UNIVERSITY OF OSLO
Department of Informatics

User Space Socket Migration for Mobile Applications

Master Thesis

Håvard Stigen Andersen

13th May 2012

User Space Socket Migration for Mobile Applications

Håvard Stigen Andersen

13th May 2012

Acknowledgement

First of all, I would like to thank my supervisors, Vera Goebel, Hans Vatne Hansen And Francisco Velázquez for their excellent guidance. Clear and concise feedback has made it possible for me to complete this thesis. Great supervision on the structure of the thesis and guidance on the proper writing style have helped a lot.

I would also like to thank my family and friends for their support. Especially a thanks to you those of who have helped me proofreading the thesis.

Håvard Stigen Andersen
University of Oslo
May 2012

Abstract

Nowadays, individuals are surrounded by several personal multimedia capable devices. This can leverage ubiquitous computing. Yet, in recent years, multimedia applications have increased their popularity and demand. These two factors have been the main motivation forces to retake process migration research. We focus on process migration to enable ubiquitous computing with multimedia application requirements, such as bandwidth and time constrains. We call applications designed for process migration *mobile applications*. This thesis addresses the connection mobility challenges in process migration between networked devices, while fulfilling multimedia applications requirements.

We present the design, implementation and evaluation of a user-space socket migration solution called SOCKMAND. SOCKMAND enables mobile applications to resume their connections on other remote nodes after a migration. The work is motivated by research on process migration for regular consumers within their own Migration Community, an overlay of personal devices. SOCKMAND supports legacy corresponding hosts, hosts which do not include any logic concerning the socket migration. This is achieved by introducing a *Migration Community Access Point* (MCAP). An MCAP acts as a proxy server between the two endpoints of a socket. SOCKMAND uses IP in UDP tunnels to transfer packets between the node with the mobile application and the MCAP. We utilize libpcap and raw sockets to achieve a user-space implementation. Libpcap and raw sockets can capture and send raw IP packets from user-space. TCP and UDP are implemented in user-space. UNIX domain sockets provide the inter-process communication between mobile applications and SOCKMAND.

We do our evaluation of SOCKMAND both by using analytical modeling as well as measurements on our implementation. The measurements are done on heterogeneous devices to determine if these devices are capable of running SOCKMAND with multimedia applications, like video conferencing.

Our evaluation shows that SOCKMAND is capable of utilizing the full bandwidth of various devices given a large enough packet size. We show that CPU load in MCAP and endpoints correlate to the number of packets per second, and not the bandwidth. This shows that application programmers should use larger packet sizes, when possible, to reduce CPU load. The round-trip time overhead introduced by Migration Community Access Points is negligible. SOCKMAND is able to support multimedia applications based on our requirements.

Contents

1 Introduction **17**
- 1.1 Background and Motivation . . . 17
- 1.2 Problem Statement . . . 18
- 1.3 Outline . . . 18

2 Background **19**
- 2.1 Mobile Applications . . . 19
- 2.2 The TRAMP Project . . . 20
  - 2.2.1 TRAMP Real-time Application Mobility Platform . . . 20
  - 2.2.2 Migration Communities . . . 21
  - 2.2.3 Real-Time Multimedia Applications . . . 21
- 2.3 Operating System Abstractions . . . 22
- 2.4 User Datagram Protocol . . . 24
- 2.5 Transmission Control Protocol . . . 24
- 2.6 IP Fragmentation . . . 25
- 2.7 Connection Handover . . . 26
- 2.8 Requirement Analysis of Background Knowledge . . . 26

3 Related Work **29**
- 3.1 Socket Migration . . . 29
  - 3.1.1 MIGSOCK . . . 29
  - 3.1.2 SockMi . . . 29
  - 3.1.3 Migratory TCP . . . 30
  - 3.1.4 Reliable Sockets . . . 30
  - 3.1.5 Socketless TCP . . . 30
- 3.2 Transport Layer Mobility . . . 31
  - 3.2.1 UPMT . . . 31
  - 3.2.2 TCP-R . . . 31
- 3.3 Emerald . . . 32
- 3.4 Analysis of Related Work . . . 32

4 Design **35**
- 4.1 Requirements . . . 35
- 4.2 SOCKMAND . . . 36
  - 4.2.1 Migration Community Access Point . . . 36
  - 4.2.2 IP in UDP Tunnels . . . 37
  - 4.2.3 SOCKMAND Architecture . . . 37

4.2.4 Different Packet Flows Through SOCKMAND 39
4.3 UUID of a Socket 39
4.4 Migrating a Socket 40
4.5 Message Passing During Migration 41
4.6 IP Fragmentation and SOCKMAND 42
4.7 Detailed Design 43
4.7.1 SOCKMAND API 44
4.7.2 SOCKMAND Library 45
4.7.3 UDP Controller 47
4.7.4 TCP Controller 47
4.7.5 IP Controller 47
4.7.6 SOCKMAND Core 48
4.7.7 Libpcap Handler 48
4.7.8 Rawsocket Handler 48
4.7.9 Tunnel Handler 49
4.7.10 Signal Handler 49
4.7.11 Migrator API 49

5 Implementation **51**
5.1 Environment 51
5.2 Overview 51
5.3 SOCKMAND components 51
5.3.1 Event Handler 52
5.3.2 SOCKMAND Core 53
5.3.3 SOCKMAND API 58
5.3.4 Libsockmand 58
5.3.5 UDP Controller 61
5.3.6 TCP Controller 65
5.3.7 Libpcap Handler 67
5.3.8 Migrator API 68

6 Evaluation **69**
6.1 Evaluation Goals 69
6.2 Analysis Approach 70
6.3 Evaluation Metrics 71
6.3.1 CPU Load 71
6.3.2 Round-Trip Time 72
6.3.3 Socket Migration Time 72
6.3.4 Packet Loss in the MCAP 72
6.4 Evaluation Factors 72
6.4.1 Packet Sizes 72
6.4.2 Packets per Second 73
6.4.3 Node Specifications 73
6.4.4 Packet Direction Through the MCAP 73
6.5 Evaluation Setups 74
6.5.1 CPU Load of SOCKMAND and Mobile Application . . 74
6.5.2 CPU Load and Packet Loss on MCAP 76
6.5.3 Socket Migration Time 77

6.5.4 Round-Trip Time 80
6.6 Results 80
6.6.1 CPU Load of SOCKMAND and Mobile Application . . 80
6.6.2 CPU Load and Packet Loss on MCAP 82
6.6.3 Socket Migration Time 85
6.6.4 Round-Trip Time 87
6.7 Discussion 89

7 Conclusion **93**
7.1 Contributions 93
7.2 Critical Assessment 94
7.3 Future Work 94
7.3.1 Short-term Goals 94
7.3.2 Long-term Goals 95

REFERENCES **95**

APPENDIX **100**

A Abbreviations **101**

B Performance Evaluation Results **103**

C Measurement Applications **107**
C.1 RTT Test Applications 107
C.2 Packetloss Test From Tunnel to Rawsocket 110
C.3 Packetloss Test From Libpcap to Tunnel 116
C.4 SOCKMAND and Mobile Application Test Applications 121

D Source Code **127**

List of Figures

2.1 TCP/IP encapsulation . . . 23
2.2 UDP Header [32] . . . 24
2.3 TCP Header [34] . . . 25
2.4 IPv4 Header [33] . . . 26

4.1 Contents of a tunnel packet . . . 37
4.2 System Architecture . . . 38
4.3 SOCKMAND Architecture . . . 38
4.4 Different packet flows through SOCKMAND . . . 40
4.5 Order of calls when an application is migrated . . . 41
4.6 Message passing when the source node is also the MCAP . . . 42
4.7 Message passing when the destination node is also the MCAP 43
4.8 Message passing when the MCAP is neither the source nor destination node . . . 44

6.1 Node setup CPU load by SOCKMAND and Mobile Application 75
6.2 Node setup CPU load and Packet loss on MCAP . . . 76
6.3 Measuring Socket Migration Time 1 . . . 77
6.4 Measuring Socket Migration Time 2 . . . 78
6.5 Measuring Socket Migration Time 3 . . . 79
6.6 Ping results . . . 80
6.7 CPU load by SOCKMAND and Mobile Application on Desktop Computer with 100B payload . . . 81
6.8 Desktop Computer Packet loss on MCAP from tunnel to rawsock . . . 82
6.9 Desktop Computer CPU Load on MCAP from tunnel to rawsock 83
6.10 Netbook Packet loss on MCAP from tunnel to rawsock . . . 84
6.11 Netbook CPU Load on MCAP from tunnel to rawsock . . . 84
6.12 Factors of RTT shown in Table 6.5 . . . 88
6.13 UPMT anchor node packet loss comparison [6] . . . 90
6.14 UPMT anchor node CPU load comparison [6] . . . 91

B.1 Desktop CPU Load on MCAP from libpcap to tunnel . . . 103
B.2 Desktop Packet loss on MCAP from libpcap to tunnel . . . 104
B.3 Netbook CPU Load on MCAP from libpcap to tunnel . . . 104
B.4 Netbook Packet loss on MCAP from libpcap to tunnel . . . 105
B.5 CPU load by SOCKMAND and mobile application on desktop computer with 1400B payload . . . 105

B.6 CPU load by SOCKMAND and mobile application on netbook with 100B payload 106
B.7 CPU load by SOCKMAND and mobile application on netbook with 1400B payload 106

List of Tables

2.1 Bandwidth requirements for Skype [35] 22
2.2 Bandwidth requirements for Google Hangout [11] 22

3.1 Comparison of Socket Migration Systems 32

6.1 Maximum packets per second based on packet size and bandwidth 73
6.2 Node Specifications 74
6.3 RTT between node A and B with the MCAP at node A 87
6.4 RTT between node A and B without SOCKMAND 87
6.5 RTT between node A and B with the MCAP at node C 88

Listings

5.1 Event Handling 52
5.2 Handle IP Controller Packet 53
5.3 Handle Tunneled Packet 54
5.4 Handle Tunneled Packet 54
5.5 Exporting Socket State 55
5.6 Importing Socket State 55
5.7 Migrate Socket 57
5.8 Sockets table 58
5.9 sm_socket( ) 59
5.10 sm_rebuildsocket( ) 60
5.11 sm_recvfrom( ) 61
5.12 UDP Control Block 62
5.13 UDP binding 62
5.14 UDP Sending 63
5.15 UDP Receiving 64
5.16 Delivering Buffered Packets 64
5.17 TCP Control Block 65
5.18 TCP Timeout Structure 66
5.19 TCP importing timeouts 66
5.20 Initializing libpcap 67
5.21 Dropping reset packets 68
C.1 Round-trip Time Client 107
C.2 Round-trip Time Server 109
C.3 Packetloss Test From Tunnel to Rawsocket Client 110
C.4 Packetloss Test From Tunnel to Rawsocket Server 114
C.5 Packetloss Test From Libpcap to Tunnel Client 116
C.6 Packetloss Test From Libpcap to Tunnel Server 119
C.7 SOCKMAND load client 121
C.8 SOCKMAND load server 124

Chapter 1

Introduction

1.1 Background and Motivation

Today, individuals are surrounded by several heterogeneous personal multimedia capable devices. This can make the ubiquitous computing paradigm shift possible. Recently, multimedia applications have increased their popularity and demand. These two factors have been our main motivation forces to focus our research on process migration. Process migration involves transferring a running process from one computer to another. We call applications designed for process migration *mobile applications*. We focus on process migration to enable ubiquitous computing with multimedia application requirements, such as bandwidth and time constraints. The field of process migration has been extensively studied over the last decades and several different approaches exist. However, process migration in a consumer setting has never been widely adopted [30].

There exist several scenarios where process migration is useful, such as accessing more processing power, exploitation of resource locality, resource sharing, fault resilience, system administration and mobile computing [30]. The scenario that is discussed in this thesis is mobile computing. This is user initiated process migration that enables users to migrate applications between their own heterogeneous personal devices. We claim that by enabling users to migrate their favorite multimedia application between their many personal devices will enhance the overall user experience.

There are several user centric scenarios that can benefit from process migration. A user may want to bring an application containing an ongoing video call from her desktop computer to her cellphone when leaving home. A real-time online game may be played on a cellphone and migrated to a desktop computer when arriving back home. Applications could be migrated to a technical support officer if there is a problem.

Although some applications increase its usability from process migration, other applications such as text-editors and graphical editors do not benefit as much from process migration. The main issue that applications benefits from process migration from a user perspective have in common is that they often are multimedia applications such as IP-telephony, video conferencing and video and music streaming applications. Such applications are increasing in popularity and demand, we claim that enabling support for process migration in such applications will further increase their popularity, and build public awareness of process migration. Milojicic [30] claim that a *killer application* is needed for process migration to become popular in the marketplace and we claim that a multimedia mobile application is that *killer application*.

A process on one node may use several resources bound to that particular node. Such resources may be a file system, peripheral devices, sockets and inter-process communication. When migrating a process, these resources must be accessible on the destination node in order for the process to have the same functionality as on the source node. Some resources, such as peripheral devices, are physically bound to the source node. Other resources, such as sockets, are logically bound to the source node due to the protocol they use. This thesis will focus on making sockets available on the destination node after a process migration.

1.2 Problem Statement

Applications, also mobile applications, can communicate with other applications over the Internet. Maintaining such connections after a process migration must be dealt with in a proper way. An endpoint of a transport layer connection is defined by an IP address and a port, together they are known as a socket. Migrating this socket along with the process is crucial for the process to maintain its connections after a migration. This is known as socket migration. The Internet does not natively provide any solutions to this problem, since it assumes that the two communicating applications never move to a new host.

This is a problem because mobile applications must be able to resume their connections to the remote applications they are connected to after they have migrated to a new node. If they are not able to do so, many of todays Internet centric multimedia applications will not be able to benefit from process migration.

1.3 Outline

The following parts of the thesis are organized as follows: Necessary background material and terminology are described in Chapter 2. Chapter 3 gives an overview of related work on the subject. Chapter 4 presents our requirements and design while Chapter 5 describes the implementation of our design. The implementation is evaluated against our requirements in Chapter 6 and finally our conclusions are drawn in Chapter 7.

Appendix A contains an abbreviations list. Appendix B shows additional performance evaluation results which supplement the results presented in Chapter 6. The source code of our meassurement applications are presented in Appendix C. Instructions on getting the source code of SOCKMAND and our measurement tools are presented in Appendix D.

Chapter 2

Background

In this chapter, we look at terminology and related technologies necessary as a basis to understand the rest of the thesis. Mobile applications in general are described in Section 2.1. The TRAMP project, which sets the context for our work is described in Section 2.2. Operating Systems and their abstractions are described in Section 2.3. Section 2.4 and 2.5 give a brief description of the two most common transport layer protocols. Useful terminology about handovers is described in Section 2.7. We summarize the requirements from this Chapter in Section 2.8.

2.1 Mobile Applications

A mobile application, not to be confused with an application designed for a cellphone, is in our context an application capable of migrating between nodes during execution.

The lifetime of a mobile application can be summarized as follows.

1. The application is started on a node 1.
2. The process is executed for some time
3. The running process is migrated from node 1, called source node, to another node 2, called destination node.
4. The two previous steps may be repeated infinitely until the application is terminated.

A process has a state that changes over time. This state includes dynamic data, the current state of the user interface, open file-descriptors and active network connections. When a process migrates, the state and the compiled code of the application must be transferred to the destination node. This state can be transferred in several different ways. It can be transferred directly from the old instance of the application to the new instance of the application through a TCP/ IP socket. Another alternative is to let the operating system extract the state of the application and send it to the operating system on the new node where it is inserted in the application. The last alternative is to send the state from the application through a supporting migrator system, elaborated later, which forwards it to the new node.

Regular operating systems and conventional programming languages do not support application mobility out of the box. To support mobile applications, the operating system needs to be changed, as in MOSIX, the programming language needs to be changed as in Emerald [14] or a combined solution where the application is designed to support mobility supported by a third application called a migrator system.

A migrator system organizes the migration of mobile applications. A migrator System is responsible for transferring the state and code, terminating the old instance of the process and starting the new process with the correct state on the destination node. If the migrator system runs in user-space, the state of the mobile application can only be accessed through inter-process communication (IPC) with the mobile application. The mobile application must expose an interface towards the migrator system where the state can be exported.

Mobility through a programming language provides programmers to migrate their application or parts of their application. Parts of the process, such as objects, may be moved within an overlay network of connected nodes. Although programming languages such as Emerald were primarily designed to provide fine grained mobility, it can also be utilized to migrate the entire process. Emerald provides fine grained mobility, which means that single objects can be migrated to another node. Emerald introduces a set of mobility related primitives which gives the programmer control over the location of objects. Examples of such primitives are *move*, *fix*, *locate* and *attach*.

Operating systems can be created or modified to support process migration. Since operating systems have full overview of processes and their data, implementing process migration in the operating system level decouples the logic of process migration from the application. A notable example of an operating system supporting process migration is MOSIX [3].

2.2 The TRAMP Project

TRAMP (TRAMP Real-time Application Mobility Platform) is a research project at the DMMS group at the Department of informatics at the University of Oslo. The project focuses on migration of real-time, user centric applications within a trusted migration community, described in Section 2.2.2.

2.2.1 TRAMP Real-time Application Mobility Platform

Unlike most of the previous systems designed for process migration, our platform TRAMP Real-time Application Mobility Platform, is implemented in user-space. The main reason for this is that the migration platform should support heterogeneous operating systems and devices. A user-space implementation also eases the installation of TRAMP. The platform organizes the migration of applications between trusted nodes.

Since the platform is not located in the kernel and the applications are running in user-space, the platform has no overview of the process stack, register values and address space. In traditional process migration, both the code and all the previous mentioned data are transferred during migration. The data needed to restart the process after migration is exported from the process to the migrator platform and transferred to the new node. This forces the applications to be migration-aware. The mobile applications are therefore designed to support this migration platform [30]. Even though the applications are designed to support the platform, they should also be able to run without a supporting migration platform.

Since we want our platform to be able to run in a regular consumer's home network, no special hardware equipment should be needed.

2.2.2 Migration Communities

TRAMP lets users migrate applications between nodes in their private migration community. The migration community can be organized in a peer-to-peer overlay network or in any other suitable fashion. For any migration to take place, the migration community must have at least two members. How the migration community is organized is out of scope for this thesis.

An instance of a migration community can include all of one user's devices, such as laptops, cellphones, tablets and desktop computers. The user is then able to migrate applications between all of these devices since they are members of the given migration community.

2.2.3 Real-Time Multimedia Applications

TRAMP is designed to support real-time multimedia applications. Such applications have specific requirements in terms of delay. Our use case is based on video conferencing. When using video conferencing, the highest acceptable end to end delay when using video conferencing is 100 ms [2]. If the end to end delay exceeds 100 ms, it will become noticeable for users.

In addition to strict requirements for delay, video conferencing also has bandwidth requirements. The bandwidth requirements depend on many factors such as audio codec, video codec and desired resolution. Although Skype is a closed protocol, they provide us with some minimum and required bandwidths for different scenarios. Table 2.1 shows minimum and recommended bandwidth for different scenarios when using Skype. From the table we see that the minimum download and upload speed for high quality video calling is 400 kbit/s while the recommended download and upload speed for an HD video call is 1.5 Mbit/s. We use this as our minimum and recommended bandwidth requirement. Table 2.2 shows minimum and recommended bandwidth for Google Hangouts. The bandwidth requirements for Google Hangouts are approximately the same as the bandwidth requirements for Skype.

Call type	Minimum download / upload speed	Recommended download / upload speed
Calling	30 kbit/s / 30 kbit/s	100 kbit/s / 100 kbit/s
Video calling / Screen sharing	128 kbit/s / 128 kbit/s	300 kbit/s / 300 kbit/s
Video calling (high-quality)	400 kbit/s / 400 kbit/s	500 kbit/s / 500 kbit/s
Video calling (HD)	1.2 Mbit/s / 1.2 Mbit/s	1.5 Mbit/s / 1.5 Mbit/s
Group video (3 people)	512 kbit/s / 128 kbit/s	2 Mbit/s / 512 kbit/s
Group video (5 people)	2 Mbit/s / 128 kbit/s	4 Mbit/s / 512 kbit/s
Group video (7+ people)	4 Mbit/s / 128 kbit/s	8 Mbit/s / 512 kbit/s

Table 2.1: Bandwidth requirements for Skype [35]

	Minimum bandwidth required	Ideal bandwidth for the best experience
Outbound from the participant	230 kbit/s	900 kbit/s
Inbound to the participant	380-500 kbit/s	1.2 Mbit/s

Table 2.2: Bandwidth requirements for Google Hangout [11]

2.3 Operating System Abstractions

An operating system (OS) provides applications with a set of APIs to underlying hardware and operating system services. The OS acts as an abstraction layer between software and hardware. These abstractions ease the use of hardware and OS services for application developers. Examples of such abstractions are file system management, memory access, inter-process communication (IPC) and sockets.

Applications access the file system through the OS. This simplifies the file access as it is not necessary for the applications to seek directly for the data on the hard-drive, but just ask the OS for a specific file. Memory is provided to applications through virtual memory. The OS provides a set of primitives for memory usage such as malloc(), bzero(), memcpy() and free(). IPC is provided to applications through pipes, signals, shared memory and sockets. The logic of communicating with other processes is hidden by the OS. The applications only use a set of simple primitives.

An operating system provides applications with the possibility of communicating with other remote or local applications using sockets. Applications are provided with a set of primitives that enables them to interact with the socket. These primitives are socket(), bind(), accept(), listen(), write(), read() and close(). Applications specify which transport protocol it wishes to use, and thereafter the OS kernel handles the details of this specific transport protocol, such as filling out necessary headers. In addition to handling the transport protocol, the OS, in cooperation with network hardware, handles the layers below the transport layer. Making changes to a transport protocol is therefore only possible if the OS kernel on both endpoints of a connection has been altered with the relevant changes to the protocol.

Applications only receive the payload of the transport layer protocol when receiving data from a socket, see Figure 2.1. The application is unaware of how the header of this protocol looks like, as well as how the headers of underlying protocols such as IP and Ethernet looks like. There are however solutions to receive the entire packet including the headers of

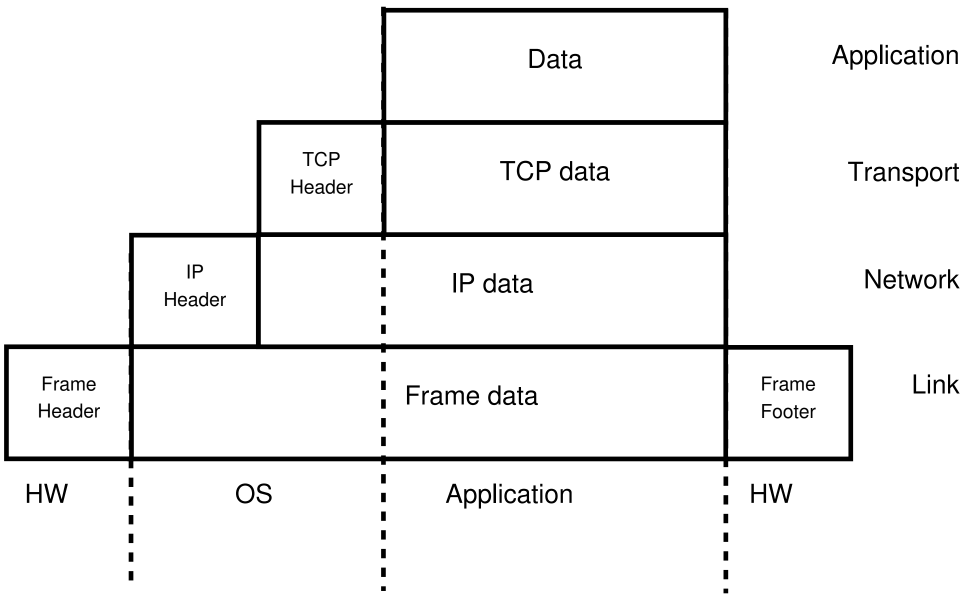

Figure 2.1: TCP/IP encapsulation

the different protocols. *Libpcap* is an OS independent library that enables packet capture to user-space applications.

Similarly, applications only write the payload of the transport layer packet to the OS. The OS is in charge of adding the correct headers to this packet before it is sent out on the network. There are solutions for user level applications to create their own custom network packets including their own headers on various operating systems, this is known as raw sockets. This is trivial on LINUX systems. Some Windows versions have however blocked this functionality due to media criticism that claim that raw sockets are a security hazard for the Internet [8][12].

OS Abstractions and Mobile Applications

The abstractions described above are designed to be used by conventional, non-mobile applications. Linux based operating systems, and other, are not designed with mobile applications in mind. Enabling mobile applications to utilize these abstractions on the destination host with the same expected behavior as on the source host is essential. If this is not possible, the state of the mobile application will not be the same on the destination host after a migration. If the application state is not the same, the migration must be seen as incomplete.

To support usage of the abstractions described above to mobile applications, the OS needs to be modified, or additional supporting software must be introduced. The abstractions are different by nature, and therefore must be solved in different ways.

Motivated by our previous work of creating a mobile Java application which handled the reconnecting of sockets in the application itself, we decide to work further with socket migration in an attempt to decouple this functionality from the application itself. The rest of this thesis will focus on how to enable mobile applications to use sockets.

2.4 User Datagram Protocol

User Datagram Protocol (UDP) is a connectionless transport layer protocol defined in RFC768[32]. The protocol enables applications to send datagrams to an IP address and a port. UDP is unreliable in the sense that the protocol itself does not guarantee that packets will arrive at the destination. UDP packets may also arrive in different order than they were sent.

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

Figure 2.2: UDP Header [32]

UDP packets include a simple header of 8 bytes. Figure 2.2 shows the UDP header. The header includes source port, destination port, packet length and a checksum field.

UDP is typically used for real time services such as video conferencing, IP telephony, streaming and online gaming.

2.5 Transmission Control Protocol

Transmission Control Protocol (TCP) is a transport layer protocol defined in RFC793 [34]. Unlike UDP, TCP is a connection oriented protocol.

With a TCP connection, data packets may be sent to an IP address and a port. The packets that are sent are guaranteed to be delivered to the application in the same order as they were sent. To ensure this reliability, packets must be acknowledged by the receiver. An acknowledgement may be sent in a packet containing no data. However, if it is a data packet waiting to be sent, the acknowledgement is piggybacked on that packet. Any packet that has not been acknowledged within a time calculated on the basis of the round trip time will be resent by the sender. The resend algorithm is defined by RFC6298 [31].

