Simple Protocol - Java userspace implementation

Student: **Andriy Padiy**

Student ID: **M00290790**

Supervisor: **Dr Glenford Mapp**

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Networks.

Abstract

Processing power of mobile devices and network speeds are rapidly growing and this has driven the urge to look into user space implementation of network protocols. User level implementations enable applications to have direct impact on communication process by adjusting QoS and level of reliability. This is particularity valid in Local Area Networks were client server application demands are growing and seeking higher performance from conventional protocols that are becoming outdated.

A new transport protocol called Simple Protocol (SP) has emerged and aims to satisfy demands of Local Area Networks with low latency and high throughput while offering custom reliability and QoS.

This paper analyses benefits and drawbacks of user level protocol implementations in contrast with conventional kernel level protocols. SP protocol is then implemented in Java and performance tests are carried out.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Network connectivity technology has made significant progress and the improvements in latency and throughput are apparent. Inter device connectivity speeds are outperforming transfer speeds of some of the conventional internal device connectivity mechanisms. For instance 10 Gbs Ethernet connectivity is becoming common place, whereas the latest Serial Advanced Technology Attachment (SATA) version specification, used for hard drive connectivity, only supports speeds of up to 6 Gbs. Storage Area Networks (SAN) and Network Attached Storage (NAS) are no longer considered to be a niche technology aimed exclusively at large enterprises. Demands for exceptional performance in Local Area Networks (LAN) connectivity, wireless

or wired, are growing and are driven not only by storage applications but also by the increase of richness and widespread of end user applications.

We also see growth in popularity of smart phones, tablets and mobile devices which are becoming more powerful and almost reaching the same level of processing power as conventional desktop and laptop computers. This means that the notion of berrying communication protocols at kernel level of the operating system for performance gains can be challenged. Having processing power to run network communication protocols in user space and fast network connectivity allows applications to have tighter integration in the communication process. For instance reliability and QoS of the connection can be dynamically adjusted at the application level depending on user requirements, whereas typically this has to be tweaked by the system administrator in the operating system or configured at the network level.

The aim of this report is to implement transport level protocol called Simple Protocol (SP) [16] in user space using Java Object Oriented Programming (OOP) language and to carry out performance tests of the protocol in comparison to some of the well know transport level protocols implemented in the kernel such as TCP and UDP. Implementation of SP uses unreliable UDP protocol for transportation but some the tests have shown minimal overhead of this approach and further work will be carried to insert SP payloads directly into Ethernet frames.

1.2 Aims and Objectives

The primary aim of the project is to contribute to development of a new flexible transport level protocol SP by implementing a Java version in user space while applying Object Oriented Programming concepts. This should help make SP more accessible and easy to understand to those wishing to undertake further development or implementations.

The secondary aim is to carry out performance testing which will reveal if user space protocols are a viable option for communication in heterogeneous client server environments where applications need to have more control at the transport level of communication. Also generally poor performance of conventional reliable transport protocols like TCP in wireless environments demands a new lightweight transport protocol [8] [7]. Objectives of this project are to

- Analyse existing SP specification and implementatation
- Implement SP protocol in Java
- Make SP codebase modular using OOP concepts
- Carry out performance tests against kernel level transport protocols (TCP and UDP)

1.3 Method

Literature review was carried out to identify pros and cons of a user level implementation of a transport protocol. Demands for user protocol implementation were also established based on the literature review. Original version of SP programmed in C was analysed and studied. To port C implementation to Java concepts of data structures were translated into Java objects, all of the functions defined in C code were separated into classes depending on which of the objects they operated on. Complete Java version of SP was tested by transfering payloads of various sizes and the same experiments were repeated for TCP and UDP. These tests are also carried out in different network environments. Results were discussed and analysed.

1.4 Resources

Access to the library and IEEE Xplore online database for research were required. The development of the protocol was carried out on a consumer grade Laptop and Desktop PCs with dual core CPUs and 4GB of RAM running Linux operating system and Eclipse Java IDE with. The tests were first performed on the above mentioned laptops and Desktop PCs. 1Gbs tests were done in a Middlesex University labs on identical PCs, to isolate any differences in hardware performance, connected via 1Gbs switch. The same

test were also repeated using 802.11g access point and wireless USB adaptors. To plot test results JavaScript charting library "Highcharts JS" [17] was used. Text editor and LaTeX were used to create the report. Doxygen [12] source documentation software was employed to analyse existing C code implementation of SP.

1.5 Deliverables

Fully functional version of SP coded in Java. Results of the performance testing, and thesis report detailing all the work carried out.

1.6 Report Structure

In Chapter 2 user level implementations of network protocols are reviewed. Pros and cons of this approach are discussed. Chapter 3 looks at SP protocol specification and C implementation is analysed. Java implementation is discussed in Chapter 4 followed by performance analysis in Chapter 5. Conclusions are drawn in Chapter 6 and futute work recommendations are given.

Chapter 2

Literature Review

2.1 Demands for new network framework

There are research papers highlighting the fact that existing communication model based on OSI does not suite fast evolving heterogeneous environments [1] [9]. It is thought that because of the variety of devices and types of communication at the "end" networks of the Internet there needs to be specialised or customisable communication model opposed to the conventional model used in the backbone (the core) of the Internet. Authors in [9] point out that for instance at the transport level of communication, TCP is not ideal in specific network environments, like wireless networks, where protocol straggles to cope with dynamics of wireless environments and needs optimisation. This was the primary focus behind the design of SP in [8]. Authors of

[1] propose a new architecture to accommodate for heterogeneous networking environments. It is called Y-Comm and breaks down conventional OSI model into two, one for peripheral networks and one for core networks or the backbone.

The idea emphasises what has been raised in [9] and proposes a precisely defined communication architecture. Majority of the peripheral networks nowdays are wireless (WLAN, Bluetooth, WiMax, 3G, 4G) and this means that end users with mobile devices, which become predominant and extremely capable in terms of processing power, need to be able to seamlessly switch between these types of networks at the periphery of the Internet.

The ability to switch seamlessly between these networks is known as vertical handover. The authors of [1] point out that because of differences in characteristics of peripheral and core networks it is difficult to offer efficient transport services to the application and QoS during vertical handover. They also claim that TCP is not a suitable transport protocol for wireless environments.

2.2 User Level Protocols

Number of papers look at user level protocol implementation for number of reasons but all have a primary focus of finding out whether performance

penalties of non kernel implementation are worth the benefits of having tighter integration with the application [2] [3] [4] [5]. There is evidence of benefits of this approach in multimedia and video streaming applications where application has direct control of communication at the transport level [3]. Some of the other reasons to implement communication protocol at user-level include improvements of the ability to develop, debug and port protocols [6].

Crucially there has also been evidence of user-level protocols outperforming kernel level protocols [3]. Another aspect of application and protocol tight integration is to enable a better QoS. Some papers raise concerns that kernel level protocols are very abstract and need to have awareness of application demands [9]. This is in particular valid in heterogeneous and wireless environments where number of variables affecting communication between devices is high. Transport protocols also need to have awareness of the context of the data flowing between devices.

Another arguments is the ability to support multiplicity of protocols to achieve simultaneous communication over the same protocol with different configuration [5]. And finally implementation of the protocol at the user-level means high level programming languages can be used which results in the ability to utilise some of the programming techniques like OOP. This allows introduction of higher complexity without compromising debugging

and portability of the protocol [6].

Another downside of kernel level design is the lack of influence from application and poor flexibility as pointed out in [5]. As soon as the piece of data has been passed on to the protocol , applications has no control over that segment of data. Real time media applications for instance may need to dynamically adjust retransmission settings for a protocol to reflect real time demands. Authors of [5] have developed a user-level transport functionality for real-time streaming media applications. The framework consists of two main components : supportive framework and selective retransmission methods. Based on unreliable transport protocol framework allows application access to the packets transmission and enabled selective retransmission of the packets dynamically. Authors argue that in time sensitive streaming applications packets are only valid for a short period of time and wasting resources to buffer and retransmit unnecessary packets is not optimal. Paper goes on to say that because of this it makes sense to let application decide packet retransmission times dynamically. Another prominent feature of this framework is selective reliability. It is highlighted that latency and reliability are contradicting characteristics that are required by streaming media applications. The contradiction lies in the fact that to maintain reliability packets have to be retransmitted, and the more retransmissions occur the higher the latency. To maintain a well balanced combination, authors argue that it is

best to let application decide the level of reliability dynamically based on the needs of content. Selective retransmission is implemented using two methods : deadline based and priority based. It is thought that every packet has it's deadline in terms of purpose to the application. Once the lifetime of the packet goes beyond this deadline, unacknowledged packet is discarded. This is deadline based retransmission control. Priority based selective retransmission ensures that when congestion windows in transport protocol reduces due to packet loss, packets with higher priority are retransmitted.

