High-Level Languages for Low-Level Protocols

Iain Oliphant

A dissertation submitted in part fulfilment of the requirement of the Degree of Master in Science at the University of Glasgow

Department of Computing Science,
University of Glasgow,
Lilybank Gardens,
Glasgow, G12 8QQ.

CS5M - April 2008

Abstract

Network protocols are often implemented in languages such as C which provide high efficiency but are difficult to maintain or extend. This project aims to show that by using a high-level language to develop a network protocol an implementation is more readable and modular and therefore is more maintainable and easier to extend. To demonstrate this an implementation of the Transmission Control Protocol (TCP) has been created in the high-level language Scala.

TCP is a highly complex transport level protocol that is generally implemented in C inside operating system kernels. Scala is a relatively new programming language which allows programmers to code in either the functional or object-oriented style, or a combination of the two. The implementation relies heavily on the Actors model of concurrency, that Scala provides, to create a highly concurrent TCP. Scala offers a rich type system which has been utilised to represent the structures of TCP in a more accessible way and in a manner that provides encapsulation at each layer, improving the overall understandability of the system.

This implementation is compared with the existing Linux and FreeBSD implementations to demonstrate the ways in which development of network protocols can be enhanced by using a high-level language.

Acknowledgements

I would like to thank Colin Perkins, my project supervisor, for proposing this project and for his continued support and guidance as it progressed.

I would also like to thank the other academics in the ENDS department for their interest, insight and questions.

Contents

Chapter 1

Introduction

This document describes the work carried out in completion of the Masters in Science research project. The aim of this project is to establish that high-level languages can be used to program network protocols and, further, that in doing so the quality of the code improves. The project describes an implementation of the Transmission Control Protocol (TCP) in the high-level programming language Scala. In analysing both the development of this implementation as well as the resulting code, an argument for the improvement in code quality is presented.

Network protocols are implemented in languages that do not address the complexity of their design. Large protocols such as the Transmission Control Protocol consist of thousands of lines of complicated code, holding large amounts of state and performing many concurrent operations and yet C is the still the language of choice for implementing it. The aim of this project is to demonstrate that not only can high-level languages be used to develop network protocols but also that their use in development can lead to a code base that is easier to read, more modular and therefore more maintainable or extensible.

Network protocols are frequently implemented in languages like C and C++ due to the high performance that these languages provide [23]. This efficiency is granted at the sacrifice of many important features provided by high-level languages. Notably C has no memory management above allocation and deallocation of the memory space by the user. This enables programmers to write code that runs quickly but that is insecure and highly unportable.

There is good reason for using efficient languages for network protocol implementation. Firstly a fast network protocol stack leads to the ability to transfer data around the network at greater speed, assuming the network connections are sufficiently fast as to outweigh the speed of the stack itself. Secondly the network protocol stack must, at some point, be able to access hardware devices, e.g. to transmit data on an Ethernet network the datalink layer must be able to interact with the network card that provides the physical Ethernet connection, this is not easily achieved in high-level languages. Lastly the network protocol stack gener-

ally operates within an operating system's kernel, which is likely to be written in a language like C or C++, leading to ease of compatibility in maintaining the same language of implementation.

Equally there are many arguments for the use of high-level languages in any implementation. Perhaps most importantly high-level languages provide abstractions over memory access. In C a programmer would have to manually allocate a predefined amount of memory and maintain a pointer to that location until it is no longer required, at which point that memory must be deallocated. In a high-level language a programmer can simply instantiate their variables, structures or objects and allow the language to allocate the memory required for its type. Furthermore the programmer need not explicitly delete anything as a garbage collector can identify unreachable objects and free the occupied memory on a periodic basis. The use of high-level languages can lead to a better design by supporting one particular design paradigm, e.g. object-based, object-oriented, etc. This means that the programmer is able to better model the problem and this in turn leads to more modular, less tightly coupled code. An improvement in design combined with good syntax and structure can contribute to provide more readable code and this in turn makes this code more maintainable and extensible by being easy to interpret and understand.

There are, therefore, many reasons for using high-level languages at some level in the network protocol stack. The Transmission Control Protocol (TCP) lies in the transport layer of the OSI model and is at a sufficiently high-level to not be constrained by a need to access hardware devices or operate with kernel level privileges. TCP is a highly complex protocol that provides many functions to the applications that utilise it, notably guaranteed delivery and congestion control. Frequently TCP is implemented in C within operating system's kernels [26]. The code base for these implementations is large, tightly coupled and lacks modularity. These properties make understanding the source code, for maintenance or extension, very difficult.

This document proceeds as follows; chapter 2 describes the research carried out in advance of and during the implementation. This information has been used to establish a language of implementation, an understanding of the protocol and current implementations and a knowledge of the related work being carried out in this research area. Chapter 3 describes the approach taken to the implementation of TCP, in particular how Scala has been utilised and the overall design of the system. Chapter 4 discusses specifics of the implementation giving examples of the use of Scala's features. Chapter 5 evaluates the implementation, considering the quality of the software compared to that of the Linux kernel. Finally, chapter 6 summarises and concludes this work and also presents some suggestions for the future direction on research in this area.

Chapter 2

Background

In developing this project a survey of relevant work has been conducted. This survey can logically be divided into three investigations: firstly, the consideration of high-level languages which could potentially be used for a redevelopment of TCP; secondly, a study of the protocol itself; lastly, a study of work relating to the use of high-level languages within network protocols or other systems development. The result of these studies is an understanding of several mainstream languages that show potential for network protocol development, a deep understanding of the Transmission Control Protocol and knowledge of the related research recently conducted in this area. Each of these areas are discussed in the sections that follow.

2.1 Languages

In this section four languages are considered as potential candidates for the implementation of network protocols. The languages under consideration are SML, Haskell, Erlang and Scala. For each of these the relative merits are considered and their use within systems development and in particular network protocol development is considered.

2.1.1 Standard ML

Standard ML (SML) [8] is a typed functional programming language which means that it regards functions as first class values, such that all values are functions and all functions are values. SML uses automatic memory management and a garbage collector to abstract over memory providing higher-level functionality.

SML provides support for modular design by having modules, or structures, which can define code which is logically separate from other code. The components of a structure, e.g. functions or values, are then accessed by calling them with the

structure name as a prefix. Data abstraction is achieved by placing a signature, or interface, into the structure which separates the implementation from the definition. Any code that uses a structure must then conform to the signature for that particular structure.

SML supports higher order functions, which are functions where at least one of the parameters or the return value is a function. This makes logical sense in a functional programming languages as all functions are regarded as first class values. This leads to the ability to curry functions. A curried function is one in which the provision of a subset of the total parameters will return a function which takes the remaining parameters as input. This means that functions can be partially evaluated leading to the ability to create functions in a more convenient manner. For example, given a simple function `add` that takes two integers if the function is applied to a single integer, e.g. `add 2`, it will return a function that takes a single parameter and, in this case, adds 2 to its value.

SML is an interesting functional language that provides many features which could be useful in protocol development. However it remains a quite basic language, in particular in comparison to languages like Java or Scala. Furthermore the syntax, while common in other functional programming languages, is awkward and therefore difficult to read and understand. This leads to it being somewhat in contrast to the aims of this project.

2.1.2 Haskell

Haskell [24] is a functional programming language which provides support for currying and higher order functions, as in SML. The language has been used to some success in systems development due to its strong efficiency properties. Haskell is a lazy functional programming language meaning that it uses lazy evaluation, i.e. a statement is only evaluated if and when it is needed. The basis for this approach is that if a statement is not required for the program to execute then the statement should not be evaluated. This allows Haskell to have slightly improved performance compared to languages which do not perform lazy evaluation.

Haskell abstracts over memory making it more secure and reliable than programming in C, it uses a garbage collector to facilitate memory management. The language has a very concise syntax not unlike that used in SML and this leads to arguments that the language is more readable than others. However, as with SML this syntax is quite unfamiliar. Source code indentation is used to aid function definition. Thus instead of enclosing functions within braces as is common in C-style languages Haskell relies on all code within a function definition appearing below and right of the start point of the function signature.

In [12] Haskell is discussed as a language of implementation for replacing C in systems development. In particular the paper discusses the possibility of using Haskell in the development of operating systems, the principles of which transfer

well to the development of network protocols. The paper discusses the ways in which Haskell can be utilised to produce efficient and modular code. However, this serves to identify the need for careful development in using Haskell to achieve properties that come more naturally with other languages.

Haskell is an interesting language that has been used successfully for systems development. It presents a strong option for this project due to this track record and its high-level features. Furthermore, Haskell is likely to produce a high-level implementation of a network protocol that retains much of its efficiency. It is important however, that this project not pursue this efficiency at the sacrifice of readability or understandability. The other languages under consideration in this section move on from Haskell to a yet higher level and as such present a better option for implementation in this project.

2.1.3 Erlang

Erlang [14] is a functional programming language designed to support highly concurrent, distributed and soft real-time applications. The language, therefore, has been used in many systems level programming tasks, with some success. [20] is one such example where the authors used Erlang to construct the TCP/IP stack, this implementation will be discussed further in section 2.3.