Figure 2.3 shows the TCP header. The TCP header is 20 bytes long if no options are used. Like the UDP header, it includes fields for source port, destination port and checksum. The TCP header however lacks the length field that is present in the UDP header. The length of the

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.3: TCP Header [34]

payload can however be computed by using the length field in the IP header and subtracting the TCP header length. The sequence number identifies which byte number in the stream that is sent in a given packet. The acknowledgement number specifies which byte number in the stream the sender expects to receive next. Several flag bits can be set. The most commonly used are ACK, SYN and FIN.

TCP connections are established using a 3-way handshake. If A want to connect to B, a packet with the SYN flag is set. B replies with SYN/ACK and finally A replies with ACK. In addition to the flags, the initial sequence numbers are exchanged.

Tearing down a connection can be done in several ways. The most common way is that A sends FIN/ACK to B which replies with an ACK. B then sends a FIN/ACK to A which then replies with an ACK.

TCP is typically used for web browsing, email, instant messaging and file transferring.

2.6 IP Fragmentation

IP fragmentation occurs when the size of an IP packet is larger than the Maximum Transmission Size (MTU) of the link layer protocol. The payload of the IP packet are split into several IP packets, called fragments, which are less or equal to the MTU of the link layer protocol. IP fragmentation can occur both at the source node and at any intermediate nodes where the outgoing link has a smaller MTU than the ingoing link.

Figure 2.4 shows the IPv4 header. IP fragments are identified by the more fragments flag or that the fragment offset field is not zero. Different fragments are linked together using the Identification field.

If the *do not fragment flag* is set in the IPv4 header, the packet cannot be fragmented. If the packet size exceeds any MTU in the path with this flag

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.4: IPv4 Header [33]

set, it is then simply dropped.

2.7 Connection Handover

When a node changes its access network it is known as a handover. A handover can occur when the node itself is mobile and moves between different access points.

A horizontal handover is a handover within a homogeneous network, such as a handover between two cells in a GSM network. Horizontal handovers are handled in layer 3 and below [28]. In contrast, a vertical handover is a handover between two heterogeneous networks. A vertical handover needs to be handled in layer 3 or above [28].

A soft handover is a handover where the new connection is established before the old connection is disconnected. In contrast, a hard handover is when the old connection is disconnected before the new connection is established. These terms are also known as *make before break* and *break before make* [28]. Some handover techniques are described in Section 3.2.

2.8 Requirement Analysis of Background Knowledge

A requirement from TRAMP is to implement the socket migration system in user space. This requirement enables the system to be easily deployable across different platforms. The socket migration system should also have a well-defined interface towards any migrator system.

Since applications should be able to run without TRAMP installed, they should also be able to run without the socket migrator system installed. No special network equipment or computer equipment should be needed to support TRAMP.

To support video calling a system should at a minimum be able to support a 400 kbit/s bidirectional data stream and ideally a 1.5 Mbit/s bidirectional data stream. The maximum acceptable delay is 100 ms.

A requirement from the Migration Community is that at least two nodes must be present in the community for any migration to take place.

Chapter 3

Related Work

There have been research efforts on socket migration techniques and the closely related field of transport layer mobility. In this Chapter, we look at previous work within these fields. In Section 3.1 we look at related work within the field of socket migration. In Section 3.2 some related transport layer mobility work is described. We describe the mobility concepts of Emerald in Section 3.3. Finally the chapter is summarized in section 3.4.

3.1 Socket Migration

Socket migration is the process of moving an endpoint of a live socket from one node to another. Several different approaches to this has been done, some of which are described here.

3.1.1 MIGSOCK

MIGSOCK enables applications to transfer an endpoint of a live socket from one host to another [15]. The system is implemented in Kernel space and therefore needs modifications to the hosts. New signals are added to the TCP protocol that signals that a socket migration will occur. MIGSOCK sockets demands that both endpoints of the socket are implemented using MIGSOCK. Migrating endpoints that are communicating with a legacy host is therefore not possible. The solution is fully distributed without the need of any additional hosts.

3.1.2 SockMi

SockMi is another solution for transferring an endpoint of a live socket from one host to another [5]. The solution depends on a Linux Kernel Module (LKM) and a daemon, a background process. The LKM translates and forwards incoming IP packets from any legacy Corresponding Host on the exporting host using Destination NAT to the importing host. Packets sent from the importing host to the Corresponding Host and translated using Source NAT. After a migration of a socket has occurred, all the packets from the corresponding host are sent in a triangular fashion. The source and destination node of the socket endpoint that is migrated needs the SockMi module and daemon installed. The Corresponding Host does not need any modifications or additional modules.

Sending IP packets using Source NAT may cause firewalls to drop these packets due to a source IP address originating outside its network. The firewall may therefore see this as an attempt of IP spoofing and therefore regard it as a security risk. SockMi may therefore become nonfunctional in certain scenarios [9].

3.1.3 Migratory TCP

Migratory TCP (M-TCP) is an extension to the TCP/IP stack which aim to provide Service Continuity for end users [36]. The authors claim that TCP's error recovery scheme is insufficient for users that are more interested in continuous service than being connected to a particular server. The extension aims to let the server endpoint of the TCP connection to be migrated to another server which provides the same service to the end user. M-TCP enables server endpoints to migrate by request of the client. When a connection is established, the server sends a migration certificate and a list of cooperating servers to the client. This certificate can then be used by the client to migrate the connection to one of the other cooperating servers. If such a migration is initiated by the client, the original server exports a state snapshot to the new server. Unlike the other socket migration techniques described in this chapter, M-TCP allows one endpoint to initiate the migration of the other endpoint.

3.1.4 Reliable Sockets

Reliable Sockets (rocks) enable sockets to be reestablished after a disconnection. Sockets may be reestablished after a migration, change of IP address or host crashes [38]. Rocks is implemented in user space and lie between applications and the kernel. Reestablishment of connections is transparent to applications. A Diffie-Hellmann key exchange happens when the sockets are established. This key is then used to reestablish the connection if a disconnection occurs.

The rocks API must be present on all of the involved hosts during a migration to work. Rocks works with legacy applications allowing them to resume connections due to change of IP address or an host crash. If a change of IP address or a host crash occurs on both endpoints of the socket, rocks will not be able to resume the connection.

3.1.5 Socketless TCP

Socketless TCP decouples the sockets from a TCP connection by identifying the connection using a Connection Identifier instead of the traditional IP address and port pair [4]. The TCP connection no longer rely on the network layer to identify a connection, therefore changes in the network layer due to network changes or a migration no longer limit the lifetime of the connection.

3.2 Transport Layer Mobility

Transport layer mobility differs slightly from socket migration. The endpoint of the socket is never moved from one node to another. Transport layer mobility enables applications to resume their connections if the node has received a new IP address due to a change of access network, known as a vertical handover.

Transport Layer Mobility is similar to socket migration when we view the migration or handover from the viewpoint of the corresponding host. In both cases, the corresponding host must deal with that the other endpoint changes its IP address.

3.2.1 UPMT

Bonola and Salano has created a system the allows applications to change access network on a node on a per-application basis [7]. Their universal per-application mobility management solution using tunnels (UPMT) is targeted towards handover management on a per application basis when multiple heterogeneous networks are available. The solution allows different applications on one host to use different access networks depending on a given policy. UPMT is completely transparent to corresponding hosts due to a tunnel to an Anchor Node. The Anchor Node is a dedicated server acting as a proxy between the application and the corresponding host. When changing the access network, UPMT still tunnels the traffic through the same Anchor Node, thus the handover is not seen by the corresponding host and is fully transparent. However this solution does not cover mobile applications moving from one node to another, but the principles applied to the handover management are useful for our work. A similar concept is shown in [16].

UPMT provides both a daemon adapter and an UPMT socket interface. The daemon adapter emulates a network interface so that legacy applications can utilize the UPMT solution without altering the source code.

> "The UPMT socket interface extends the traditional socket interface by adding the means to explicitly control the mobility. It fully supports the traditional socket interface, without changing the signature of the methods or of the functions implementing the interface. Existing application [sic] could be modified at the source code level to use the new interface, becoming UPMT aware application." [7]

3.2.2 TCP-R

TCP-R is a TCP redirection mechanism that enables host mobility support in the transport layer [10]. The goal is to maintain active TCP connections even though the IP address changes due to a change in the access network. During the initial handshake, authentication keys are exchanged. These keys are used for authentication if a later handover needs to be handled. TCP-R adds a set of redirect operations to standard TCP protocol. The standard TCP characteristics are maintained. To provide continuous operation due to a change of IP address, both endpoints need to be TCP-R enabled.

3.3 Emerald

In the Emerald programming language, mobile objects move freely between nodes in an overlay network [14]. This creates situations that are fairly similar to a handover, for example when an object on node *A* invokes an object that recently moved from node *B* to node *C*, node *A* will send the invocation message to node *B*. Node *B* will then have stored a forwarding address for that object. The message is therefore forwarded to node *C* and node *C* replies to node *A* directly.

This obviously does not work if node *B* has crashed. In that case a *cascading search algorithm* is initiated throughout the overlay network to find the lost object.

When trying to locate objects while the objects are moving from host to host and so on, the locate packet will "chase" the object until it catches it.

The solution is designed for fine grained mobility. It is not an option if all of the corresponding hosts do not share an overlay network.

3.4 Analysis of Related Work

Even though the systems wish to accomplish the same goal, namely migrating one or possibly both endpoints of a connection from one node to another, they differ in several ways: Where is the system implemented? Does the system need to be present on the corresponding host? Can both endpoints be migrated? Kuntz and Rajan [15] describes three general categories of socket migration, namely Proxy Based Forwarding (PBF), Packet Spoofing (PS) and Host-to-Host Migration Support (HHMS). Table 3.1 compares the different socket migration systems previously mentioned based on these criteria.

System	**Implementation**	**Legacy CH**	**Symmetric**	**Category**
MIGSOCK	Kernel space	No	Yes	HHMS
SockMi	Kernel module	Yes	No	PS
Reliable sockets	User space	No	Yes	HHMS
Migratory TCP	Kernel space	No	No	HHMS
Socketless TCP	Kernel space	No	Yes	HHMS

Table 3.1: Comparison of Socket Migration Systems

A performance evaluation of UPMT [6] shows that such a solution benefits, with respect to processing power, by being implemented in kernel space. One problem is larger package losses in the Anchor Node in the user-space implementation.

Only SockMI supports a legacy corresponding host and is the only one using a packet spoofing solution. All the other systems use a form of Host-to-Host Migration Support forcing both endpoints to be migration aware and thereby excluding legacy corresponding hosts.

UPMT introduces us to a tunneling solution with Anchor Nodes. By introducing an Anchor Node in addition to the Connection Manager, complete transparency for the corresponding host can be achieved. This must however be weighed against a centralized or decentralized solution.

UPMT is designed specifically to support Legacy Corresponding Hosts. Providing a daemon adapter, as in UPMT, to mobile applications is not necessary since the applications anyway need to be migration aware and therefore may as well utilize the proposed language specific framework. Legacy applications do not support mobility out of the box and therefore needs modification.

Chapter 4

Design

In this chapter, we present the design of our socket migration system called SOCKMAND. We start by summarizing our requirements from Chapter 3 and Chapter 2 in Section 4.1. The overall design and architecture of SOCKMAND are described in Section 4.2. Section 4.7 goes into details of the different components of SOCKMAND.

4.1 Requirements

This section describes the requirements for our socket migration system called SOCKMAND. The requirements originate from our findings in Chapter 3 and Chapter 2. The following requirements must be fulfilled in SOCKMAND:

SOCKMAND must not need altering of the OS Kernel By altering the OS kernel, adaptation of the system will be harder for regular users. When altering the OS kernel, the system will be less portable to other operating systems. The system must therefore be implemented in user-space.

The network must not be altered No special network equipment must be needed. The system must be able to run on regular devices.

It must be possible to communicate with legacy applications on legacy hosts Applications must be able to communicate with legacy servers that are not migration aware. By enabling mobile applications to communicate with Skype etc, we can enhance the usability of the client software of these systems without needing to modify the server endpoint.

The solution must be fully distributed No central entity must control the system. The system must only rely on the nodes that are already present for the migration of an application to take place.

Socket Migrations must be transparent to the corresponding host Applications on Corresponding Hosts must not know if an application migrates. This provides privacy of application location.

The API provided to the mobile application must resemble regular sockets If APIs are known to programmers, it will be easier to adopt. Similarity to regular sockets also simplifies the modification of already existing applications to support socket migration. Without software supporting the system, the system will not be adopted.

Migration Community The Migration Community must consist of at least two nodes for any migration to take place. There is no upper limit on how many nodes there may be in the migration community. The limitation will lie in the implementation of the migration community.

Delay The maximum acceptable end to end delay during video calls delay is 100 ms. This means that the maximum acceptable round trip time is 200 ms.

Bandwidth To support video calling, the system should at minimum be able to support a 400 kbit/s or ideally a 1.5 Mbit/s data stream in both directions simultaneously.

4.2 SOCKMAND

This section describes the design of our Socket Migration system called SOCKMAND (SOCKet MANager Daemon). A daemon is a user-space background process. The system is a user space proxy based forwarding socket migration system.

4.2.1 Migration Community Access Point

Using an Anchor Node is necessary when communicating with a legacy application on a legacy node or for transparency reasons. However the use of a single centralized anchor node is not an optimal solution when we want a fully distributed solution. We want the system to select one of the members in the Migration Community as the Anchor Node. We call the node in the Migration Community acting as an Anchor Node the Migration Community Access Point (MCAP).

In cases when it is only one node in the Migration Community, this node will always act as the MCAP. If a mobile application is running on this node, called source node, it will use this node as the MCAP. At a later point, a new node can enter the community and the application is migrated there. In this case, we will leave residual dependencies on the source node. If the user decides to turn the source node off, the connection will break.

Selecting the best possible Migration Community Access Point for the given situation is important. The system may learn that if someone starts a call on their laptop at a certain time of day, they will most likely migrate the application to their cellphone. In such a case, selecting the cellphone as the MCAP is a convenient choice. If it is not likely that the application will migrate, it is beneficial to use the original node of the application as the Anchor Node. The decision making for selecting the best Migration Community Access Point is out of scope in this thesis.

4.2.2 IP in UDP Tunnels

If a mobile application is located on another node than its Migration Community Access Point, tunnels are used to tunnel packets between the two nodes. These tunnels use UDP as its transport protocol. UDP tunnel packets are exchanged directly between two instances of SOCKMAND. These packets contain the IP packets the application has sent through TCP or UDP. Since UDP is unreliable, packets sent over the tunnel may be dropped. If the packet contains a TCP packet and then gets dropped, the TCP protocol will handle this itself.

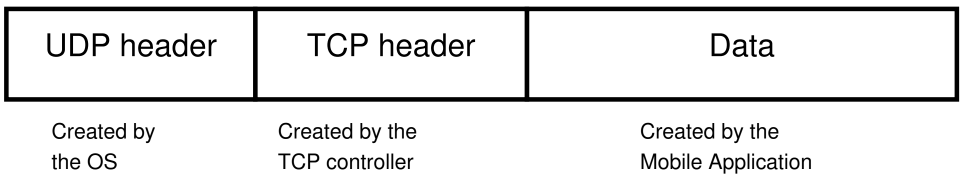
UDP header
TCP header
Data
Created by the OS
Created by the TCP controller
Created by the Mobile Application

Figure 4.1: Contents of a tunnel packet

Figure 4.1 shows the content of a IP packet sent between two nodes using SOCKMAND. The data is created by the mobile application and sent to SOCKMAND. The TCP controller then creates a TCP header and the IP Controller an IP header. When SOCKMAND sends the packet to the Operating System through a regular UDP socket, the UDP header is added.

4.2.3 SOCKMAND Architecture

SOCKMAND is a daemon responsible for handling socket migration if a process with active sockets migrates. The main task of the daemon is to add a layer of abstraction between the migratable applications and the logic involved in the socket migration. SOCKMAND consists of a daemon and language dependent libraries that will act as a framework in different languages. These frameworks must resemble the languages own implementation sockets.

Figure 4.2 shows the interaction of our SOCKMAND system in interaction with other applications and systems. The figure shows three nodes. Node A includes a mobile application, TRAMP and SOCKMAND. The mobile application is communicating with a Legacy Application on Node C. Node B is in this scenario acting as an Migration Community Access Point

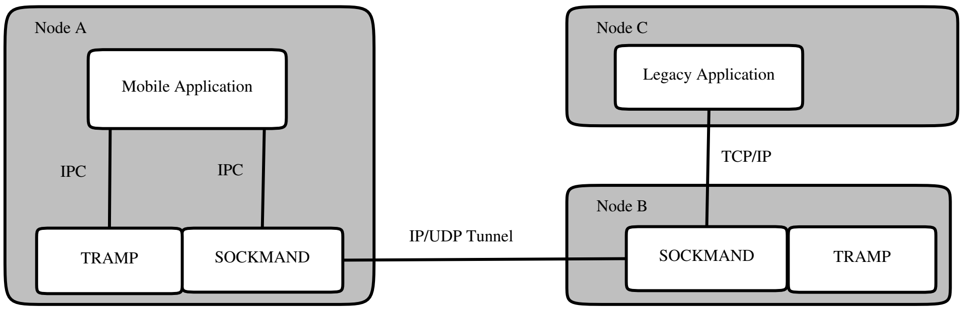

Figure 4.2: System Architecture

(MCAP), described in Section 4.2.1. IP packets are tunneled over UDP from Node A to Node B. Node B then write the contents of the tunneled packet out on raw sockets to Node C. In the opposite direction, Node C is writing regular TCP/IP packets to Node B which captures the entire packet, including IP and TCP headers. Node B then tunnels this packet over UDP to Node A where the data is delivered to the mobile application.

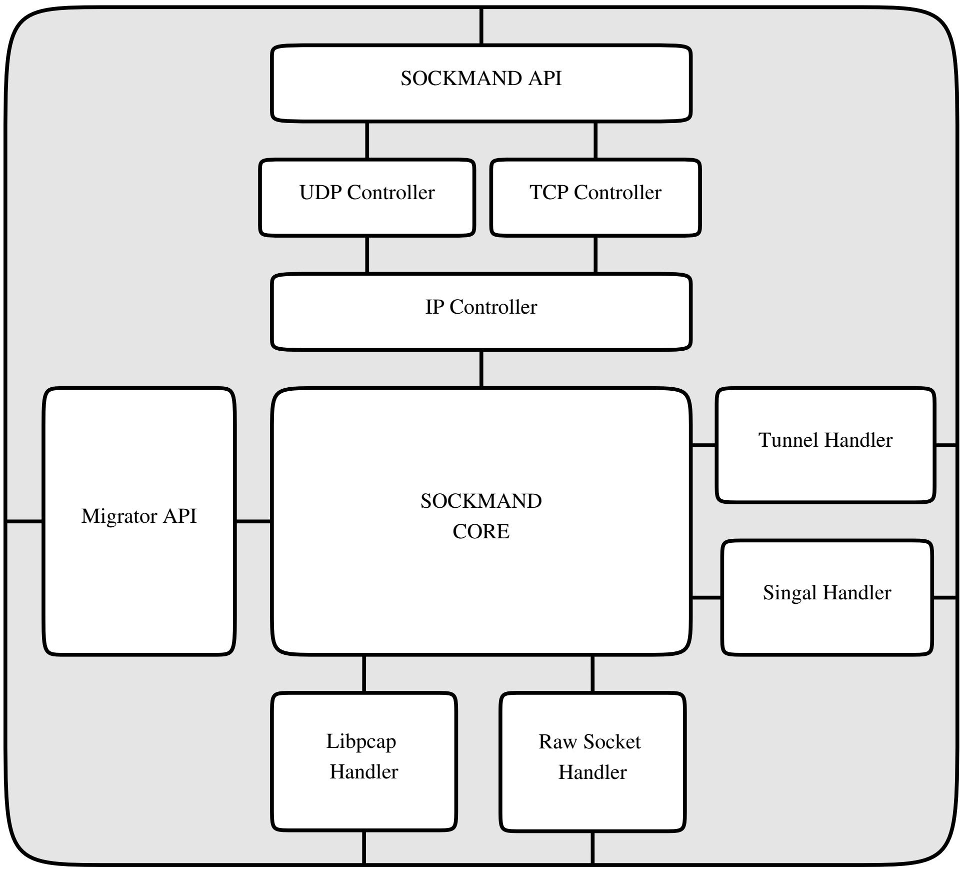

Figure 4.3: SOCKMAND Architecture

Figure 4.3 shows the different components in SOCKMAND. The main component in SOCKMAND is SOCKMAND core. This component is responsible for the routing between applications, the Rawsocket Handler, the Libpcap Handler and the tunnels.

SOCKMAND core exchanges data with the mobile applications via the SOCKMAND API. Packets from the SOCKMAND API are sent to the UDP or TCP controller which handles the transport layer headers and are then sent to the IP controller which handles the IP headers. Packets from the IP controller are sent to the Core and then either to the Tunnel or the Raw Socket Handler depending on where the Migration Community Access Point is located.

If a mobile application migrates to a new host, the Migrator API is responsible for exporting the state of the connections of the Mobile Application to the migrator. It also receives the state of newly arrived applications. This information is used to rebuild sockets of the mobile application.

The Signal Handler is responsible for the signaling between different SOCKMANDs within a Migration Community. The different signals are described in Section 4.7.10.

4.2.4 Different Packet Flows Through SOCKMAND

There are three different routes a packet may take through SOCKMAND. Which route a packet takes depends on the location of the mobile application and the Migration Community Access Point of the specific socket of that mobile application. The flows are illustrated in Figure 4.4

1. Between the mobile application and the Libpcap Handler or Raw-socket Handler

2. Between the mobile application and the Tunnel Handler

3. Between the Libpcap Handler or Rawsocket Handler and the Tunnel Handler

Packets are routed between the mobile application and the Libpcap Handler or Rawsocket Handler when the application is located on the same node as the MCAP is for a given socket.

If the MCAP of a socket is located on another node than the mobile application, packets on the node where the mobile application is located are routed between the mobile application and the Tunnel Handler.

The last scenario is if a node acts as an MCAP for an application located on another node. Packets on this node will then be routed between the Libpcap Handler or Rawsocket Handler and the tunnel handler.

4.3 UUID of a Socket

To simplify implementation, we decide to introduce a unique identifier for each socket. This identifier follows the socket when it is migrated to new nodes. The identifier is created when a socket is created. The identifier consists of a 32 bit random integer value. It is a small possibility that two

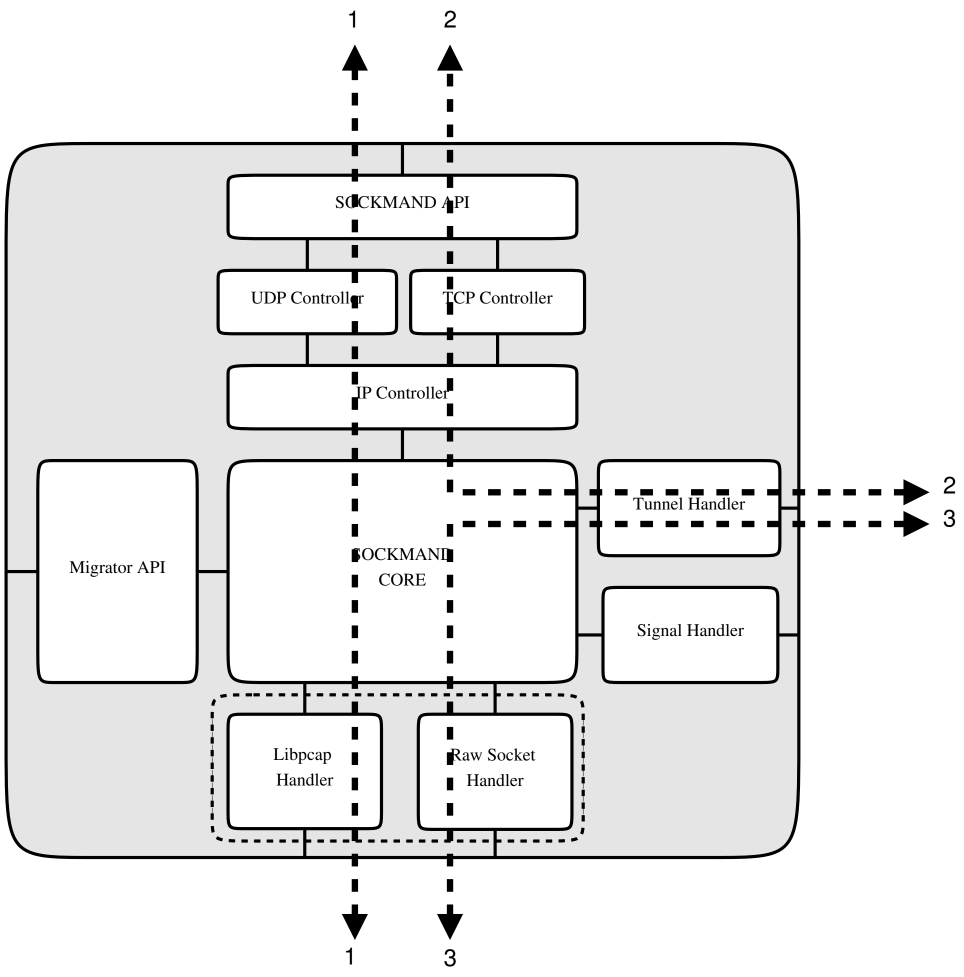

Figure 4.4: Different packet flows through SOCKMAND

similar UUIDs will be created. This is however highly unlikely, and error detection can be introduced to prevent this.

4.4 Migrating a Socket

When the migrator migrates a mobile application, it kills the instance of the application on the source node. SOCKMAND cannot see the difference between this action and when the mobile application actually exits. The migrator must therefore inform SOCKMAND about a migration before it kills the application. To migrate a socket using SOCKMAND, the migrator system must interact with both the mobile application and SOCKMAND in a specific order. Figure 4.5 shows the order of these calls as they are executed.

1. The migrator retrieves the UUID of the sockets used in the Mobile Application

2. The migrator asks SOCKMAND to migrate these sockets to the destination node

3. The migrator kills the mobile application on the source node

Any other tasks the migrator must perform with the mobile application during this process can happen in any order, and do not affect the socket migration.

Two alternative solutions exist. SOCKMAND can use a timer when it notices that an application has exited. If the migrator calls SOCKMAND before the timer goes out, SOCKMAND will know it is a migration. If the timer goes out, SOCKMAND will consider the application exited and then close TCP connections.

Another solution is to catch SIGTERM signals in the SOCKMAND library used in the application. When these signals are caught, the library notifies SOCKMAND through the UNIX domain sockets described in Section 4.7.1. This way, SOCKMAND will know that the application are migrating and not exiting.