There are strong emphasis on implementation of user space transport protocol applying OOP paradigm in [6], authors points out that any transport protocol at abstract level has two components : units of information exchange, also known as packets, and structures and procedure to manage the information exchanged in packets. This means that protocols that offer reliability would typically have high levels of structure and procedures to manage the information exchange and the opposite is true for less reliable protocols. Authors argue that by implementing transport protocol in user space and applying OOP techniques the following benefits are achieved : portability,rapid prototyping and debugging, adaptability, configurability and readability [6].

One of the drawbacks of user level implementation is the ability to handle multi-user environments. As discussed in [6] traditionally kernel implemen-

tations of transport protocols demultiplex packets and pass them on to the appropriate process, if the protocol to be run in user-space there needs to be a mechanism to handle multi-user environments at the kernel level to demultiplex the packet and pass it on for processing at the user level. Nonetheless in the context of modern heterogeneous wireless environments, most of the mobile devices are personal and are single user.

2.3 User Level Protocol Performance Considerations

In [5] after carrying out performance testing conclusion is made that primary performance bottleneck is crossing from user to kernel level which requires system calls and buffer copying. To identify performance penalty latency tests were conducted by transferring data using XTP protocol implemented in user level and over IP and transferring the same amount of data using inside IP payload only. The results showed about 2 msec processing overhead for packets that could fit into Ethernet frame and did not require fragmentation. This number raised with payload increase due to fragmentation overhead. Study [3] explains that in order for any protocol to guaranty quality of service end systems at the basic level need to be able to efficiently sched-

ule application and protocol processing to the CPU and efficiently move the data. Based on these principles, authors investigate in detail methods of improving these aspects and come up with a technique of implementating conventional transport protocols in user space. The end result is user space TCP/IP implementation that outperforms kernel space TCP/IP.

At the core of the research is real-time upcall (RTU) technique that allows user space protocol to guarantee efficient processing and user-kernel shared memory that provides efficient data movement mechanisms. The primary advantages of RTU is the ability to offer guaranties to data stream requirements, simplified code and improved runtime efficiency. The shared memory technique reduces memory copy overhead that normally occurs in conventional kernel level protocol implementations where data needs to be copied from network device buffer to kernel buffer and then to user space application buffer.

When analysing efficiency authors highlighted the fact that to ensure good quality of service at transport level, executing protocol at higher priority is not enough and both application and protocol stack need to be given scheduling priority to process network data.

The main benefits of kernel protocol implementation is integration with I/O system, ease of demultiplexing data to user processes and security, however lack of priority levels, and precedence of kernel execution over user level pro-

cesses means that application will always lag behind protocols in processing network data . This results in priority inversion where lower priority traffic, from applications perspective, will be processed before higher priority traffic at the kernel level, since the kernel has no awareness of priorities set by the application [3]. To implement RTU based user-level TCP/IP authors used existing NetBSD kernel source code for TCP/IP protocol stack and ported it as a user level library that could be used by the applications. Authors carried out performance tests by measuring throughput of the TCP/IP with RTU in user space as well as conventional TCP/IP as kernel implementation. User level protocol outperformed kernel level by 20 Mb/s and according to the authors this was due to the fact that extra memory copy operation was eliminated by shared memory technique used in user level protocol.

Another effort to bring network protocols closer to the application is icTCP [10]. The research focus is on development of the framework that exposes access to TCP information as well as control of some of the TCP variables from user-level. This enabled development of user-level libraries that act as extensions to TCP built to satisfy particular needs, one of the examples is wireless environments where duplicate acknowledgements can be controlled in a different manner compared with standard TCP. Authors also argue that by deploying "extensions" to TCP at user-level, TCP stack becomes more flexible, composable and more deployable. The final results

show that in terms of CPU overhead the penalties are minimal and services employing this framework scale well.

In terms of level of exposure of TCP information, framework allows access to all TCP variables as per TCP specification (sequence numbers, unacknowledged sequence numbers and more). In addition to these, information about the each packet is also presented to the user level.

With regards to control of TCP, authors offer access to a range of TCP variables that are modifiable by conventional means (sysctl, tcp_retries etc..). However to keep the values safe and not to create abnormal behaviour of the protocol, authors limit modifiable variables.

As an evaluation of this framework authors carry out a number of tests. Tests are : ease of conversion of conventional TCP to icTCP in a safe manner; CPU overhead introduced by icTCP; how extensible is icTCP in terms of functionality; ease of development of the user libraries.

In terms of CPU overhead authors evaluate icTCP implementation of TCP-Vegas [11] and carry out bandwidth versus CPU tests for TCP Reno kernel implementation and notice only 0.5 percent vraiance.

With regards to scaling, comparison was done by creating a number of connections with icTCP TCP Reno in kernel implementation. The results showed that when icTCP had 4 connections open CPU utilisation reached 100 percent wheras TCP Reno only utilised 80 percent of the CPU with 4

connections. 100 percent utilisation was reached with 16 Reno connections. Throughput degradations became apparent with 4 connections in icTCP and authors claim that with 96 connections TCP Reno throughput is up to 30 percent higher. Authors conclude that by introducing user level exposure to TCP without huge compromises in performance network protocols can be more flexible and more deployable.

Chapter 3

SP Design

SP protocol has been designed with flexibility, minimum complexity and focus on reliability with low latency to operate in LAN environments [8]. In this section SP design will be discussed, design ideas behind SP will be presented, then functional design explained followed by a review of SP implementation in C.

SP was designed to be message based rather than stream based like TCP. Communication between entities would consists of exchange of messages which are broken down into blocks. The idea behind this is that (LAN) communications are primarily transaction based rather than stream-based like in Wide Area Networks (WAN) therefore it would make sense for a transport protocol to follow this model [16].

Reliability aspect of SP is covered by the usage of transmission acknowl-

edgements (ACK packets) and missing parts of the stream are indicated by negative acknowledges (NACK packets).

SP functionality is packet and connection driven like any other transport protocol [6], communication properties between two parties are maintained in a connection state and blocks are used to carry data between the entities. Blocks with data contain headers with information describing the attributes of the block.

3.1 SP Protocol Definition

3.1.1 Header

Packet header strcuture can be seen in Figure 3.1. The header values are described below.

Figure 3.1: SP Packet Header [16]

- DST_ID

  This is packet destination reference.

- SRC_ID

  Packet source reference.

- PK_TYPE

  There are eight packet types in SP which are used for different purpose and are discussed later.

- PRI

  There are four levels of priority in SP, and packets with highest priority are dealt first.

- CB

  Used for Explicit Connection Notification support.

- FLAGS

  This field is used to assign additional properties to the packet. These will be discussed later.

- CHKSUM

  Same technique as used in conventional TCP to offer means of verifying integrity of the packet (or just the header).

- TOTAL_LEN

  Length of the whole packet.

- PBLOCK

  Present block, which indicates which part of the whole message this packet belongs to.

- TBLOCK

  Total block indicates the number of blocks inside the message.

- MESS_SEQ_NO

  This field show the last message sent.

- MESS_ACK_NO

  This field indicates sequence number of the last received message.

- SYNC_NO

  Indicates number of ACKs received and incremented every time the ACK is received.

- WINDOW_SIZE

  Dictates the maximum size of the transfer window.

3.1.2 Packet Types

SP uses different packet types to fulfil particular communication functions. Here is a summary of all the types and their roles within SP.

- START

  Used to establish communication between two entities, combination of FLAG values is used in addition.

- REJ.

  This packet is used to indicate that the connection has been rejected.

- CNTRL

  This type is used for exchange of control information about protocol and is always reliable

- DATA

  Use to carry payload.

- ACK

  Packet that indicates successful receipt of a packet.

- NACK

  Packets are used for flaging any missing blocks of the message.

- ECHO

Used for connection auditing and administration. Measures RRT and verifies established connectivity between the entities.

- END

  The opposite of a START packet, used to tear down the connection between two entities. SP has the following flags to further enhance capabilities by using these flags as additional attributes of the packet.

3.1.3 Flags

- WINDOWS_VAL_ID

  If this flag is set, it means that window size is valid.

- ST_CKS

  By setting this flag SP will be required to checksum the whole packet

- ST_RTR

  Used to indicate that this packet needs to be retransmitted if the checksum is not valid or missing.

- ST_RETRANS

  This flags indicated that the packet has been retransmitted.

- REMOTE_RESET

  When set, it means that connection has been reset at the other end.

- REPLY_REQUESTED

  If this flag is set communicating entity is asked for a reply to this packet.

- REPLY

  Shows that this packet is a reply.

- END_MESSAGE

  Signals that this packet is the last block/packet of the message.

3.1.4 Timers

There are a number of timers employed to guarantee efficient delivery of the packets by executing specific action when some packets are not dealt with on one side or another.

- CONN_TIMER

  Used when establishing new connections. When first START packet is sent with REPLY_REQUESTED flags set, this timer is launched. If no response received back from the other end, START packet is resent. The process is repeated 3 times and the connection opening is stoped.

