
  name:           "TCP",
  close:          tcp_close,
  connect:        tcp_v4_connect,
  disconnect:     tcp_disconnect,
  accept:         tcp_accept,
  ioctl:          tcp_ioctl,
  init:           tcp_v4_init_sock,
  destroy:        tcp_v4_destroy_sock,
  shutdown:       tcp_shutdown,
  setsockopt:     tcp_setsockopt,
  getsockopt:     tcp_getsockopt,
  sendmsg:        tcp_sendmsg,
  recvmsg:        tcp_recvmsg,
  backlog_rcv:    tcp_v4_do_rcv,
  hash:           tcp_v4_hash,
  unhash:         tcp_unhash,
  get_port:       tcp_v4_get_port,
};
```

A.4 udp_prot

```
struct proto udp_prot = {
  name:           "UDP",
  close:          udp_close,
  connect:        udp_connect,
  disconnect:     udp_disconnect,
  ioctl:          udp_ioctl,
  setsockopt:     ip_setsockopt,
  getsockopt:     ip_getsockopt,
  sendmsg:        udp_sendmsg,
  recvmsg:        udp_recvmsg,
  backlog_rcv:    udp_queue_rcv_skb,
  hash:           udp_v4_hash,
  unhash:         udp_v4_unhash,
  get_port:       udp_v4_get_port,
};
```

Appendix B

User Controlling Program

B.1 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 from 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 restart
 * from 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 <sys/time.h>  /* for gettimeofday */
#include "migsock_user.h"

int main(int argc, char *argv[]) {
  int fd; /* file descriptor for /dev/migsock */
  int ret; /* return val */
  /* a socket counter and number of sockets defined from the cmd arg */
  int i, num_sockets;
  char filename[15];  /* name for the file where data is serialized */
  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; /* file descriptor for serialized data */
```

```
struct timeval start_time, stop_time;

ret = -1;
if (argc < 3 || argc > 4)
  {
    printf ("Usage : user <pid> <sockfd> [sockfd2]\n");
    goto done;
  }

arg->pid = atoi(argv[1]);
arg->sockfd[0] = atoi(argv[2]);

/* default number of sockets is 1 */
num_sockets = 1;

/* we give users the choice of using one or two sockets */
if (argc == 4) {
  arg->sockfd[1] = atoi(argv[3]);
  num_sockets = 2;
}

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

/* now open the migsock device */
fd = open("/dev/migsock", O_RDONLY);
printf("FD = %d\n", fd);
if (fd < 0) {
  printf("Bad FD.\n");
  goto done;
}

/* now send migration request and serialize data for each socket */
for(i = 0; i < num_sockets; ++i) {
  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));

  arg->flag = i;

  /* 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 %d.\n", ret);
      goto done;
    }

    /* start timing how long it takes to serialize */
    gettimeofday(&start_time, NULL);
    /* 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");
      goto done;
    }

    sprintf(filename, "fdata%d.bin", i+1);
    fout = open(filename, O_WRONLY | O_CREAT);
    if (fout == -1) {
      printf("%dth socket: Failed to create %s.
      Migration aborted\n", i + 1, filename);
      goto done;
    }

    /* write this serialized junk in arg->data to MIGSOCK_FILENAME */
    ret = write(fout, (void *)(arg->data), MIGSOCK_SERIAL_DATA_SIZE);
    if (ret == -1) {
      printf("%dth socket: Failed to write to %s.
      Migration aborted\n", i + 1, filename);
      close(fout);
      goto done;
    }

    /* stop timing, end of serialization */
    gettimeofday(&stop_time, NULL);
    printf("start_migrate: Total time in millisecond %ld\n",
    stop_time.tv_usec/1000 + stop_time.tv_sec * 1000
    - start_time.tv_usec/1000 - start_time.tv_sec * 1000);
    printf("%d bytes of serialized data written to %s\n", ret, filename);

    close(fout);
  }
  printf("Migration initiated.\n");
 done:
  close(fd);
  return ret;
}
```