Erlang has a concurrency model different from the thread based style of languages such as C and Java. Instead of relying on altering some shared memory to allow threads to interact Erlang has the notion of different processes communicating with one another via asynchronous message passing. Correctly utilised this model reduces the possibility of deadlocks or race conditions. Each process is referred to as an actor and each actor has control over a certain amount of state. If another actor requires that a piece of state outwith its control be changed it sends a message to the relevant actor which processes the message. Reduced chance of deadlock means the need for synchronisation of resources is also reduced, as this removes a significant amount of code from highly concurrent programs, e.g. locks, monitors, mutexes, etc.

Erlang's model of concurrency provides levels of safety that would be useful in network protocol development. Concurrency is inherently present in protocols as at least required are a sender and a receiver. Erlang could aid this project firstly in providing an abstraction over memory and memory management, aided by its garbage collector, and secondly by providing a concurrency model that reduces the possibility of deadlocks and race conditions. Such behaviour is difficult to verify in the thread based concurrency model provided by the pthreads package in C. Furthermore, concurrency is built in to Erlang and the creation of processes is a

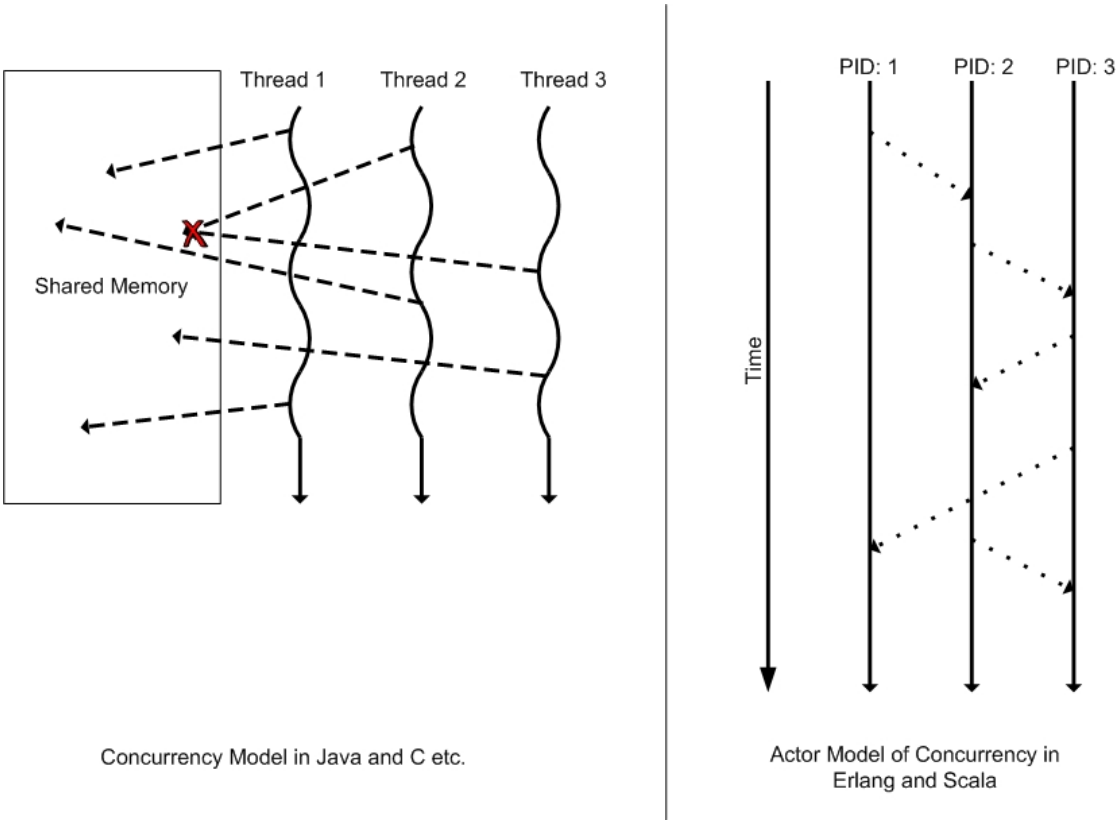


Figure 2.1: Comparsion of models of concurrency

light-weight activity, i.e. it uses few resources and processes are created quickly. Thus, the amount of concurrency in the design of a network protocol could be increased, perhaps providing a better design, but certainly giving designers an increased number of options. The readability and code size are likely to be reduced because of the concurrency model also. Creating, managing, grabbing and relinquishing locks requires lines of code around that code which performs the work required. Erlang will not require such constructs and so the code required for locking is removed and therefore the readability should be increased as a reader need not take the time to understand how and why the programmer has controlled access to variables accessible by many threads.

```
loop()
    receive
        {source, msg} ->
            source ! dealWithMsg(msg);
            loop();
    end
```

Figure 2.2: Erlang source code showing an actor which receives a message from some source and then responds by passing back the result of a function.

2.1.4 Scala

Scala [17] is a new programming language that combines the functional and object-oriented paradigms. Programmers are free to use a functional programming style, where immutable variables are passed through a series of functions or an object-oriented style where state is stored in a series of classes which extend each other and interfaces. Scala interacts with the Java programming language to provide access to the massive set of APIs that Java provides. Furthermore Scala is able to directly interact with Java classes. This is true to the extent where a Scala class can extend a Java object, or vice-versa. Scala also compiles to Java bytecode making it as highly portable as Java itself as it will run on a Java Virtual Machine (JVM). Scala has been designed to be a scalable language, meaning that large programs can be easily constructed in a modular way that leaves them easy to maintain. Its syntax is a combination of C and Java style, for the object-oriented programming and an Erlang style for functional programming. This combination of styles is provided in a manner that means that they can be combined at the programmer's leisure. This ability means that the design of network protocols could follow a semi object-oriented, semi-functional design with the paradigms intertwined in a way that best suits the protocol in question. Essentially Scala

is offering a flexibility that allows for each of the design techniques to be used where they are most appropriate. Previous approaches to network development have taken a single paradigm approach as they were constrained by the language used.

In Scala type declaration differs from that of C and Java in adopting the *name : type* style of declaration seen in languages such as Ada. Scala uses a static type system that allows the type of an object to determine the function available to it. It also provides support for generics and abstract encapsulation via abstract types. A static type system means that Scala code is significantly more reliable than its dynamic counterparts. Static type systems rule out the chances of certain run-time errors by being able to identify the capabilities of a particular type at compile time. In dynamic languages the type of a variable cannot be identified until run-time as the variable does not have a type declaration. This means that a higher percentage of code can fail at run-time. Furthermore, run-time errors have the possibility of not being found during testing as a particular branch of the code that is not regularly taken may be missed, thus the code is more likely to fail during real operation with a dynamic type system. Scala supports parametric polymorphism and dynamic dispatch, just as Java does.

Scala's object-oriented structures are very much like Java. This consistency with Java makes the combination of the two languages simple. The ability to combine Scala code with Java code is potentially useful for this project. Firstly, Java supplies one of the largest sets of APIs of any programming language, much of which has been well optimised. Using these APIs reduces the programming burden as well as providing encapsulation of those aspects of the development which can be handed off to the libraries. Perhaps more importantly though, much work has been done with Java in and around the networking area. This includes several accessible implementations of systems which give access to raw sockets through Java compatible code. This means that the difficulty in accessing the IP layer can be alleviated by using an open-source tool that exposes a Java raw sockets wrapper.

Scala adopts the same model of concurrency as Erlang, the actors model. It also adopts a similar syntax to Erlang in dealing with actors (see figure 2.3). Scala actors are extensions of the *Actor* class and any messages that are sent to a class which extends this class are dealt with by an *act* function. The *act* function contains a *receive* or *react* clause which subsequently contains a series of case statements, each of which use pattern matching to identify which action is to be taken with a particular incoming message. This method of pattern matching could provide a valuable way of discriminating between different TCP segment types when sending to the IP layer, for example. Of course, as Scala is able to interact so closely with Java a programmer is free to use the standard thread and lock method provided by the Java concurrency APIs. As noted in section 2.1.3, the actors model of concurrency will result in a smaller amount of code by eliminating that used up by locking mechanisms.

```
def act() {
    while(true){
        receive {
            case (source: Actor, msg: String) =>
                source ! dealWithMsg(msg);
            case msg =>
                println("Unrecognised message received")
        }
    }
}
```

Figure 2.3: Scala source code showing an actor which receives a message from some other actor (source) and then sends a message back with the result of a function.

The security offered by Scala will also be improved compared to that of other languages. Other languages discussed here support automatic memory management thus eliminating buffer overflows as potential security risks, but Scala's compiling into bytecode means that it offers increased security compared to these languages.

Scala therefore presents the most attractive language for use in this project. It is the language at the highest level of all those considered, it offers a combination of programming styles and enhanced security and reliability. Furthermore, its interaction with the Java programming language and its relatively common style of syntax means that more programmers will be able to find the code to be accessible for maintenance, extension or just for understanding.