- END_TIMER

  Is used when the connection closure is initiated. END packet is sent and this timer is launched.

- ACK_TIMER

  Is launched when the and ACK packet is explicitly requested. If there is no response from the other side, ACK packet is sent with REPL_REQUESTED flag set and this process is repeated three times until either the response to the original packet is received or the data is retransmitted.

- RETRANS_TIMER

  Is started when the packets are sent for retransmission. The data is retransmitted 3 times and the connection is closed if it is unsuccessful after third attempt.

- ECHO_TIMER

  Used for internal connection metrics and to ensure that packets of ECHO type are delivered.

3.1.5 Connections States

The state of the connection and all of the characteristics are kept in SP in a connection data type. There are 7 possible states that connection can be in during it's lifetime. These are :

- NOT_IN_USE.

  Connection is not is a usable state and has been shutdown.

- CONN_REQUESTED

  Connection is in this stage during the initialisation stages of opening SP. When the START packet is sent and connection is waiting for the other side to reply, it is set to be in this state.

- CONNECTED.

  This state is assigned to connection when two parties have exchanged START packets and the connection is successfully established.

- END_REQUEST_LOCAL

  The END packet was sent to a remote host and the local connection is waiting for END packet reply.

- END_REQUEST_REMOTE

  END packet has been sent in response to END packet from the other side and the connection is waiting for the stack to complete the closure.

- CLOSING

  Both sides have exchanged END packets and the connection is about to shut down, awaiting to process retransmitted and unacknowledged packets.

- CLOSED

  The connection is closed with no associated packets in the stack.

3.1.6 Reliability Flags

The flexibility of changing reliability "on the fly" is implemented in practice via FLAG fields. Depending on a combination of ST_RTR and ST_CKS FLAGs reliability can be changed. When ST_RTR and ST_CKS are not set, the connection is considered UNRELIABLE. On the other hand when ST_RTR and ST_CKS are both set the connection is RELIABLE. When only ST_CKS is set the connection is forward error corrected on the other side of the connection. When ST_RTR is set and ST_CKS is not packets are retransmitted if are found to contain errors (checksum is not valid).

There is a number of prominent SP features that set it apart from conventional transport protocols like UDP and TCP. With regards to synchronisation, the protocol is considered to be asynchronous

3.2 C Implementation Analysis

There is a successful implementation of user level SP protocol in C. And the primary drive behind this project is to prot this implementation to Java. The reason behind is due to the fact that Java is cross platform and deployment friendly. Meaning that a wide range of devices support Java from custom mobile devices to desktop computers.

In order to port this implementation to Java, existing code based had to be

analysed and studied. This section will document the analysis carried out on the existing source code and describe documentation produced during the process. Due to the nature of C as a programming language the source code is not intuitive and naturally not modular, so understanding complex implementation requires lengthy analysis. As discussed in the literature review sections this is one of the reasons many studies aimed to complex protocol implementations to user level.

The C version of user level implementation of SP consists of two C source files. One called *sp_new.h* and the other *sp_new.c*. These two files comprise a functioning protocol implementation excluding the timer module.

The timer features detailed in the previous sections of SP specification are implemented using a linux kernel module that acts like a server that accepts timer request packets on a UDP socket. If the timer has not been cancelled, by sending a cancellation packet, and the specified period of time expires, this kernel module then sends out a packet back to SP to let protocol know of the expired timer. When SP receives this packet it identifies it as high priority packet, by looking at the UDP source port, and calls appropriate routine to deal with expired timer. This kernel module will not be discussed in detail in document as the implementation of timer features was done using inbuilt Java Runtime Environment (JRE) timer.

sp_new.h file is a header file that contains all data types and structure defi-

nitions of SP protocol. So SP header, SP connection, SP packet and buffers structures are defined in this file. In addition all the type definitions, like packet types, reliability options, protocol ports are also defined in *sp_new.h*. *sp_new.c* file is the main source file that contains all the functions that comprise the SP protocol functionality or the core of the protocol. The file contains over 4500 lines of code with minimal documentation which was challenging to understand.

3.2.1 Source Code Analysis with Doxygen

To aid the with the code complexity issue, documentation tool called Doxygen [12] was employed. Doxygen, when run against the source code files, is able to produce source code documentation in a number of formats. In this case the preferred output was HTML. Doxygen primarily gathers comments from the source code and produces a documentation manual of the source files in a collection of linked html files that then can be accessed in a web browser. This however only works for well documented source, which is not the case in this scenario. However there number of other striking feature of Doxygen that are relevant in this particular case.

Doxygen is able to separate C data structures and present them on separate html pages with all the defined data fields. Doxygen also produced an Index

html page that contained a list of all functions, variables, defines and typedefs alphabetically sorted. These could also be listed separately (i.e functions only). Doxygen also produced an HTML file for each source code file that it finds. Within each there are number of useful features. A list of all variables, defines and functions present in that file are shown. The most useful features on this page are the Caller and Called graphs that are automatically generated for each function present in the code for this source file. Call graph presents all the functions called within this particular function. Caller graph does the reverse and shows how this function is called within this file by other functions. Relationship of all other not directly related functions is also shown. Doxygen also generates HTML enabled source code viewing with syntax highlighting where all functions, variables and defines are hyperlinks to their dedicated section with Caller and Called graph.

In addition to *sp_new.c* and sp_new.h there are two other files : *server.c* and *client.c* which demonstrate SP usage and act as a server and client to exchange messages using SP.

These two files were also present in the Doxygen output.

The code analysis process begun by browsing Doxygen Call and Called graphs starting from the *client.c* file main function. The main function of *client.c* protocol is the starting point of SP protocol life from the client point of view and the caller graph displays all the functions called during the lifetime of SP

protocol from the client perspective. The downside of the Call graph is the lack of order indication of how these functions were called and this required manual process of analysing the source code. The same principle was applied to *server.c* main function which was the starting point of SP life from server perspective.

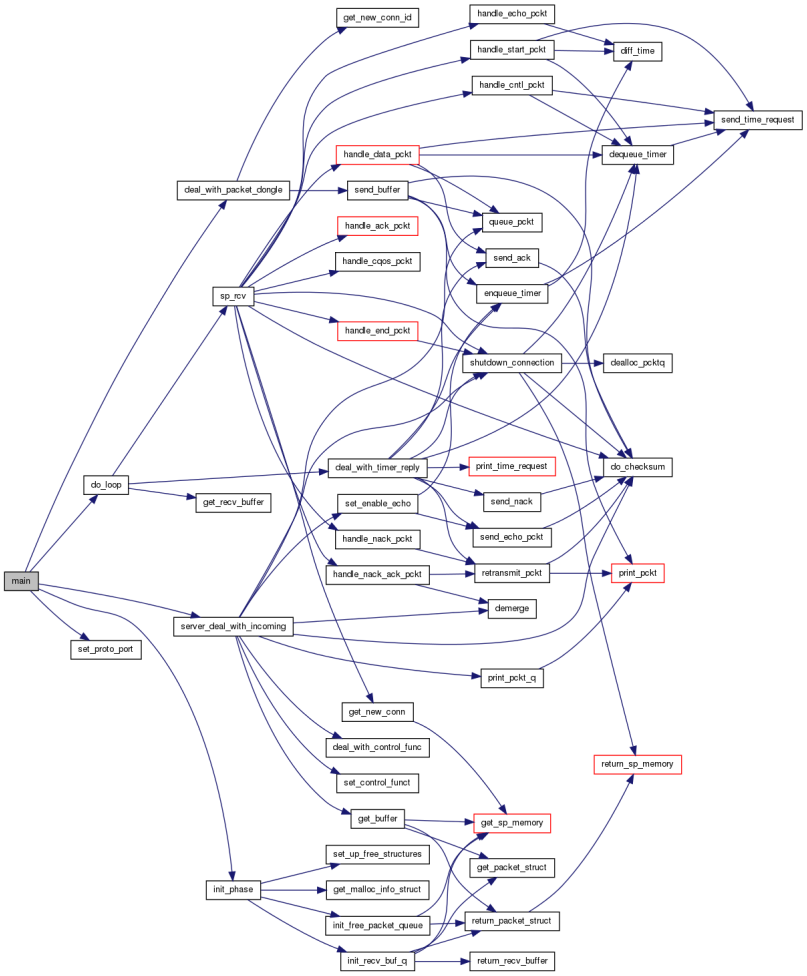

Figure 3.2: main() function Caller graph in server.c. Generated using Doxygen

Having established the order in which all the functions were called and how they related to each other, primitive flow diagram was generated to describe the process of establishing SP connection between *client.c* and *server.c* programs.

3.2.2 Data Types

Having figured the source code structure the following conclusions were drawn. SP C implementation had hierarchical structure for packet and header management. At the bottom of the hierarchy were buffers which are C data types and contained a pointer to the start of the packet data content as well as the pointer to SP packet itself. SP packet in it's turn contained SP header variable which is at the top of hierarchy. SP connection was self contained data type with only a number of references to other data types like packet buffers, queues and timer requests.