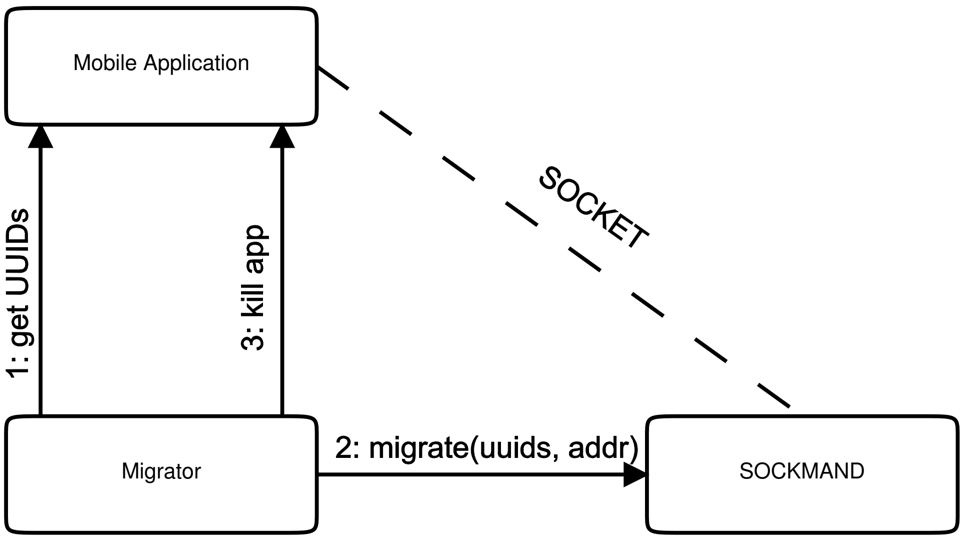

Figure 4.5: Order of calls when an application is migrated

4.5 Message Passing During Migration

There are three different migration scenarios in our design. The scenarios differ on where the MCAP is located during migration.

Figure 4.6 shows the messages that are passed if the source node is the MCAP during a migration. Note that data is sent using regular TCP to Node 1 prior to migration. Data sent during the migration phase is tunneled from node 1 to node 2. This data does not get acknowledged until the socket has been reconnected with the application on node 2. Note that this acknowledgement is tunneled to node 1 which then sends the content of the tunneled packed to the legacy corresponding host.

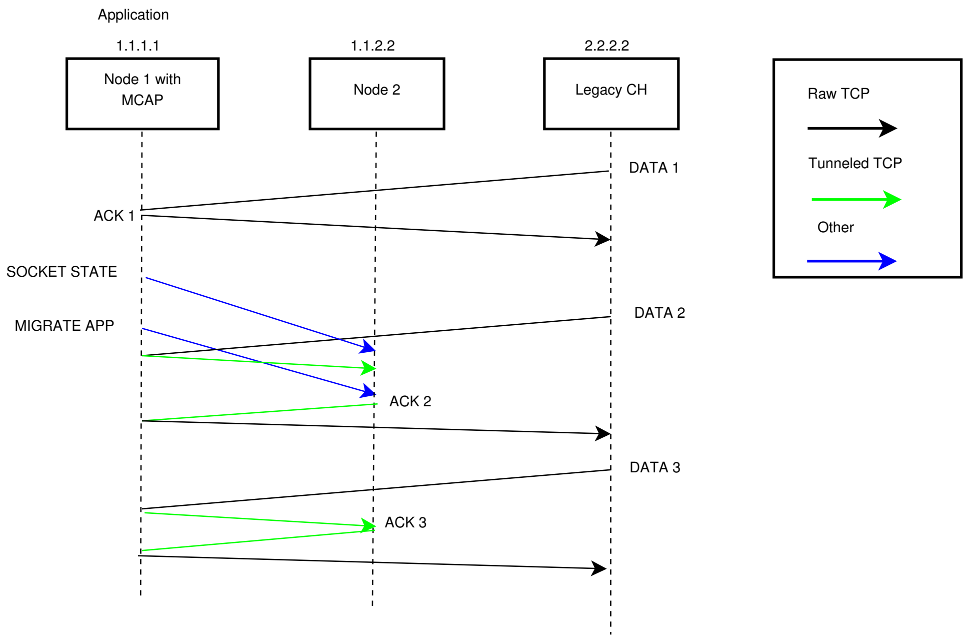

Figure 4.6: Message passing when the source node is also the MCAP

Figure 4.7 shows the messages that are passed if the destination node is the MCAP during a migration. Note that data sent prior to the migration is sent to Node 1 which tunnels it to Node 2 where the application is located. The acknowledgement of this packet is tunneled from Node 2 to Node 1 and then sent to the legacy corresponding host. Packets sent during the migration phase are buffered on Node 1 and not acknowledged until the application is done migrating.

The last scenario is when the MCAP is located on a node that does not participate in the migration. The message flow of this scenario is shown in figure 4.8.

4.6 IP Fragmentation and SOCKMAND

When IP packets are fragmented, the transport layer header is only included in the first IP fragment. This causes problems when we are inspecting the transport layer header of each packet at an intermediate node since the transport layer header is missing from all the packets except from the first.

Two solutions to this exist. We can buffer IP fragments until we have all fragments in the MCAP. When all the fragments have arrived, we forward all of them based on the transport layer header. A problem with this solution is that it introduces an additional delay for the IP fragments. If a fragment gets dropped, the fragment will not be resent by the sender. Because of this, it must be a timer in the MCAP that flushes the buffer if not all of the IP fragments have arrived within a given time.

The other solution is to store the identification field, source and

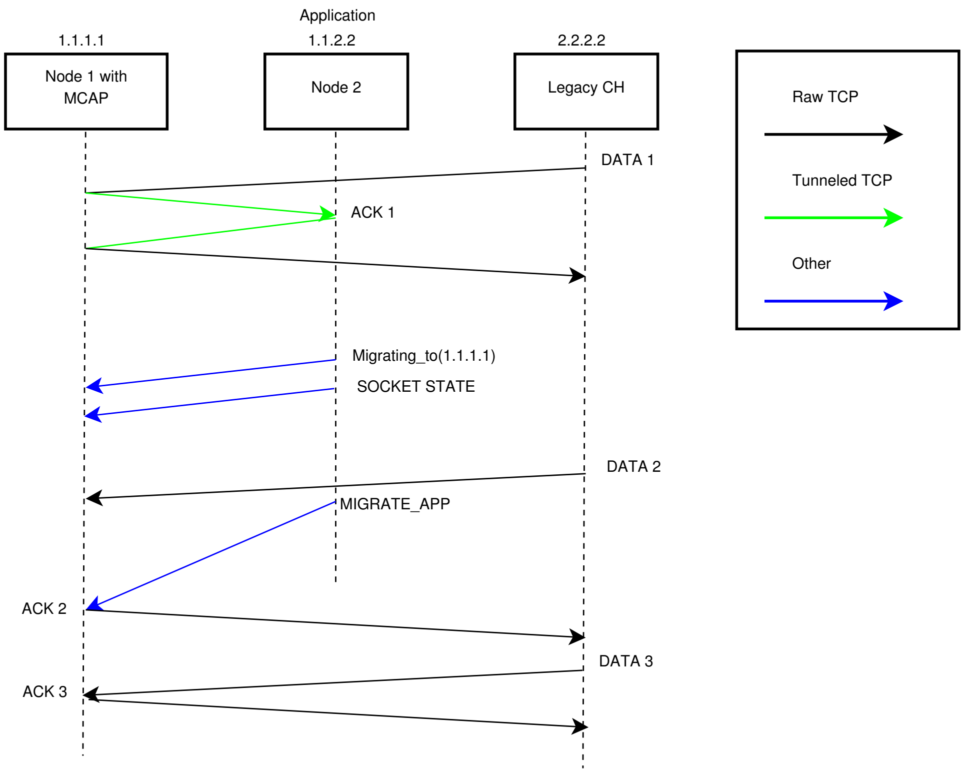

Figure 4.7: Message passing when the destination node is also the MCAP

destination address of the first packet with the routing information based on the transport layer header in a dedicated IP fragment routing table. This will however only work if the first fragment actually arrives first. IP does not guarantee ordering of packets so the first fragment may arrive last. A solution to this is to buffer fragments that we do not currently have routing information about.

Due to time limitations we will not implement support for IP fragmentation. Instead we make sure that none of our IP packets exceeds the Ethernet MTU of 1500 Bytes. This does not affect TCP since a TCP packet size has no meaning for the application layer. UDP packet sizes will however be limited. The maximum payload of a UDP packet sent through SOCKMAND is limited by the Ethernet MTU and the two IP headers and UDP headers that are included when a UDP packet is tunneled from SOCKMAND to an MCAP. The UDP header size is 8 Bytes and the IP header size is 20 Bytes The maximum UDP payload size is:

$1500B - 2 * (20B - 8B) = 1444Bytes$

4.7 Detailed Design

This section goes into the details of the different components of SOCKMAND. Since many of the components in our design are dependent on each other, there is no linear order in which the components can be described. The components will be described in the order of which they are used in the

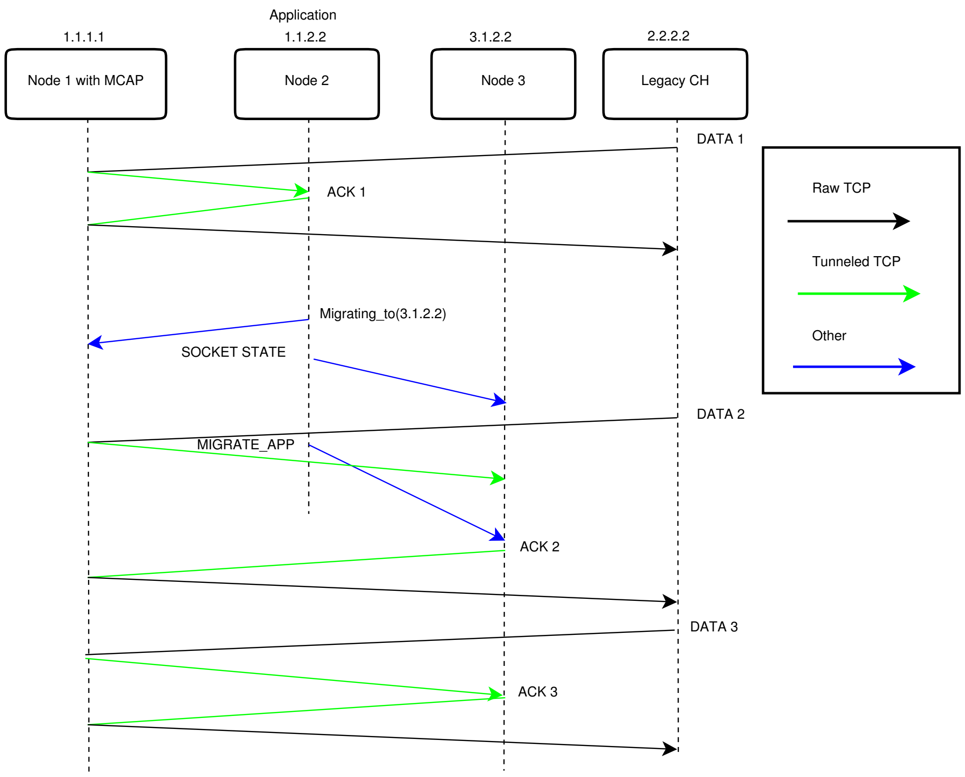

Figure 4.8: Message passing when the MCAP is neither the source nor destination node

first described scenario in Section 4.2.4.

4.7.1 SOCKMAND API

The SOCKMAND API is the component where applications connect to SOCKMAND. In addition to the API component, a library is provided to applications wishing to connect to the SOCKMAND. This library manages the IPC between the SOCKMAND API and the application.

We use UNIX domain sockets as the form of IPC between applications and the Socket Manager. This is convenient since they provide a file descriptor which can be used as a regular file-descriptor by the application. The API provided to the mobile applications therefore resembles regular sockets since they both use file-descriptors which are selectable.

If UNIX domain sockets are unavailable on a given system, local sockets can be used. UNIX domain sockets are however faster than local sockets since they bypass the network stack [37].

The SOCKMAND API receives calls over a UNIX domain socket from the SOCKMAND library, executes the specific operation by calling a function in the UDP or TCP controller and sends a return value to the application.

4.7.2 SOCKMAND Library

The SOCKMAND Library consists of the several functions. Most of these functions are designed to behave exactly as their POSIX equivalent.

sm_socket

The sm_socket function should work as described in [26]. The function returns a file descriptor which can be used by the writing and reading function described later in this chapter. However we introduce a fourth argument: mcappolicy. This argument is used to determine where the Migration Community Access Point will be located. If the mcappolicy argument is 0, the MCAP will be located on the same node as the application currently resides.

sm_bind

The sm_bind function is similar to the POSIX bind function described in [17]. Calling sm_bind will bind a socket to a port number. The POSIX bind function will only bind a socket to a port on the local node, but the sm_bind function will bind a socket to a port number on the MCAP that was selected by the mcappolicy argument in the sm_socket function.

sm_listen

The sm_listen function is similar to the POSIX listen function described in [20]. It marks the socket as a passive socket that will wait for an incoming connection. The POSIX function allows the application to provide a backlog integer argument which states how many unaccepted connections that can be queued on the file descriptor. Due to time limitations and that this is not vital for the key concepts presented in this thesis, we do not implement the queuing functionality. Notice that the port the socket is bound to may be located on another node where the MCAP is located.

sm_accept

The sm_accept function is similar to the POSIX bind function described in [22]. The function will block until another node connects to the port number the socket is bound to. On success, the function returns the file-descriptor of the accepted socket. On error, the function returns -1. After the function has returned successfully, we can use the sm_write an sm_read functions on the socket.

sm_connect

The sm_connect function is similar to the POSIX connect function described in [19]. The function can only be used with TCP connections. It initiates a three way handshake with a corresponding host set in the parameters. The three way handshake is done by sending a SYN packet to the corresponding host. The corresponding host replies with a SYN/ACK packet which is replied with an ACK. During this three way handshake, sequence numbers are initiated. When the sm_connect is successfully executed, the TCP connection will be in a connected state. If the sm_bind function has not been called yet, SOCKMAND will bind the socket to a random port.

sm_close

The sm_close function is similar to the POSIX connect function described in [18]. It can be used on both UDP and TCP sockets. If the socket is using TCP, it will initiate a tear down sequence of FIN/ACK and ACK packets that will set the socket in a closed state. If the socket is UDP, it will simply unbind the socket so that no further packets can be sent or received on the socket.

sm_read

The sm_read function is similar to the POSIX connect function described in [21]. It can be used on both UDP and TCP sockets. If the socket is using TCP, it will try to read at most *count* bytes from the data stream into *buf*. It will return the number of bytes that actually were read. If the socket is using UDP, it will do a call to sm_recvfrom with the src_addr and addrlen arguments set to 0. This means that it will receive the next datagram and discard the metadata of the datagram.

sm_write

The sm_write function is similar to the POSIX connect function described in [27]. It can only be used on connected TCP sockets. If the socket is using TCP, it will try to write at most *count* bytes from the data stream from *buf*. It will return the number of bytes that were written.

sm_sendto

The sm_sendto function is similar to the POSIX connect function described in [25]. It can only be used on UDP sockets. It will send a datagram to the destination set in the *dest_addr* structure. If the sm_bind function has not been called yet, SOCKMAND will bind the socket to a random port.

sm_recvfrom

The sm_recvfrom function is similar to the POSIX connect function described in [23]. It can only be used on UDP sockets that have been bound. It will read a datagram from a socket into *buf*. If *src_addr* and *addrlen* is not equal to 0, it will read the metadata of the datagram into the *src_addr* structure.

sm_getuuid

The sm_getuuid function is intended to be used when the mobile application exports its state to a migrator. The function returns the UUID of a socket, described in Section 4.3. The uuid that is returned must be used to rebuild the socket on the destination node using the sm_rebuildsocket function.

sm_rebuildsocket

The sm_rebuildsocket function is intended to be used when the mobile application rebuilds a socket after arriving at the destination node. the function must be called on the destination node after the socket has been migrated there. On success, the function will return a file descriptor which can be used by the writing and reading function described later in this chapter. If the socket has not been migrated to the destination node, the function will return -1.

4.7.3 UDP Controller

The UDP Controller contains an UDP implementation in user space. Data are received from the SOCKMAND API and then packed into a UDP packet and sent to the IP Controller, described in Section 4.7.5. When UDP packets enter SOCKMAND via either the tunnel or the MCAP, UDP packets are received from the IP Controller and the payload is sent to the SOCKMAND API for delivering to the mobile applications. The state of UDP ports are maintained in this component. A port can either be bound to an application or unused. When an application migrates away from a host, the state of its UDP ports are exported from this component.

4.7.4 TCP Controller

The component contains a full TCP implementation in user space. This involves handshaking, acknowledging and tearing down connections. The TCP controller is responsible for handling the state of TCP connections.

Data are received from the SOCKMAND API and then packed into a TCP packet and sent to the IP Controller. When TCP packets enter SOCKMAND via either the tunnel or the MCAP, TCP packets are received from the IP Controller and the payload is sent to the SOCKMAND API for delivering to the mobile applications. The states of the TCP connections are maintained in this component. When an application migrates, the state of its TCP connections are exported from this component.

4.7.5 IP Controller

The IP controller receives IP packets from the UDP and TCP controller and adds the correct checksum. The packets are then forwarded to the SOCKMAND Core described in Section 4.7.6.

The IP controller receives IP packets from the SOCKMAND Core. The IP packets are sent either to the TCP or the UDP Controller depending on the protocol of the packet.

4.7.6 SOCKMAND Core

The SOCKMAND Core is the main component that routes IP packets between the IP Controller, the Tunnel Handler, the Libpcap Handler and the Rawsocket Handler. Packets are routed depending on where the Mobile Application is located. SOCKMAND Core also handles signals received through the Signal Handler.

The component is responsible for the exporting of socket states to other nodes. States can be exported to another node in two cases; when the socket is migrated and when a new socket is created and the MCAP is located on another node. The node with the MCAP must know where to route the packets it captures.

4.7.7 Libpcap Handler

The Libpcap Handler component is responsible for receiving raw IP packets. Raw IP packets are received using the cross platform pcap library. When connections are established to a corresponding host, a pcap filter captures all incoming packets on this connection and delivers them to the MCAP component. The packets are then sent to the SOCKMAND core for further routing and processing.

Packets captured by the pcap library also enter the kernel. The packet we receive is a duplicate of the packet that enters the kernel. When UDP packets enter a kernel which has no application bound to the destination port of that packet, they are just dropped by the kernel. However, TCP packets that enters a kernel which has no application bound to the destination port of that packet are replied with an TCP packet with the RESET flag set. Since we handle the TCP packet in user space, we must stop the kernel from sending these RESET packets. This is done by calling the iptables for Linux or ipfw for OSX through the system() call and adding a filter that drops outgoing RESET packets on the ports we are using from SOCKMAND. Instead of calling iptables, we can use the libiptc library which enables us to modify the firewall table directly from the source code. This library can however change without notice and will only work on Linux and not on OSX.

4.7.8 Rawsocket Handler

The Rawsocket Handler component is responsible for sending raw IP packets. The component receives IP packets from the SOCKMAND Core. These packets may originate from a tunnel or from a local mobile application. The packets include both the IP header and the transport layer header. These packets should now be sent to their destination. The packets are sent through raw sockets since they already include transport layer and IP headers.

4.7.9 Tunnel Handler

The Tunnel Handler is responsible for sending and receiving packets on the IP/UDP tunnel described in Section 4.2.2. The Tunnel Manager listens on a single UDP port, and all incoming packets are received on this port. Packages received on the tunnel are delivered to the Socket Manager Core which routes the packet either to the IP Controller or the MCAP.

4.7.10 Signal Handler

The signal handler is responsible of send signals and socket states between nodes. The component connects to SOCKMAND instances on other nodes using a regular TCP/IP socket. The two different signals can be sent are described in the following subsections.

migrating_to signal

The migrating_to signal is sent from the node where the endpoint of a socket is located. The signal is sent to the MCAP of that particular node when the endpoint of the socket will migrate to a new node which is not the MCAP. The signal must be sent so that the MCAP can update the current location of the endpoint of that particular socket. The signal is synchronous, so the sender blocks until the receiver has processed the signal. The parameters are the UUID of the socket that will migrate and the new address of the endpoint of the socket.

socketstate signal

The socketstate signal is sent from the node where the endpoint of a socket is located. The signal includes the state of a socket. It can be sent in two cases. The first case is when a socket is established and the MCAP needs to be informed about the socket and the current endpoint of the socket. The other is when the socket is migrated to a new node. The destination node must then be informed about the state of the socket. The signal is synchronous, so the sender blocks until the receiver has processed the signal.

4.7.11 Migrator API

The Migrator API is responsible for the interaction between a migrator and SOCKMAND. Like the SOCKMAND API, the migrator API can use UNIX domain sockets to provide inter process communication between the migrator and SOCKMAND. Since we do not have any implementation of a migrator yet, we simulate the migrator API by user interaction in SOCKMAND using the keyboard.

migrate_socket function

The migrator can call the migrate_socket function in SOCKMAND. The parameters used are int UUID and unsigned int address. The call is synchronous, so the migrator blocks until SOCKMAND is done migrating the socket. The function returns 0 on success and -1 if there is an error.

get_anchoraddress function

SOCKMAND can call the get_anchoraddress function in the migrator. The parameter used is int mcap_policy. The parameter describes a given policy the migrator should base the selection of an MCAP on. The function returns the address of the node in the overlay network that best matches the policy. The implementation of this function in a migrator is out of scope of this thesis.

Chapter 5

Implementation

In this chapter we describe the implementation of SOCKMAND. Section 5.1 describes the implementation environment. In Section 5.2 we present an overview of the implementation and Section 5.3 presents the implementation details of the various components in SOCKMAND.

5.1 Environment

We have implemented SOCKMAND for Linux. SOCKMAND is implemented in the C programming language. It is portable to Mac OSX with minor modifications. As mentioned in Section 2.3, Windows does not support raw sockets which prevent us from adding support for Windows.

5.2 Overview

Our implementation closely follows the architecture presented in Figure 4.3. Each of the components in the architecture has a corresponding C source file and header file. In addition, the event handling and a utility file is implemented in their own C source and header files.

The most important data-structures are the appsocket structure described in Section 5.3.3 and the transport layer control blocks described in Sections 5.3.5 and 5.3.6. The most important functions are in the SOCKMAND Core described in Section 5.3.2. There we describe the routing functions and the functions concerning the actual migration of a socket. The other components mostly contain functionality necessary for the user space transport layer implementations.

5.3 SOCKMAND components

In this section we go through some implementation details of SOCKMAND. The implementation of some components is trivial and is not necessary to present in this chapter in order to understand the thesis. They are therefore not described in detail.

5.3.1 Event Handler

Events in SOCKMAND are handled using the POSIX select function. The select function is described in [24]. We build a set of all the file descriptors where new data can arrive. These file descriptors include the tunnel, the libpcap socket, signal sockets and UNIX domain sockets connected to applications and the migrator. In addition it handles timeouts in the TCP controller. Listing 5.1 shows the main event loop in SOCKMAND.

Listing 5.1: Event Handling

```
void handle_events( ){
    //optimization: use libevent
    fprintf(stderr,"Starting main event loop\n");
    while( running ){
        build_select_list();
        struct timeval timeout;

        timeout.tv_sec = 0;
        timeout.tv_usec = CLOCKGRANULARITY * 1000;

        int retval = select( max_fd + 1, &read_set, 0, 0, ↩
            &timeout );

        switch( retval ){
            case -1 :
                break;
            case 0 :
                checktcptimeout();
                break;
            default :
                //Check if tunnel packets has arrived
                if(FD_ISSET(tunnel_sockfd, &read_set)){
                    read_tunneled_packet( );
                }
                //Check if packets have been captured
                if(FD_ISSET(pcap_fd, &read_set)){
                    read_raw_packet( );
                }
                //Check timeouts
                checktcptimeout();

                //Check if a new signal connection is here
                if(FD_ISSET(listenfd, &read_set)){
                    handlesignalconnection();
                }

                //Check for keyboard
                if(daemonized == 0){
                    if(FD_ISSET(STDIN_FILENO, &read_set)){
                        handle_keyboard( );
                    }
                }

                //Check for local IPC
                if(FD_ISSET(app_sockfd, &read_set)){
                    handleappconnection();
                }
                int i;
                //Check for new data from apps
                for(i = 0; i < napps; i++){
                    if(appsockets[i].active)
                        if(FD_ISSET(appsockets[i].unixfd, &read_set)) {
                            handleapp(appsockets[i].unixfd);
                        }
                }

                //Check for any new signals
                for(i = 0; i < nsignals; i++){
                    if(signalsockets[i].active)
                        if(FD_ISSET(signalsockets[i].fd, &read_set)) {
                            handlesignal(signalsockets[i].fd);
                        }
                }
                break;
        }
    }
}
```

Using select is satisfactorily for a small set of file descriptors. If the set is very large, we can benefit from using libevent instead [29].

5.3.2 SOCKMAND Core

SOCKMAND Cores main responsibility is managing the routing between the IP Controller, the Tunnel Handler, the Libpcap Handler and the Rawsocket Handler.

Listing 5.2: Handle IP Controller Packet

```
int handle_ip_packet(struct ipheader *ippacket){
    if(is_my_address(ippacket->ip_src)){
        return write_ip_packet(ippacket);
    }
    else{
        return write_tunneled_packet(ippacket, ippacket->ip_src);
    }
}
```

Listing 5.2 shows how packets that are received from the IP Controller are handled. They are sent either to the Rawsocket Handler or to the Tunnel

Handler based on the location of the MCAP. The address of the MCAP is located in the ip_src field of the IP packet. If the ip_src field contains the IP address of the local node, the local node is the MCAP and the packet is sent using raw sockets. If the ip_src field contains the IP address of a remote node, the packet is tunneled to the remote node which is acting as the MCAP.

Listing 5.3: Handle Tunneled Packet

```
void handle_tunneled_packet(struct ipheader *ippacket){
    if(is_app_present(ippacket)){
        read_ip_packet(ippacket);
    }
    else{
        if(is_my_address(ippacket->ip_src)){
            write_ip_packet(ippacket);
        }
        else{
            //Major error, should not be possible
            printf("Unexpected packet in handle tunneled ↩
               packet\n");
        }
    }
}
```

Listing 5.3 shows how packets that are received from the Tunnel Handler are handled. If the endpoint of the socket is located at the current node, the packet is delivered to the IP controller. The packet was in this case tunneled to the current node from the MCAP. Otherwise, the packet is sent to its final destination using raw sockets. In the last case, this node is acting as an MCAP of the socket.