2.2 Transmission Control Protocol

TCP [22, 25] has been selected for implementation in this project due to its complexity relative to other network protocols. Other protocols occupying the transport layer, such as UDP [21] or DCCP [15], do not have the same complexity as TCP due to its nature of storing state. TCP provides a guaranteed end to end service for the application layer, this is not true of the other major transport protocols. UDP provides a service that attempts to send but makes no attempt to confirm the arrival of the data, instead it just continues to attempt to send. This is useful for many applications, including multimedia streaming where not every packet need arrive as the difference in quality of data is not discernible to human senses. DCCP is essentially an extension that provides congestion control to UDP. However, there are many applications where this type of service is unacceptable. TCP provides a service where all data is required to arrive at its destination, assuming non-exceptional circumstances. To do this however it must store a large amount of state and it must communicate regularly with the receiving

host to identify which bits of data have gotten through and which have failed. This project intends to show that the complexity of network protocol development can be represented in a better manner by using a high-level language, for this reason the protocol with the most complexity is the protocol of choice to highlight the differences.

Another possibility for implementation in this project is a protocol below the transport layer or an application above it. TCP has been chosen over these options on the basis that the likely candidate in the network layer is the Internet protocol (IP). IP while being complicated also relies on some interaction with hardware. This is not achievable with the language chosen for this project and so C (or similar) would need to be used to provide some functionality. This is at odds with the principles of this research and so a protocol that is above hardware interaction is more desirable. An application running onto TCP or UDP is not desirable to implement as a high number of applications at this level are already commercially available in high-level languages and as such this would not represent an advancement of this research.

The following sections describe TCP and the extensions which have been made to it including congestion control techniques. Attention is drawn to the areas which this project would be required to implement, and those which it will be likely to ignore due to time constraints or the lack of the feature's use.

2.2.1 Connection Establishment and Data Transfer

A TCP connection is established via a three-way handshake. The three-way handshake is so named as it consists of the originating, or active, host sending a synchronise (SYN) request to the receiving, or passive, host. This SYN is then responded to with an acknowledgment (ACK) indicating that the passive host is willing to make the connection. Finally, the active host confirms the receipt of this confirmation via another ACK. At each end of the connection a TCP exists holding a transmission control block (TCB) which holds the state of connection. A connection, and therefore a TCB, is identified by its initial sequence number (ISN) and the port numbers upon which information is received and sent. TCP specifies 2^{16} port numbers upon which connections can be established therefore allowing multiple TCP connections to be run concurrently.

Data is transferred in segments, the size of which is determined by the receiving host. Each byte of data has an associated sequence number with which the receiving host can ascertain the order in which the data was meant to arrive. This allows for reordering of the segments should they be rearranged as they are passed through the network. To facilitate guaranteed delivery each byte of data is acknowledged by the receiver. This takes the form of a segment, with data or without, which has the sequence number of the received byte in the acknowledgment field of the header. The sender is then aware that every byte of data with sequence number

less than the acknowledged sequence number has been received. As a sender is also aware of which sequence numbers have been sent it can resend any packets that it deems to have been lost. Furthermore, if a receiver receives a byte with sequence number higher than the expected sequence number it can send a duplicate acknowledgment, indicating that it is missing the specified sequence number.

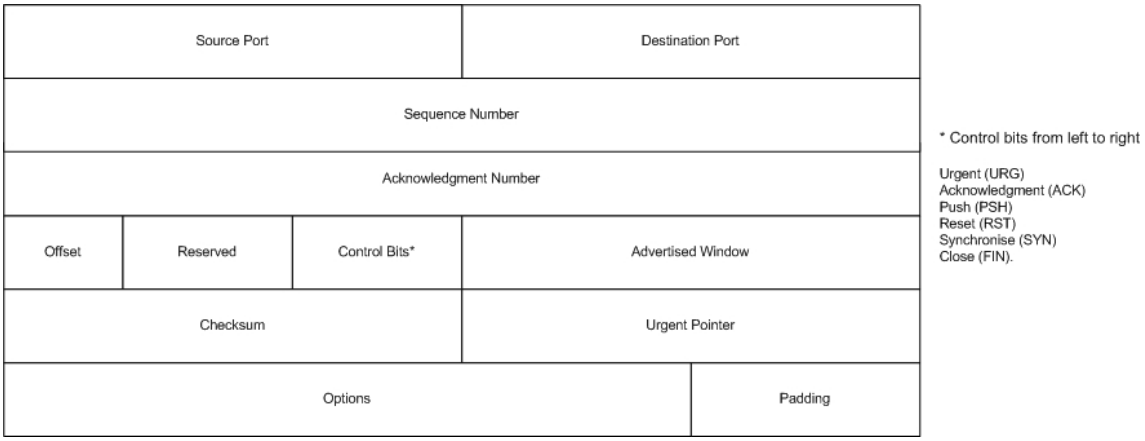


Figure 2.4: TCP Header format

The details provided in this section outline the behaviour of a standard TCP implementation with no extensions. Figure 2.4 shows the format of a TCP header. Any implementation of TCP must at the very least conform to the described functionality and the format of the TCP header as described in the diagram. Various extensions to TCP also exist, some of which will be discussed in the next section.

2.2.2 Enhancements and Extensions

This section discusses the enhancements and extensions that have been made to TCP. In particular it dicusses the congestion control mechanisms which are now required of any real TCP implementation and an experimental extension, Selective Acknowledgment, which attempts to provide more communication between two associated TCPs to allow for more efficient sending of segments.

RFC 4614 [10] is a roadmap of TCP and its extensions. It lists and summarises each of the major TCP RFCs and indicates which of them must be implemented as part of a standard TCP implementation. It goes on to detail the RFCs which

indicate extensions to TCP which are optional and which are suggested as well as those that are experimental or are not generally implemented. Some of the extensions in that document are outlined in this section, in particular those that are relevant to this project.

There are several extensions to TCP that are now required of any complete implementation, one set of which is specified in RFC 2581 [3]. These extensions are designed to control the effects of network congestion on the ability of the TCPs to communicate. Network congestion occurs when routers on the network receive packets at a rate that exceeds the rate at which it can forward them. A router under these circumstances will react by queuing incoming packets but, should the incoming rate persist, inevitably the space available for these queues is exhausted and the router has to drop packets.

TCP employs a congestion control mechanism originally outlined by Jacobson in [13]. Its congestion control mechanism is actually four separate algorithms which work in combination to combat or prevent congestion at the various stages at which it can occur in a connections life span. The first of the algorithms is *slow start*. Slow start is a preventative measure aimed at reducing the likelihood of a TCP contributing to a congested network. Prior to the inclusion of slow start a TCP selected its rate of sending based on the rate at which the receiver can handle the data (this information is contained in the passive host acknowledgment of the three-way handshake). If a receiving TCP is able to receive faster than the rate at which a router on the path can process packets then the connection is immediately doomed to enter congestion. Slow start alleviates this by increasing the send rate by one segment for every acknowledgment received up to some predefined threshold. By doing this slow start reduces the chance of starting in a congested state.

Once this threshold is reached the next of the congestion control mechanisms, congestion avoidance, begins execution. Congestion avoidance continues to increase the send rate of the sending TCP, but at a slower rate than slow start. In actuality it is increasing the send rate by one segment every round-trip time (RTT). In either of these modes, slow-start or congestion avoidance, it is possible for congestion to occur. This is in fact the reason for the slow start and congestion avoidance algorithms, they are probing the network to see at what rate congestion occurs. A packet loss signals that congestion has occurred and the send rate is adjusted appropriately.

The new send rate after packet loss is dependent on the remaining congestion control algorithms, fast retransmit and fast recovery. Fast retransmit is defined to respond to the situation where a packet is dropped on the network because of conditions other than congestion. In this situation the segment is retransmitted and the fast recovery algorithm begins execution. Fast recovery has control until a non-duplicate acknowledgment is received, i.e. the receiver has received the missing