3.2.3 do_loop Function

At the core of the functionality is a do_loop function that utilises conditional while loop depending on whether it was launched by the client or the server and acted accordingly.At the core of it's functionality is a UDP listen socket command. When launched by the server it would run and listen on the

UDP socket until the program is interrupted or an unexpected error occurred inside the protocol. When lunched by the client, do_loop() would only listen on the socket if the client is expecting a packet from the server. As soon as the packet is picked up inside do_loop() from UDP socket, it is passed to sp_recv() function which then identifies the type of packet and passes it to a "handle" type functions.

3.2.4 "handle" Functions

Depending on the type of the packet appropriate handle function is called. Apart from START packet, there is one for each packet type. Also, before the packet is passed on to "handle" function, checksum is verified. When packet hits it's destined handle function it goes through a number of routines which are as follows.

- handle_cntrl_pckt()

  Used to process CNTRL packets, does routine checks to see of REPLY_REQESTED is set and send the packet back, cancels timers id REPLY is set, updates appropriate connection fields.

- handle_data_pckt()

  Used for DATA packet processing. Does check to see if REPLY_REQUESTED is set and sends an ACK back, checks to see if the packet is first

DATA packet for this connection, verifies and updated MESS_SEQ and MESS_ACK numbers on the connection. Check to see if the packet is out of order and needs to be retransmitted, checks to see which part of the messages this packet is and if there are any gaps in which case NACK packets are sent. Timers are dealt with appropriately too.

- handle_ack_pckt()

  Timers are dealt with, and the buffer queue administrative work is done to find the appropriate packet and remove it from the queue, if the packet contains REPLY_REQUESTED, response packet is sent back.

- handle_nack_pckt()

  Appropriate packet is found in the send queue and retransmitted. handle_echo_pckt() If the packet contains REPLY_REQUESTED, the packet is simply returned back to the sender. Otherwise we are receiving this packet in response to our request to carry put connection metrics. Values like RTT are measured and updated on a connection status fields.

- handle_cqos_pckt()

  If this is the packet from the other side with REPLY_REQUESTED, then the window size is updated on the connection to that of the current packet and the packet is sent back to the sender with a REPLY field.

Otherwise the packet is simply ignored for now.

- handle_end_pckt()

  Generally speaking connection is shutdown but administrative work is carried out depending on the current state of the connection.

- handle_start_pckt()

  The function is called at the start of sp_recv function. If it's a START packet with REPLY_REQUESTED, a new connection is established and the internal status is pre-populated. Otherwise we are receiving a REPLY to our START packet which initiated a connection. The function then deals with the CONN_TIMER and updates pre-prepared connection values.

The other packet that does not have dedicated "handle" function is REJ packet. When this packet received, connection is simply shutdown from within sp_recv() function.

3.2.5 deal_with_incoming Functions

When the packet has been dealt with appropriately and connection values updated, the next step inside do_loop() is to run a deal_with_incoming() function which updates the connection states of the connection that recently processed packet belongs to. This is triggered by setting global variable "incoming" at

any point of the "handle" function. The contents of deal_with_incoming() function is presently assigned at the "application" level. *server.c* has a server_deal_with_incoming() function and *client.c* appropriately sets it to client_deal_with_incoming().Both of these functions are dfined in *sp_new.c* file.

There are a number of possible operations inside server_deal_with_incoming() function that are dependant on the state of the connection. Administrative work is done on the connection state and the buffers, and the return value is given back to do_loop(). server_deal_with_incoming() function would always return CONTINUE_LOOP value which dictates for the do_loop() to iterate and return to listening state. client_deal_with_incoming() on the other hand would carry out minimal connection status administrative work and would return DISCONTINUE_LOOP to do_loop() which would result in do_loop() while loop breaking and following further down the *client.c* code. The only time client_deal_with_incoming() returns CONTINUE_LOOP for client is when the connection state is not in anticipated state.

This covers the core of the the protocol functionality. There are a number of supportive functions that are called within handle and deal_with_incoming() functions that are not covered and can be found in the Appendix section.

Chapter 4

Java implementation of SP

After having analysed C implementation the goal of porting protocol to Java was twofold, to make code modular and easier to understand, develop and debug; and by implementing it in Java make it cross platform and suitable for mobile devices. There were a number of code structure changes that had to be done due to a diverse nature of two programming languages C and Java. Java is purely object oriented and enforces adherence to Object Oriented Programming methodology. This meant that the *sp_new.c* single file implementation had to be broken down into modules or classes. The Classes that were created followed data type definitions from *sp_new.h* file.

4.1 Classes

This section describes Java classes that have been created in this implementation of SP with any important structural changes to those of C implementation.

4.1.1 SpConnection

This Class defines fields to describe SP connection structure and functions that perform any operations on the state of SP connections. All the functions present in C version and are directly related to the connection entity were ported here and are listed in Table 4.1 with a brief description of what they do.

4.1.2 SpHeader

Similar to SpConnection class, structure of SP header is defined using variables and any functions that directly perform operations on the header have been ported here. However due to a difference in supported data types and lack of support for unsigned data types in Java there is a difference in definition of the header variables. As it can bee seen in a Table 4.2 all variables are defined as signed and either integer or short. Unlike in C implementation where header flags, cb, priority and packet type are "binary packed"

add4Retransmission()	Adds a packet to a retransmit queue for this connection
dealWithAck()	Performs administrative work on the ACK packet, updates queues, order etc.
dealWithEndOfMessage()	Performs administrative work on the packets (next and last) for the the last packet of the message (END_OF_MESSAGE flag)
dealWithSend()	Performs administrative work on the "lastSent" packets of this connection and closes connection if no "lastSent" packets present
sendAck()	Sends ACK for this connection
sendEcho()	Sends ECHO for this connection
sendNack()	Sends NACK for this connection
sendNackFromHandleDataPacket()	One of the secondary functions that sends NACK packet from handle function for Data packets.
shutDownConnection()	Shuts down this connection and sends an ACK packet
SpConnection(bool)	Optional constructor, that initiates the queue and adds default values to some of the variables of this connection
SpConnection()	Default constructor, only initiates receive queue
toString()	Used for debugging purposes and prints out values of all variables set for this connection

Table 4.1: SpConnection Functions

into one variable flags are stored using standard Java library ArrayList class which contains ENUM Flag types defined as part of this class. PacketType is also stored an ENUM type variable that has been defined as part of the class to contain all packet types. All the other types are integers and shorts. This approach introduces inability of C implementation to communicate with Java implementation. Details of this issue will be discussed in subsequent sections. There is only one functions in this class : toString() which returns a String of all SpHeader variable values for debug purposes.

4.1.3 SpPacket

This Class defines the packet structure which uses SpHeader class and a number of functions that perform manipulations on the packet. List of all variables and functions can be seen in 4.3

4.1.4 SP

Core Class that contains main logic of SP protocol. All "handle" functions have been defined here as well as do_loop() equivalent and some other supportive functions. Memory management functions and any buffer related operations in the code that are present in C source were simply left out. This is because Java has inbuilt memory management capabilities. All

short	destId
short	srcId
int	pri
int	cb
List SpHeader.headerFlags	flags = new ArrayList SpHeader.headerFlags()
packetType	pkType
short	chkSum
short	totalLen
int	pblock
int	tblock
short	messSeqNo
short	messAckNo
int	syncNo
int	window

Table 4.2: SpHeader Variables

Public Types	
enum	spPacketCommand { FREE_BUFFER, RETURN_TO_SENDER, DO_NOTHING }
Public Member Functions	
byte[]	packetToByte ()
SpPacket	byteToPacket (byte[] stream)
String	toString ()
void	myFree ()
void	queuePacket (LinkedList¡ SpPacket ¿ queue)
Public Attributes	
SpHeader	header = new SpHeader()
SpPacket	next = null
SpPacket	prev = null
String	data
char	retrans = 0
spPacketCommand	command = SpPacket.spPacketCommand.DO_NOTHING

Table 4.3: SpPacket Variables

global definitions like SP_PORT and DatagramSocket (UDP socket) are defined here. Complete list can be found in Table 4.4. The logic inside SP Class has been ported from C implementation and do_loop() is replaced with run() function. The number of functions has significantly reduced and apart from core handle functions that deal with the packet types are resequence(), merge() and demerge() functions that perform administrative work on the queues and dispatchPacket() function that is used as an interface for applications wishing to transport packets using SP. Another noticeable difference is a single dealWithConnection() function that is a combined version of sever_deal_with_incoming() and client_deal_with_incoming() functions in C implementation. Functions takes a boolean value "server" as a switch and acts accordingly.