B.2 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 restart from CRAK. It
 * will reestabilsh connection to the remote host that was talking to the
 * start_migrate side.  First it wakes the process that CRAK restart 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 <sys/time.h>  /* for gettimeofday */
#include "migsock_user.h"

int main(int argc, char *argv[]) {
  int fd; /* file descriptor for /dev/migsock */
  int ret; /* return val */
  /* a socket counter and number of sockets defined from the cmd arg */
  int i, num_sockets;
  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; /* file descriptor for de-serialized data */
  char filename[15]; /* name for the file where data is serialized */
  struct timeval start_time, stop_time;

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

  arg->pid = atoi(argv[1]);
  arg->sockfd[0] = atoi(argv[2]);

  num_sockets = 1;
```

```
/* we give users the choice of using one or two sockets */
if (argc == 4) {
  arg->sockfd[1] = atoi(argv[3]);
  num_sockets = 2;
}


fd = open("/dev/migsock", O_RDONLY);
if (fd < 0) {
  printf("Bad FD.\n");
  goto done;
}

for(i = 0; i < num_sockets; ++i) {
  /* indicate which socket */
  arg->flag = i;
  /* start timing how long it takes to de-serialize
   * De-serialize timing includes:
   * 1) Reading from the serialized file.
   * 2) Doing actual de-serialization.
  */
  gettimeofday(&start_time, NULL);
  sprintf(filename, "fdata%d.bin", i+1);
  fin = open(filename, O_RDONLY);
  if (fin == -1) {
    printf("Failed to open %s.  Migration aborted\n", MIGSOCK_FILENAME);
    goto done;
  }

  /* read in file contents to fdata buffer */
  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);
  goto done;
    }
  close(fin);

  /* stop timing, end of serialization */
  gettimeofday(&stop_time, NULL);

  printf("finish_migrate: Total time in microsecond to deserialize %ld\n",
  stop_time.tv_usec + stop_time.tv_sec * 1000000
  - start_time.tv_usec - start_time.tv_sec * 1000000);
```

```
/* start timing how long it takes to wake up the remote process */
gettimeofday(&start_time, NULL);
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 %d.\n", ret);
  goto done;
}
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");
  goto done;
}
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");
  goto done;
}

/* stop timing, end of notifying the remote process to wake up */
gettimeofday(&stop_time, NULL);
printf("finish_migrate: Total time in millisecond to wake up
      the server %ld\n", stop_time.tv_usec/1000 + stop_time.tv_sec * 1000
      - start_time.tv_usec/1000 - start_time.tv_sec * 1000);
    }

    printf("end of migration\n");
    ret = 0;
   done:
    if(ret == -1)
      close(fin);
    close(fd);
    return ret;
  }
```

B.3 start_proc_migrate.c

```
/* start_proc_migrate.c
 *
 * This user program is run on a process and a socket that is connected
 * to a remote host before crak from 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_proc_migrate.c is called from the
 * new host and process, after the process has been migrated using restart
 * from CRAK.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/time.h>  /* for gettimeofday */
#include "migsock_user.h"

#include <dirent.h> /* for opendir */

#define PROC_ROOT_DIR "/proc" #define PROC_PROCESS_STAT_FILE
"stat"

struct pidlist {
  pid_t cur;
  int flag;  // to indicate whether it's a controlling thread
  struct pidlist *next;
};


pid_t get_ppid(pid_t pid) {
  char filen[256];
  int i, f, dummy, dummy2;
  char data[256], d[5], e[5];
  pid_t ppid;
  sprintf(filen, PROC_ROOT_DIR "/%d/" PROC_PROCESS_STAT_FILE, pid);
  f = open(filen, O_RDONLY);

  if (f < 0) {
    printf("can't open file %s", filen);
    return -1;
  }
  read(f, data, sizeof(data));
  sscanf(data, "%d %s %s %d %d %*s", &dummy, d, e, &dummy2, &ppid);
  close(f);
  return ppid;
}