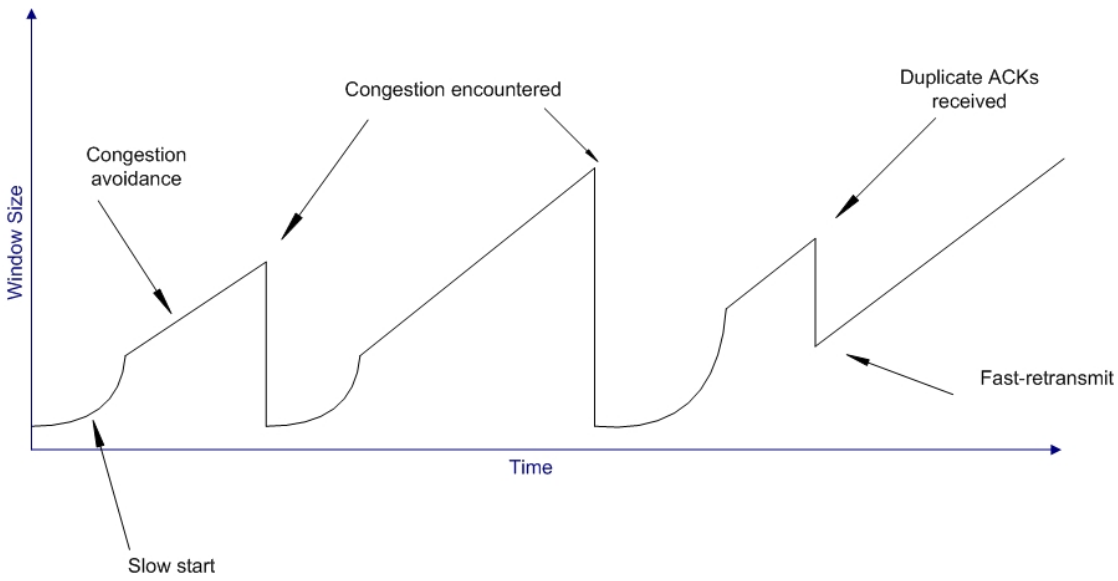


Figure 2.5: TCP Congestion Control behaviour

segment. Once this occurs fast recovery sets the send rate to be half the current rate and hands control over to the congestion avoidance algorithm. Packet loss not due to congestion is detected if a receiving TCP sends more than two duplicate acknowledgments as the sending TCP can infer that although the receiving TCP has not received a particular sequence number, it has received subsequent sequence numbers (hence the duplicate acknowledgments) and therefore the packet loss must have been the result of a temporary network problem. The contrasting situation is that the packet loss was due to congestion. In this situation the TCP responds by setting the slow start threshold to half the current send rate and beginning slow start.

The congestion control algorithms described above are now standard in TCP and as such must be implemented by this project to allow for comparisons to be made. They also add significant complexity to the implementation by introducing more timers, and relatively complex execution paths dependant on a high number of variable situations.

Selective Acknowledgment (SACK) [18, 11, 6] is a TCP option. SACK intends to extend a sending TCP's knowledge of the segments which have arrived at the receiving TCP. Without SACK a sending TCP is aware only of the latest contiguous block of segments which have arrived, i.e. all bytes up to the latest acknowledged sequence number have successfully arrived. However, in a network where packet loss or reordering is commonplace it may be beneficial for a sending TCP to be aware of those segments which arrived but which cannot be acknowledged due to a missing segment before it. The aim of this extension is to allow a sending TCP

to avoid sending segments which have already been successfully received after a packet loss or reorder.

SACK uses the ability of TCP to provide options. Where options are present they are signified by the options field in the header and occupy a set additional amount of space. SACK uses this space to specify each contiguous block of segments that it has received by denoting a start sequence number and an end sequence number. It is then the responsibility of the sending TCP to use this information to send on only those segments which will fill the gaps in incremental order, i.e. all data between the acknowledgment and the first SACK block, then between first block and second, etc.

SACK would represent an area where high-level languages can help improve the maintenance of network protocol code. SACK is inherently an extension, in that an acknowledgment format exists in the specification but SACK tries to improve this in a different location. Therefore to be compatible with other TCPs an implementation of it for this project would also be an extension to the original protocol. This means that by focusing on the SACK implementations of a C implementation and comparing it to the one created in completion of this project, the ease of maintenance could be assessed. SACK is a complicated extension and this makes it appealing for using a high-level language. However, the complexity of SACK means that the time taken to implement it will mean that it is outside the scope of this project.

2.3 Related Work

This section discusses work that has already been undertaken in this area. In particular it dicusses implementation of network protocols in SML, Erlang and Prolac. For each of these languages the merits of the implementations have been discussed and where possible the failures and/or areas for further work have been picked up on to allow this project to progress using hindsight gained from the previous work. Each of the implementations have different approaches to the design of the protocol and so these features are also of particular interest to this project.

2.3.1 FoxNet

The FoxNet project [4, 5] is the implementation of a TCP/IP stack in Standard ML (SML). The project was aimed at producing an implementation of TCP/IP that could boast good performance and good structure. SML supports this modularity as well as providing memory management and garbage collection and so FoxNet has aims in line with the project proposed here. The structure of the implementation in FoxNet is based on the x-kernel [19]. X-kernel provides a structured design that

has each layer of the stack producing the same signature to the layers above and below it. In this way protocols can be mixed, e.g. to allow TCP to run directly over Ethernet without ever using the IP layer.

The design of implementation takes a form very much in line with standard kernel implementations and the original specification of the protocols. This would appear to be a strange decision as SML provides good support for high modularisation which would allow this implementation to reduce the coupling and make the code more readable and maintainable. Instead these properties are enhanced only by the arguably improved readability of the SML syntax and the removal of any memory management code due to SML's memory management system and garbage collector.

The performance of FoxNet is measured in [9]. This technical report performs a comparison between FoxNet and the TCP/IP stack from Digital Unix. The results of the experiments show that while the throughput for the Unix stack is better over small transfers, the FoxNet implementation in SML fast approaches the same throughput as the file size increases to approximately 1MB.

The FoxNet project provides essentially the first step towards moving protocol development into high-level languages. The language chosen, SML, is able to maintain a reasonably high efficiency compared to C implementations making it more attractive than other arguably higher-level languages. However, the project proposed here intends to move on to attempt to use a general purpose language that provides a high number of features, with the aim of investigating the impact on efficiency rather then preserving it.

2.3.2 Erlang TCP/IP Implementation

In [20] an implementation of the TCP/IP stack has been implemented in Erlang. The implementation has been created to allow the authors to research the provision of distributed fault tolerance for TCP connections. As discussed in section 2.2 a TCP connection records a high amount of state to provide its services. To provide fault tolerance a failing host must have its connections copied quite precisely at some fall back host. As a server crash can be unpredictable and undetectable there is no way for a failing server to guarantee transfer of connection state to another host. The authors suggest several methods for replicating this state, ranging from routing all TCP connections through a central server to having a host observing packets to obtain connection state information.

The Erlang TCP/IP implementation uses multiple processes which communicate with each other via message passing, using the actors model of concurrency. Actors are created for send and receive processing in each of the protocols, these can then communicate with the layers above and below by passing messages in to the actor for the desired protocol's sender or receiver, as appropriate. TCP in their implementation also has send and receive actors but, different from the other

implemented protocols, it relies on an additional process which manages the state of TCP connections, essentially this process controls the TCBs.

It is worth noting that not all of the implementation is achievable in Erlang. At the lower levels of the implementation, where hardware access is required, C wrappers are used to allow Erlang code to interact with the hardware. C is also used for checksum computation, due to the authors understanding that a significant performance impact would be made by repeatedly performing the computation in Erlang. This project will take the approach of investigating the impact of such computations by attempting to use the chosen high-level language at all conceivable levels (of TCP). A useful study would be to identify the areas of high or sustained computational difficulty and attempt to optimise the high-level language to compensate in those areas, perhaps as further work.

The focus of experimentation for the Erlang implementation was efficiency. A 1GB file was transferred between hosts with throughput being measured. The authors achieved a throughput of 150Mbps compared to the 615Mbps that the Linux TCP/IP stack was able to achieve on the same equipment. The paper reports here that the 615Mbps result for throughput of the Linux kernel is limited by the network card installed, not by the software. This means that although the paper claims that a throughput of one quarter that of the Linux TCP/IP stack has been achieved, the results do not support it. To effectively convey the relative speed of the implementations a new network card would have to have been installed which was able to support the Linux stack until the point at which the software is limited.

The Erlang TCP/IP stack was intended for use in support for distributed fault tolerance of connections and the paper briefly describes a potential model for achieving this. The paper describes a model whereby there are three levels of synchronisation between primary and backup servers. Upon connection establishment some data will need to be synchronised with the backup server, presumably this data will be port numbers, ISN etc. After this some data will need to be synchronised after each change is made, e.g. sequence numbers of last byte received and the size of the receive window. Other data can be synchronised after the connection has failed, by sending data to the opposing TCP to solicit an ACK from which more data can be computed or derived. Notably whilst this model is described in detail there is no indication from the authors that the model has ever been constructed. This means that, although appealing in concept, its potential for real world application is entirely untested and to claim that it will work is unfounded.

The implementation makes good use of the actors model of concurrency (discussed in section 2.1.3) with many different threads of activity communicating via asynchronous message passing. Given that Scala has a similar model of concurrency this design could potentially be a valuable starting point for this project. Furthermore the Erlang implementation reports a strong performance in efficiency, achieving 150Mbps. Although the use of Scala will limit the throughput further still, an interesting comparison will be to see how the two languages differ in speed,

given a similar design.

2.3.3 Prolac TCP/IP Implementation

TCP has also been implemented in Prolac [16]. Prolac is a language developed purely for protocol implementation with the aim of making protocols more readable, understandable, modular and extensible whilst maintaining an efficiency comparable to that of C protocol implementations. Prolac is compiled into C leading to an arguably more readable C code base, and access to C debuggers and compilers. This means that whilst Prolac is more readable and is object-oriented, the object based C code which it compiles to retains some of the performance that is seen in C implementations of TCP.