Listing 5.4: Handle Tunneled Packet

```
void handle_captured_packet(struct ipheader *ippacket){
    if(is_app_present(ippacket)){
        read_ip_packet(ippacket);
    }
    else{ //App somewhere else
        write_tunneled_packet(ippacket, get_app_location(↩
           ippacket));
    }
}
```

Listing 5.4 shows how packets that are received from the Libpcap Handler are handled. If the endpoint of the socket is located at the current node, the packet is delivered to the IP controller. Otherwise, the packet is tunneled to its final destination. In both cases, this node is acting as an MCAP of the socket.

In addition to routing packets between components, SOCKMAND core also contains logic for exporting socket states, importing socket states and migrating sockets. Listing 5.5 shows how a socket state is exported. Socket state can be exported either when binding a socket to a port and the MCAP remote or when a socket is being migrated to a new node. Notice that the size of the socket state depends on the transport protocol.

Listing 5.5: Exporting Socket State

```c
void send_socket_state(struct connectioninfo *c, unsigned ↩
    int address){
     int cblen = 0;
     if(c->protocol == IPPROTO_TCP)
          cblen = sizeof(struct tcp_cb);
     else if(c->protocol == IPPROTO_UDP)
          cblen = sizeof(struct udp_cb);

     int unackedpacketslength = 0;
     if(c->protocol == IPPROTO_TCP && address == c->↩
         current_address){
          unackedpacketslength = ↩
              get_serialized_timeout_length((struct tcp_cb*)↩
              c);
     }

     int length = sizeof(struct socketstate) + cblen + ↩
         unackedpacketslength;

     struct socketstate* ss = malloc(length);
     ss->signaltype = SOCKETSTATE;
     memcpy(ss+1, c, cblen);
     ss->protocol = c->protocol;
     ss->length = length;

     if(unackedpacketslength != 0){
          char* timeouts = ((char*)ss) + cblen;
          serialize_timeouts(timeouts, get_cb_by_pointer(c))↩
              ;
     }

     sendsignal((char*)ss, length, address);
     free(ss);
}
```

Listing 5.6: Importing Socket State

```c
void import_socket_state(struct socketstate* ss){
     struct connectioninfo *c = (struct connectioninfo *) (↩
         ss+1);

     c->sockmandapi_id = -1; //No app has connected here ↩
         yet

     if(c->current_address != get_my_addr()){
         c->status = MIGRATEDAWAY; //This node will be the anchornode
         fprintf(stderr,"Acting as MCAP for socket %d\n", c->uuid);
     }
     else{
         fprintf(stderr,"Socket %d migrated here\n", c->uuid);
     }


     int cblen = 0;
     if(c->protocol == IPPROTO_TCP)
         cblen = sizeof(struct tcp_cb);
     else if(c->protocol == IPPROTO_UDP)
         cblen = sizeof(struct udp_cb);


     struct connectioninfo *oldinfo = get_connection_info_by_uuid(c->uuid);
     if(oldinfo!=0){
         //The socket has either use this node as an MCAP or has been here before
         memcpy(oldinfo, c, cblen);
         if(c->protocol == IPPROTO_TCP){
             int timeoutslen = ss->length - sizeof(struct tcp_cb) - sizeof(struct socketstate);
             if(timeoutslen>0){

                 char* timeouts = (char* )ss + sizeof(struct tcp_cb);
                 deserialize_timeouts(timeouts, get_cb_by_pointer(oldinfo));
             }
         }
     }
     else{
         if(c->protocol == IPPROTO_TCP){
             int cb = tcp_importstate((struct tcp_cb*)c);
             int timeoutslen = ss->length - sizeof(struct tcp_cb) - sizeof(struct socketstate);
             if(timeoutslen>0){
                 char* timeouts = (char* )ss + sizeof(struct tcp_cb);
                 deserialize_timeouts(timeouts, cb);
             }

         }
         else if(c->protocol == IPPROTO_UDP)
             udp_importstate((struct udp_cb*)c);
     }
 }
```

Listing 5.6 shows how a socket state is imported. It may be cases where we import the Socket State of one socket several times. Therefore we copy over the old socket state if it exists to prevent duplicate information. If the socket state has not been present on the node before, the UDP controller or the TCP controller imports the state into a new control block.

Listing 5.7: Migrate Socket

```
int migrate_socket(int uuid, unsigned int newaddr){
    if(newaddr == get_my_addr()){
        printf("Migrating to here, not possible\n");
        return -1;
    }

    struct connectioninfo *c = get_connection_info_by_uuid↩
       (uuid);
    if(c==0){
        printf("Unknown uuid\n");
        return -1;
    }

    c->current_address = newaddr;

    if(c->mcap_address != get_my_addr() && c->mcap_address↩
        != newaddr){
        //Send signal to the MCAP
        struct signalheader s;
        s.signaltype = MIGRATING_TO;
        s.newaddress = newaddr;
        s.uuid = uuid;
        sendsignal((char*)&s, sizeof(struct signalheader),↩
            c->mcap_address);
    }

    send_socket_state(c, newaddr);
    c->status = MIGRATEDAWAY;
    c->sockmandapi_id = -1;
    return 0;
}
```

Listing 5.7 shows how a socket is migrated. First we update the control block with the new location of the application. The state of the socket is then sent to the destination node. If the MCAP is neither the source nor the destination node, the MCAP is notified of the new address of the socket.

5.3.3 SOCKMAND API

Listing 5.8: Sockets table

```c
struct appsocket{
    int unixfd;
    int controlblock; //Id in either tcp or udp cont
    int uuid; //random number
    short active;
    short protocol;
    unsigned int mcappolicy;
};
```

The SOCKMAND API component contains a data structure that has information about all the sockets that are connected to mobile applications. A new element is added to the structure either when the sm_socket function is called or the sm_rebuildsocket function is called. The data structure is shown in Listing 5.8 The unixfd field contains the file descriptor of the UNIX domain socket that is used to transfer data between SOCKMAND and a mobile application. This file descriptor is added to the file descriptor set mentioned in Section 5.3.1. The control block field contains the id of either a UDP control block or a TCP control block. It is used when data is sent to either the TCP controller or the UDP controller. If it is -1, the socket has not yet been bound. The mcappolicy field contains the mcappolicy that is sent as an argument through the sm_socket function.

The event handler checks for new UNIX domain socket connections by listening to the path defined in SOCK_PATH and for new data on the existing UNIX domain sockets. If new data arrives, the handleapp function reads the message and determines which function that has been called in libsockmand by checking a message header which contains the information about the call. The SOCKMAND API then executes the call by calling functions in either the TCP controller or the UDP controller. The details of the implementation of the different calls are found in the source code.

5.3.4 Libsockmand

Libsockmand is the library that must be included in mobile applications to connect to the SOCKMAND API. We have written libsockmand in C, but it is easily implemented in other languages such as Java to support mobile applications written in Java. All the functions except from sm_read write the arguments to the SOCKMAND API using a UNIX domain socket and read a data structure from the SOCKMAND API. This means that the calls between libsockmand and the SOCKMAND API are synchronous.

sm_socket

Listing 5.9 shows how the sm_socket function is implemented in libsockmand. The sm_socket function creates the connection between libsockmand and the SOCKMAND API. It connects to the SOCKMAND API using the path defined in SOCK_PATH. The protocol and mcappolicy arguments are then sent to the SOCKMAND API and the SOCKMAND API returns a UUID of the socket. The function returns the actual UNIX domain socket file descriptor which is used later when calling the other functions in libsockmand. Since we return an actual file descriptor, we are able to use the select function on the migratable socket.

Listing 5.9: sm_socket( )

```
int sm_socket(int domain, int type, int protocol, int ↩
    mcappolicy){
    if(domain != AF_INET)
        return -1;
    if(type == SOCK_STREAM){
        if(!(protocol == 0 || protocol == IPPROTO_TCP))
            return -1;
        if(protocol == 0)
            protocol = IPPROTO_TCP;
    }
    else if(type == SOCK_DGRAM){
        if(!(protocol == 0 || protocol == IPPROTO_UDP))
            return -1;
        if(protocol == 0)
            protocol = IPPROTO_UDP;
    }
    else{
        return -1;
    }

    int s, len;
    struct sockaddr_un remote;

    if((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1){
        perror("socket");
        return -1;
    }

    remote.sun_family = AF_UNIX;
    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.↩
        sun_family);
    if(connect(s, (struct sockaddr *)&remote, len) == -1){
        perror("connect");
        return -1;
    }

    struct socketmsg msg;
    msg.msgtype = SOCKET;
    msg.protocol = protocol;
    msg.mcappolicy = mcappolicy;

    write(s, &msg, sizeof(struct socketmsg));

    socketprotocol[s] = protocol;
    uuid_of_fd[s] = readint(s);
    return s;
}
```

sm_rebuildsocket

Listing 5.10 shows how the sm_rebuild function is implemented in libsockmand. It is fairly similar to the sm_socket function, but instead of sending the mcappolicy and protocol to the SOCKMAND API, we send a message with the UUID of the socket we want to rebuild. The SOCKMAND API will on a successful rebuild return the protocol of the socket and on error return -1. Notice that we in this function also return the UNIX file descriptor to the mobile application.

Listing 5.10: sm_rebuildsocket( )

```
int sm_rebuildsocket(int uuid){
    int s, len;
    struct sockaddr_un remote;

    if((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1){
        perror("socket");
        return -1;
    }

    remote.sun_family = AF_UNIX;
    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.↩
        sun_family);
    if(connect(s, (struct sockaddr *)&remote, len) == -1){
        perror("connect");
        return -1;
    }

    struct rebuildmsg msg;
    msg.msgtype = REBUILD;
    msg.uuid = uuid;

    write(s, &msg, sizeof(struct rebuildmsg));

    int ret = readint(s);
    if(ret == -1){
        close(s);
        return ret;
    }
    uuid_of_fd[s] = uuid;
    socketprotocol[s] = ret;
    return s;
}
```

sm_recvfrom

Listing 5.11 shows how the sm_recvfrom function is implemented in libsockmand. It first reads a message containing the metadata of a UDP datagram. If the application programmer requests it, the metadata is copied into the structure pointed to by src_addr. The actual payload of the datagram is then read and finally the number of bytes read is returned. The other functions in libsockmand are implemented using the same technique, we will therefore not go into further details about the other functions.

Listing 5.11: sm_recvfrom( )

```c
int sm_recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen){
    struct recvfrommsg m;
    read(sockfd, &m, sizeof(struct recvfrommsg));

    if(flags!=0){
        fprintf(stderr, "Flags not yet suppported, use 0\n");
        return -1;
    }

    //fill in the src addr struct if possible
    if(src_addr != 0 && addrlen !=0 && *addrlen >= sizeof(struct sockaddr_in)){
        struct sockaddr_in *servaddr =  (struct sockaddr_in*) src_addr;
        bzero(servaddr,sizeof(struct sockaddr_in));
        servaddr->sin_family = AF_INET;
        servaddr->sin_addr.s_addr = m.sourceaddr;
        servaddr->sin_port = m.sourceport;
    }

    //If we ask for more or equal to what was available
    if(m.len <= len){
        return read(sockfd, buf, m.len);
    }

    //If we ask for less than what was available
    else{
        int ret = read(sockfd, buf, len);
        read(sockfd, buf, m.len-len); //Discard the unread part, this is standard POSIX
        return ret;
    }
}
```

5.3.5 UDP Controller

The UDP Controller component contains a UDP implementation in user space. UDP is a simple transport layer protocol and is also simple to implement. Most of the functions are implemented as they would have been implemented with no migration support.

Listing 5.12 shows the UDP Control Block structure. The sockmandapi_id field refers to the id of the socket in the data structure in the SOCKMAND API component. If the sockmandapi_id field is set to -1, the application has not yet called the sm_rebuildsocket function and connected to the endpoint of the socket. The current_address field contains the actual location of the endpoint of the socket and is used to forward packets from an MCAP to the actual location of the application. The mcap_address field contains the location of the MCAP of the socket. The bufferedpacketsfirst points to the first buffered packet that has arrived at the endpoint before an application has called sm_rebuildsocket. The bufferedpacketslast points to the last buffered packet that have arrived at the endpoint before an application has called sm_rebuildsocket. When sm_rebuildsocket is called, the packets pointed to by the bufferedpackets pointer are delivered to the application.

Listing 5.12: UDP Control Block

```
struct udp_cb{
    int uuid;
    int sockmandapi_id;
    int protocol;
    short status;
    unsigned short sourceport;
    unsigned int current_address;
    unsigned int mcap_address;
    struct bufferedpacket* bufferedpacketsfirst;
    struct bufferedpacket* bufferedpacketslast;
};
```

Listing 5.13 shows how UDP sockets are bound. Notice that we check if the mcap_address is on the current node. If it is not on the current node, the information about the socket must be exported to the MCAP so that the MCAP is able to tunnel the packets to the correct node. If the MCAP is located on the current node, we start capturing packets with the source port in the destination field of the UDP header.

Listing 5.13: UDP binding

```
int udpbind(int sockmandapi_id, int uuid, short sourceport↩
    , unsigned int mcap_address){
    int i;
    for(i = 0; i < next_udpcb; i++){
        if(udp_cbs[i].status == UDPACTIVE && udp_cbs[i].↩
            sourceport == sourceport && udp_cbs[i].↩
            mcap_address == mcap_address){
            return -1;
        }
    }

    udp_cbs[next_udpcb].sockmandapi_id = sockmandapi_id;
    udp_cbs[next_udpcb].uuid = uuid;
    udp_cbs[next_udpcb].mcap_address = mcap_address;
    udp_cbs[next_udpcb].current_address = get_my_addr();
    udp_cbs[next_udpcb].sourceport = sourceport;
    udp_cbs[next_udpcb].status = UDPACTIVE;
    udp_cbs[next_udpcb].protocol = IPPROTO_UDP;
    udp_cbs[next_udpcb].bufferedpacketsfirst = 0;
    udp_cbs[next_udpcb].bufferedpacketslast = 0;

    if(is_my_address(mcap_address))
        register_udp_capture(sourceport);
    else
        send_socket_state((struct connectioninfo *)&↩
           udp_cbs[next_udpcb], mcap_address);

    return next_udpcb++;
}
```

Listing 5.14 shows how UDP packets are sent. The function is called from the SOCKMAND API. The function creates an IP packet and creates the IP header and UDP header. The data that should be sent are copied into the packet and then the packet is sent to the IP controller.

Listing 5.14: UDP Sending

```
int udpwrite(int cb, const void *buf, int len, unsigned ↩
   int destaddr, short destport){
    if(len> MAXUDPPAYLOAD){
        printf("Major error, to large udp packet\n");
        return -1;
    }

    char buffer[ETHERNETMTU];
    bzero(&buffer, ETHERNETMTU);
    struct ipheader *ip = (struct ipheader *) buffer;
    struct udpheader *udp = (struct udpheader *) (ip + 1);
    char *data = (char*)(udp + 1);

    build_ip_packet(ip,
                    IPPROTO_UDP,
                    udp_cbs[cb].mcap_address,
                    destaddr,
                    sizeof(struct udpheader) + len);

    build_udp_header(udp, udp_cbs[cb].sourceport, destport↩
        , len);

    memcpy(data, buf, len);
    send_ip_packet(ip);
    return len;
}
```

Listing 5.15 shows how UDP packets are received. Notice that if the sockmandapi_id field is equal to -1, the packet is buffered for later delivery. The buffered packets are stored in a singly linked list, but we maintain the pointer to the last entry in the list to avoid looping through the entire structure when we want to add one element.

Listing 5.15: UDP Receiving

```
void udpread(struct ipheader *ip){
    struct udpheader *udp = (struct udpheader *) (ip + 1);
    int cb = get_udp_cb(ip);
    int payloadlen = ntohs(udp->uh_len) - sizeof(struct ↩
        udpheader);
    if(udp_cbs[cb].sockmandapi_id != -1){
        deliver_udp_data( udp_cbs[cb].sockmandapi_id,
                          (char*)(udp + 1),
                          payloadlen,
                          udp->uh_sport,
                          ip->ip_src);
    }
    else{
        struct bufferedpacket *t = malloc(sizeof(struct ↩
            bufferedpacket));
        t->next = 0;
        t->ip = malloc(ip->ip_len);
        memcpy(t->ip, ip, ip->ip_len);

        if(udp_cbs[cb].bufferedpacketsfirst == 0){
            udp_cbs[cb].bufferedpacketsfirst = t;
            udp_cbs[cb].bufferedpacketslast = t;
        }

        else{
            udp_cbs[cb].bufferedpacketslast->next = t;
            udp_cbs[cb].bufferedpacketslast = t;
        }
    }
}
```

Listing 5.16 shows how buffered packets are delivered to an application when it has called sm_rebuildsocket after arriving on the destination node.

Listing 5.16: Delivering Buffered Packets

```
void udp_deliverbufferdpackets(int cb){
    struct bufferedpacket* t = udp_cbs[cb].↩
        bufferedpacketsfirst;
    if(t==0)
        return;

    while(t!=0){
        struct ipheader* ip = t->ip;
        struct udpheader *udp = (struct udpheader *) (ip +↩
             1);
        int payloadlen = ntohs(udp->uh_len) - sizeof(↩
            struct udpheader);

        deliver_udp_data( udp_cbs[cb].sockmandapi_id,
                          (char*)(udp + 1),
                          payloadlen,
                          udp->uh_sport,
                          ip->ip_src);

        struct bufferedpacket* tmp = t->next;
        free(ip);
        free(t);
        t=tmp;
    }
}
```

5.3.6 TCP Controller

The TCP Controller component contains a TCP implementation in user-space. The implementation follows RFC 791 [33], RFC 5681 [1] and RFC 6298 [31]. These RFCs define the minimum functionality to fulfill the protocols requirements. TCP is a well-defined standard, but we claim that it is hard to implement due to many low level details. Although the TCP implementation is not a key element in this thesis, most of the time spent on the implementation was spent implementing TCP in user space.

Listing 5.17 shows the TCP Control Block structure. Notice that the fields that are similar to those in the UDP Control Block shown in Listing 5.12 is located at the same place in the structure. This is to simplify importing and exporting of socket states regardless of the transport layer protocol. In addition to the fields similar to those in the UDP control block, several fields needed to maintain the state of a TCP connection are included.

Since the TCP protocol itself is not a key element in this thesis, we will not dig in to the details of the standard implementation, but focus on the additional functionality added to support socket migration.

Listing 5.17: TCP Control Block

```
struct tcp_cb{
    int uuid;
    int sockmandapi\_id;
    int protocol;
    short status;
    unsigned short sourceport;
    unsigned int current_address;
    unsigned int mcap_address;

    unsigned int dest_address;
    unsigned short dest_port;
    unsigned int nextseq; //Next seqnum
    unsigned int lastack; //That we sent
    unsigned int lastremoteseq; //That we received
    unsigned int lastremoteack; //That we received
    unsigned short remote_windowsize;
    unsigned short congestion_window;
    int dupacks;
    unsigned short sstresh;

    int rtt; //Round trip time
    long int tmp; //Used to calculate rtt
    int RTO; //Retransmission timeout in milliseconds
    struct timeouts* packets;
    struct bufferedpacket* bufferedpackets;
};
```

Unlike the fixed UDP socket state size, the TCP socket state size is arbitrary due to possible unacknowledged packets that need to be migrated along with the socket. Listing 5.18 shows the data structure which holds information about unacknowledged packets. The elements are organized as a singly linked list. This linked list must be migrated with the unacknowledged packets since it contains important information such as when the resend timer times out and how many times we have tried to resend the packet. When exporting the data structure we serialize the linked list and the unacknowledged packets. The structure is deserialized and imported on the destination node.

Listing 5.18: TCP Timeout Structure

```
struct timeouts{
    long int timesout;
    short resent_times; //Number of times it has been ↩
        resent
    short packettype;
    struct ipheader* ip; //Pointer to the packet
    struct timeouts* next; //0 if last
};
```

When importing the data structure a problem occurs. The machine clock on the source node and destination node are most likely not synchronized. Therefore, the value in the timesout field in the TCP timeout structure has lost all meaning. We therefore chose to update all the timesout fields in the data structure on the destination node with the current time on the destination node plus the retransmission timeout value located in the TCP control block. This is shown in Listing 5.19.

Listing 5.19: TCP importing timeouts

```
void deserialize_timeouts(char* buf, int cb){
    struct timeouts* t = (struct timeouts*)buf;
    struct timeouts* prev = 0;
    while(1){
        int last = 0;
        if(t->next==0)
            last = 1;

        struct timeouts* tmp = malloc(sizeof(struct ↩
           timeouts));
        tmp->next = 0;
        int iplen = t->ip->ip_len;
        tmp->ip = malloc(iplen);
        memcpy(tmp, t, sizeof(struct timeouts));
        t += sizeof(struct timeouts);
        memcpy(tmp->ip, t, iplen);
        t += iplen;

        //Set a new timeout value
        tmp->timesout = getmillis() + tcp_cbs[cb].RTO;

        if(prev == 0)
            tcp_cbs[cb].packets = tmp;
        else
            prev->next = tmp;

        prev = tmp;
        if(last)
            break;
    }
}
```

Alternative and possibly better approaches for determining the new value of the timesout field should be investigated, but due to the scope of this thesis, we leave that for future work.

5.3.7 Libpcap Handler

Listing 5.20 shows how we initialize the Libpcap Handler. pcap_fd is the file descriptor where new captured packets will arrive. Notice that we initialize the library with a dummy filter to prevent all packets on the node from being captured. When new sockets are created, we append a new filter argument to the filter character array and call the set_filter function again.

Listing 5.20: Initializing libpcap

```
int libpcaphandler_init(){
    char *dev;

    dev = pcap_lookupdev(errbuf);
    if(dev == NULL){
        fprintf(stderr,"%s\n",errbuf);
        return(-1);
    }

    pcap_lookupnet(dev,&netp,&maskp,errbuf);

    descr = pcap_open_live(dev,BUFSIZ,0,-1,errbuf);
    if(descr == NULL){
        printf("pcap_open_live(): %s\n",errbuf);
        return(-1);
    }

    pcap_fd =  pcap_get_selectable_fd(descr);
    if(pcap_fd < 0){
        fprintf(stderr,"Error getting pcap selectable\n");
        return(-1);
    }

    bzero(filter,MAX_FILTERSIZE);
    memcpy(filter, "tcp dst port 1", 15);
    //Setting a dummy filter to stop all packets from ↩
        arriving
    set_filter();

    printf("pcap init done\n");
    return 0;
}
```

As mentioned in Section 4.7.7, we need to drop TCP reset packets sent from the OS kernel when we are implementing TCP in user space. Listing 5.21 shows how a filter is added that drops TCP reset packets from a given source port.

Listing 5.21: Dropping reset packets

```
void droprstpackets(short sport){
    int port =  ntohs(sport);
    char comm[1000];
    snprintf(comm, sizeof(comm), "iptables -A OUTPUT -p ↩
       tcp --sport %d --tcp-flags rst rst -j DROP", port)↩
       ;
    if(system(comm)<0){
        perror("System() ");
        exit(-1);
    }
}
```

5.3.8 Migrator API

As mentioned in Section 4.7.11, this component is only simulated in our implementation. We let users define their preferred MCAP in as an argument when starting SOCKMAND. To migrate a socket, users can use the keyboard interface to simulate a call to the migrate_to function.

Chapter 6

Evaluation

In this chapter we present the evaluation of SOCKMAND described in Chapters 4 and 5. Section 6.1 describes the goals of our evaluation. We present the common performance evaluation approaches in Section 6.2. The different performance metrics we use are described in Section 6.3. In Section 6.4 we present the different factors we vary. Our evaluation setups are presented in Section 6.5 and the results of them are presented in Section 6.6. Finally, in Section 6.7 we discuss the results presented in Section 6.6.

6.1 Evaluation Goals

The main goals of our evaluation are to determine if SOCKMAND can be used for real-time applications and to evaluate the load of SOCKMAND in various scenarios. We present four specific goals for our evaluation:

- **Goal 1:** To determine if SOCKMAND can be used for real-time applications, see Section 2.2.3, we need to analyze bandwidth restrictions and additional delay introduced by SOCKMAND and verify that SOCKMAND fulfills the metric requirements presented in Section 4.1. SOCKMAND should at least support a 400 kbit/s and ideally a 1.5 Mbit/s data stream in both directions simultaneously. An MCAP must therefore be able to process the double of these numbers, minimum 800 kbit/s and ideally 3.0 Mbit/s. SOCKMAND must not introduce an additional delay which causes the round-trip time to exceed 200 ms. We therefore look at the impact SOCKMAND has on round trip latency, both the delay during regular operation and the delay introduced during a socket migration and if SOCKMAND limits the bandwidth.

- **Goal 2:** We evaluate the CPU load of SOCKMAND and packet loss in MCAPs on various heterogeneous devices. We us the results to determine which devices that can fulfill the requirements presented in Section 4.1.

- **Goal 3:** Bonola and Salsano [6] show that packet loss is closely related to the CPU load, we therefore investigate the relationship between CPU load and packet loss in SOCKMAND. We compare our results with the results presented in the performance evaluation of the UPMT solution [6] which has a similar architecture.

- **Goal 4:** We investigate the difference in terms of CPU load when using the Rawsocket Handler and the Libpcap Handler. When a node is acting as an MCAP between a mobile application and a legacy application, packets flow through the MCAP in two directions. In one direction the Rawsocket Handler is used, in the other direction the Libpcap Handler is used. If there are differences in the CPU load, packet loss may occur faster in one direction than in the other, and the MCAP will become asymmetric.

- **Goal 5:** We investigate if there is any difference in CPU load when using SOCKMAND and when using regular sockets.

6.2 Analysis Approach

When doing performance evaluation on a solution to a problem, there are three different approaches that must be considered, namely modeling, simulation and measurement [13].

Modeling is the theoretical approach used in performance evaluation. We present a mathematical model of the performance of the solution and apply mathematical methods to evaluate the performance. Modeling is good for simple solutions, but if the solution is complex, the model also becomes more complex.

Simulation is testing a solution in a controlled simulated environment. This is suitable if a simulation environment exists for the domain of the solution. When doing a simulation, we have control over all the factors in the environment. This means that if we are doing two simulations with the same parameters, we will get the same result. Simulations are good if the solution is easy to implement in a simulator environment. For solutions depending on many factors, such as hardware and operating systems, accurate simulations are difficult to achieve.

When simulations are difficult due to many dependencies on uncontrollable factors, our third choice is to do measurements. A measurement is done on a real implementation of the solution in a real life environment. Due to the real life environment, uncontrollable factors can affect our measurements. To minimize this, we should terminate unnecessary background processes which affect CPU load and available network bandwidth. Unlike simulations, measurements are hard to reproduce, especially when it is not run on the same equipment. Other side effects that are coming from the OS, other applications and network load can alter the results.