// read all the pids from /proc
// get pid and its threads
int get_threads (struct pidlist *plist, const pid_t ppid) {
  int pid;
  struct dirent *this_entry;
  char path[50];
  struct stat status;
  struct pidlist *id;
  int flag;  // to mark the controlling thread
  // opendir
  DIR* p_dir = opendir(PROC_ROOT_DIR);
  flag = 0;
  if (!p_dir) {
    perror("Can't open "PROC_ROOT_DIR);
    return -1;
  }

  while ((this_entry = readdir(p_dir)) != NULL) {
    pid = atoi(this_entry->d_name);
    if (pid <= 0)
      continue;

    strcpy(path, PROC_ROOT_DIR);
    strcat(path, "/");
    strcat(path, this_entry->d_name);

    if (stat(path, &status) != 0) {
      printf("Failed getting stat from %s", this_entry);
      continue;
    }
    else if (S_ISDIR(status.st_mode) && (get_ppid(pid) == ppid) && pid != ppid) {
      id = malloc(sizeof(struct pidlist));
      id->cur = pid;
      if(flag == 0) {
        // this is used to signal the controlling thread
        id->flag = 1;
        flag = 1;
      }
      else {
        id->flag == 0;
      }
      plist->next = (struct pidlist *) id;
      plist = plist->next;
    }
  }
  closedir(p_dir);
  return 0;

}

int main(int argc, char *argv[]) {
  int fd; /* file descriptor for /dev/migsock */
  int ret; /* return val */
  int i; /* a socket counter */
  char filename[15];  /* name for the file where data is serialized */
  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; /* file descriptor for serialized data */

  int parent;
  struct pidlist pid_list, *pid_ptr;
  int pid = 0;

  ret = -1;
  if (argc != 7)
    {
      printf ("Usage : user <pid> <sockfd> <sockfd2> <sockfd3>
      <sockfd4> <sockfd5> <sockfd6>\n");
      goto done;
    }

  arg->pid = atoi(argv[1]);
  arg->sockfd[0] = atoi(argv[2]);
  arg->sockfd[1] = atoi(argv[3]);
  arg->sockfd[2] = atoi(argv[4]);
  arg->sockfd[3] = atoi(argv[5]);
  arg->sockfd[4] = atoi(argv[6]);

  pid_list.cur = atoi(argv[1]);
  pid_list.flag = 0;
  parent = atoi(argv[1]);
  get_threads(&pid_list, pid_list.cur);
```

```
// the list contains the parent process as well
for(pid_ptr = &pid_list; pid_ptr != NULL; pid_ptr = pid_ptr -> next) {
  if(pid_ptr->cur != 0) {// && pid_ptr->cur != parent ) {
    printf("pid %d\n", pid_ptr -> cur);

    if(kill(pid_ptr->cur, SIGSTOP) == -1) {
  printf("Couldn't Suspend Process %d.  Aborting migration.\n", pid_ptr->cur);
  goto done;
    }

  }
}

/* now open the migsock device */
fd = open("/dev/migsock", O_RDONLY);
printf("FD = %d\n", fd);
if (fd < 0) {
  printf("Bad FD.\n");
  goto done;
}

i = 0;
/* now send migration request and serialize data for each socket */
for(pid_ptr = &pid_list; pid_ptr != NULL; pid_ptr = pid_ptr -> next) {
  // eliminate the parent process; there is no controlling thread to eliminate
  if(pid_ptr->cur != 0 && pid_ptr->cur != parent) {
    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));

    arg->flag = i;

    /* 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 %d.\n", ret);
  goto done;
    }

    /* 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");
  goto done;
    }
```

```
      sprintf(filename, "fdata%d.bin", ++i);
      fout = open(filename, O_WRONLY | O_CREAT);
      if (fout == -1) {
    printf("%dth socket: Failed to create %s.
    Migration aborted\n", i, filename);
    goto done;
      }

      /* write this serialized junk in arg->data to MIGSOCK_FILENAME */
      ret = write(fout, (void *)(arg->data), MIGSOCK_SERIAL_DATA_SIZE);
      if (ret == -1) {
    printf("%dth socket: Failed to write to %s.
    Migration aborted\n", i, filename);
    close(fout);
    goto done;
      }