The Prolac TCP implementation is inserted into the Linux kernel, with all the functions it provides overriding the standard kernel implementation. The design of TCP is based on that described in [25, 26] but redeveloped into an object-oriented structure. The TCB for example is divided up into several modules (C implementations often only have a single TCB definition) each module is an extension to another providing additional functionality resulting in a chain of modules that make up all of the TCB functionality. This leads to a more readable code base, as each component of the TCB has its own location and does not have code within it that relates to some other function of the TCB.

TCP in Prolac has good efficiency due to its compilation into C. The Prolac version is compared with the Linux kernel version and the paper reports that end-to-end latency is comparable, whilst Prolac produced an implementation which uses fewer cycles to process a packet. The authors put this down to the way in which their timers are designed, using only two compared to the multiple used in the Linux kernel implementation. The throughput of the connection produced by the two implementations is also examined with the Prolac implementation lagging significantly behind the Linux implementation, 8Mbyte/s compared with 11.9MByte/s respectively.

The Prolac TCP implementation is interesting from the perspective of tackling protocol implementation in an object-oriented way. However, the implementation does not take this design far enough. Instead of reworking the design from scratch, a C implementation is used as a basis and as such the code is not as effective as it could be in utilising an object-oriented design. It is attractive to see the TCB in a more modular design and this will be something that is wished to be achieved in this project. Prolac TCP focuses on an implementation efficiency which they have been unable to achieve. Furthermore the readability of the language is also questionable. The syntax mixes syntax from C and functional programming languages to produce a form which is only readable to those who know Prolac, this makes it very inaccessible to the majority of programmers.

2.3.4 A Metric for Software Readability

An important part of this project is to assess the readability of the final implementation against that of standard C implementations. Readability is the assessment of how readable a section of code is to an individual and as such a metric cannot be supplied to determine how readable code is, as the individuals may differ on opinion. [7] discusses how certain properties of code impact the readers assessment of its readability.

This paper takes *snippets* of Java code (three *simple* Java statements) and asks a group of annotators to score each of the snippets in terms of its readability. For each of the snippets its properties are determined and this information is compared to the readability score. The properties in question are, for example, the number of variables being referred to, the amount of whitespace and the number of comments. Snippets are deliberately chosen to exhibit at least some of these qualities, for example it would be useless to provide a snippet which contains three `import` statements as the readability would be high despite it having no features from which a readability trait can be established. Similarly, statements which are not simple statements are not included in the count of three. This means that the opening of a loop or the declaration of method appear in the snippet but will be followed by at least three statements which are deemed to be simple, e.g. method calls or variable assignments.

The annotators, consisting of 120 computing science students of varying levels of progress in their studies and therefore with differing familiarities with the Java programming language, were asked to rate 100 code snippets on their readability on a scale of 1 to 5. Importantly, the scores were not averaged on their absolute value, this is because while an annotator may give a score of 5 and another a score of 2 for a particular snippet the relative readability may be the same if the latter annotator gave a score of 1 to every other snippet. This leads to the snippets being scored as either more or less readable, rather than having an absolute value on the 1 to 5 scale.

The first observation made after this experiment is that, for the most part, people agree on what is readable code. This allows for a threshold to be extracted, from which the snippets can be separated into "less readable" and "more readable". Once these snippets have been established they are examined to extract a correlation between certain features and readability.

The most prominent features are the average number of variables, i.e. the fewer variables to understand the more readable a piece of code is, and the average length of a line, i.e. a snippet with short lines is more readable than a snippet with long lines. For the most part these results confirm common notions of what is readable and what is not. An interesting result is that the number of comments in a snippet has less impact on the readability than does the whitespace.

This paper presents some interesting results in the area of readability, confirming

via experimentation that the features that are thought to affect readability do have the most impact. However, the short snippet size means that the readability is a concept over a very small piece of code. It may be the case that a highly readable snippet within a larger piece of code may in fact be less readable than this metric suggests. Another point of interest would be to examine how the complexity of code affects the impact that these features have on readability. It seems reasonable that, for example, comments would have more bearing on the readability when particularly complex mathematical code is present.

Chapter 3

Approach

This chapter intends to explain how this project aims to show that network protocols can be successfully created in a high-level language and how doing so increases the quality of the software produced. As part of this project an implementation of the Transmission Control Protocol has been created in the high-level language Scala. This implementation is then to be analysed to both consider firstly if it shows that a network protocol can be successfully created in Scala, and then if this can be generalised to all high-level languages. Furthermore, by utilising the features provided by Scala it is hoped to be shown that not only does the implementation adopt a better design, but also that the code is more accessible in terms of readability and modularity. Consequently, this should indicate that the software is more maintainable and extensible.

The chapter proceeds by discussing the features within the Scala programming language which have been directly utilised within the code, that is those features which are visible to the programmer. After the use of the language has been established the chapter continues to discuss the high-level design of TCP that has been taken and how it differs from standard implementations.

3.1 Language Use

The Scala programming language provides many constructs which this implementation endeavours to utilise heavily to provide a readable and understandable TCP implementation. Each of these is discussed in the sections to follow.

3.1.1 Functional Programming

Scala supports both the functional and object-oriented programming styles to allow programmers flexibility in their implementation. This implementation has opted

for a mainly functional programming style to provide clarity to code and to allow the use of Scala's optimisations for functional programming.

Pattern Matching

An important concept in functional programming is the use of pattern matching and Scala provides advanced mechanisms for supporting this. In particular it uses a `match` statement, similar in form to the `switch` statement in Java to allow the programmer to pattern match on any type.

This implementation makes extensive use of pattern matching in three areas. Firstly, pattern matching is used to determine actions to be taken based on incoming parameters to a function. Essentially, where a function receives a parameter which determines its behaviour a match statement appears cleaner than an series of `if`'s. Secondly, pattern matching is used in recursive functions to detect the terminating case. Figure 3.1 demonstrates how a pattern match can be used in a function to break the recursive cycle, in this case when a list becomes empty or the correct entry is located. The third and most common use of pattern matching is in the reaction to incoming messages to actors. Pattern matching is particularly useful in this final instance as it allows the programmer to indicate which types are expected to be sent to this actor in at any given time and also, what the behaviour is given the different types. Unlike with the former two uses of pattern matching, it is unlikely that the behaviour of an actor would be remotely the same upon receipt of two messages containing parameters of different types and so the `match` statement provides a significantly neater representation than would an `if` statement.

```
def contains(s : SomeType, list : List[SomeType]) : Boolean ={
   (list) match {
      case List() => //empty list
         false;
      case (x :: xs) =>
         if (x == s)
            true;
         else
            contains(s,xs);
   }
}
```

Figure 3.1: A function which utilises pattern matching to terminate the recursive cycle.

Tail Recursion

The implementation makes use of tail recursion throughout the code. Tail recursion is recursion where the calling statement to the function is the final statement in that same function. Figure 3.2 shows the difference in using tail recursion and non-tail recursion. The top function has two possible return statements, either a non-recursive terminating "`return 0`" or the recursive "`(list.head + sum(list.tail))`". As the latter of these return statements contains the recursive call to the sum function, the function must be re-instantiated and run to provide the result to the overall expression. In the second of the two functions the recursive call to `sum` is the entire return statement, this allows Scala to reuse the current instantiation of the function, simply replacing the parameters. The `sum` function can therefore be run recursively in the memory space required for a single function call, regardless of the size of the `list` parameter.

```
def sum(list : List[Int]) : Int ={
   if (list.size == 0)
      return 0;
   else
      return list.head + sum(list.tail);
}


def sum(total : Int, list : List[Int]) : Int ={
  if (list.size == 1)
     return total+list.head;
  else
     return sum(total+list.head, list.tail);
}
```

Figure 3.2: Demonstration of the use of match statements and case classes in the Connection object.

3.1.2 Actors

In using Scala a programmer is encouraged to develop in a highly concurrent way using actors to abstract over that concurrency. Making TCP highly concurrent in a language such as C would be problematic and its behaviour potentially unpredictable. Clearly a certain level of concurrency exists in the standard C implementations of TCP, hence the ability to run multiple connections simultaneously,

however this project endeavours to create an implementation where not only is a connection in a separate logical process but so too are many of its components.

The difficulty in designing a C implementation of high concurrency lies in the model which it uses to enable this concurrency. The `PThreads` provides the ability to create a thread which can execute any piece of code. However, difficulties arise when the code in one thread needs to interact with that of another thread. In C interactions are handled by one or more threads writing to a memory location (potentially abstracted over by pointers) and one or more threads reading from that same memory location. Conflicts thus arise over the use of this memory location as a writer may leave data for a reader which is then overwritten by another writer or a reader may read the location prior to the writer putting the data there. The solution to these problems is to provide a lock or mutex over that memory location, thus the potential for deadlocks arises.

The actors model, as discussed in section 2.1.3, alleviates all of these issues and eliminates some. In Scala any class or object can be an Actor by simply extending the class `scala.actors.Actor` and providing a function, `act`. This `act` function is the entry point for the Actor. Actors are started by either calling the `act` function, the `start` function or by sending a `start` message to the Actor's mailbox. Communication then proceeds using this same mailbox, where Actors send messages between one another to facilitate sharing data.