Our analysis of SOCKMAND uses both analytical modeling and measurements. Which approach we are using depends on the characteristics of the metric we analyze. Modeling or simulating CPU load is difficult due to many factors which is hard to model or simulate. We therefore choose to measure the CPU load using our implementation of SOCKMAND running on the different environments described in Section 6.4.3

We present a simple model of the round-trip latency since the round-trip latency consists of easily definable steps. We also do measurements on the implementation of SOCKMAND which we compare with the model. By comparing the measurement with the model, we can quantify the factors presented in the model.

We choose to model the socket migration time since it consists of a few easily definable steps, namely sending the socket state to the destination node and in one case sending a message to the MCAP. Doing a measurement of the socket migration time will only produce results valid for a particular scenario, a model will however explain the socket migration time in details valid for all scenarios.

Modeling or simulating packet loss is difficult due to many factors which are hard to model or simulate. We therefore choose to measure the packet loss using our implementation of SOCKMAND.

6.3 Evaluation Metrics

We evaluate SOCKMAND against a set of different performance metrics, The performance metrics are *CPU load*, *round-trip time*, *socket migration time* and *packet loss in the MCAP*. Each of the metrics and its analysis approach are described in the following subsections

6.3.1 CPU Load

CPU load is measured using the top command on UNIX. Top measures CPU load in percentage of available CPU time. Using top, we can measure both the mobile application and SOCKMAND at once. Notice that when top measures the CPU load of an application, it reports the percentage used of one CPU. Single-threaded applications can therefore only achieve 100% CPU load although there is still available CPU time in other processors. Multi-threaded applications can however achieve more than 100% CPU load; for instance in a dual core environment top can report that an application uses up to 200% CPU load.

When measuring the CPU load using top, we need to make sure that we get the correct average CPU load. This is done by starting the task we want to measure and the top tool at the same time. We run the task for 90 seconds and then stop top immediately. The beginning and the end of the measurement will include measurements which reports the standby CPU load and up to the average CPU load. These numbers do not reflect the actual average CPU load and we consider them to be outliers. We therefore ignore the first and last 10 seconds from the output before we calculate the average CPU load. By doing so, we remove the impact that the beginning and the end of the measurements have on the average CPU load.

The applications we use to introduce CPU load are presented in Sections C.2, C.3 and C.4 in Appendix C.

6.3.2 Round-Trip Time

We define round-trip latency as the time it takes for one packet to be sent from a mobile application to a legacy application, the packet is then replied instantly by the legacy application and when the reply packet arrives at the mobile application, we have calculated the round-trip latency. Unlike the ping tool, which operates on the network layer, our measurement includes transport layer overhead and context switches into a user space application. Our measurement application designed to measure the round-trip latency is shown in Section C.1 in Appendix C.

6.3.3 Socket Migration Time

Socket migration time is the time it takes from the migrator asks SOCKMAND to migrate a socket from node A to node B until SOCKMAND on node B is ready to accept a rebuildsocket() call from a mobile application. We define the socket migration time as the time it takes to execute the migratesocket() function on node A. Note that this time excludes any time spent migrating the mobile application itself.

6.3.4 Packet Loss in the MCAP

When a node receives more packets than it can process, packet loss occur. We visualize the packet loss by plotting a line with the number of packets sent per second on the x axis and the number of packets received at the destination on the y axis. If the rates are the same, there is no packet loss and the line is linear. If the sending rate is larger than the receiving rate, packet loss has occurred.

We create an application which sends UDP packets at a given rate to a destination via SOCKMAND acting as an MCAP. Another application receives these packets and reports the number of packets it receives per second. To get an average result, we run the applications for 90 seconds and report the average number of packets received per second. Our measurement applications designed to measure the packet loss are presented in Sections C.2 and C.3 in Appendix C.

6.4 Evaluation Factors

When evaluating SOCKMAND we vary some factors to see what effect changing the factor have on the results. The factors are *packet size*, *packets per second*, *node specifications* and *the packet direction through the MCAP*. Each of the factors are described in the following subsections.

6.4.1 Packet Sizes

We vary between a small packet size and a large packet size to see if the packet size has any impact on the CPU load. Since we have not implemented support for IP fragmentation, the packet size is limited by the

Ethernet Maximum Transmission Unit (MTU). As mentioned in Section 4.6, the maximum UDP payload size we can send is 1444 Bytes.

6.4.2 Packets per Second

We vary the number of packets per second that are sent through SOCKMAND to see what effect the number of packets per second has on the CPU load and packet loss. The number of packets we can send per second is limited by the bandwidth and the packet size. Our test environment has a 1000 Mbit/s bandwidth which limits the packets we can send per second. This formula gives us the number of packets per second (μ) based on a bandwidth (β) in Mbit/s and a packet size (γ) in bytes.

$$\mu = \frac{10^6\beta}{8\gamma}$$

	100 Mbit/s	1000 Mbit/s
128 Bytes	97656	976563
156 Bytes	80128	801282
1428 Bytes	8754	87535
1456 Bytes	8585	85852

Table 6.1: Maximum packets per second based on packet size and bandwidth

Table 6.1 shows some calculated numbers that are necessary to understand the results described later in this chapter.

6.4.3 Node Specifications

We vary the node specifications between a high-end desktop computer, a netbook and a high performance computer connected with a 1000 Mbit/s line. We vary the node specifications to get a view of the CPU load of SOCKMAND on different computers with different processing powers. The high performing computer is used to count packets that arrive, reply packets that are used for round-trip time measuring and to send packets at a given rate. No measurements of CPU load are done on the high performing computer. The specifications of the different nodes are shown in Table 6.2.

The netbook has significantly lower specifications than the desktop computer. The netbook's specifications are in the range of high-end mobile devices such as cellphones and tablets.

6.4.4 Packet Direction Through the MCAP

When a node is acting as an MCAP between a mobile application and a legacy application, packets flow through the MCAP in two directions. When packets are sent from a mobile application to a legacy application through an MCAP, packets first arrive at the Tunnel Handler in the MCAP. The packets are then forwarded to the SOCKMAND Core which forwards the

	Desktop Computer	Netbook	High Performing Computer
Model	HP Compaq 8100 Elite	MSI Wind U-100	
Operating System	Ubuntu 11.04 64 bit	Ubuntu 11.11 32 bit	Linux 2.6.1 x86_64
CPU	Intel Core i7 CPU 870 @ 2.93GHz	Intel Atom CPU N270 @ 1.60GHz	64 x Intel Xeon CPU L7555 @ 1.87GHz
Cores	4	1	8
Threads	8	2	
Memory	4 GiB	2 GiB	123 GiB
Ethernet interface	82578DM Gigabit	RTL8101E	
Ethernet capacity	1 Gbit/s	100 Mbit/s	

Table 6.2: Node Specifications

packet to the Rawsocket Handler. The packets are then sent to the legacy application using raw sockets.

In the other direction when packets are sent from a legacy application to a mobile application through an MCAP, packets first arrive at the Libpcap Handler. The packets are then forwarded to the SOCKMAND Core which forwards the packets to the Tunnel Handler. The packets are then sent to the SOCKMAND on the node of the mobile application using UDP tunnels.

As we can see from the description above and from Figure 4.4, we are using different components and technologies depending on the direction a packet is sent through an MCAP. We therefore investigate if there are any differences in terms of CPU load for the different directions.

6.5 Evaluation Setups

We use different evaluation setups and scenarios to analyze the different performance metrics. The evaluation setups are *CPU load of SOCKMAND and mobile application*, *CPU load and packet loss on MCAP*, three *socket migration time* scenarios and *round-trip time*. The different setups are described in the following subsections.

6.5.1 CPU Load of SOCKMAND and Mobile Application

The first scenario is where a mobile application MA on node A communicates with a legacy application LA on node B with the MCAP located on node A. This scenario is shown in Figure 6.1.

We measure the following metrics.

- CPU Load on SOCKMAND on node A
- CPU Load on the mobile application on node A

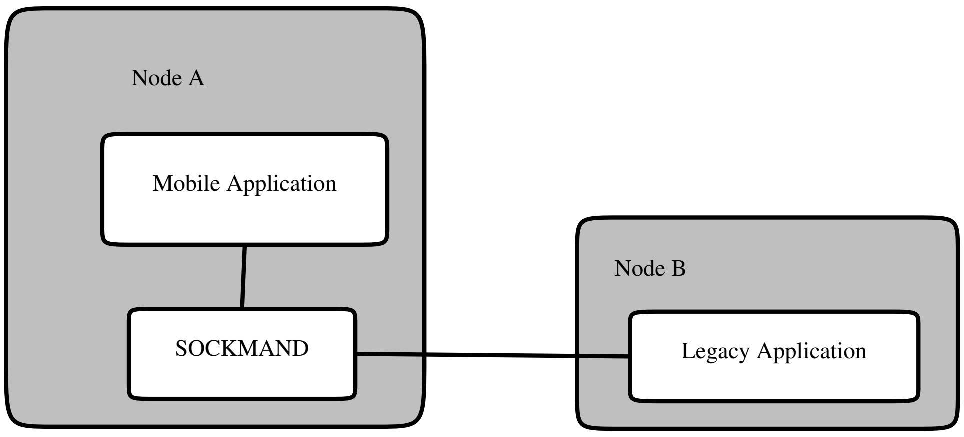

Figure 6.1: Node setup CPU load by SOCKMAND and Mobile Application

We vary these factors to see what effect they have on the CPU load of SOCKMAND and the mobile application.

- Node specifications of node A
- Packet size
- Packets per second

When running this experiment, the legacy application on node B sends UDP packets to node A. The legacy application sends packets with the given packet size at a specific rate. The packets are sent using regular POSIX sockets. When packets arrive at node A, they are captured by the Libpcap Handler. The packets are then sent to the SOCKMAND Core which sees that they are intended for a local mobile application. The packets are then sent to the IP controller which sends the packets to the UDP Controller which delivers them to the SOCKMAND API. The SOCKMAND API sends the packets to the mobile application using UNIX Domain sockets. The mobile application then reads the packet using the sm_recvfrom function. No further processing is done in the mobile application which only calls sm_recvfrom in an infinite loop.

We vary the payload of the packets between 100B and 1400B. Note that only a UDP header and a IP header are added, so the total packet size is 128B and 1428B. The number of packets per second varies between 1000 and the maximum packets number of packets we are able to send per second on a given link or the maximum number of packets we are able to send before the CPU load reaches 100%.

In addition we measure the CPU load of a legacy application that does the same as the mobile application, namely receiving packets in an infinite loop. This measurement provides a basis for comparison between a user space socket implementation and a kernel socket implementation.

6.5.2 CPU Load and Packet Loss on MCAP

We measure the packet loss on a node used only as an MCAP. The MCAP will receive tunneled packets and forward them through the MCAP component. The incoming number of packets per second is varied. The number of packets that arrive at the destination is measured. We assume that the packets that do not arrive at the destination were dropped in the MCAP. In addition the CPU load is measured. The packet size is varied to see if this has any impact on the CPU load and packet loss. The scenario is shown in Figure 6.2.

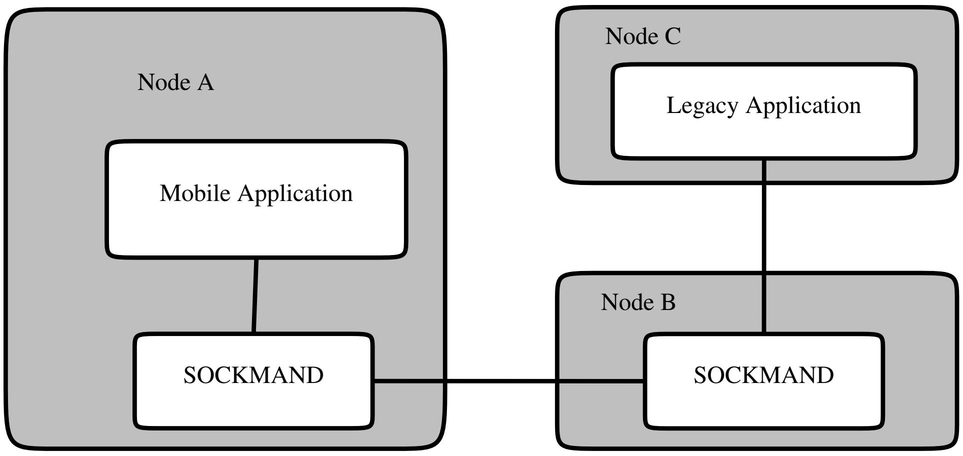

Figure 6.2: Node setup CPU load and Packet loss on MCAP

We measure the following metrics. The incoming and outgoing packets are measured so we can calculate the packet loss.

- CPU Load on SOCKMAND on node B
- Incoming packets on node B
- Outgoing packets from node B

We vary these factors to see what effect they have on the CPU load of SOCKMAND and the packet loss.

- Node specifications of node B
- Packet size
- Packets per second
- Packet direction through the MCAP

We run this experiment with two different payload sizes, 100B and 1400B. When packets are between node A and node B, there are two IP and UDP-headers in the packet. The total packet size is therefore 156B and 1456B. When packets are sent between node B and node C, no matter of the direction, the packets include one IP header and UDP header. The total packet size is then 128B and 1428B. The number of packets per second varies between 1000 and the maximum number of packets we are able to send per second on a given link or the maximum number of packets we are able to send before the CPU load reaches 100%.

6.5.3 Socket Migration Time

We chose to model the socket migration time because it consists of a few simple actions. We do a measurement to verify our model. When modeling the socket migration time, there are three different scenarios to consider. These scenarios are the same as described in Section 4.5.

Scenario 1

The first scenario is when the socket is migrated from node A to node B where node A is the MCAP of that socket. The socket is communicating with a legacy host on node C. The scenario is shown in Figure 6.3. After the migration, one endpoint of the socket is located on node B with the MCAP of that socket on node A. The other endpoint of the socket is still located on node C.

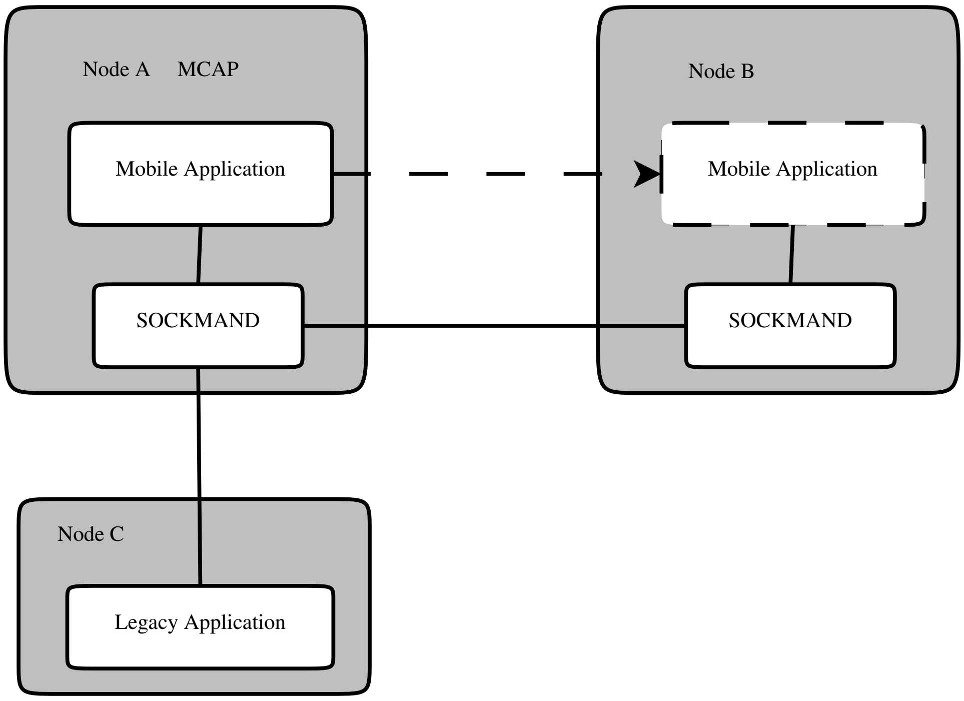

Figure 6.3: Measuring Socket Migration Time 1

Scenario 2

The second scenario is when the socket is migrated from node A to node B where node B is the MCAP of that socket. The socket is communicating with a legacy host on node C. The scenario is shown in Figure 6.4. After the migration, one endpoint of the socket is located on node B with the MCAP of that socket on node B. The other endpoint of the socket is still located on node C.

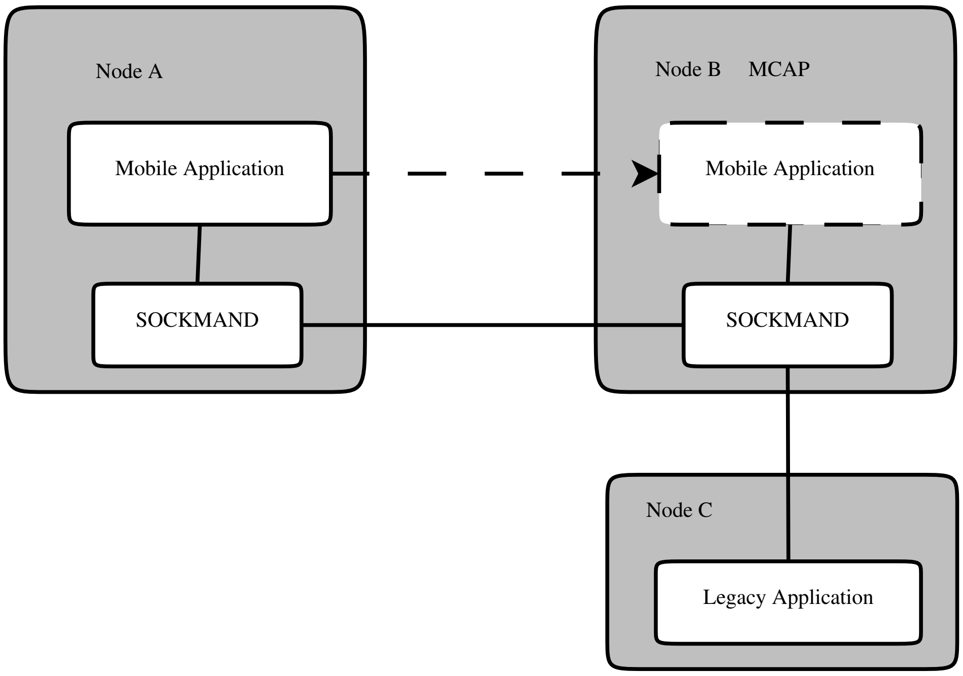

Figure 6.4: Measuring Socket Migration Time 2

Scenario 3

The third scenario is when the socket is migrated from node A to node B where node C is the MCAP of that socket. The socket is communicating with a legacy host on node D. The scenario is shown in Figure 6.5. After the migration, one endpoint of the socket is located on node B with the MCAP of that socket on node C. The other endpoint of the socket is still located on node D.

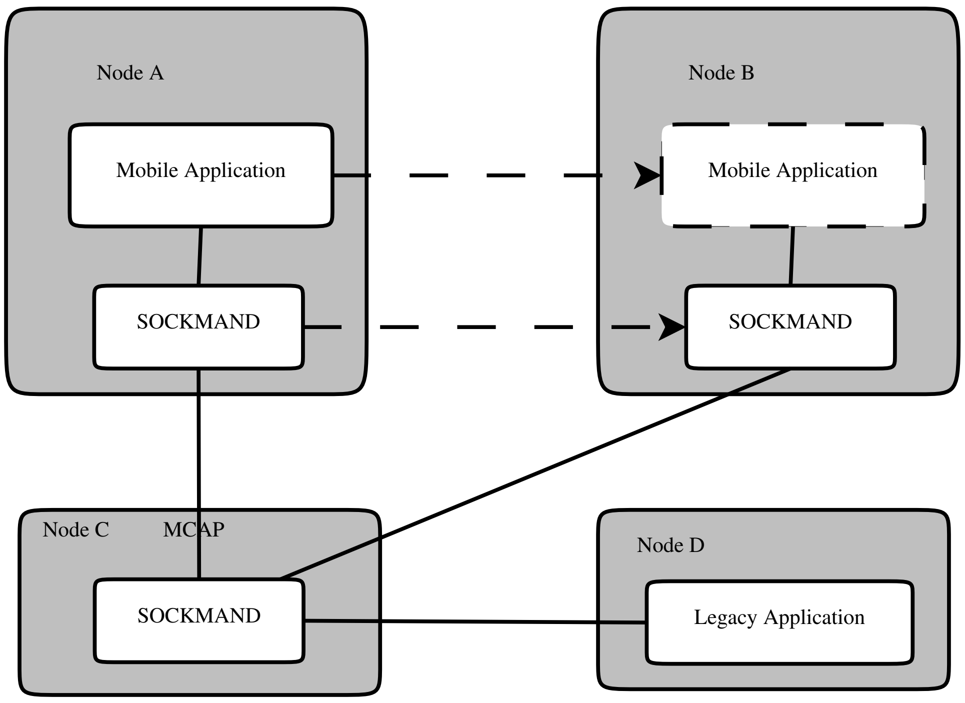

Figure 6.5: Measuring Socket Migration Time 3

6.5.4 Round-Trip Time

In this scenario we measure the round-trip time between two nodes A and B. Several factors are varied to see what effect they have on the round-trip time:

- Node specifications of node A
- Whether the MCAP is on node A or node C
- Node specifications of MCAP node C
- Whether SOCKMAND is used or not

Figure 6.6 shows the RTT reported by the ping tool between the nodes used in our test environment. The RTT is the average of 100 ping requests. The results shown in Section 6.6.4 are the average of 100 measurements.

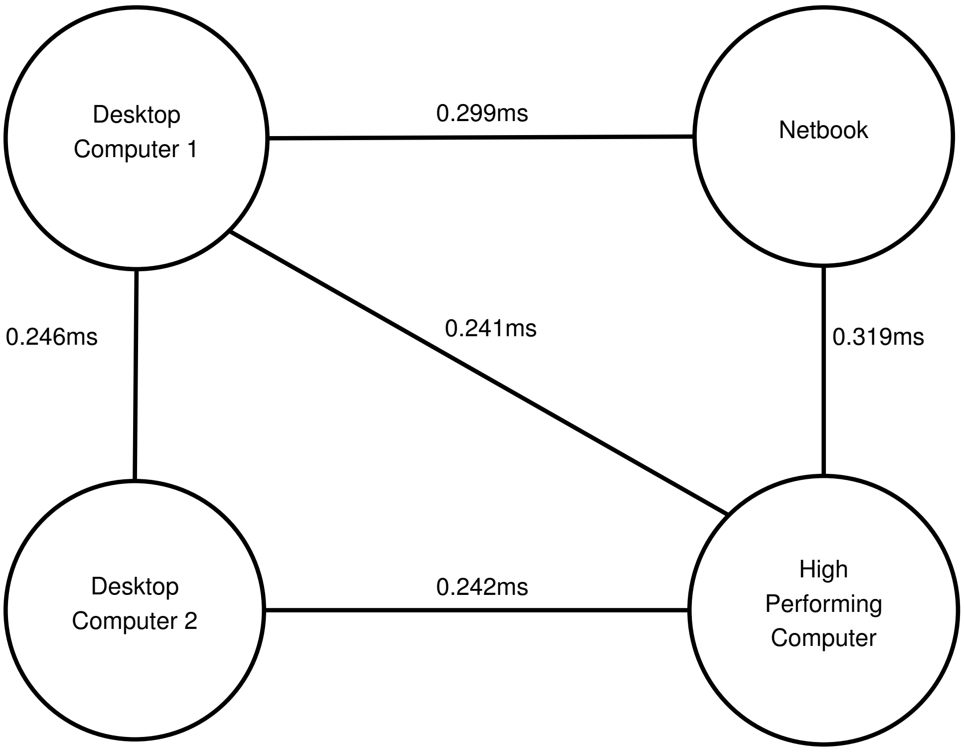

Figure 6.6: Ping results

6.6 Results

This section presents the results of the evaluations described in Section 6.5.

6.6.1 CPU Load of SOCKMAND and Mobile Application

Figure 6.7 shows the CPU load by SOCKMAND and a mobile application based on the number of packets per second with 100B payload size. The

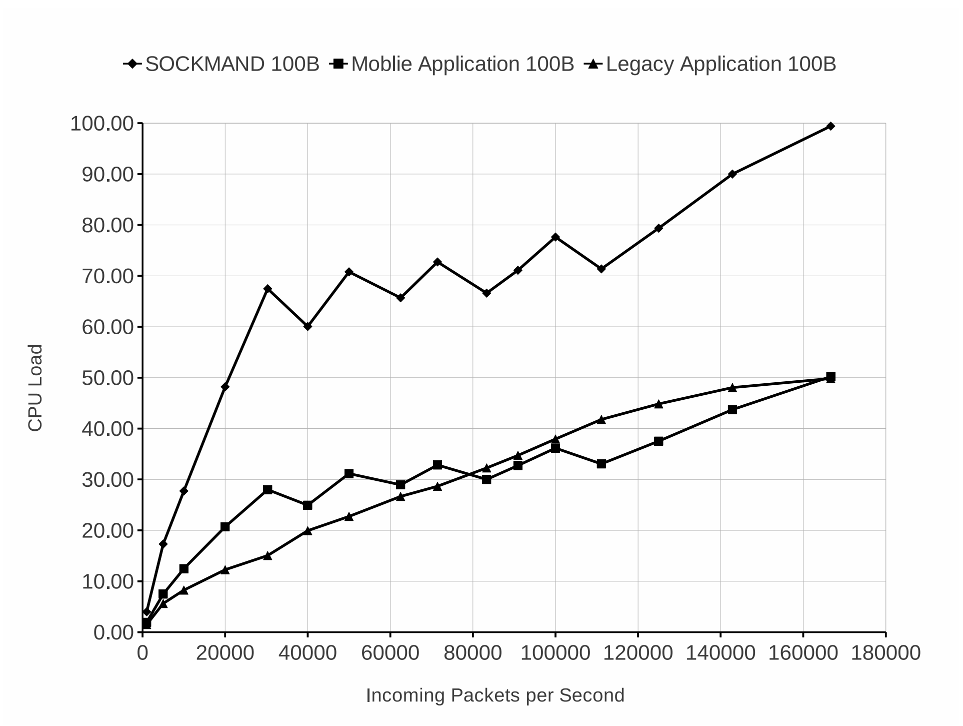

Figure 6.7: CPU load by SOCKMAND and Mobile Application on Desktop Computer with 100B payload

CPU load by a legacy application that uses regular sockets is also included in the graph for comparison. From Figure 6.7 we can see that there is little difference between the CPU load of a mobile application and a legacy application doing the same task.

When a legacy application is using regular sockets, all the CPU load is by the legacy application. When a mobile application is using SOCKMAND, the total CPU load is the sum of the CPU load of SOCKMAND and the mobile application.