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

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

B.4 finish_proc_migrate.c

```
/* finish_proc_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 restart from CRAK. It
 * will reestabilsh connection to the remote host that was talking to the
 * start_migrate side.  First it wakes the process that CRAK restart 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 <sys/time.h>  /* for gettimeofday */
#include "migsock_user.h"
#include <sys/stat.h> /* for stat */
#include <dirent.h> /* for opendir */
#include <signal.h> /* for SIGCONT */
#define PROC_ROOT_DIR "/proc"
#define PROC_PROCESS_STAT_FILE "stat"

struct pidlist {
  pid_t cur;
  int flag;  // to indicate whether it's a controlling thread
  struct pidlist *next;
};


pid_t get_ppid(pid_t pid) {
  char filen[256];
  int i, f, dummy, dummy2;
  char data[256], d[5], e[5];
  pid_t ppid;
  sprintf(filen, PROC_ROOT_DIR "/%d/" PROC_PROCESS_STAT_FILE, pid);
  f = open(filen, O_RDONLY);

  if (f < 0) {
    printf("can't open file %s", filen);
    return -1;
  }
  read(f, data, sizeof(data));
  sscanf(data, "%d %s %s %d %*s", &dummy, d, e, &ppid, &dummy2);
  close(f);
  return ppid;
}

// read all the pids from /proc
// get pid and its threads
int get_threads (struct pidlist *plist, const pid_t ppid) {
  int pid;
  struct dirent *this_entry;
  char path[50];
  struct stat status;
  struct pidlist *id;
  int flag;  // to mark the controlling thread
  // opendir
  DIR* p_dir = opendir(PROC_ROOT_DIR);
  if (!p_dir) {
    perror("Can't open "PROC_ROOT_DIR);
    return -1;
  }
  flag = 0;
  while ((this_entry = readdir(p_dir)) != NULL) {
    pid = atoi(this_entry->d_name);
    if (pid <= 0)
      continue;

    strcpy(path, PROC_ROOT_DIR);
    strcat(path, "/");
    strcat(path, this_entry->d_name);

    if (stat(path, &status) != 0) {
      printf("Failed getting stat from %s", this_entry);
      continue;
    }
    else if (S_ISDIR(status.st_mode) && (get_ppid(pid) == ppid) && pid != ppid) {
      id = malloc(sizeof(struct pidlist));
      id->cur = pid;
      id->flag = 0;
      if(flag == 0) {
        id->flag = 1;
        flag = 1;
      }
      plist->next = (struct pidlist *) id;
      plist = plist->next;
    }
  }
  closedir(p_dir);
  return 0;

}

int main(int argc, char *argv[]) {
  int fd; /* file descriptor for /dev/migsock */
  int ret; /* return val */
  int i; /* a socket counter */
  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; /* file descriptor for de-serialized data */
  char filename[15]; /* name for the file where data is serialized */
  int parent;
  struct pidlist pid_list, *pid_ptr;

  ret = -1;
  if (argc != 7)
    {
      printf ("Usage : user2 <pid> <sockfd> <sockfd2>
      <sockfd3> <sockfd4> <sockfd5>\n");
      goto done;
    }

  arg->sockfd[0] = atoi(argv[2]);
  arg->sockfd[1] = atoi(argv[3]);
  arg->sockfd[2] = atoi(argv[4]);
  arg->sockfd[3] = atoi(argv[5]);
  arg->sockfd[4] = atoi(argv[6]);

  pid_list.cur = atoi(argv[1]);
  parent = atoi(argv[1]);
  get_threads(&pid_list, pid_list.cur);

  fd = open("/dev/migsock", O_RDONLY);
  if (fd < 0) {
    printf("Bad FD.\n");
    goto done;
  }

  i = 0;
  // the list contains the parent process as well
  for(pid_ptr = &pid_list; pid_ptr != NULL; pid_ptr = pid_ptr -> next) {
```

```
// eliminate the parent process; there is no controlling thread to eliminate
if(pid_ptr->cur != 0 && pid_ptr->cur != parent) {
  arg->pid = pid_ptr->cur;
  arg->flag = i;

  sprintf(filename, "fdata%d.bin", ++i);

  fin = open(filename, O_RDONLY);

  if (fin == -1) {
printf("Failed to open %s.  Migration aborted.\n", filename);
goto done;
  }
  /* 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");
  goto done;
}

  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 %d.\n", ret);
goto done;
  }
  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");
goto done;
  }
  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");
          goto done;
        }
        close(fin);
      }
    }
    printf("end of migration\n");
    ret = 0;
   done:
    if(ret < 0)
      close(fin);
    close(fd);
    return ret;
  }
```