In this implementation the approach has been taken that each Actor is an abstraction over a particular set of data. For instance, TCP specifies that any segment which is not acknowledged within a reasonable time be resent. This means that the implementation must maintain a queue of segments which are to be retransmitted. This queue is easily encapsulated within an Actor and this means that the objects which must interact with it are not concerned with the way in which the queue works. Furthermore it increases the modularity of the implementation by not having functions within other objects managing retransmissions on top of the work that the object is primarily designed for.

The use of Actors is of particular importance when timers are brought into consideration. TCP defines several timers of different or variable timeouts. By representing the behaviour of these timers as a group of Actors it has been possible to model the timeout of these timers with an interrupt style. In a C implementation such behaviour would require a far more complex design involving callback functions or a polling approach could be taken. This implementation of TCP makes extensive use of the actors model to provide clarity between components and to increase modularity by dividing logical components into separate actors which communicate asynchronously.

3.1.3 Case classes

Case classes in Scala allow the programmer to represent what would be a small class in another object-oriented language, as a single line. The case class is declared in the form:

`case class` *TypeName*`(`*param1 : S*`,`*param2 : T*`, ...)`

The justification for the introduction of this type of class is that it compresses a class into a small amount of space when it is required to do very little. Take, for example, a small Java class which stores two instance variables and provides getters and setters, thus taking up 10+ lines and an additional source file for very little reward.

The main use of case classes in the Scala TCP implementation is in passing of messages. As case classes are essentially a means of differentiating between two possible types on some method of input, they have been used a means of identifying a type of message between two actors. Figure 4.3 demonstrates how case classes are declared and how an actor can process messages by identifying the possible case classes that may be passed as messages and then prescribing behaviour based on that type. Also note that in the `case` statement the parameters of the case class are listed, this gives subsequent code full access to these values, enabling it to proceed without the need to call getter methods in the holding class.

It is worth noting that this use of case classes is likely to increase the readability of the `match` statements in actors code. This is because an actor is able to receive a message of any type including native or common types, the purpose of which will be distinctly less clear to a reader than if that common or native type is annotated by a descriptive type name. For example, sequence and acknowledgement numbers are represented as type `Long` in this implementation, if an actor receives a long it may or may not be clear that this is a sequence number, an acknowledgement number or some completely unrelated value. By prefixing Ack to that value as a case class with a `Long` typed parameter, it is immediately clearer that the message being received is an acknowledgement. This notion has been used throughout this implementation to aid readability and to prevent case classes from having to differentiate between messages that have the same type.

3.1.4 Modularity

Implementations of TCP are generally lacking in modularity with few functions and very few source files. This project aims to produce a TCP that makes use of a sensible package structure, to be described in the next section, as well as abstracting out reasonable components of the implementation into different classes.

```
case class Number(i : Int) extends Message;
case class Letter(c : Char) extends Message;
...
actor {
   react{
      case Number(i) =>
         //do something with ‘i’
      case Letter(c) =>
         //do something with ‘c’
   }
}
```

Figure 3.3: Example of the use of case classes in Scala.

In part the use of Actors while help to facilitate this modularity. Further, by approaching this problem with a functional style the programmer is encouraged to remove complicated sections of code and replace them with well defined functions.

Wherever possible classes will be created to abstract over and encapsulate logical components, e.g. headers, windows etc. A functional style will be taken throughout and all values will be held as immutable objects, this further encourages the programmer to use functions as the natural way of dealing with immutable state is to move it from function to function. Modularity and loose coupling is a generally accepted measure of software quality and emphasis on this in the design will ensure that the quality of this implementation is improved relative to C-based implementations.

Focusing on the design principles as laid out in this section should ensure that the software is created to a high quality. It should also ensure that Scala is being used to its full potential and that the rich features which it provides are able to enhance the implementation.

3.2 TCP Design

This section intends to discuss the overall design for the implementation of TCP. The design principles draw on the use of the Scala language as described in the preceding section. The main focus is on the communication between the major components of the system and how these components operate together to produce TCP behaviour.

3.2.1 High-level design

From the application level TCP can be thought of as being the ability to establish a connection to another remote host on the network and send and receive data. This understanding provides three main processes that are work for any TCP connection. Firstly, the connection itself, that is the static state of the connection such as port numbers, IP addresses etc. Secondly, a receiver which is able to capture incoming segments and process them into a meaningful format. Lastly, a sender which puts data into the correct format for transmission so that TCPs of this and other implementations can understand the intent.

The implementation uses functions to store state and each of the three components, connection, receiver and sender, can have a state independent but related to that of the others. Each of the components will be represented by an Actor and will communicate using a combination of messages and case class messages as described in section 3.1.3. State changes will be caused by the contents of each of these messages and a state change will occur upon each receipt, i.e. the receipt of each message is terminated by a function call, which may or may not be a call to the currently executing function.

The `Receiver` object will be responsible for taking receiving segments passed down via the connection manager (see section 3.2.1). Furthermore it will be responsible for disassembling these segments into the component parts, i.e. sequence number, acknowledgement number, advertised window etc. Two possibilities for dealing with each of these components can be considered, firstly, the components can be represented as a single packaged message and sent along. This representation leads to the possibility that a message type will need to exist for each of the possible combinations of segments, i.e. a segment containing only a sequence number and no acknowledgement number would be sent differently from a message containing both. At some stage with this representation these messages would have to be broken down into their constituent parts and it is deemed wise to do this at the receiver because, as mentioned above, part of its remit is to disassemble and pass on segments. The alternative, and the selected, approach is to have each constituent part given a message type. Thus, if a segment contains a sequence number and an acknowledgement number in normal data transfer then two messages will be transmitted from the receiver to the connection to be processed independently of one another. Note at this stage that the receiver has absolutely no access to the sender, and vice-versa, all communication is directed through the central connection object, this is evident from the system diagram shown in figure 3.4.

Another important function, related to the deconstruction of the segments, is the supply of data to the connection, and through the connection, the application layer. This is achieved through the `ReceiveBuffer`. The `ReceiveBuffer` abstracts some

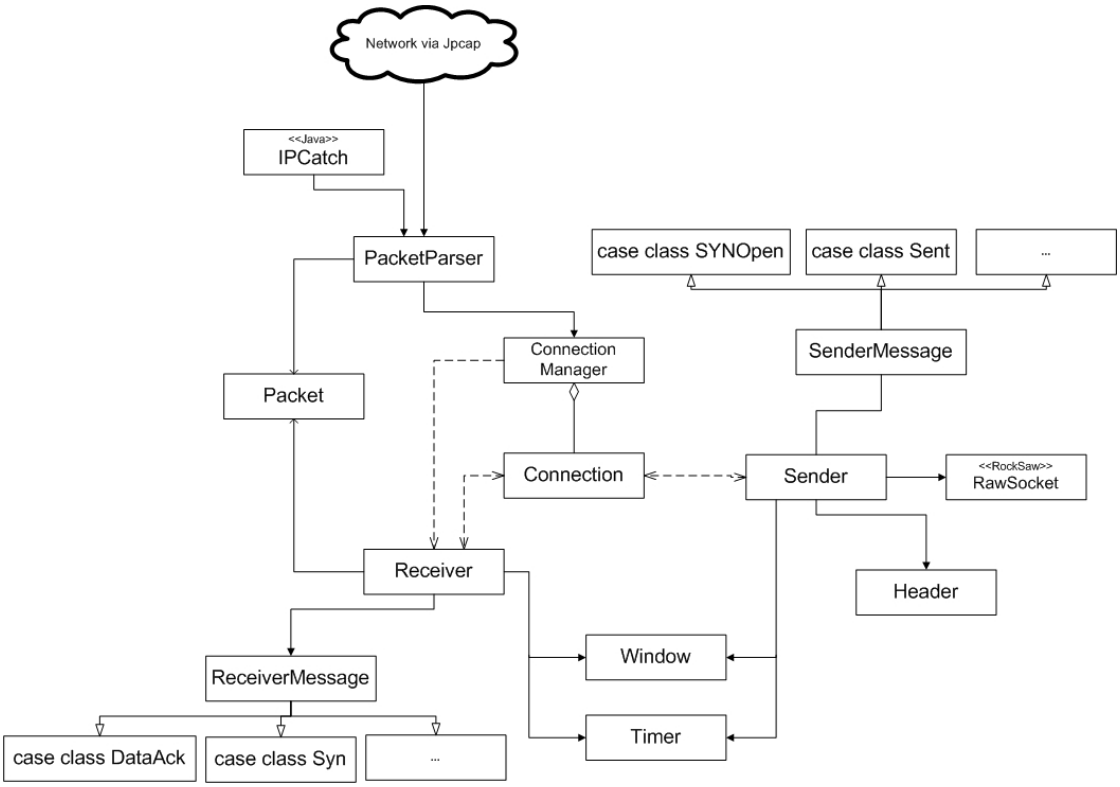


Figure 3.4: Overall design of the Scala TCP implementation.

important work away from the receiver in facilitating the data reordering which may be required as packets can be reordered by following different paths through the network or by packet loss.