For each packet arriving at the node when we are running the mobile application and SOCKMAND, the packet must be transferred from the kernel to SOCKMAND, from SOCKMAND to the kernel and from the kernel to the mobile application. This is a total of three copies of the packet. When packets are delivered directly to a legacy application, it only needs to be copied once from kernel to the legacy application. This means that when a mobile application is using SOCKMAND, each packet must be copied three times more than when a legacy application uses regular sockets.

This pattern is visible in the results. The combined load of SOCKMAND and the mobile application for a given packet rate is slightly above three times as large as the CPU load of a legacy application doing the same task. We see that the CPU load of the mobile application and the legacy application is approximately the same. We observe that the CPU load of the mobile application fluctuates while the CPU load of the legacy application is more linear, we do not have a clear explanation for the fluctuation, it may be related to the UNIX domain sockets. However the CPU load of SOCKMAND is slightly above twice the CPU load of the mobile application.

When calling the sm_recvfrom function, there are two reads of the UNIX domain socket. As mentioned in Section 5.3.4, the mobile application first reads the metadata of the packet, and then the payload itself. This may explain why the CPU load is slightly larger than three times as larger.

The same pattern as in Figure 6.7 is visible in Figure B.5, Figure B.6 and Figure B.7 in Appendix B with different packet sizes and node specifications.

6.6.2 CPU Load and Packet Loss on MCAP

Figure 6.8 shows the packet loss on node B. In this experiment the packets on node B are received using the Tunnel handler and sent using the raw socket handler. Node B is a desktop computer. The measurement of 1400B packets stops at 85852 packets per second since the bandwidth is fully loaded. Figure 6.9 shows the CPU load of SOCKMAND on node B in the same experiment. Each measurement is running for 90 seconds with the first and last 10 seconds removed from the result.

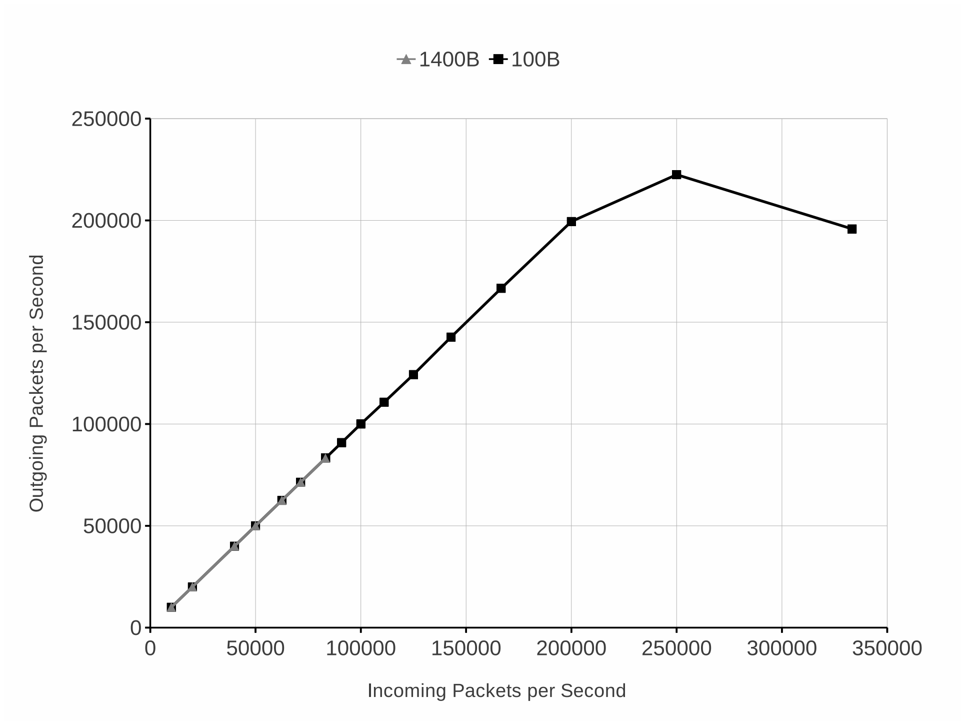

Figure 6.8: Desktop Computer Packet loss on MCAP from tunnel to rawsock

In Figure 6.9 we observe that the packet size does not affect the CPU load. For a given number of packets per second, the CPU load is the same for packets with different payload sizes. We are unable to reach a CPU load of 100% when we are using packet sizes of 1456B. This is because we are not able to send the number of packets needed since we are limited by our

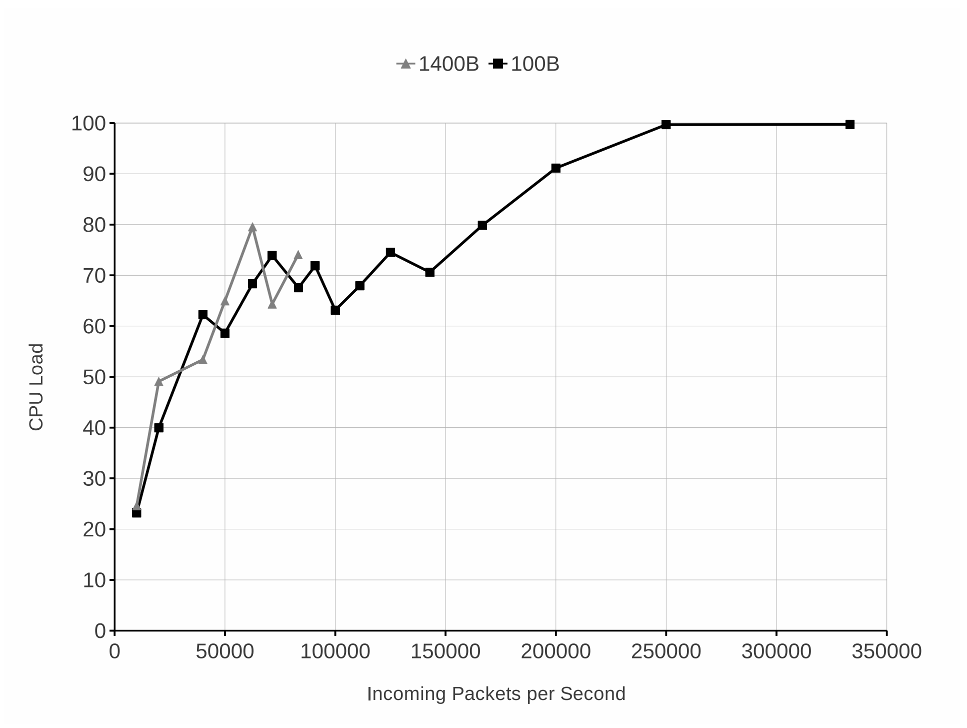

Figure 6.9: Desktop Computer CPU Load on MCAP from tunnel to rawsock

1000Mbit/s line. We see from Table 6.1 that the total number of packets we are able to send with this packet size is 85851.

When we compare the two graphs, we see that packet loss for 100b payload packets starts when the CPU load reaches 100%. No packet loss occur when we use 1400B payload packets since we are not able to reach 100% load with these packet sizes due to our bandwidth limit.

Figure 6.10 shows the packet loss on node B. In this experiment the packets on node B are received using the Tunnel handler and sent using the raw socket handler. Node B is a netbook computer. The line which represents 1400B payload sizes flattens out at 8585 packets per second since the bandwidth is fully loaded. Figure 6.11 shows the CPU load of SOCKMAND on node B in the same experiment. Each measurement is running for 90 seconds with the first and last 10 seconds removed from the result.

When comparing the CPU load on the netbook versus the CPU load on the desktop computer, we see that the CPU load for 100B payload packets reaches 100 much earlier than on the desktop computer due to the slower CPU. Note that the Ethernet card of the netbook only supports 100 Mbit/s. We see from Table 6.1 that the total number of packets the netbook is able to receive per second with a payload size of 100B is 80128. From the same table we see that the total number of packets the netbook is able to receive per second with a payload size of 1400B is 8585.

The netbook is not able to process packets of 100B payload size when the Ethernet card is receiving on full capacity, but the netbook is able to

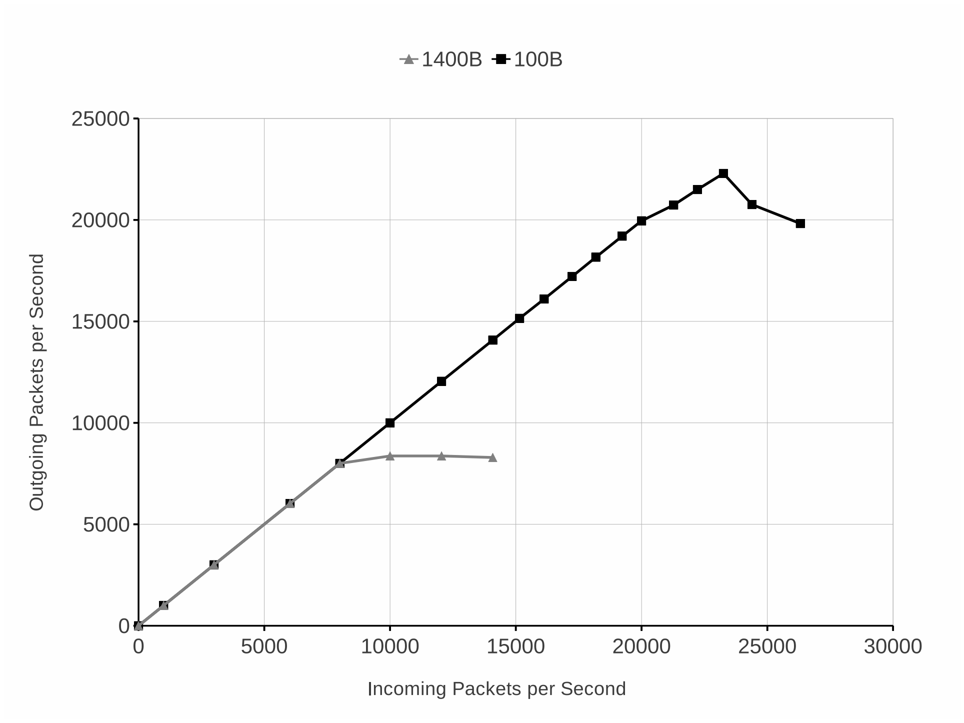

Figure 6.10: Netbook Packet loss on MCAP from tunnel to rawsock

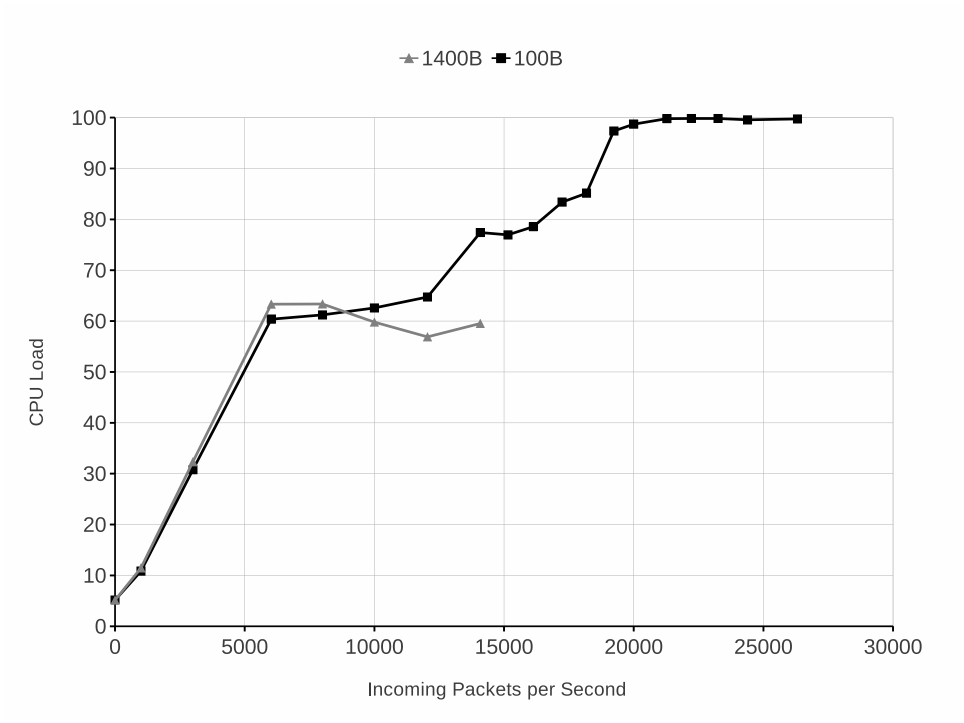

Figure 6.11: Netbook CPU Load on MCAP from tunnel to rawsock

process packets of 1400B payload size when the Ethernet card is receiving on full capacity. This is because sending packets with a 100B payload at

100 Mbit/s creates 80128 packets per second. On the other hand, sending packets with a 1400B payload at 100 Mbit/s creates 8585 packets per second. The CPU load is dependent on the number of packets per second, not the amount of data per second, so although the same amount of data is sent, we can only process this amount of data if the packets are large enough.

Figure B.2 shows the packet loss on node B. In this experiment the packets on node B are received using the Libpcap Handler and sent using the Tunnel Handler Node B is a Desktop computer. Figure B.1 shows the CPU load of SOCKMAND on node B in the same experiment.

Figure B.4 shows the packet loss on node B. In this experiment the packets on node B are received using the Libpcap Handler and sent using the Tunnel Handler Node B is a netbook computer. Figure B.3 shows the CPU load of SOCKMAND on node B in the same experiment.

The results from the experiments where packets on node B are received using the Libpcap Handler and sent using the Tunnel Handler do not differ from the results that we see when packets on node B are received using the Tunnel handler and sent using the Rawsocket Handler. It shows us that the direction of a packet through an MCAP does not affect the CPU load or packet loss. This means that the performance of receiving packets with libpcap is equal to the performance of sending using raw sockets. This ensures that the performance of the MCAP is symmetric, meaning that the direction of packets through an MCAP does not have an impact on performance.

6.6.3 Socket Migration Time

To migrate a socket, the MCAP must be notified and the destination node must receive the state of the socket. Information is passed between the source node, the MCAP and the destination node using a regular TCP/IP socket. We assume that the TCP/IP sockets are already connected between all the nodes in the Migration Community. Because of this, no three way handshakes must be done when migrating a socket.

For TCP connections through SOCKMAND, any unsent packets or unacknowledged packets must be sent along with the socket state to the destination node. The size of this data will vary depending on the characteristics of that particular TCP connection.

We will only need to notify the MCAP about the new location of the socket endpoint when the MCAP is neither source nor destination. The size of the message that is sent to the MCAP is 12 bytes. This will always fit within one TCP packet. In addition a reply is sent from the MCAP that acknowledges that the message was received.

When sending the socket state to the destination node, the size of the socket state will vary depending on whether it is a TCP or UDP socket. The size of a UDP socket state is 32 bytes and the size of a TCP socket state is 96 bytes.

Modeling the time used to send a given number of bytes over a TCP connection is difficult without analyzing that specific TCP implementation.

In addition to the time spent transferring data, we also need to consider the time spent importing the socket state and the context switches when sending and receiving messages. These times will be so small that they are negligible.

Scenario 1

The first scenario is when the socket is migrated from node A to node B where node B is the MCAP of that socket. The socket is communicating with a legacy host on node C. In this scenario it is not necessary to send any message to the MCAP since the MCAP is the destination node. The total time spent migrating the socket can be described as:

$c + d + e + f$

Where c is the time spent exporting the state on the source node, d is the time spent sending the state to the destination node, e is the time spent importing the state on the destination node and f is the time spent sending an acknowledgement back from the destination node to the source node.

If the state does not have many unacknowledged packets that must be sent along with the state, a good approximation of the socket migration time will be the round-trip time between node A and B.

Scenario 2

The second scenario is when the socket is migrated from node A to node B where node A is the MCAP of that socket. The socket is communicating with a legacy host on node C. In this scenario it is not necessary to send any message to the MCAP since the MCAP is the source node. Because of this, the model is the exact same as in Scenario 1.

Scenario 3

The third scenario is when the socket is migrated from node A to node B where node C is the MCAP of that socket. The socket is communicating with a legacy host on node D. Since the MCAP is neither source nor destination node, the MCAP must be notified of the migration.

The total time spent migrating the socket can be described as:

$a + b + c + d + e + f$

Where a is the time spent sending a migrating to message from the source node to the MCAP, b is the time spent sending sending an acknowledgement back from the MCAP to the source node. c is the time spent exporting the state on the source node, d is the time spent sending the state to the destination node, e is the time spent importing the state on the destination node and f is the time spent sending an acknowledgement back from the destination node to the source node.

If the state does not have many unacknowledged packets that must be sent along with the state, a good approximation of the socket migration time will be the sum of the round-trip time between node A and B and the sum of the round-trip time between node A and C.

6.6.4 Round-Trip Time

The measurements are divided into three scenarios. RTT between node A and B with the MCAP at node A is shown in Table 6.3, RTT between node A and B without SOCKMAND is shown in Table 6.4 and RTT between node A and B with the MCAP at node C is shown in Table 6.5.

Node A	Node B	Average RTT	Standard Deviation
DC 1	HPC	0.395ms	0.045ms
Netbook	HPC	0.667ms	0.044ms

Table 6.3: RTT between node A and B with the MCAP at node A

Node A	Node B	Average RTT	Standard Deviation
DC 1	HPC	0.280 ms	0.040 ms
Netbook	HPC	0.347 ms	0.054 ms

Table 6.4: RTT between node A and B without SOCKMAND

When we compare the results in Table 6.3 with Table 6.4 we see that the RTT increases when using SOCKMAND on node A with the MCAP at node A. This increase comes from the additional copies of packets as mentioned in Section 6.6.1. We can calculate the difference in RTT when SOCKMAND is used and when regular sockets are used by subtracting the average RTT in Table 6.4 from the average RTT in Table 6.3. The difference in RTT for a desktop computer is:

$0.395ms - 0.280ms = 0.115ms$

The difference in RTT for a netbook is:

$0.667ms - 0.347ms = 0.320ms$

We see that the overhead is almost three times as large in the netbook as in the desktop computer. This difference is explained by the slower CPU in the netbook. The overhead time is however small when compared to RTT experienced even when pinging nodes in other regions of Norway. The RTT reported by ping between the University of Oslo and the Norwegian University of Science and Technology in Trondheim 500 km away on April 24th was on average 8.165 ms with a standard deviation of 0.123 ms.

By comparing the results in Table 6.4 with the RTT reported by the ping tool shown in Figure 6.6 we see the difference in RTT reported by our UDP RTT tool and the RTT reported by the ping tool which operates at the IP layer. We calculate this difference by subtracting the average RTT reported by the ping tool shown in Figure 6.6 from the average RTT reported by our UDP RTT measurement tool in Table 6.4. We use the difference in reported RTT by the two measurement tools later in this section to calculate the overhead introduced by the MCAPs.

The difference in reported RTT where node A is a desktop computer is:

$0.280ms - 0.241ms = 0.039ms$

The difference in reported RTT where node A is a netbook is:

$0.347ms - 0.299ms = 0.048ms$

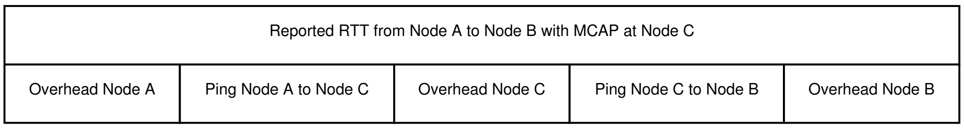

Figure 6.12: Factors of RTT shown in Table 6.5

Node A	Node B	Node C (MCAP)	Avg RTT	Std dev
DC 1	HPC	DC 2	0.671 ms	0.069 ms
DC 1	HPC	Netbook	0.774 ms	0.055 ms
Netbook	HPC	DC 1	0.859 ms	0.052 ms

Table 6.5: RTT between node A and B with the MCAP at node C

The round-trip times presented in Table 6.5 are the sum of the factors presented in Figure 6.12 The sum of the overhead in node A and node B is calculated in the previous paragraph. The ping times between the nodes are shown in Figure 6.6. We can calculate the overhead in Node C, the MCAP, for different nodes using this formula:

```
Overhead in MCAP = (RTT from node A to Node B) -
(overhead in node A) - (ping from node A to node C) -
(ping from node C to node B) - (overhead in node B)
```

Using the formula we can calculate the overhead in desktop computer 2 when it is acting as an MCAP. The RTT from node A to node B is located in the first row of Table 6.5. The sum of overhead in node A which is a desktop computer and the overhead in node B is calculated above and is 0.039 ms. The ping between desktop computer 1 and desktop computer 2 is shown in Figure 6.6 and is 0.246 ms. The ping between desktop computer 2 and the high performing computer is shown in Figure 6.6 and is 0.242 ms. The overhead in desktop computer 2 when it is acting as an MCAP is:

```
Overhead in desktop computer 2 =
0.671 ms - 0.039 ms - 0.246 ms - 0.242 ms =
0.144 ms
```

We calculate overhead in the netbook when it is acting as MCAP in the same way. The numbers are from Table 6.5 and Figure 6.6. The overhead in the netbook when it is acting as an MCAP is:

```
Overhead in netbook =
0.774 ms - 0.039 ms - 0.241 ms - 0.319 ms =
0.175 ms
```

We calculate overhead in desktop computer 1 when it is acting as MCAP in the same way. The numbers are from Table 6.5 and Figure 6.6. Unlike in the previous calculations, node A is in this case the netbook. The sum of overhead in node A which is a netbook and the overhead in node B is calculated above and is 0.048 ms. The overhead in desktop computer 1 when it is acting as an MCAP is:

```
Overhead in desktop computer 1 =
0.859 ms - 0.048 ms - 0.299 ms - 0.241 ms =
0.271 ms
```

Since desktop computer 1 and desktop computer 2 are both similar desktop computers we would expect to see the same results. They are however not the same. It is also unexpected that the overhead in desktop computer 2 is larger than in the lower performing netbook. This may be because there have been some external factors affecting the results when desktop computer 2 was the MCAP. However since the result is in such a small timescale, we claim that the difference is negligible.

Using an MCAP does increase the RTT. The increase of RTT that comes from processing in the MCAP is small, less than 0.271 ms. The most notable difference is that the MCAP introduces an additional hop on the path from node A to node B. If the MCAP is located close to one of the nodes in terms of RTT, the increase in RTT from node A to node B is small. If the MCAP is located far away from both nodes in terms of RTT, the increase in RTT from node A to node B is significant.

6.7 Discussion

In this section we verify that the results presented in Section 6.6 fulfill our evaluation goals presented in Section 6.1. We look at each of our evaluation goals and verify that we have fulfilled it.

Our first goal is to verify that SOCKMAND can be used for real-time applications. Section 6.6.2 shows that with 1400B packet sizes the desktop computer can transfer data at a rate of 1000 Mbit/s with a CPU load of around 80% and the netbook can transfer data at a rate of 100 Mbit/s with a CPU load of around 64%. Note that this is not the percentage of total available CPU load since the measurements are done on multi-core and multi-threaded architectures. The results mean that our architecture is able to utilize the full network capacity of these nodes if the packet size is sufficiently large. These results show that our implementation supports large enough bandwidth to support real-time applications such as video streaming. Our recommended bandwidth requirement for real-time applications is 1.5 Mbit/s, this requirement is fulfilled.

Our round trip time requirement for real-time applications is 200 ms. Section 6.6.4 shows that the round trip time overhead introduced by processing packets in an MCAP is negligible. However, by using an MCAP, the latency will be increased since the packets must be routed by the MCAP. This means that it is important to have an algorithm that selects an MCAP close, in terms of RTT, to one of the end points. If the MCAP is close to one of the end points, the additional cost of routing packets through the MCAP will be lower than if the MCAP is located far away from both of the endpoints.

The socket migration time will, as mentioned in Section 6.6.3 depend on a fixed set of factors. It is impossible to present a number for these factors since they will vary based on the environment and in the TCP case, the characteristics of the TCP socket. We claim however that the socket migration time is low enough to support real-time applications because of the small size of the messages that must be sent between nodes. The major factor of the socket migration time is the time it takes to send the state and any messages to other nodes. In the worst case scenario, data must be sent to two other nodes from the source node. As long as the sum of the round trip time to both of the nodes from the source node is lower than 200 ms, the socket migration time will not exceed 200 ms. Note that when migrating a socket, other states related to the mobile application are also transferred from the source node to the destination node. By migrating the socket in parallel with the mobile application, we minimize the effect the socket migration has on the total process migration time.

Goal 2 is to verify what devices that are able to fulfill the requirements presented in Section 4.1. Section 6.6.2 shows that both the desktop computer and the netbook is able to fulfill the bandwidth requirement of 1.5 Mbit/s. Section 6.6.4 shows that there is no noticeable difference in the round-trip time overhead by the different devices. This shows us that a low end device such as the netbook is able to fulfill our requirements.

Goal 3 is to look at the relationship between packet loss and CPU load. Section 6.6.2 clearly shows the correlation between CPU load and packet loss in the MCAP. When the CPU load of the MCAP comes near 100%, packet loss occurs. The results also show that the payload size does not affect the CPU load of the MCAP. There is no notable difference in CPU load of payload sizes of 100B and 1400B. The data rate is the product of the packet size and the number of packets per second. The smaller the packet sizes the faster the CPU load will reach 100% and packet loss will occur. This suggests that application programmers should send fewer and larger packages, and not many small packages. Whether the application programmer can do that, completely depends on the domain of that specific application.