B.5 start_thread_migrate.c

```
/* start_thread_migrate.c
 *
 * This user program is run on a process and a socket that is connected
 * to a remote host before crak from 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_thread_migrate.c is called from the
 * new host and process, after the process has been migrated using restart
 * from CRAK.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/time.h>  /* for gettimeofday */
#include "migsock_user.h"

#include <dirent.h> /* for opendir */

#define PROC_ROOT_DIR "/proc"
#define PROC_PROCESS_STAT_FILE "stat"

struct pidlist {
  pid_t cur;
  int flag;  // to indicate whether it's a controlling thread
  struct pidlist *next;
};


pid_t get_ppid(pid_t pid) {
  char filen[256];
  int i, f, dummy, dummy2;
  char data[256], d[5], e[5];
  pid_t ppid;
  sprintf(filen, PROC_ROOT_DIR "/%d/" PROC_PROCESS_STAT_FILE, pid);
  f = open(filen, O_RDONLY);

  if (f < 0) {
    printf("can't open file %s", filen);
    return -1;
  }
  read(f, data, sizeof(data));
  sscanf(data, "%d %s %s %d %d %*s", &dummy, d, e, &dummy2, &ppid);
  close(f);
  //printf("get ppid %d\n", ppid);
  return ppid;
}

// read all the pids from /proc
// get pid and its threads
int get_threads (struct pidlist *plist, const pid_t ppid) {
  int pid;
  struct dirent *this_entry;
  char path[50];
  struct stat status;
  struct pidlist *id;
  int flag;  // to mark the controlling thread
  // opendir
  DIR* p_dir = opendir(PROC_ROOT_DIR);
  flag = 0;
  if (!p_dir) {
    perror("Can't open "PROC_ROOT_DIR);
    return -1;
  }

  while ((this_entry = readdir(p_dir)) != NULL) {
    pid = atoi(this_entry->d_name);
    if (pid <= 0)
      continue;

    strcpy(path, PROC_ROOT_DIR);
    strcat(path, "/");
    strcat(path, this_entry->d_name);

    if (stat(path, &status) != 0) {
      printf("Failed getting stat from %s", this_entry);
      continue;
    }
    else if (S_ISDIR(status.st_mode) && (get_ppid(pid) == ppid) && pid != ppid) {
      //printf("getchildren %d\n", pid);
      id = malloc(sizeof(struct pidlist));
      id->cur = pid;
      if(flag == 0) {
    // this is used to signal the controlling thread
    id->flag = 1;
    flag = 1;
```

```
        }
        else {
      id->flag == 0;
        }
        plist->next = (struct pidlist *) id;
        plist = plist->next;
      }
    }
    closedir(p_dir);
    return 0;

  }

  int main(int argc, char *argv[]) {
    int fd; /* file descriptor for /dev/migsock */
    int ret; /* return val */
    int i; /* a socket counter */
    char filename[15];  /* name for the file where data is serialized */
    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; /* file descriptor for serialized data */

    int parent;
    struct pidlist pid_list, *pid_ptr;
    int pid = 0;

    ret = -1;
    if (argc != 7)
      {
        printf ("Usage : user <pid> <sockfd> <sockfd2>
        <sockfd3> <sockfd4> <sockfd5> <sockfd6>\n");
        goto done;
      }

    arg->pid = atoi(argv[1]);
    arg->sockfd[0] = atoi(argv[2]);
    arg->sockfd[1] = atoi(argv[3]);
    arg->sockfd[2] = atoi(argv[4]);
    arg->sockfd[3] = atoi(argv[5]);
    arg->sockfd[4] = atoi(argv[6]);

    pid_list.cur = atoi(argv[1]);
    pid_list.flag = 0;
    parent = atoi(argv[1]);
```

```
get_threads(&pid_list, pid_list.cur);

// the list contains the parent process as well
for(pid_ptr = &pid_list; pid_ptr != NULL; pid_ptr = pid_ptr -> next) {
  if(pid_ptr->cur != 0) {
    printf("pid %d\n", pid_ptr -> cur);

    // stop the entire thread family
    if(kill(pid_ptr->cur, SIGSTOP) == -1) {
  printf("Couldn't Suspend Process %d.  Aborting migration.\n", pid_ptr->cur);
  goto done;
    }

  }
}


// now open the migsock device
fd = open("/dev/migsock", O_RDONLY);
printf("FD = %d\n", fd);
if (fd < 0) {
  printf("Bad FD.\n");
  goto done;
}

i = 0;
// now send migration request and serialize data for each socket
for(pid_ptr = &pid_list; pid_ptr != NULL; pid_ptr = pid_ptr -> next) {
  // eliminate the parent process and the controlling program
  if(pid_ptr->cur != 0 && pid_ptr->flag == 0 && pid_ptr->cur != parent) {
    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));

    arg->flag = i;

    // 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 %d.\n", ret);
  goto done;
    }

    // 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");
    goto done;
      }

      sprintf(filename, "fdata%d.bin", ++i);
      fout = open(filename, O_WRONLY | O_CREAT);
      if (fout == -1) {
    printf("%dth socket: Failed to create %s.
    Migration aborted\n", i, filename);
    ret = -1;
    goto done;
      }

      // write this serialized junk in arg->data to MIGSOCK_FILENAME
      ret = write(fout, (void *)(arg->data), MIGSOCK_SERIAL_DATA_SIZE);
      if (ret == -1) {
    printf("%dth socket: Failed to write to %s.
    Migration aborted\n", i, filename);
    close(fout);
    goto done;
      }

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

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

B.6 finish_thread_migrate.c