The `Connection` object is responsible for the overall state of the system, with its currently executing state being deemed as the state for the overall connection. Furthermore, the `Connection` object uses message passing to facilitate the passing of information from the receiver to the sender. The sender itself is responsible for the outgoing data in the system. Data is passed down through the `Connection` object which the sender is then able to access via the `SendBuffer`. Once again this buffer is represented as an actor and processing proceeds via a series of message passes between the three involved objects, ensuring that data's integrity is maintained throughout.

Figure 3.5 displays the communication model for the `Sender`, `Receiver` and `Connection` object. The sender and receiver communicate directly with the connection via message passing and provide data via the `ReceiverBuffer` and `SendBuffer`. The sender and receiver are ultimately unaware of each other and communication between them is handled by the connection. This enables the connection to have

an overall view of the system and its current execution, hence why it is able to maintain the overall connection state.

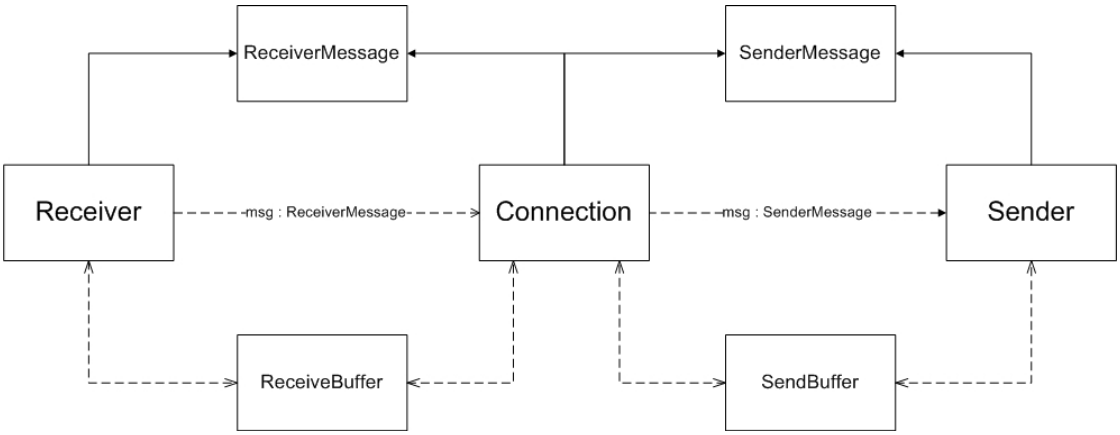


Figure 3.5: Communication model for interaction between Receiver, Connection and Sender objects.

Window Management

The sender must maintain its send window. This means keeping track of those segments which have been sent and of those, which have been acknowledged. Furthermore, through communication with the receiver (via the `Connection` object) it must be constantly aware of the changing advertised window included in each segment. The window maintenance functionality has been encapsulated inside the `Window` class. This allows the sender to simply call the relevant function within the `Window` class whenever a message which affects it is received, e.g. receiver has passed a new advertised window or sender has sent a new segment.

Timers

TCP specifies many different timers for a variety of functions throughout the system. These timers are provided to the sender are receiver via the top level `Timer` object. `Timer` specifies a function for each of the timers with different parameters allowing the timers to be customised. For example, many of the timers have a String parameter which allows the caller to specify a unique timeout message, thus enabling them to react differently by differentiating between the timeouts. Additionally, one of the parameters for each of the timers is an Actor, this allows

the caller to specify where the timeout message should be sent to, usually the caller itself. Thus, the mode of operation of the timers is for the sender or receiver to make a single function call and be prepared to receive a message of timeout, this allows the sender and receiver to continue processing whilst the timer counts down.

Connection Manager

TCP can run many connection simultaneously and so it is required that there be some way of identifying which segments are intended for which connections. This functionality is achieved in this implementation by the `ConnectionManager`. Once again this object is held as an Actor within the system, creating links to connections by passing messages about as appropriate to register the connection and to pass segments throughout the system. Connection identification is achieved by locating the source and destination ports and determining whether or not the connection exists within the system and if so, where to send that segment if a connection does exist.

Connections must register with the manager on instantiation after which point any segments with a port pattern matching theirs will be forwarded to it, via their receiver. In order to improve the efficiency of the segment forwarding the connection manager takes the approach of assuming that any arriving segment is most likely to belong to the connection which was most recently forwarded a segment. Thus, it maintains a single connection which, upon receipt of a segment, is checked for compatibility. If the segment does not belong to that connection the remaining connections are checked and whichever connection the segment belongs to is then held as the last to have had a segment forwarded.

Chapter 4

Implementation

This chapter aims to explain the key features of the implementation of TCP in Scala. Importantly it will try to demonstrate how the language has been used in the way described in 3.1. Each component of the system is discussed in high detail and examples of the code have been provided to further demonstrate the use of the Scala programming language. The chapter begins by describing the lowest level of the implementation where IP layer access is controlled and then moves up to describe the controllers that sit on top of this.

4.1 Segments

TCP sends data in segments which are made up of the TCP header and TCP data, concatenated together. The TCP header provides the source and destination ports as well as the sequence number of the leading byte of data and an acknowledgment for the last byte of data that was received. Furthermore, it also provides all the control flags to indicate what additional information can be taken from the header such as sequence number synchronisation or a connection close signal. It is therefore necessary for this implementation to conform strictly to the specification of the TCP header in its construction for sending segments so as to ensure successful communication between different TCP implementations.

Segments are passed down to the IP layer as an array of `Byte`. The `Byte` representation is of little use in a high-level programming environment where the types available for use are richer and can better represent the individual components of the segment. Take for example the source port component of the TCP segment header. The source port is a 16-bit value whose 8 most significant bits are held in the first byte and the 8 least significant bits are held in the second byte. The natural internal representation of such a value in C would be an unsigned short, this is not the case in Scala. Scala, and indeed Java, do not provide unsigned native types and so a different method of storage is required. The solution chosen in this im-

plementation is to store each of the values in a type which is larger than required, e.g. a value of 8-bits is stored in a 16-bit `Short`. In this way normal arithmetic on these values is possible throughout the remainder of the code at the expense of some difficult operations within the `Header` class. If this step of abstraction had not been taken then all code which accesses values from the segment header would be required to account for the possibility of receiving a negative value in place of a large positive one (e.g. port number 65536 would appear as -1). Another potential solution to this problem would have been to create classes for 8-bit, 16-bit and 32-bit values which carried out the arithmetic as required. However, the cost to memory of storing a value in a native type twice its size is smaller than the cost of instantiating repeatedly an object and performing complex calculations on that value. Furthermore, due to the functional nature of this implementation such a class would be required to return values of its own type, thus increasing that cost.

Importantly, although the components of the TCP header are required throughout the system the need for them to be contained as a single `Header` object is only present in incoming packets. Port numbers are statically defined upon instantiation and the other values of the header can be calculated based on the state in which the TCP is currently operating. The same cannot be said about incoming segments as firstly, although the port numbers are statically defined for a connection the connection manager needs to be able to read the port numbers to determine which connection the segment is to be forwarded to. Secondly, as the order in which packets are received is not necessarily the order in which the packets arrive the sequence number of the segments must be determined from the header value and not simply the current state, as in outgoing segments which do leave consecutively. Further, the control bits may indicate a required change in state (e.g. if a `FIN` flag is present).

To best model these two different ways of dealing with headers the implementation makes use of the ability to define an Object and a Class in Scala. The Object definition of the header specifies static methods by which segments can be created. The necessary tasks are to define the base header (all values in place except the checksum), and then to finalise the header by adding data and performing the checksum calculation, and subsequently placing the checksum value into the correct holder in the header. All these operations treat a header as an array of `Byte` as this is the format in which the segment must be to be passed down to the IP layer.

The `Header` class is used to decipher incoming packets. The packet object is received, via jpcap (see section 4.2.2), and parsed by the `Packet` class which retains some of the IP information for further use (primarily this is required to determine the IP address of the active host as the passive host is unaware of this upon instantiation of the connection, instead it operates in the LISTEN state until this information has been obtained).

4.1.1 Checksum

Previous work in developing TCP in high-level languages has taken the approach of developing the majority of the protocol in the given language but retaining the C implementation of the checksum calculation [20, 5]. The basis for this decision is that the checksum calculation is carried out very regularly, at the sending of each segment and the receipt of each segment, and the C implementation, due to its low-level access can perform this operation at higher efficiency than a high-level language. This implementation has opted for the approach that wherever possible the development of TCP will proceed in the development language, Scala.

The checksum for TCP is the one's complement of the one's complement sum of each 16-bits making up the header, data and pseudo-header. The pseudo-header consists of the source and destination IP addresses as well as the protocol number for TCP and the length of the segment, not including the 12 bytes of the pseudo-header. The checksum function developed for this implementation utilises recursion and pattern matching over lists to provide a concise and easy to follow piece of code, included in figure 4.1.