4.1.5 SpLogFormatter

There has been a number of new features added in Java implementation and one of them is a logging capability which enables 3 levels of logging to a file or console output. Java inbuilt logging Logger Class was utilised and the default formatting of the output was modified using SpLogFormatter Class. Author felt that having Logging capability development and debugging of the protocol would be much more efficient. There are three levels of logging :

Public Member Functions	
	SP (int portNo, boolean type)
	SP (int portNo, boolean type, Level logLevel, String LogFile)
boolean	resequence (LinkedList SpPacket queue, SpPacket packet, SpConnection connection)
void	run ()
void	handleStartPacket (SpPacket packet, InetAddress othersideIP, int othersidePORT)
void	handleRejPacket (SpConnection activeConnection, SpPacket incomingPacket)
void	handleCntrlPacket (SpConnection activeConnection, SpPacket incomingpacket)
void	handleDataPacket (SpConnection activeConnection, SpPacket incomingpacket)
void	handleAckPacket (SpConnection activeConnection, SpPacket incomingpacket)
void	handleNackAckPacket (SpConnection activeConnection, SpPacket incomingpacket)
void	handleNackPacket (SpConnection activeConnection, SpPacket incomingpacket)
void	handleEchoPacket (SpConnection activeConnection, SpPacket incomingpacket)
void	handleCqosPacket (SpConnection activeConnection, SpPacket incomingpacket)
void	handleEndPacket (SpConnection activeConnection, SpPacket incomingpacket)
int	dealWithConnection (boolean server, SpConnection incoming)
void	dispatchPacket (SpConnection activeConnection, SpPacket packet2send, int pblock, int tblock)
void	closeSP (SpConnection connection2close)
SpConnection	openSP (String destIP, int connectionType, int maxMTU, char priority)
Static Public Member Functions	
static short	doChecksum (SpPacket packet)
static void	sendPacket (SpPacket packet, DatagramSocket socket, InetAddress othersideIP, int othersidePORT)
static void	demerge (LinkedList SpPacket mq, LinkedList SpPacket lq)
static void	merge (LinkedList SpPacket mq, LinkedList SpPacket lq)

Table 4.4: SP Class Functions

INFO, WARNING, CRITICAL. Logging output is added in the critical sections of the code and depending on the outcome of the function appropriate level is raised. INFO level is specified across all of the code base whereas WARNING and CRITICAL is mostly put in places where SP protocol might malfunction. The logging can be disabled when the protocol is launched by instantiating SP Class. For performance purposes it is advisable to disable logging as entries are written to a file on a file system. An example output of verbose INFO level logging can be seen in Figure 4.1

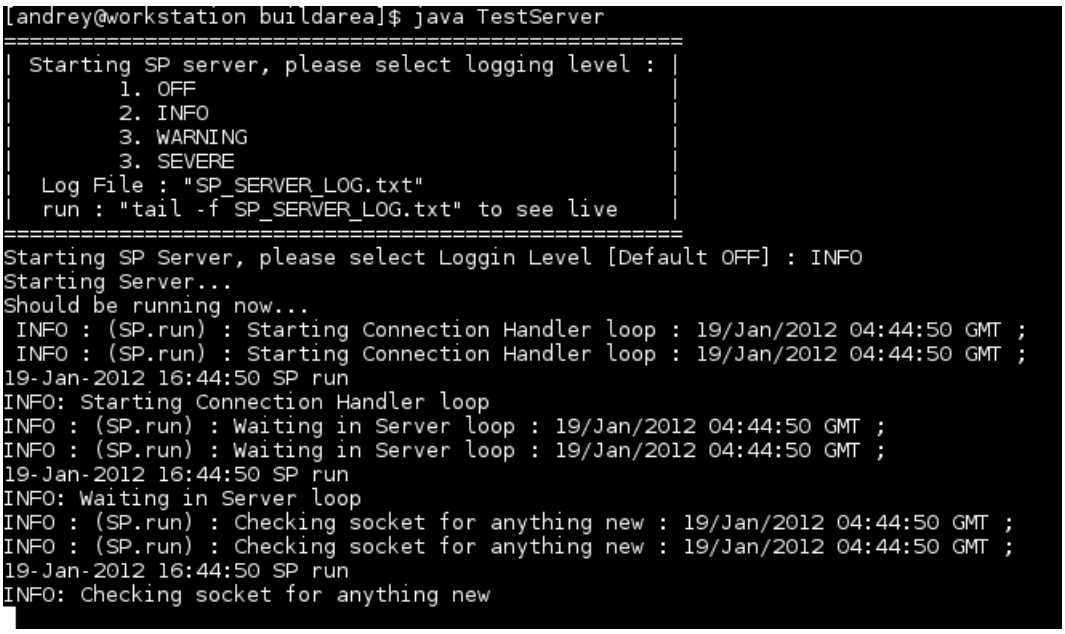

Figure 4.1: INFO level Logging enabled

4.1.6 SpTimer

Another new feature is the internal timer support. C implementation relies on the specially designed kernel module that listened on a UDP socket and accepted specially crafted packets with timer requests and would send this packets back when timers expired. This mechanism has been replaced with a new one that relied on internal Java timer. SpTimer Class has been defined to represent timers. Just like with a C version and as per SP specification a number of timer types have been defined. Multithreading concept was also utilised to aid with a timer requests. The main logic behind the new timer implementation with multithreading is explained next. Anywhere within the code an instance of SpTimer Class can be created. Using this object a new timer of any type can be launched using enque() function which then calls sendTimeRequest() function. The latter function will prepare a new thread and will schedule it to start in the specified by the timer amount of time. The scheduling is achieved using Java inbuilt ScheduledExecutorService Class. When the timer expires, the thread is launched and executes the block of code found inside run() function defined in SpTimer Class. run() function contains several sub sections of code that perform operations depending on the type of timer, these are caught by if statements. SendTimeRequest function also deals with cancellation of the already launched timers. This is achieved by

simply running "cancel" function on an instance of ScheduledExecutorService Class instantiated when the timer was launched. There were a few caveats to this implementation. The first one is the concurrency issue with the state of the connection. The argument was that the state of the connection should only be altered by one entity at the time and it should be atomic. In other words, the changes to the state of the connection should be "all or nothing". This was achieved by implementing another Java inbuilt Class called Lock. Every connection had a lock variable which was of a Class type Lock. When a new thread launched by the timer had to perform any operations on the connection, an attempt was made to "grab" a lock. The thread would wait in the background until the lock is obtained. Once the lock is obtained, changes to the connection are made and the thread releases the lock.

The second issue with using threads to execute timer code was that there was a possibility of a connection being shut down while some timers threads were still waiting to execute. This problem was overcome by introduction of a new SpConnection variable called timerTheadCount. This was a simple integer value that defaulted to 0 when connection was created and every time a new timer thread was launched is increased. Before connection state could be changed to CLOSED and connection can be shutdown, the value of the timerTheadCount variable is checked. Providing it's 0 the connection can be closed, otherwise it will wait in CLOSING state for all the timer threads to

execute. timerTheadCount value is reduced whenever the thread executes it's code or the timer is cancelled.

4.2 Queues

The are a number of queues that are used for internal packet and connection management. These were implemented using following inbuilt into Java library classes :

- Vector
- LinkedList
- BlockingQueue
- List

Using these classes opposed to implementing own like in C version meant that some of the inbuilt features like updating the size of the queue, accessing and modifying content of the queue was easier and already existed which reduced the amount of code.

4.3 Java to Java interface

The underlying communication between two devises using SP is done on top of UDP protocol just like with a C version of SP. UDP packets are loaded with SpPacket objects and transferred unreliably. Before SpPacket is put inside UDP datagram, it is serialised. Serialisation process is carried out inside packetToByte() and ByteToPacket() functions in SpPacket class. The process is as follows : ByteArrayOutputStream and ObjectOutputStream objects are created, ObjectOutputStream is then instantiated with ByteArrayOutputStream as a parameter, meaning that anyhting written to this ObjectOutputStream will be written to ByteArrayOutputStream. SpPacket object is then written to ObjectOutputStream and convered to ByteArray using inbuilt toByteArray() function in ByteArrayOutputStream class. This function then resturns output of toByteArray() and it is then packed into UDP datagram. The reverse is perfirmed using ByteArrayInputStream and ObjectInputStream classes in byteToPacket function which returns an instance of SpPacket object. This ofcourse restircts this implementation of SP to Java only, the workarounds are discussed int he next section.

4.4 Java to C interface

For simplicity and ease of development the initial implementation of SP header in Java varied to that of C and simply used a separate variable for each of the header item. Once the protocol inner workings were implemented and the base layout of the code was established an attempt was made to enable Java to C communication. The logic behind was to first be able to read C packets from the network and convert them into SpPacket objects inside Java. To achieve this byteToPacket() functions that was called as soon as run() function inside SP class read from the UDP socket had to be modified. Instead of deserialisig to SpPacket object like in a pure Java version, the data had to be read from the stream of bits and values extracted by individual bits. There were a few obstacles to this approach. First if the fact that C implementation used host (aka little endian) order to transfer data whereas Java uses network (big endian) order. To overcome this problem a new class was introduced to the code base called LittleEndianDataInputStream [13]. This class when instantiated takes InputStream and allows access to the data that is not conventionally available in Java. As can be seen in Tables 4.5 and 4.6 there is a mismatch between the data types, however LittleEndianDataInputStream class contains a number of functions that allow access to some data types that are not present to Java conventionally. For instance

there is a method called readUnsignedShort() which, by manipulating bits in the stream using SHIFT and AND operations converts unsigned short read from InputStream into Java integer.