```
/* finish_thread_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 restart from CRAK. It
 * will reestabilsh connection to the remote host that was talking to the
 * start_thread_migrate side.  First it wakes the process that CRAK restart 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 <sys/time.h>  /* for gettimeofday */
#include "migsock_user.h"
#include <sys/stat.h> /* for stat */
#include <dirent.h> /* for opendir */
#include <signal.h> /* for SIGCONT */
#define PROC_ROOT_DIR "/proc"
#define PROC_PROCESS_STAT_FILE "stat"

struct pidlist {
  pid_t cur;
  int flag;  // to indicate whether it's a controlling thread
  struct pidlist *next;
};


pid_t get_ppid(pid_t pid) {
  char filen[256];
  int i, f, dummy, dummy2;
  char data[256], d[5], e[5];
  pid_t ppid;
  sprintf(filen, PROC_ROOT_DIR "/%d/" PROC_PROCESS_STAT_FILE, pid);
  f = open(filen, O_RDONLY);

  if (f < 0) {
    printf("can't open file %s", filen);
    return -1;
  }
  read(f, data, sizeof(data));
  sscanf(data, "%d %s %s %d %*s", &dummy, d, e, &ppid, &dummy2);
  close(f);
  //printf("get ppid %d\n", ppid);
  return ppid;
}

// read all the pids from /proc
// get pid and its threads
int get_threads (struct pidlist *plist, const pid_t ppid) {
  int pid;
  struct dirent *this_entry;
  char path[50];
  struct stat status;
  struct pidlist *id;
  int flag;  // to mark the controlling thread
  // opendir
  DIR* p_dir = opendir(PROC_ROOT_DIR);
  if (!p_dir) {
    perror("Can't open "PROC_ROOT_DIR);
    return -1;
  }
  flag = 0;
  while ((this_entry = readdir(p_dir)) != NULL) {
    pid = atoi(this_entry->d_name);
    if (pid <= 0)
      continue;

    strcpy(path, PROC_ROOT_DIR);
    strcat(path, "/");
    strcat(path, this_entry->d_name);

    if (stat(path, &status) != 0) {
      printf("Failed getting stat from %s", this_entry);
      continue;
    }
    else if (S_ISDIR(status.st_mode) && (get_ppid(pid) == ppid) && pid != ppid) {
      id = malloc(sizeof(struct pidlist));
      id->cur = pid;
      id->flag = 0;
      if(flag == 0) {
    id->flag = 1;
    flag = 1;
      }
      plist->next = (struct pidlist *) id;
      plist = plist->next;
    }
  }
  closedir(p_dir);
  return 0;