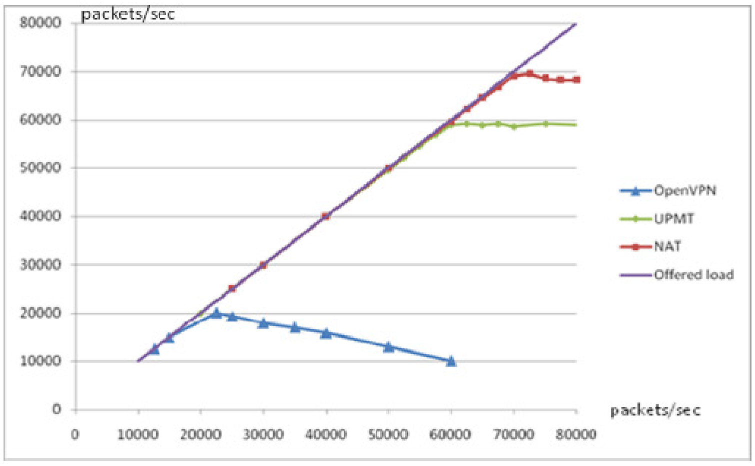

Figure 6.13: UPMT anchor node packet loss comparison [6]

Figure 6.13 and Figure 6.14 show packet loss and CPU load in an anchor

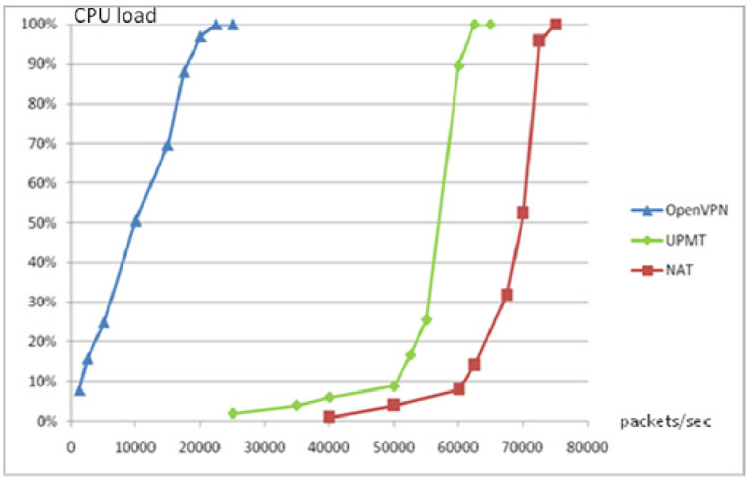

Figure 6.14: UPMT anchor node CPU load comparison [6]

node for various tunneling solutions presented in [6]. The specifications of the anchor node in that experiment are PC VIA C7-D Processor 1.5 GHz, 2 NICs Ethernet 100 Mb/s [6]. In terms of processing power it is slightly slower than the netbook used in our experiments. We must be careful when comparing our results with the results shown in [6] due to the difference in equipment and test environment, OpenVPN, which is a user space tunneling solution, reaches 100% CPU load when the incoming packets arrive at a rate of approximately 21000 packets per second. Figure B.3 shows that SOCKMAND reaches 100% CPU load at the same packet rate. This shows that SOCKMAND performs as good as the Open VPN user space tunneling solutions.

Our fourth goal is to investigate the difference in terms of CPU load when using the Rawsocket Handler and the Libpcap Handler. The results presented in Section 6.6.2 show us that the direction of a packet through an MCAP does not affect the CPU load or packet loss. The amount of CPU time used for receiving a packet with the Tunnel Handler and sending it with the Rawsocket Handler is equal to the CPU time used for capturing a packet with the Libpcap and sending it with the Tunnel Handler, this means that the MCAP performs symmetrically. The direction of the data through an MCAP does not affect the CPU load or packet loss.

Our last goal is to compare the CPU load when using SOCKMAND and when using regular sockets. From Section 6.6.1 we learn that using SOCKMAND in user space creates a significant overhead versus using the OS implementation of sockets. Our evaluation shows that the CPU load is at least three times as large when SOCKMAND is used due to additional copying of packets between kernel space and user space. This knowledge shows us that the performance of SOCKMAND will increase if SOCKMAND is implemented in kernel space. However this will limit the portability of SOCKMAND.

Chapter 7

Conclusion

In this chapter we conclude our work and the findings from this master thesis. In Section 7.1 we present our contributions. We do a critical assessment of our work in Section 7.2. Finally, we present some topics that can be looked at in the future in Section 7.3.

7.1 Contributions

We have developed a design and implemented a socket migration system which satisfies our requirements from Chapter 2 and Chapter 3. The main requirements are that the system must not need altering of the OS Kernel, the network must not be altered, it must be possible to communicate with legacy applications on legacy hosts, socket migrations must be transparent to the corresponding host and a set of metric requirements regarding the support of multimedia applications, like video conferencing.

Our work is aimed to make the ubiquitous computing paradigm possible by supporting process migration of real-time multimedia applications between heterogeneous devices. The contribution to the scenario from this thesis is our user-space socket migration system which will support the migration of the sockets used by the mobile real-time multimedia application.

Our main contribution is a user-space socket migration system, called SOCKMAND. Our design enables sockets to be migrated from one node to another while maintaining the connection to a legacy corresponding host. This is achieved by introducing Migration Community Access Points. SOCKMAND uses IP in UDP tunnels to transfer packets between the nodes with the mobile application and the MCAP.

Since SOCKMAND is implemented in user-space, it does not need any altering of the kernel and is therefore portable. The architecture is designed as a building brick in a process migration system. SOCKMAND should be used with a process migration system which takes care of the migration of the process, while SOCKMAND is the part which takes care of the migration of the sockets.

We have implemented this design and evaluated it in Chapter 6. The evaluation shows that SOCKMAND satisfies our requirements for real-time multimedia applications. This means that SOCKMAND can be used to migrate sockets that have high requirements in terms of delay and bandwidth.

We have shown that the difference in CPU load between a user-space and a kernel-space socket implementation is approximately three times as high in the user-space implementation. This is due to the additional transfers of the packet between kernel-space and user-space.

The evaluation shows that the CPU load depends on the number of packets per second and not on the data rate. This means that two data streams of 50 Mbit/s, can create different CPU loads if the packet sizes are different.

7.2 Critical Assessment

In the beginning of the thesis work, we focused on process migration in general. It took some time before we decided to look at socket migration in particular. The process of selecting the specific subtopic of process migration should have been done faster.

When we initially started searching for related work, we were not able to locate most of the work mentioned in Chapter 3. This was because we were unfamiliar with the term *socket migration*. We first found this term and work related to it well into our design process. Although we found the related work well into the design process, our design differs from the already existing work that we have found. Our work supplements the existing knowledge with a new socket migration solution.

As mentioned in Section 5.3.6, the implementation of TCP in user space proved to be harder than expected. This estimation error forced us to use more time implementing than originally planned, and therefore a bit less time on evaluation and writing.

7.3 Future Work

Our implementation of SOCKMAND only includes the core functionality needed to support socket migration. There are extensions and improvements to SOCKMAND that can provide further functionality, increased performance and new fields of research. We categorize the future work in short-term and long-term goals, where short term is anything shorter than a couple of months.

7.3.1 Short-term Goals

If increased performance is desirable, we will benefit from an implementation of SOCKMAND in kernel space. Most of the functionality that is present in SOCKMAND is already present in the kernel. Since sockets are already a part of the kernel, only the functionality needed for the actual migration of the socket is needed. A positive side effect of implementing

SOCKMAND in kernel space is that the TCP implementation in the kernel already includes TCP options which we have not yet implemented in SOCKMAND due to time constraints. Such TCP options are selective acknowledgments, window scaling and TCP timestamps. Although a kernel space implementation is desirable in terms of performance, the portability of the system will decrease.

Network Address Translation (NAT) support is missing from our design. SOCKMAND may fail if some nodes are behind a NAT device. Support for nodes hidden behind a NAT must be added to enable SOCKMAND to be used in these cases.

When socket migration fails due to unexpected errors, this must be handled in the proper way. SOCKMAND should notify the process migration system and the system should, if possible, decide on whether or not to abort the process migration.

We use the *break before make* strategy when migrating sockets using SOCKMAND. It may be better in some cases to use *make before break*. We should investigate the possibility to use *make before break* in SOCKMAND. Whether or not this is possible may depend on how the actual process is migrated.

7.3.2 Long-term Goals

Due to time constraints, we have left the selection of MCAP for future work. An algorithm that chooses a good MCAP within a Migration Community is essential for performance and low round trip delay. The algorithm should take a policy as input. The policy can for instance state whether up time should be preferred over delay. Since a Migration Community is a generic overlay that can be implemented in several ways, there may be need for an algorithm for each of the different overlay implementations.

TCP is designed as an end-to-end transport layer protocol. The protocol reacts when changes in the route between the two endpoints occur. This can be changes in round-trip time, increased packet loss and lower bandwidth. After a migration of one of the endpoints, the characteristics of the route between the endpoints will change. A possible future research topic is to look at how we can reconfigure the state variables of the TCP connection after a migration to ensure that the TCP connection is functioning optimally on the new route.

We have not looked at security issues in our design. Security features must be added to prevent unauthorized *socket hijacking*. Whether this should be done by adding keys or encryption functionality to SOCKMAND, if it should be a feature of the Migration Community overlay or by other solutions is open for future research.

Bibliography

[1] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[2] Mario Baldi and Yoram Ofek. End-to-end delay analysis of videoconferencing over packet-switched networks. *IEEE/ACM Trans. Netw.*, 8(4):479–492, August 2000.

[3] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[4] E. Beda and N. Ventura. Socketless tcp - an end to end handover solution. In *Networks, 2005. Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication., 2005 13th IEEE International Conference on*, volume 2, page 6 pp., nov. 2005.

[5] Massimo Bernaschi, Francesco Casadei, and Paolo Tassotti. Sockmi: a solution for migrating tcp/ip connections. In *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, pages 221 –228, feb. 2007.

[6] M. Bonola and S. Salsano. Per-application mobility management: Performance evaluation of the upmt solution. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pages 2249 –2255, july 2011.

[7] M. Bonola, S. Salsano, and A. Polidoro. Upmt: Universal per-application mobility management using tunnels. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pages 1 –8, 30 2009-dec. 4 2009.

[8] Seth Fogie Cyrus Peikari. Raw sockets revisited: The day the internet died. http://www.airscanner.com/pubs/rawsockets.pdf, 2003. Accessed online Feb 27 2012.

[9] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice), May 2000. Updated by RFC 3704.

[10] D. Funato, K. Yasuda, and H. Tokuda. Tcp-r: Tcp mobility support for continuous operation. In *Network Protocols, 1997. Proceedings., 1997 International Conference on*, pages 229 –236, oct 1997.

[11] Google. System requirements for hangouts. http://http://support.google.com/plus/bin/answer.py?hl=en&answer=1216376, 2012. Accessed online Apr 24 2012.

[12] Ian Griffiths. Raw sockets gone in xp sp2. http://www.interact-sw.co.uk/iangblog/2004/08/12/norawsockets, August 2004. Accessed online Feb 27 2012.

[13] Raj. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling / Raj Jain.* Wiley, New York :, 1991.

[14] E. Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, University of Washington, 1988.

[15] Bryan Kuntz and Karthik Rajan. Migsock migratable tcp socket in linux. Master's thesis, Carnegie Mellon University, 2002.

[16] D.A. Maltz and P. Bhagwat. Msocks: an architecture for transport layer mobility. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1037 –1045 vol.3, mar-2 apr 1998.

[17] The Linux man-pages project. *BIND(2) man page*, December 2007.

[18] The Linux man-pages project. *CLOSE(2) man page*, December 2007.

[19] The Linux man-pages project. *CONNECT(2) man page*, December 2008.

[20] The Linux man-pages project. *LISTEN(2) man page*, November 2008.

[21] The Linux man-pages project. *READ(2) man page*, February 2009.

[22] The Linux man-pages project. *ACCEPT(2) man page*, September 2010.

[23] The Linux man-pages project. *RECV(2) man page*, August 2010.

[24] The Linux man-pages project. *SELECT(2) man page*, August 2010.

[25] The Linux man-pages project. *SEND(2) man page*, August 2010.

[26] The Linux man-pages project. *SOCKET(7) man page*, June 2010.

[27] The Linux man-pages project. *WRITE(2) man page*, August 2010.

[28] J. Manner and M. Kojo. Mobility Related Terminology. RFC 3753 (Informational), June 2004.

[29] Nick Mathewson and Niels Provos. libevent - an event notification library. http://libevent.org/, February 2012.

[30] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32:241–299, September 2000.

[31] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[32] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[33] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[34] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[35] Skype. How much bandwidth does skype need? https://support.skype.com/en-us/faq/FA1417/How-much-bandwidth-does-Skype-need, February 2012. Accessed online Apr 17 2012.

[36] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: connection migration for service continuity in the internet. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 469 – 470, 2002.

[37] Kwame Wright, Kartik Gopalan, and Hui Kang. Performance analysis of various mechanisms for inter-process communication. Technical report, Department of Computer Science, Binghamton University, 2007.

[38] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, MobiCom '02, pages 95–106, New York, NY, USA, 2002. ACM.

Appendix A

Abbreviations

ACK	Acknowledgement
AN	Anchor Node
B	Bytes
CH	Corresponding Host
CPU	Central Processing Unit
DN	Destination Node
DC	Desktop Computer
IP	Internet Protocol
IPC	Inter Process Communication
LA	Legacy Application
MA	Mobile Application
MCAP	Migration Community Access Point
MH	Mobile Host
NAT	Network Address Translation
P2P	Peer-To-Peer
RTT	Round-Trip Time
SN	Source Node
TCP	Transmission Control Protocol
TRAMP	TRAMP Real-time Application Mobility Platform
UDP	User Datagram Protocol

Appendix B

Performance Evaluation Results

This appendix includes additional results which supports performance evaluation results in Chapter 6. All the results shown in this appendix are referenced in Chapter 6.

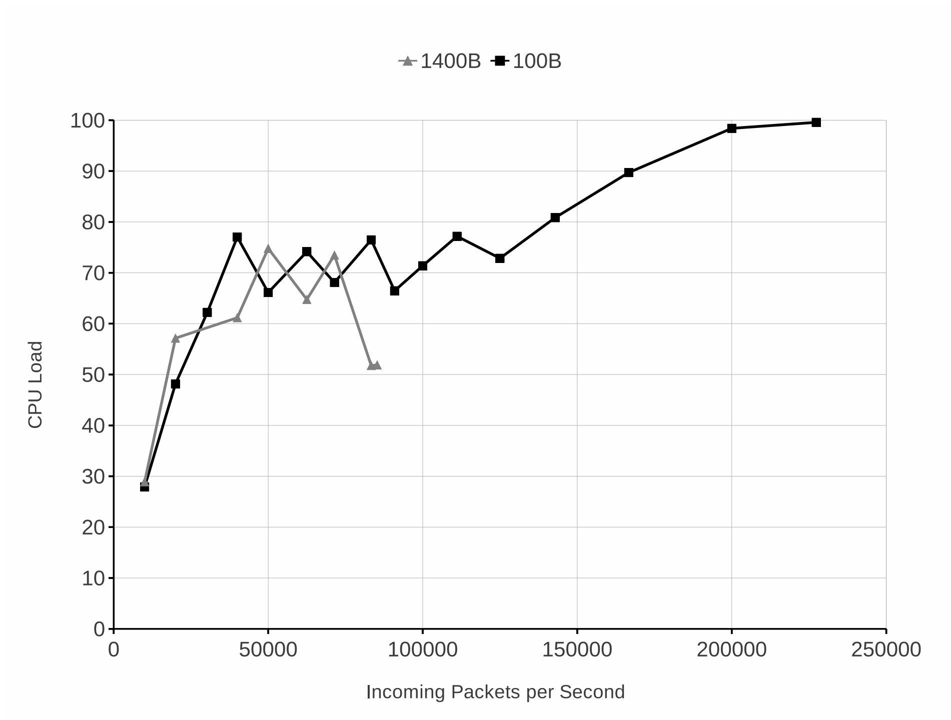

Figure B.1: Desktop CPU Load on MCAP from libpcap to tunnel

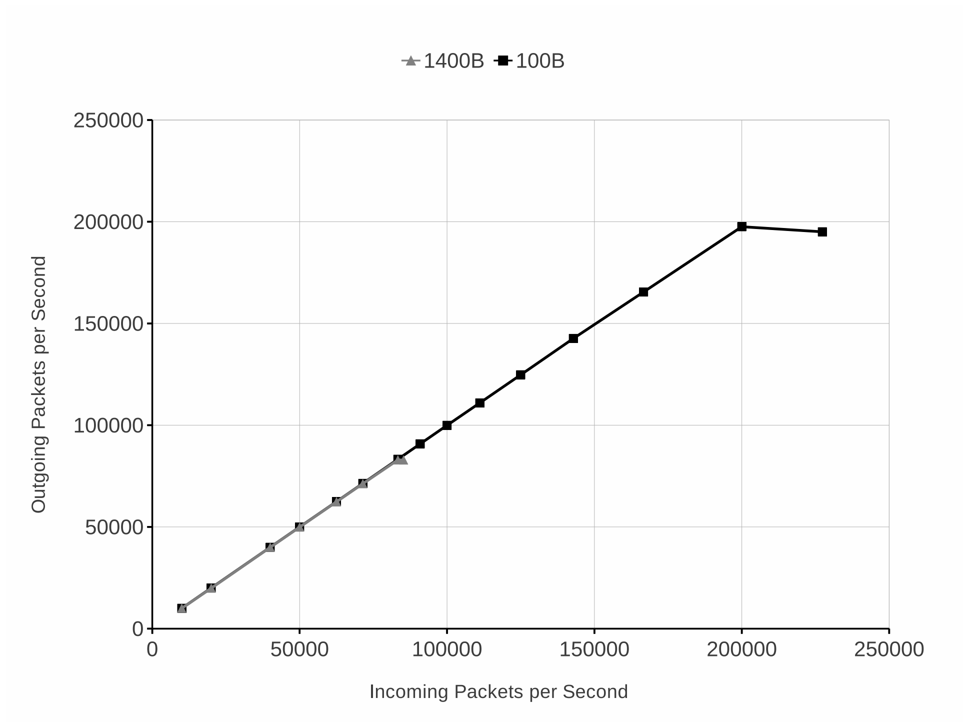

Figure B.2: Desktop Packet loss on MCAP from libpcap to tunnel

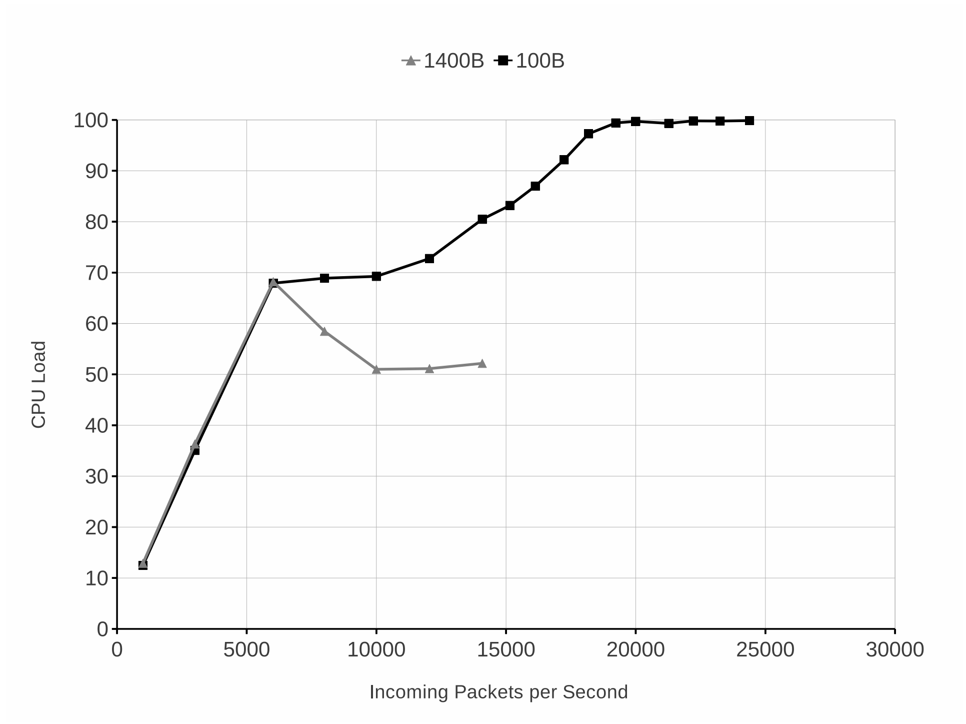

Figure B.3: Netbook CPU Load on MCAP from libpcap to tunnel

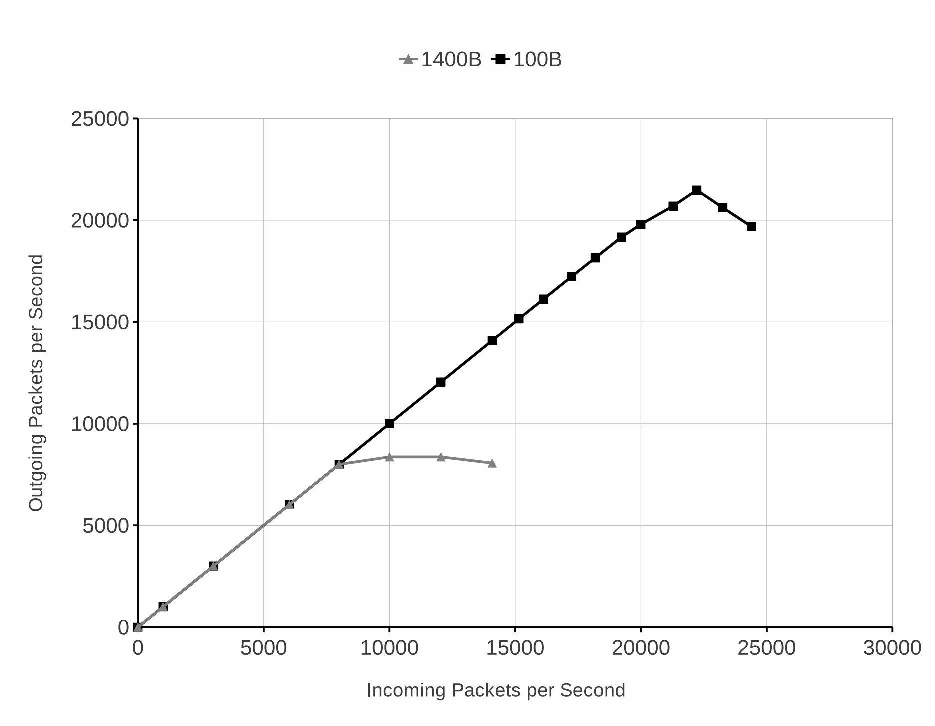

Figure B.4: Netbook Packet loss on MCAP from libpcap to tunnel

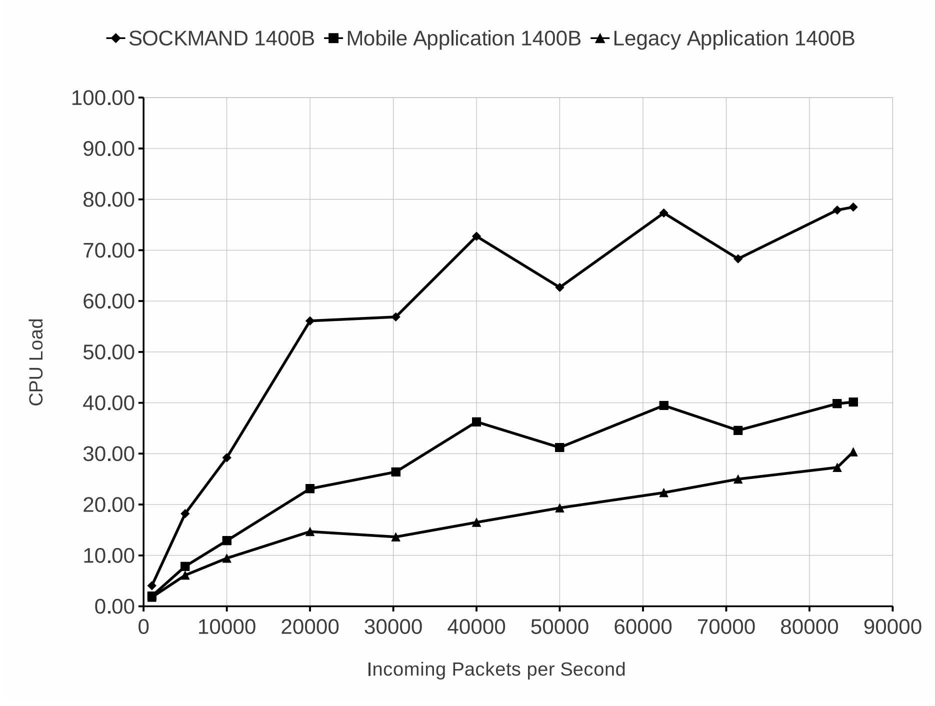

Figure B.5: CPU load by SOCKMAND and mobile application on desktop computer with 1400B payload

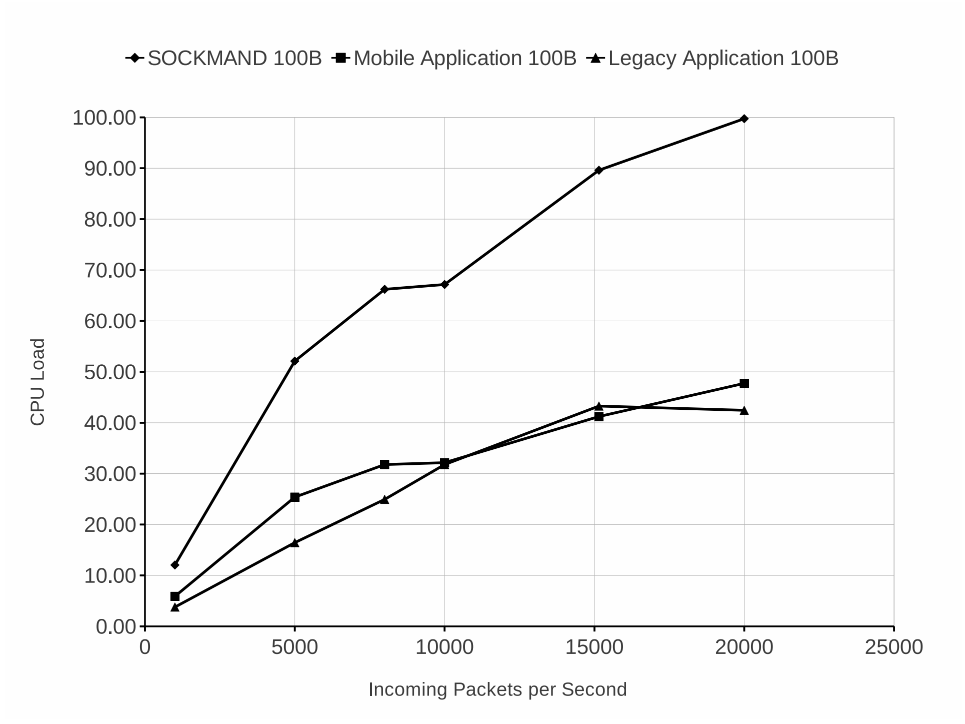

Figure B.6: CPU load by SOCKMAND and mobile application on netbook with 100B payload

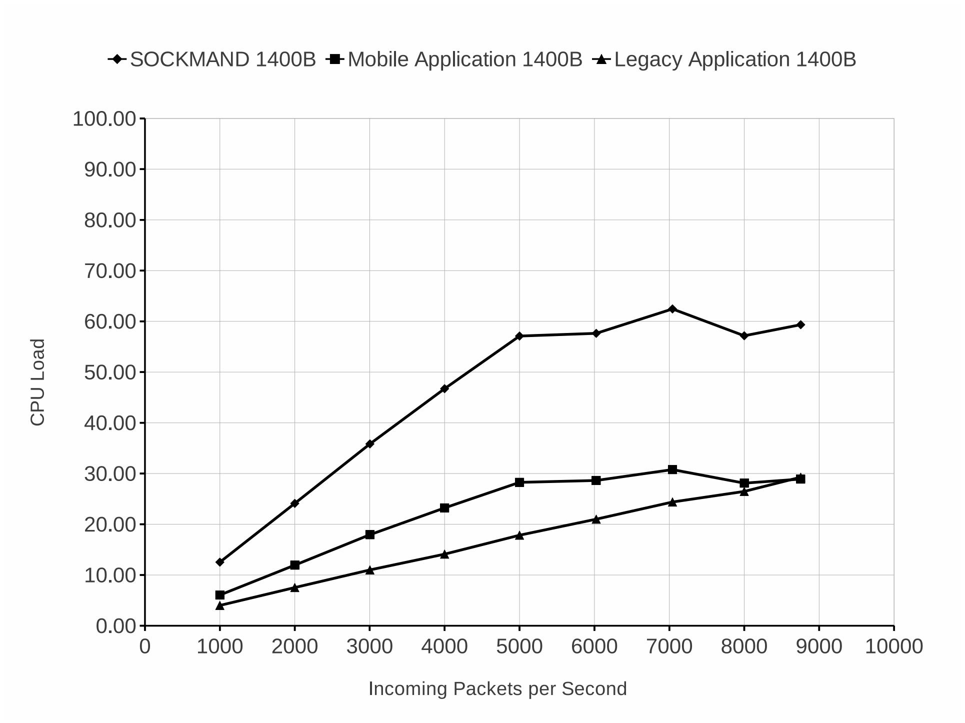

Figure B.7: CPU load by SOCKMAND and mobile application on netbook with 1400B payload

Appendix C

Measurement Applications

In this appendix we present our measurement applications used in Chapter 6. We describe what the applications do and how to execute them. Instructions on how to get the source code files are presented in Appendix D.