Name	Size (bits)	Range
boolean	1	true or false
char	16	0 to 65535
byte	8	-128 to 127
short	16	-32768 to 32767
int	32	-2147483648 to 2147483647
long	64	-9223372036854775808 to 9223372036854775807

Table 4.5: Java Data Types [14]

Adopting this technique a few more functions were added to LittleEndianDataInputStream class. readTypePriCb() and readSyncNoWindow(). These were designed to read "binary packed" SP headers from C. By placing unsigned C variable into a twice larger signed Java variable and performing SHIFT and AND with all 1s operations on the bits values can be extracted and converted into conventional Java variables. For instance to convert SP SYNC_NO and WINDOW headers that are "binary packed" into one 32 bit long unsigned integer in C, readSyncNoWindow() function reads four bytes

Name	Size (bits)	Range
char	8	signed: -128 to 127 unsigned: 0 to 255
short int (short)	16	signed: -32768 to 32767 unsigned: 0 to 65535
int	32	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	32	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	8	true or false
wchar_t	16 or 32	1 wide character

Table 4.6: C Data Types [15]

(32bits) from the InputStream. Each byte is then casted into int and AND operation is performed with 0x000000FF. To read the signed value of the first 24 bits (WINDOW header) the first byte is SHIFTED by 16, second is shifted by 8 and all are joined together with the third by bitwise OR operation and casted to a long. The content of the fourth byte is simply cast into signed long. This conversion results in two signed long variables with values extracted from "binary packed" unsigned integer. The same technique was used in readTypePriCb() function to extract packet type, priority and cb from C SP packets. Extracted value were then cast again into more efficient data types as per Table 4.2. Initial tests shown successfull results and consistent conversions, however with more thorough testing a few issues were discovered. When one of "binary packed" values in C was assigned to it's maximum value and the other to it's minimum, the result of the conversion in Java would not be accurate. This approach was not investigated any further and will be listed in "Future work section" of this report.

Chapter 5

Performance Testing and Evaluation

5.1 Test Environment Overview

To gauge the performance of Java implementation, two applications were written. One application acted as a server listening for SP packets, picking them up from the socket and printing output of the DATA packets out. The client application would generate and send SP DATA type packets. Applications are TestServer and TestClient Java classes. TestServer prompts user for debugging level as seen in 4.1, then object of SP class is created with construct parameters of "true" for server boolean variable and specifying logging level and log file location. SP.run() function is then launched, which causes

protocol to initiate and to launch run() loop and listen on the socket for packets. This behaviour is identical to that of C implementation and do_loop() function. TestClient class is at the core of tests. It contains a number of for loops that are used to initialise an instance of SP protocol, with server boolean value set to false to indicate client usage. A new SpConnection is then created using openSP() function found in SP class. This function takes an IP address of the server, connectionType (reliability), maximum packet size, and priority. Return value of this function is a SpConnection object. dispatchPacket() function from SP class is used next. It takes SpConnection obeject, returned from openSP() function, and SpPacket packet loaded with String type data. Pblock and tblock values are also taken. Using this method payloads of varying sizes are sent. The maximum payload size during tests is set to 32KB. This means that tests with payloads below 32KB will be sent as a single message with a single packet (pblock 0, tblock 1). Payloads larger than 32KB are broken down into a number of packets that are sent multiple times by invoking dispatchPacket function multiple times with pblock=n and tblock=((total payload)/32KB) . Where n is in the range from 0 to (((total payload)/32KB)-1). Test values were doubled from 1024 bytes to 262144 bytes. To generate payloads a single character of 1byte size was filled into SpPacket "data" variable of String type using a for loop. The measurement was carried out by taking a timestamp value as soon as proto-

col is initialised, SP class instantiated, and after a connection is closed using function SP.closeSP after having transferred all the payload. The time difference is then substructure and added to an array. For accuracy the same tests are performed number of times and then are printed out from an array to the console in CSV format and averages are also calculated. Number of tests were carried out in different environment to see how Java implementation compares to conventional kernel level implementations of UDP and TCP. These tests were carried out in a lab using identical PCs with Linux Fedora 15 operating system. The results are discussed in the next sections.

5.2 TCP and UDP Tests

To gauge performance in comparison to TCP and UDP a similar TCP and UDP test client and server applications were written in Java. New classes were created, TCPServer and TCPClient, UDPServer and UDPClient. Client applications had identical structure to those of SP but used TCP and UDP to transfer payloads. Server applications utilised a while loop and a listening socket which would accept payloads, buffer them and print the output onto console.

5.3 Loopback Interface

Loopback network interface tests were carried out to test SP and TCP performance without any influence of external network issues.

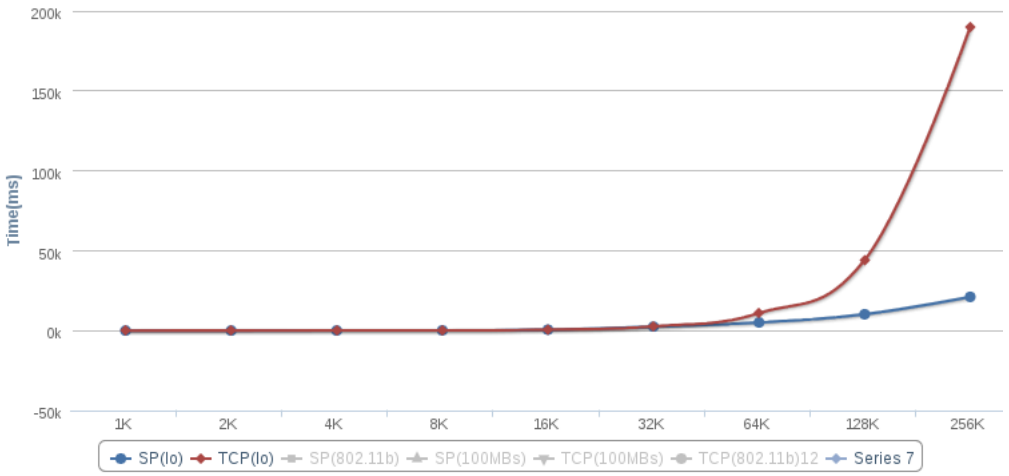

Figure 5.1: SP vs TCP loopback interface

Results show that TCP starts to degrade in performance as soon as packets reach sizes above 16KB. SP maintenance a steady increase of the transfer times until the packet size reaches 64KB in size. These tests are averages of 100 transfers of each packet type.

5.3.1 1GBs LAN

The tests were identical to those of loopback interface, for every payload size starting from 1KB and doubling the value to 256KB packets were transferred using SP and TCP. The average values from 100 runs were taken for both

SP and TCP runs. The results can be seen in Figure 5.2

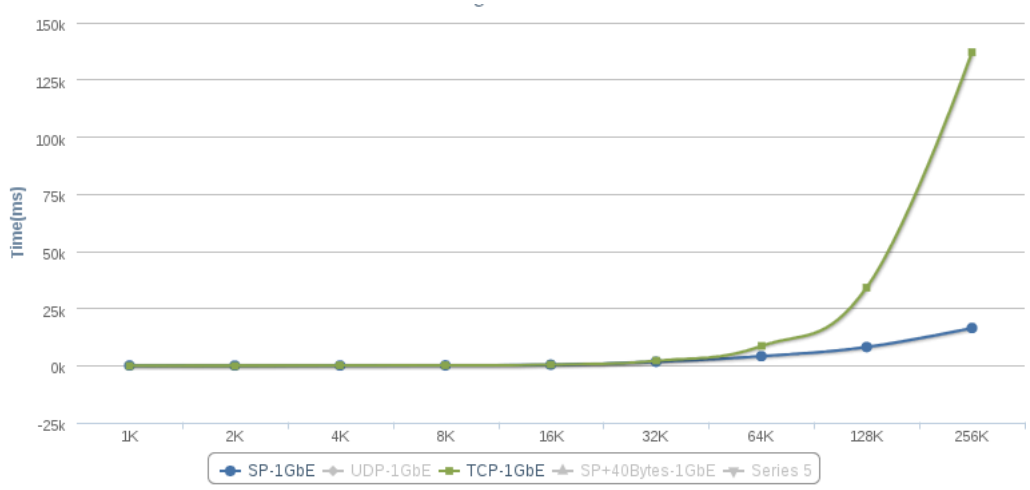

Figure 5.2: SP vs TCP 1GBs average of 100 runs.