}

int main(int argc, char *argv[]) {
  int fd; /* file descriptor for /dev/migsock */
  int ret; /* return val */
  int i;/* a socket counter */
  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; /* file descriptor for de-serialized data */
  char filename[15]; /* name for the file where data is serialized */
  int parent;
  struct pidlist pid_list, *pid_ptr;

  ret = -1;
  if (argc != 7)
    {
      printf ("Usage : user2 <pid> <sockfd> <sockfd2>
      <sockfd3> <sockfd4> <sockfd5>\n");
      goto done;
    }

  arg->sockfd[0] = atoi(argv[2]);
  arg->sockfd[1] = atoi(argv[3]);
  arg->sockfd[2] = atoi(argv[4]);
  arg->sockfd[3] = atoi(argv[5]);
  arg->sockfd[4] = atoi(argv[6]);

  pid_list.cur = atoi(argv[1]);
  parent = atoi(argv[1]);
  get_threads(&pid_list, pid_list.cur);

  fd = open("/dev/migsock", O_RDONLY);
  if (fd < 0) {
    printf("Bad FD.\n");
    goto done;
  }

  i = 0;
  // the list contains the parent process as well
```

```
for(pid_ptr = &pid_list; pid_ptr != NULL; pid_ptr = pid_ptr -> next) {
  // eliminate the parent thread and the controlling thread
  if(pid_ptr->cur != 0 && pid_ptr->flag == 0 && pid_ptr->cur != parent) {
    arg->pid = pid_ptr->cur;
    arg->flag = i;

    sprintf(filename, "fdata%d.bin", ++i);

    fin = open(filename, O_RDONLY);

    if (fin == -1) {
  printf("Failed to open %s.  Migration aborted.\n", filename);
  goto done;
    }

    // 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");
    goto done;
  }

    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 %d.\n", ret);
  goto done;
    }
    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");
  goto done;
    }
    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");
      goto done;
        }
        close(fin);
      }
  }
  printf("end of migration\n");
  ret = 0;
 done:
  if(ret < 0)
    close(fin);
  close(fd);
  return ret;}
```

Bibliography

[1] Netfilter: Firewalling, NAT and packet mangling for Linux 2.4.

[2] L. A. Amnon. The mosix scalable cluster file systems for Linux.

[3] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner. *Linux Kernel Programming.* Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, 2002.

[4] J. Crowcroft and I. Phi. *TCP/IP & Linux Protocol Implementation: Systems Code for the Linux Internet.* John Wiley & Sons, first edition, 2001.

[5] B. Kuntz and K. Rajan. Migsock : Migratable TCP socket in Linux. Technical Report TR-2001-4, Carnegie Mellon University, Pittsburgh, PA. USA, 2002.

[6] A. Leon-Garcia and I. Widjaja. *Communication Networks.* McGraw-Hill Science/Engineering/Math, second edition, 2003.

[7] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. pages 1037–1045, 1998.

[8] T. S. Mitrovich, K. M. Ford, and N. Suri. Transparent redirection of network sockets. In *OOPSLA Workshop on Experiences with Autonomous Mobile Objects and Agent Based Systems.*

[9] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.

[10] C. E. Perkins and D. B. Johnson. Mobility support in ipv6. In *Mobile Computing and Networking*, pages 27–37, 1996.

[11] X. Qu, J. X. Yu, and R. P. Brent. A mobile TCP socket. Technical Report TR-CS-97-08, Australian National Unversity, Canberra 0200 ACT, Australia, 1997.

[12] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[13] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[14] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

[15] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: Highly available internet services using connection migration, 2002.

[16] H. Zhong and J. Nieh. Crak: Linux checkpoint/restart as a kernel module. *Technical Report CUCS-014-01, Columbia University*, November 2001.