C.1 RTT Test Applications

These applications are used measure the round-trip time between our mobile application and a legacy application. The client sends a given number of packets and measures the time it takes to get a reply from the server.

1. Compile the server.c with gcc and start it on a node A.

2. Make the client using make and run it on a node B with SOCKMAND running.

The client will send the given number of packets to node A. The packet are replied instantly. The time is measured and printed, the time is given in microseconds.

To modify the client to send the packets through an MCAP, change the fourth argument to sm_socket from 0 to 1. Make sure you have SOCKMAND running on another node and that you give the address of that node as a parameter to SOCKMAND on node B.

Listing C.1 shows the client application and Listing C.2 shows the server application.

Listing C.1: Round-trip Time Client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
```

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>

#define BUFLEN 1400
#include "libsockmand.h"

int main(int argc, char **argv)
{
    struct timeval start, end;
    long elapsed_utime;     /* elapsed time in microseconds */
    long elapsed_seconds;   /* diff between seconds counter */
    long elapsed_useconds;  /* diff between microseconds counter */

    if(argc<4){
        fprintf(stderr, "Usage: %s <destination IP address> <destport> <count>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    unsigned int dst = inet_addr(argv[1]);
    short port = htons(atoi(argv[2]));
    int count = atoi(argv[3]);
    char buf[BUFLEN] = "Ping message\0";
    int sd = sm_socket(AF_INET,SOCK_DGRAM,0,0);

    int i;
    long sum = 0;

    struct sockaddr_in servaddr;
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = dst;
    servaddr.sin_port = port;
    socklen_t slen = sizeof(servaddr);


    struct sockaddr_in s;
    socklen_t s2 =  sizeof(s);

    sm_sendto(sd, buf, BUFLEN, 0, (struct sockaddr *)&servaddr,slen); //virtual bind
    sm_recvfrom(sd, buf, BUFLEN, 0, (struct sockaddr *)&s,&s2);

    sleep(1);
    for(i=0; i < count; i++){

        gettimeofday(&start, NULL);
        sm_sendto(sd, buf, BUFLEN, 0, (struct sockaddr *)&servaddr,slen);
        sm_recvfrom(sd, buf, BUFLEN, 0, (struct sockaddr ↩
            *)&s,&s2);
        gettimeofday(&end, NULL);


        elapsed_seconds  = end.tv_sec  - start.tv_sec;
        elapsed_useconds = end.tv_usec - start.tv_usec;
        elapsed_utime = (elapsed_seconds) * 1000000 + ↩
            elapsed_useconds;

        printf("%ld\n", elapsed_utime);
        sum += elapsed_utime;
        sleep(1);
    }
    printf("\nDone doing tests\n");
    double average = (double)sum/(double)count;
    printf("Average RTT: %f microseconds\n",average);
    close(sd);
    return 0;
}
```

Listing C.2: Round-trip Time Server

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

#define BUFLEN  1400

int main(int argc, char **argv)
{
    struct sockaddr_in   si_local, si_remote;
    int port;
    socklen_t slen;
    char buf[BUFLEN];
    slen     =   sizeof(si_remote);

    if(argc!=2){
        fprintf(stderr, "Usage: %s <port number>\n", argv↩
            [0]);
        return 0;
    }

    port=atoi(argv[1]);
    if(port<0){
        fprintf(stderr, "Could not parse port\n", argv[0])↩
            ;
        return 0;
    }

    int s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    memset((char *) &si_local, 0, sizeof(si_local));
    si_local.sin_family       =  AF_INET;
    si_local.sin_port         =  htons(port);
    si_local.sin_addr.s_addr  =  htonl(INADDR_ANY);
    if (bind(s, (const struct sockaddr *)&si_local, sizeof↩
       (si_local))==-1){
        perror("bind");
        exit(EXIT_FAILURE);
    }

    while(1==1){
        int read = recvfrom(s, buf, BUFLEN, 0, (struct ↩
           sockaddr *)&si_remote, &slen);
        sendto(s, buf, read, 0, (struct sockaddr *)&↩
           si_remote, slen);
    }
    close(s);
    return 0;
}
```

C.2 Packetloss Test From Tunnel to Rawsocket

These applications are used measure the CPU load of SOCKMAND and packet losses when packets are routed through an MCAP from the Tunnel Handler to the Rawsocket Handler. The client will send a given number of packets per second with a given packet size to the server on node A through SOCKMAND on node B. The arguments should be varied to look at their affect on the CPU load and packetloss. When the client is done sending, interrupt the server using CTRL + C to view the test results. Top can also be used to monitor the CPU load of SOCKMAND on node B.

1. Compile server.c with gcc and run it on a node A.
2. Start SOCKMAND a node B.
3. Compile client.c with gcc and run it on a node C.

Listing C.3 shows the client application and Listing C.4 shows the server application.

Listing C.3: Packetloss Test From Tunnel to Rawsocket Client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
```

```c
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>

struct ipheader {
    unsigned char ip_hl:4, ip_v:4;
    unsigned char ip_tos;
    unsigned short int ip_len;
    unsigned short int ip_id;
    unsigned short int ip_off;
    unsigned char ip_ttl;
    unsigned char ip_p;
    unsigned short int ip_sum;
    unsigned int ip_src;
    unsigned int ip_dst;
};


struct udpheader {
    unsigned short int uh_sport;
    unsigned short int uh_dport;
    unsigned short int uh_len;
    unsigned short int uh_check;
};


void build_ip_packet(struct ipheader *ip, int protocol, ↩
   unsigned int srcaddr, unsigned int destaddr, int ↩
   buflen){
    ip->ip_hl = 5;
    ip->ip_v = 4;
    ip->ip_tos = 16;
    ip->ip_len = buflen;
    ip->ip_id = 0;
    ip->ip_ttl = 64;
    ip->ip_p = protocol;
    ip->ip_src = srcaddr;
    ip->ip_dst = destaddr;
}

void build_udp_header(struct udpheader *udp, unsigned ↩
   short srcprt, unsigned short destport, int buflen){
    udp->uh_sport = srcprt;
    udp->uh_dport = destport;
    udp->uh_len = htons(buflen-(sizeof(struct ipheader)));
}

int main(int argc, char **argv){
    struct timeval start, prev, now, end;
    long elapsed_utime;
    long elapsed_seconds;
    long elapsed_useconds;

    if(argc != 7){
        fprintf(stderr, "Usage: %s <MCAP IP address> <destination IP address> <dest port number> <number of seconds> <packets per seconds> <packetsize>\n", argv[0]);
        return 0;
    }

    unsigned int  mcapaddr = inet_addr(argv[1]);
    if(mcapaddr==-1){
        printf("Could not parse <MCAP IP address>\n");
        return 0;
    }
    unsigned int toaddr = inet_addr(argv[2]);
    if(toaddr==-1){
        printf("Could not parse <destination IP address>\n");
        return 0;
    }

    int destport=atoi(argv[3]);

    if(destport==-1){
        printf("Could not parse <dest port number>\n");
        return 0;
    }

    int srcport=9999;

    int seconds = atoi(argv[4]);
    if(seconds==-1){
        printf("Could not parse <number of seconds>\n");
        return 0;
    }
    int packetrate =  atoi(argv[5]);

    if(packetrate==-1){
        printf("Could not parse <packets per seconds>\n");
        return 0;
    }

    int buflen = atoi(argv[6]);

     if(buflen==-1){
        printf("Could not parse <packetsize>\n");
        return 0;
    }

    int totalpackets = packetrate*seconds;
    char buf[buflen];
    long utimebetweensend = 1000000/packetrate;

    printf("Sending %d packets\n", totalpackets);
    printf("%d packets / second\n", packetrate);
    printf("Sending for a total of %d seconds\n", seconds);
    printf("Microseconds between each packet %ld\n", utimebetweensend);

    struct ipheader *ip = (struct ipheader *) buf;
    struct udpheader *udp = (struct udpheader *) (ip + 1);

    build_ip_packet(ip,
                    IPPROTO_UDP,
                    mcapaddr,
                    toaddr,
                    buflen);

    build_udp_header(udp, htons(srcport), htons(destport), buflen);

    int tunnel_sockfd = socket(AF_INET,SOCK_DGRAM,0);

    struct sockaddr_in servaddr;
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(5000);

    bind(tunnel_sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));

    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;

    servaddr.sin_addr.s_addr = mcapaddr;
    servaddr.sin_port = htons(5000);

    long left = 0;
    int sendtosize = sizeof(servaddr);

    int i;
    gettimeofday(&start, NULL);
    sendto(tunnel_sockfd, buf, buflen, 0, (struct sockaddr *)&servaddr,sendtosize);

    gettimeofday(&prev, NULL);


    for(i=0;i<totalpackets-1;i++)
    {
        while(1){
            gettimeofday(&now, NULL);
            elapsed_seconds  = now.tv_sec  - prev.tv_sec;
            elapsed_useconds = now.tv_usec - prev.tv_usec;
            elapsed_utime = (elapsed_seconds) * 1000000 + elapsed_useconds;
            left = elapsed_utime - utimebetweensend;
            if(left > 0)
                break;
        }
        sendto(tunnel_sockfd, buf, buflen, 0, (struct ↩
            sockaddr *)&servaddr,sendtosize);
        memcpy(&prev,&now,sizeof(struct timeval));
        prev.tv_usec -= left;
    }

    gettimeofday(&end, NULL);

    elapsed_seconds  = end.tv_sec  - start.tv_sec;
    elapsed_useconds = end.tv_usec - start.tv_usec;
    elapsed_utime = (elapsed_seconds) * 1000000 + ↩
        elapsed_useconds;

    double secs = elapsed_utime/1000000.0;

    printf("Sent %d packets\n", totalpackets);
    double actualrate = (double)totalpackets/secs;
    printf("Actual packets / second %f\n",actualrate);
    printf("Elapsed time = %ld microseconds\n", ↩
        elapsed_utime);
    printf("Elapsed time = %f seconds\n", secs);
    close(tunnel_sockfd);
    return 0;
}
```

Listing C.4: Packetloss Test From Tunnel to Rawsocket Server

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <signal.h>

#define BUFLEN 1500

int npacksreceived;

struct timeval start, end;

void signal_handler(int sig){
    printf("Receieved %d packets\n", npacksreceived);

    long elapsed_utime;     /* elapsed time in microseconds↩
         */
    long elapsed_seconds;  /* diff between seconds counter↩
         */
    long elapsed_useconds; /* diff between microseconds counter */

    elapsed_seconds  = end.tv_sec  - start.tv_sec;
    elapsed_useconds = end.tv_usec - start.tv_usec;
    elapsed_utime = (elapsed_seconds) * 1000000 + elapsed_useconds;

    double secs = elapsed_utime/1000000.0;

    double actualrate = (double)npacksreceived/secs;
    printf("Actual packets / second %f\n",actualrate);
    printf("Elapsed time = %ld microseconds\n", elapsed_utime);
    printf("Elapsed time = %f seconds\n", secs);
    exit(EXIT_SUCCESS);
}

int main(int argc, char **argv){
    npacksreceived = 0;
    signal(SIGINT, &signal_handler);
    struct sockaddr_in   si_local;
    int sd;
    int port;
    char buf[BUFLEN];

    if(argc!=2){
        fprintf(stderr, "Usage: %s <port number>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    port=atoi(argv[1]);

    if ((sd=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))==-1){
        perror("socket");
        exit(EXIT_FAILURE);
    }

    memset((char *) &si_local, 0, sizeof(si_local));
    si_local.sin_family       =  AF_INET;
    si_local.sin_port         =  htons(port);
    si_local.sin_addr.s_addr  =  htonl(INADDR_ANY);
    if (bind(sd, (const struct sockaddr *)&si_local, sizeof(si_local))==-1)
    {
        perror("bind");
        exit(EXIT_FAILURE);
    }
    int first = 1;
    while(1)
    {
        if (recvfrom(sd, buf, BUFLEN, 0, 0, 0)==-1)
        {
            perror("recvfrom()");
            exit(EXIT_FAILURE);
        }
        else
        {
            npacksreceived++;
            if(first==1){
                first=0;
                gettimeofday(&start, NULL);
            }
            else{
                gettimeofday(&end, NULL);
            }
        }
    }
}
```

C.3 Packetloss Test From Libpcap to Tunnel

These applications are used measure the CPU load of SOCKMAND and packet losses when packets are routed through an MCAP from the Libpcap Handler to the Tunnel Handler. The client will send a given number of packets per second to node B through SOCKMAND on node C. The arguments should be varied to look at their affect on the CPU load and packetloss. When the client is done sending, interrupt the server using CTRL + C to view the test results. Top can also be used to monitor the CPU load of SOCKMAND on node A.

1. Start SOCKMAND a node A.
2. Start SOCKMAND on node B with node A as the MCAP address.
3. Compile server.c with gcc and run it on a node B with a port P as argument.
4. Interrupt SOCKMAND on node B using CTRL + C and press enter in the server.
5. Compile client.c with gcc and run it on a node C. Use the address of node A and the port P as arguments.

SOCKMAND on node B is killed because the packets are sent directly to the server application and not through SOCKMAND. Listing C.5 shows the client application and Listing C.6 shows the server application.

Listing C.5: Packetloss Test From Libpcap to Tunnel Client

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

```
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>



int main(int argc, char **argv){
    struct timeval start, prev, now, end;
    long elapsed_utime;    /* elapsed time in microseconds */
    long elapsed_seconds;  /* diff between seconds counter */
    long elapsed_useconds; /* diff between microseconds counter */

    if(argc != 6){
        fprintf(stderr, "Usage: %s <MCAP IP address> <dest port number> <number of seconds> <packets per seconds> <packetsize>\n", argv[0]);
        return 0;
    }

    unsigned int toaddr = inet_addr(argv[1]);
    if(toaddr==-1){
        printf("Could not parse <MCAP IP address>\n");
        return 0;
    }

    int destport=atoi(argv[2]);

    if(destport==-1){
        printf("Could not parse <dest port number>\n");
        return 0;
    }

    int srcport = 9999;

    int seconds = atoi(argv[3]);
    if(seconds==-1){
        printf("Could not parse <number of seconds>\n");
        return 0;
    }
    int packetrate =  atoi(argv[4]);

    if(packetrate==-1){
        printf("Could not parse <packets per seconds>\n");
        return 0;
    }

    int buflen = atoi(argv[5]);

    if(buflen==-1){
        printf("Could not parse <packetsize>\n");
        return 0;
    }

    int totalpackets = packetrate*seconds;
    char buf[buflen];
    long utimebetweensend = 1000000/packetrate;


    printf("Sending %d packets\n", totalpackets);
    printf("%d packets / second\n", packetrate);
    printf("Sending for a total of %d seconds\n", seconds);
    printf("Microseconds between each packet %ld\n", utimebetweensend);


    int s = socket(AF_INET,SOCK_DGRAM,0);

    struct sockaddr_in servaddr;
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;

    servaddr.sin_addr.s_addr = toaddr;
    servaddr.sin_port = htons(destport);

    long left = 0;
    int sendtosize = sizeof(servaddr);

    int i;
    gettimeofday(&start, NULL);
    sendto(s, buf, buflen, 0, (struct sockaddr *)&servaddr,sendtosize);
    gettimeofday(&prev, NULL);


    for(i=0;i<totalpackets-1;i++)
    {

        while(1){
            gettimeofday(&now, NULL);
            elapsed_seconds  = now.tv_sec  - prev.tv_sec;
            elapsed_useconds = now.tv_usec - prev.tv_usec;
            elapsed_utime = (elapsed_seconds) * 1000000 + elapsed_useconds;
            left = elapsed_utime - utimebetweensend;
            if(left > 0)
                break;
        }
        sendto(s, buf, buflen, 0, (struct sockaddr *)&servaddr,sendtosize);
        memcpy(&prev,&now,sizeof(struct timeval));
        prev.tv_usec -= left;
    }

    gettimeofday(&end, NULL);

    printf("\nDONE\n\n");

    elapsed_seconds  = end.tv_sec  - start.tv_sec;
    elapsed_useconds = end.tv_usec - start.tv_usec;
    elapsed_utime = (elapsed_seconds) * 1000000 + ↩
       elapsed_useconds;

    double secs = elapsed_utime/1000000.0;

    printf("Sent %d packets\n", totalpackets);
    double actualrate = (double)totalpackets/secs;
    printf("Actual packets / second %f\n",actualrate);
    printf("Elapsed time = %ld microseconds\n", ↩
       elapsed_utime);
    printf("Elapsed time = %f seconds\n", secs);
    close(s);
    return 0;
}
```

Listing C.6: Packetloss Test From Libpcap to Tunnel Server

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <signal.h>
#include <sys/time.h>

#include "libsockmand.h"

#define BUFLEN  512

int npacksreceived;

struct timeval start, end;

void signal_handler(int sig)
{
    printf("Received %d packets\n", npacksreceived);

    long elapsed_utime;    /* elapsed time in microseconds↩
         */
    long elapsed_seconds;  /* diff between seconds counter↩
         */
    long elapsed_useconds; /* diff between microseconds ↩
        counter */

    elapsed_seconds  = end.tv_sec  - start.tv_sec;
    elapsed_useconds = end.tv_usec - start.tv_usec;
    elapsed_utime = (elapsed_seconds) * 1000000 + ↩
        elapsed_useconds;

    double secs = elapsed_utime/1000000.0;

    double actualrate = (double)npacksreceived/secs;
    printf("Actual packets / second %f\n",actualrate);
    printf("Elapsed time = %ld microseconds\n", ↩
        elapsed_utime);
    printf("Elapsed time = %f seconds\n", secs);
    exit(EXIT_SUCCESS);
}

int main(int argc, char **argv)
{
    npacksreceived = 0;
    signal(SIGINT, &signal_handler);
    struct sockaddr_in  si_local, si_tmp;
    int  s;
    char buf[BUFLEN];

    if(argc!=2)
    {
        fprintf(stderr, "Usage: %s <destport>\n", argv[0])↩
            ;
        return 0;
    }

    int destport = atoi(argv[1]);
    if(destport == -1){
        printf("Could not parse <destport>\n");
        return 0;
    }

    int tunnel_sockfd = sm_socket(AF_INET,SOCK_DGRAM,0 ,1);

    memset((char *) &si_tmp, 0, sizeof(si_tmp));
    si_tmp.sin_family       =  AF_INET;
    si_tmp.sin_port         =  htons(destport);
    si_tmp.sin_addr.s_addr  =  htonl(INADDR_ANY);
    sm_bind(tunnel_sockfd, (const struct sockaddr *)&↩
        si_tmp, sizeof(si_tmp));
    printf("Kill local SOCKMAND and press enter\n");
    getchar();

    s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    memset((char *) &si_local, 0, sizeof(si_local));
    si_local.sin_family       =  AF_INET;
    si_local.sin_port         =  htons(5000);
    si_local.sin_addr.s_addr  =  htonl(INADDR_ANY);
    bind(s, (const struct sockaddr *)&si_local, sizeof(↩
       si_local));

    int first = 1;
    while(1)
    {
        recvfrom(s, buf, BUFLEN, 0, 0, 0);

        npacksreceived++;
        if(first==1){
            first=0;
            gettimeofday(&start, NULL);
        }
        else{
            gettimeofday(&end, NULL);
        }
    }
}
```

C.4 SOCKMAND and Mobile Application Test Applications

These applications are used to measure the CPU load of SOCKMAND and a mobile application when they are receiving packets at a given rate and with a given packet size. The client will send the a given number of packets per second to node A. The arguments should be varied to look at its affect on the CPU load. The server and SOCKMAND on node A should be monitored with top.

1. Make the server using make and run it on a node A with SOCKMAND running.
2. Compile client.c with gcc and run it on another node.

Listing C.7 shows the client application and Listing C.8 shows the server application.

Listing C.7: SOCKMAND load client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
```

```


int main(int argc, char **argv){
    struct timeval start, prev, now, end;
    long elapsed_utime;     /* elapsed time in microseconds */
    long elapsed_seconds;  /* diff between seconds counter */
    long elapsed_useconds; /* diff between microseconds counter */

    if(argc!=6){
        fprintf(stderr, "Usage: %s <ipaddress> <dest port number> <number of seconds> <packets per seconds> <packetsize>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int destport = atoi(argv[2]);
    if(destport==-1){
        printf("Could not parse <dest port number>\n");
        return 0;
    }
    int seconds =  atoi(argv[3]);
    if(seconds==-1){
        printf("Could not parse <number of seconds>\n");
        return 0;
    }
    int packetrate = atoi(argv[4]);
    if(packetrate==-1){
        printf("Could not parse <packets per seconds>\n");
        return 0;
    }
    int buflen = atoi(argv[5]);
    if(buflen==-1){
        printf("Could not parse <packetsize>\n");
        return 0;
    }

    int totalpackets = packetrate*seconds;
    char buf[buflen];
    long utimebetweensend = 1000000/packetrate;


    printf("Sending %d packets\n", totalpackets);
    printf("%d packets / second\n", packetrate);
    printf("Sending for a total of %d seconds\n", seconds);
    printf("Microseconds between each packet %ld\n", utimebetweensend);


    int s = socket(AF_INET,SOCK_DGRAM,0);

    struct sockaddr_in servaddr;
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;

    servaddr.sin_addr.s_addr = inet_addr(argv[1]);
    servaddr.sin_port = htons(destport);

    long left = 0;
    int sendtosize = sizeof(servaddr);

    int i;
    gettimeofday(&start, NULL);
    if ( sendto(s, buf, buflen, 0, (struct sockaddr *)&servaddr,sendtosize)==-1){
        perror("sendto()");
        exit(EXIT_FAILURE);
    }
    gettimeofday(&prev, NULL);


    for(i=0;i<totalpackets-1;i++)
    {

        while(1){
            gettimeofday(&now, NULL);
            elapsed_seconds  = now.tv_sec  - prev.tv_sec;
            elapsed_useconds = now.tv_usec - prev.tv_usec;
            elapsed_utime = (elapsed_seconds) * 1000000 + elapsed_useconds;
            left = elapsed_utime - utimebetweensend;
            if(left > 0)
                break;
        }
            if (sendto(s, buf, buflen, 0, (struct sockaddr *)&servaddr,sendtosize)==-1){
        perror("sendto()");
        exit(EXIT_FAILURE);
    }
        memcpy(&prev,&now,sizeof(struct timeval));
        prev.tv_usec -= left;
    }

    gettimeofday(&end, NULL);

    printf("\nDONE\n\n");

    elapsed_seconds  = end.tv_sec  - start.tv_sec;
    elapsed_useconds = end.tv_usec - start.tv_usec;
    elapsed_utime = (elapsed_seconds) * 1000000 + elapsed_useconds;

    double secs = elapsed_utime/1000000.0;

    printf("Sent %d packets\n", totalpackets);
    double actualrate = (double)totalpackets/secs;
    printf("Actual packets / second %f\n",actualrate);
    printf("Elapsed time = %ld microseconds\n", elapsed_utime);
    printf("Elapsed time = %f seconds\n", secs);
    close(s);
    return 0;
}
```

Listing C.8: SOCKMAND load server

```
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>

#include "libsockmand.h"

int main(int argc, char **argv){
    struct sockaddr_in   si_local;
    int sd;
    char buf[1500];

    if(argc!=2){
        fprintf(stderr, "Usage: %s <port number>\n", argv[0]);
        return 0;
    }

    int port=atoi(argv[1]);
    if(port==-1){
        fprintf(stderr, "Usage: %s <port number>\n", argv[0]);
        return 0;
    }


    if ((sd = sm_socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP, 0))==-1){
        printf("Error in sm_socket()");
        return 0;
    }

    memset((char *) &si_local, 0, sizeof(si_local));
    si_local.sin_family       =  AF_INET;
    si_local.sin_port         =  htons(port);
    si_local.sin_addr.s_addr  =  htonl(INADDR_ANY);
    if (sm_bind(sd, (const struct sockaddr *)&si_local, sizeof(si_local)) == -1){
        printf("Error in sm_bind()");
        return 0;
    }

    while(1){
        if (sm_recvfrom(sd, buf, 1500, 0, 0, 0)==-1){
            printf("Error in sm_recvfrom()");
            return 0;
        }
    }
    return 0;
}
```

Appendix D

Source Code

The source code of SOCKMAND can be downloaded at http://tramp-project.org/downloads/sockmand.tgz . The archive includes the source code of SOCKMAND, measurement applications and test applications. The source code is released under the GNU General Public License 3.

The source codes of the measurement tools are included so other measurements of the same metrics can be run in the same way. By using the same tools for measurements, it will be possible to compare other results with the results presented in this thesis.

The archive includes README files which gives instructions on compiling and running.

The MD5 checksum of the archive is:
efdce6f9b99a7e5a816fbfd82851304b