The results show that SP outperforms TCP as soon as the packet size is increased over 16KB. Another interesting discovery is the time it takes to transfer the first packet is considerably longer in SP compared with TCP. The former takes around 50 milliseconds to transfer the first 1K packet, whereas the latter just over a few milliseconds. This can be seen in Figure 5.3, it is believed that this is due to the fact that during SP connection establishment test client ans server Java classes begin JRE memory allocation which requires extra time.

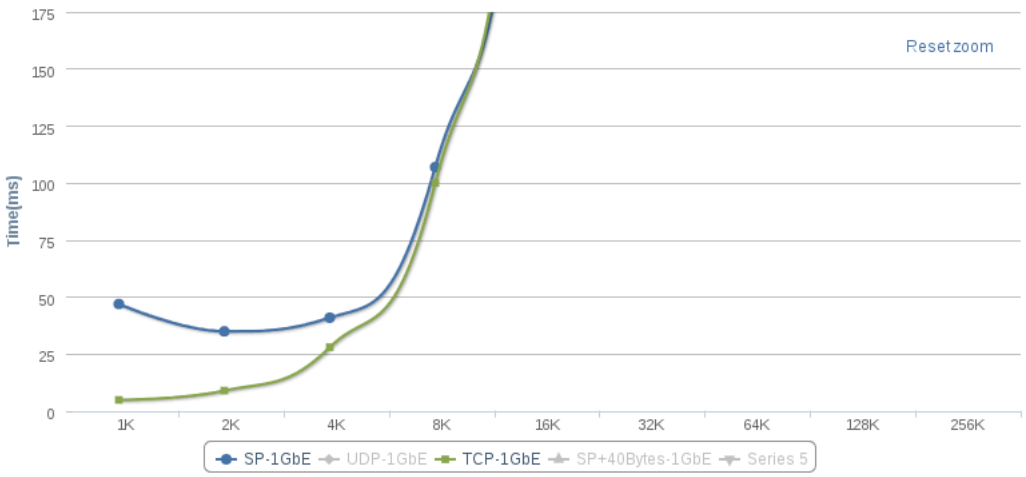

Figure 5.3: SP vs TCP 1GBs average of 100 runs. First packets.

Similar to TCP tests a set of tests was carried out to see how much overhead SP uses on top of UDP.

The tests were repeated 100 times for every payload size starting from 1KB and doubling the value to 256KB. The average values from 100 runs were taken for both SP and TCP runs. The results can be seen in Figure 5.4

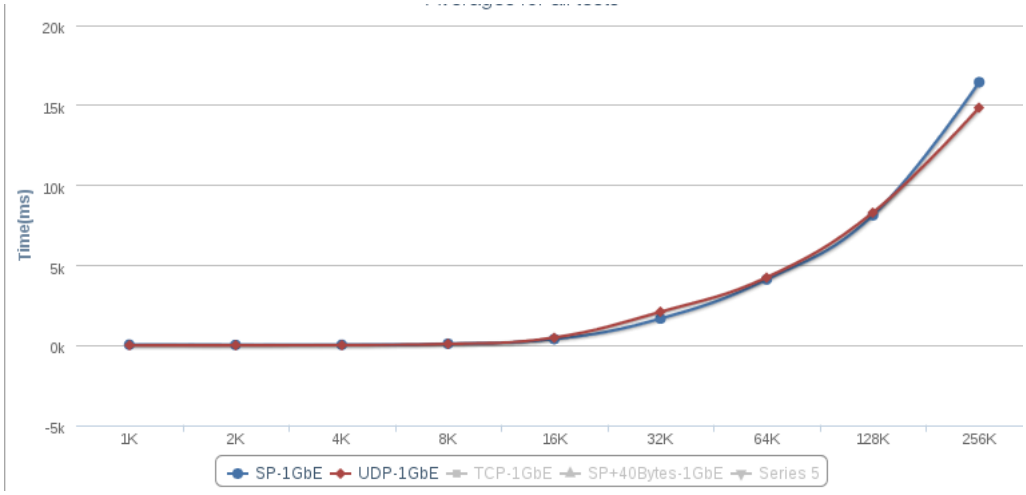

Figure 5.4: SP vs UDP 1GBs average of 100 runs.

This results reveal that there is very minimal overhead created by SP on top of UDP. The average transfer time for the largest packet, 256K, SP took 16461ms to complete the transfer, whereas UDP took 14865ms to performs the same operation on average. The "slow start" phenomena in SP is confirmed in this comparison too as UDP takes roughly the same time to transfer the first 1K packet as TCP (Figure 5.5)

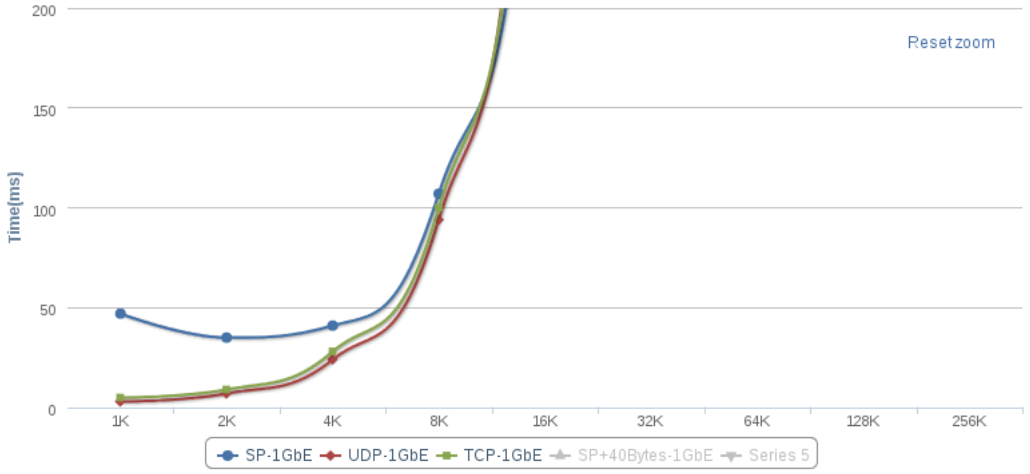

Figure 5.5: SP vs TCP vs UDP 1GBs Average of 100 runs. First packets.

5.3.2 802.11g

An attempt was also made to perform the same set of tests in wireless environment. However the results were inconclusive. Initial test displayed in 5.6 were performed on development desktop and laptop that were connected

via 802.11g wireless access point and were located close to each other. The transfers were successful. However when the same attempt was made but in a lab environment, due to a dropped packets a bug in protocol functionality was discovered. The details are described in further work section at the end this paper. The comparison was also made of TCP and SP transfers in wireless environment against test via loopback interface as shown in 5.7

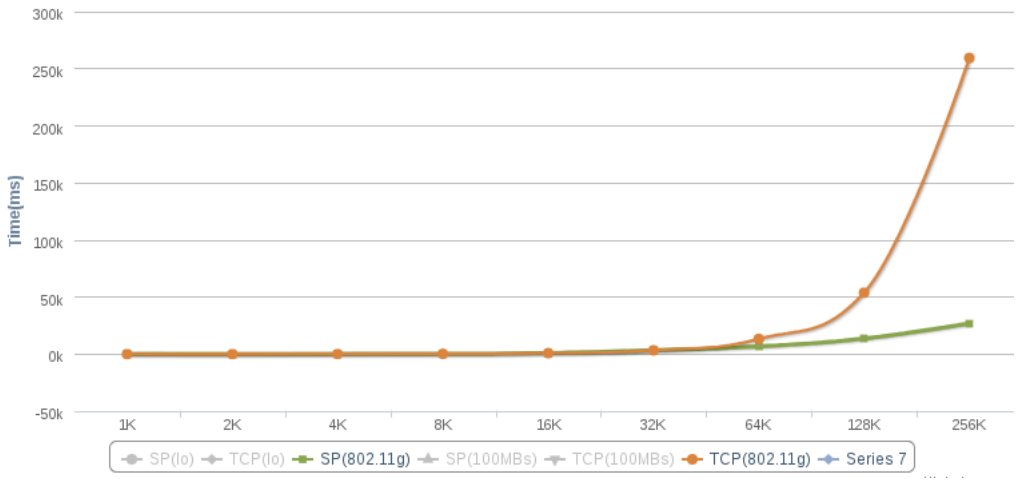

Figure 5.6: SP vs TCP 802.11g interface

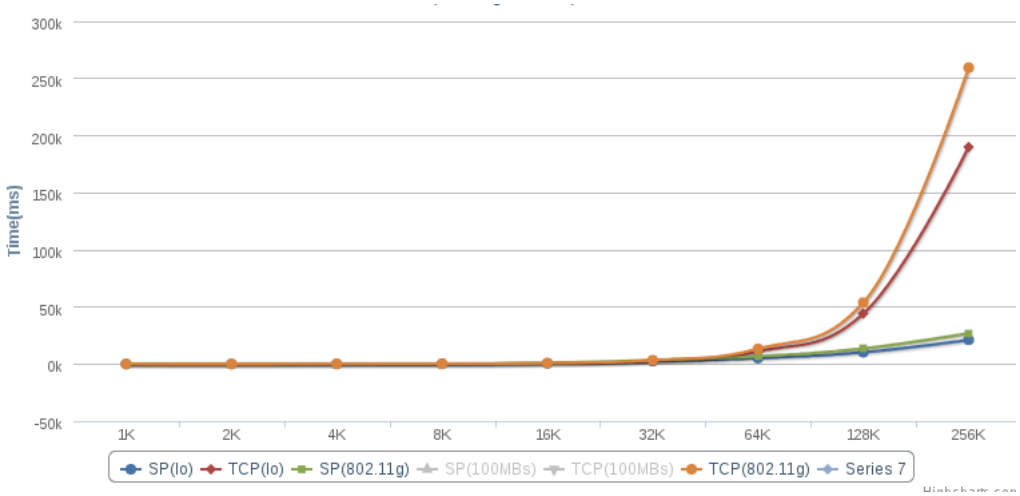

Figure 5.7: SP vs TCP 802.11g and loopback interface

5.4 Summary

Initial tests prove that Java implementation of SP is capable of transferring data and to function as a transport protocol and implementation can be considered successful. The performance tests reveled that there is very minimal overhead of SP on top of UDP. Tests performed in wireless environment were inconclusive due to discovery of a bug. TCP tests results were also unreliable as the reason of dramatic decreases in transfer speed has not been identified. The details of this are discussed in conclusion and further work section at the end of the report.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The performance tests against TCP are inconclusive and decrease of transfer time of payloads above 16 KB need to be investigated. The potential cause of this could be due to some internal congestion control mechanism in TCP or the way test application was written. On the other hand UDP results were a good indication that there is very minimal performance overhead in using SP on top. Porting the code to Java made SP more flexible in terms of deployment, development and debugging. The code does not have any external library dependencies and can run on any platform that supports Java. SP has been run on variations of Linux distributions different version of Windows as well as different versions of JRE.

6.2 Future Work and Recommendations

Further work needs to be done on Java implementation of the protocol. The outstanding issue of Java to C interface needs to be investigated. There are two potential approaches to this, one is to redesign internal data types in Java to match those in C by creating some byte level manipulation techniques to overcome signed and unsigned variable type mismatch. Another approach is to pick up from where the author left with "adaptor" interface creation inside packetToByte() and byteToPacket() functions inside SpPacket class. Some outstanding work inside the protocol design needs to be completed. Error handling functions needs improving and there needs to be a method created to measure packet sizes. Presently there are no means of detecting the true packet size of SP in Java. An attempt was made to perform this by converting the object to byte array and calculating the size, however the author felt that this was not an accurate representation as some internal Java meta data about the object is kept and the total value does not repleteness total value of the SP packet. Potentially this can be implemented along side Java to C interface where byte level manipulation performed. There was also an attempt made to create a unit testing framework around SP using Junit and Mockito object mocks to make protocol development more agile. The source files to some of these Junit test case are attached along with the code.

With regards to performance testing and evaluation, there needs to be an investigation carried out into why TCP was significantly dropping it's transfer rates on packets above 16KB. Once this problems is solved, another set of tests needs to be performed to make the test results conclusive. More thorough testing is required in wireless environments with simulation of dropped packets. The tests presented in this paper were performed with two devices in very close range to Access Point (AP). When an attempt was made to run the tests while simulating some packet loss, number of bugs was discovered with the way connection are terminated. When END packet is sent, the client waits in a do_loop() for an acknowledgment from the server. However if the END packet is lost, the client connection is not terminated and the client would not break out of the loop. This is where END_CONNECTION timer was introduced. However the re-run of the test needs to be performed. In addition, more wireless tests need to be carried out with simulation of measurable packet loss scenarios and to graph how SP responds to such scenarios. Another ambition was to port this implementation to Android device which supports JRE. This would enable performance testing in truly mobile environment.

References

[1] G.E.Mapp, F.Shaikh, J.Crowcroft, D.Cottingham and J.Baliosian, *"Y-Comm: A Global Architecture for Heterogeneous Networking Invited Paper"* In: 3rd International Conference on Wireless Internet, 22nd-24th October. 2007, Austin, Texas.

[2] A.Barros,F.Pacheco,L.M.Pinho, *"A Complex Protocol Layer as a linux User-Space Process"* In: Industrial Embedded Systems, 2006. IES '06. International Symposium, 2006 , Page(s): 1 - 4

[3] R.Gopalakrishnan, G.M.Parulkar, *"Efficient User-Space Protocol Implementations with QoS Guarantees Using Real-Time Upcalls"* In : Networking, IEEE/ACM Transactions Volume: 6 , Issue: 4, 1998 , Page(s): 374 - 388

[4] G.Mapp,S.Pope,A.Hopper, *"The Design and Implementation of a High-Speed User-Space Transport Protocol"* In : Global Telecommunications

Conference, 1997. GLOBECOM '97., IEEE,Volume: 3, 1997 , Page(s): 1958 - 1962 vol.3

[5] J.Liu, *"Toward a user-level real-time transport protocol with selective reliability "* In : Military Communications Conference, 2008. MILCOM 2008. IEEE, 2008 , Page(s): 1 - 7

[6] T. Strayer, G. Simon, and R. E. Cline Jr., *"An Object-Oriented Implementation of the Xpress Transfer Protocol"* In : XTP Forum Research Affiliate Annual Report, pages 5366, 1994

[7] G,Mapp,D,Thakker,D,Silcott, *"The Design of a Storage Architecture for Mobile Heterogeneous Devices "* In : Networking and Services, 2007. ICNS. Third International Conference,2007 , Page(s): 41

[8] L.Riley, *"A Fast Transport Protocol for Heterogeneous Networking: The Development of a Simple Transport Layer Protocol for Local Network Signaling"* Postgraduate thesis, Middlesex University, 2011.

[9] J.Crowcroft,S.Hand,R.Mortier,T.Roscoe,A. Warfield, *"Plutarch: An argument for Network Pluralism"* In : Proceedings of ACM SIGCOMM Workshops, 2003.

[10] H.S. Gunawi, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, *"Deploying Safe user-level network services with icTCP"* In Proceedings of the

USENIX Symposium on Operating Systems Design and Implementation (OSDI),2004, pages 317-332

[11] L. Brakmo,S.O'Malley,L. Peterson, *"TCP Vegas:New techniques for congestion detection and avoidance"* In Proceedings of SIGCOMM '94 Symposium, 1994, pages 24,35

[12] http://www.doxygen.org

[13] http://code.google.com/p/guava-libraries/

[14] http://softpixel.com/ cwright/programming/datatypes.java.php

[15] http://www.cplusplus.com/doc/tutorial/variables/

[16] A. Padiy, G.Mapp, L.Riley, *"yRFC The Simple Protocol (SP) Specification"* In YComm research published on "http://www.mdx.ac.uk/research/areas/software/ycomm_research.aspx"

[17] http://www.highcharts.com/

Appendices

```
import java.io.BufferedReader;

class TCPServer
{
   public static void main(String argv[]) throws Exception
      {
         String clientSentence;
         String capitalizedSentence;
         ServerSocket welcomeSocket = new ServerSocket(6789);

         while(true)
         {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStrea
            DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

            clientSentence = inFromClient.readLine();
            System.out.println("Received: " + clientSentence);
            //capitalizedSentence = clientSentence.toUpperCase() + '\n';
            //outToClient.writeBytes(capitalizedSentence);
         }
      }
}
```

Figure 6.1: TCP server source code used for testing

```java
import java.io.*;
import java.net.*;

class UDPServer
{
   public static void main(String args[]) throws Exception
      {
         DatagramSocket serverSocket = new DatagramSocket(9876);
            byte[] receiveData = new byte[2621440];
            byte[] sendData = new byte[2621440];
            while(true)
               {
                  DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
                  serverSocket.receive(receivePacket);
                  String sentence = new String( receivePacket.getData());
                  System.out.println("RECEIVED: " + sentence);
                  //InetAddress IPAddress = receivePacket.getAddress();
                  //int port = receivePacket.getPort();
                  //String capitalizedSentence = sentence.toUpperCase();
                  //sendData = capitalizedSentence.getBytes();
                  //DatagramPacket sendPacket =
                  //new DatagramPacket(sendData, sendData.length, IPAddress, port);
                  //serverSocket.send(sendPacket);
               }
      }
}
```

Figure 6.2: UDP server source code used for testing

CD Contents

Following items have been included in the CD

* Full SP Java source code (over 4000 lines)

* TCP and UDP Client and Server applications used in testing

* SVN Repository containing full revision history of the project lifecycle.

* Doxygen reports after running over C and Java versions of SP.

* Image attachments

* All test results in CSV format as well as html files with ploted graphs using JS Highcharts [17]

* LaTeX source of the report.