From figure 4.1 it can be seen that the checksum operation makes use of pattern matching and recursion to reduce the amount of amount of code required to obtain the checksum value. Furthermore, the `chksum` function has been broken up into two parts, increasing modularity and further enhancing the understandability of the function as a whole. Other implementations of this checksum operation have used a single function or indeed embedded the code amongst other operations.

The design of segments and their constituent parts is important as they are the means by which data enters and leaves the TCP system. By encapsulating the segment handling code in the way described here it has been possible to abstract above the somewhat awkward byte array handling and bit shifting code. The parts of the implementation that access this layer, to be described in the following sections, have no requirement to deal directly with the construction of segments and can instead focus on the logic of the protocol.

4.2 Raw Sockets

Raw sockets are the method by which TCP interacts with the IP layer of the TCP/IP stack thus creating sockets. In normal circumstances TCP creates a raw socket which it then sends and receives data from. In order to create TCP in Scala it was required that this layer of the kernel stack be accessed in some manner. Two tools have been used to perform this task. Firstly RockSaw[2] provides access to a raw socket implementation and secondly Jpcap[1] provides access to IP layer, allowing packets to be intercepted.

```
//Calculates the checksum for the segment 'xs'
private def chksum(xs : List[Byte]) : Short = {
    val notSum : Int = add(xs);
    return (~(shorten(notSum))).toShort;
}

//Brings a checksum value down to a Short by folding in carry bits
private def shorten(sum : Int) : Short = {
    if ((sum >>> 16) > 0)
        shorten((sum & 0xffff) + (sum >>> 16));
    else
        sum.toShort;
}

//Sums each byte pair in the segment 'xs'
private def sum(xs : List[Byte]) : Int =
   (xs) match {
       case List() => 0;
       case (y :: ys) =>
           (ys) match {
               case List() =>
                   ((y << 8) & 0xff00);
               case (z :: zs) =>
                   (((y << 8) & 0xff00) + (z & 0x00ff)) + sum(zs);
   }
```

Figure 4.1: Code snippet showing the use of recursion and pattern matching in the checksum function.

4.2.1 RockSaw

RockSaw presents a simple interface to abstract over the raw sockets implementation. After instantiation a single method call is all that is required to send segments which are subsequently packaged into IP packets. Scala's close interaction with Java allows the process of using this library to be as simple as if Java itself were being used.

The RockSaw package presents a single type, RawSocket. Once instantiated this RawSocket can have data sent in the transport level formats, e.g. TCP, UDP to be moved to the kernels IP layer. This access is achieved by using the Java Native Interface (JNI) to call C code which contains the raw socket implementation. All RockSaw access is handled by the `Sender` object of this implementation to ensure that the necessity to understand its interface is contained and encapsulated.

4.2.2 Jpcap

Jpcap is a library created on top of the libpcap library for C. It enables the Java code, or in this case Scala, to intercept IP packets. However, note that Jpcap does not offer the ability to prevent packets from progressing to the kernel and as such special provision has had to be made to ignore the response of the kernel TCP implementation. It is believed that it would be possible to configure a firewall to intercept packets before arriving at the transport level, thus disabling its interference. All attempts at this, however, have resulted in intercepting the packets too late, i.e. the kernel is still able to make responses or too early, meaning that Jpcap cannot collect the IP packets and notify the Scala implementation.

The interface for Jpcap is simple to set up a listener object over a particular network interface. All packets arriving on that network interface, and matching a configurable filter, are forwarded to a `PacketHandler` object, `PacketParser` in this implementation. The `PacketParser` then obtains the byte array format of the packet and creates a packet object which holds all the IP information (and within the data part of the packet, TCP information) that is relevant to TCP. This packet is then simply forwarded to the connection manager which distributes it to the correct areas of the system.

4.3 States and State Transitions

One of the major aims in the development of this implementation was to achieve a program in which it is easy to read how the program will execute. To achieve this the most important step is to be able to identify firstly, the different states which TCP can be in for any given connection and secondly, which states it can subsequently move to, under the given cicrumstances.

Table 4.3 shows each of the states a standard TCP can occupy at any given time, it also shows the main function of that state, although within that function each state has a different specific purpose. State transitions occur based on two major inputs, firstly, the activity of the remote TCP, for example the receipt of a `FIN` flagged message in the `ESTABLISHED` state will result in a move to the `CLOSE_WAIT` state. Secondly, instructions can be passed down from the application, in this case in the `ESTABLISHED` state a close message from the application will result in the TCP moving to the `FIN_WAIT_1` state.

Within each of the available states a different set of behaviour is defined based on the information being received from these two major sources and, additionally, the timers that are active in the system. Equally each state has a set of criteria which defines when a state transition occurs. This implementation makes extensive use of case statements, message passing and case classes to retain clarity in the code as well as enhancing the ability of a reader to establish the behaviour of the system at runtime based on the current state and the information being received.

State	Handshake	Data receipt	Data sending	Closing
SYN_RECEIVED	✓			
SYN_SENT	✓			
ESTABLISHED		✓	✓	
CLOSE_WAIT				✓
LAST_ACK				✓
FIN_WAIT_1				✓
CLOSING				✓
FIN_WAIT_2				✓
TIME_WAIT				✓
CLOSED				

Table 4.1: List of TCP states and the purpose of that state.

To provide clarity and to further the modularity of this implementation the approach taken here was to divide the work up into three different Actors, each of which are performing logically different but very much connected tasks. These are the `Connection`, the `Receiver` and the `Sender` objects (see section 3.2). Each of these objects can exist in any one of the standard TCP states but they are not required to reach these states simultaneously, thus the overall state of a TCP connection is denoted as a triple, e.g `(SYN_SENT, ESTABLISHED, SYN_SENT)` denotes a point where the receiver has just received a segment indicating firstly that the remote host wishes to synchronise with the local host and secondly that it acknowledges the receipt of the synchronise request first sent by the local host. Therefore, it would seem that this model of states is more complex than that of standard TCP implementations. However, Scala provides mechanisms which allow these multiple states to be modelled and handled in a very readable manner.

Importantly, because of the functional nature of this implementation, any given TCP state is modelled as a function and as that function proceeds it may call a new function, thus facilitating a change of state. Of course it is possible that the function calls itself to preserve the current state, this is most common in the ESTABLISHED state as this state must be re-entered every time a packet is sent or received, among many other things. In modelling the state in this manner the aim is to make progress more predictable and understandable. Furthermore, by eliminating mutable state variables Scala is able to optimise its operation, hopefully leading to better efficiency.

4.3.1 Connection & Receiver

The `Connection` object models the TCP state in the traditional way, ultimately the current state of the entire TCP can be distilled down into a single description

by simply recognising which function its connection is working in. At any given time the sender and receiver are operating at the same state as the `Connection` object, or one ahead or behind in the transitions. In the example above the receiver is one ahead of the connection as it has information regarding state transition that it will subsequently pass on to the `Connection` object. It may appear premature for the `Receiver` object to move into the ESTABLISHED state but the receiver need not know about the absolute state of the system to function effectively. It is assumed that the direction of communication is: remote host → receiver → connection → sender → remote host, in general. Thus, although the sender will eventually send an ACK segment to move firmly into the ESTABLISHED state, that segment is never acknowledged by the remote host and so the receiver can move into the ESTABLISHED state in advance of the connection. In this way the receiver is in the correct state if the remote host initiates data transfer.

The three objects `Receiver`, `Connection` and `Sender` are represented as Actors, thus allowing them to operate independantly as described above. These objects are aware of their respective roles in the TCP structure but are essentially oblivious to the work being carried out by the other two. The `Receiver` is responsible for all receipt of packets. It is aware of the states of a TCP and it is aware of the conditions for state transition, though as mentioned above it removes some of the criteria for moving between states as it is not responsible for the actions that initiate it. This can be seen as an assumption that the `Connection` object will do its job as required, in the instance above that means its connection will instruct the sender to send an ACK message. The `Receiver` is also responsible for the reordering, storing and provision of data for the `Connection` object. This role involves the first means of communication that the Actors have, a shared resource, the `ReceiveBuffer`. Although from an abstract point of view the `Receiver` is responsible for the reordering of data it is in fact the `ReceiveBuffer` which carries out this function. The buffer is a simple collection, List, of tuples containing a sequence number and a List of Bytes, the sequence number being that of the first Byte in that list. The buffer is itself an actor so receivers send a message containing a tuple to it and connections simply request data when required in the form of a message. This guarantees the integrity of the data as the buffer can only process one message at any given time, with a shared memory approach the possibility would arise for a simultaneous read/write scenario.

The second method of communication between these two Actors is through the usual message passing. For a receiver a series of messages are pre-defined as case classes, for example on receipt of a segment with the ACK flag set a receiver would send its respective connection an `DataAck` message with the relevant acknowledgement number contained within. Figure 4.2 demonstrates how the receipt of a packet can stimulate a flurry of message passing to execute the logic of the system. The method of receipt of messages is simply to have each state contain a match (or case) statement. Although these case statements can be quite expansive, with ten or more potential messages, they make reasoning about behaviour very easy. Figure 4.3 shows how these case classes and case statements can be combined to

represent the available functions in the given states. In this example it can be seen that although the `DataAck` message is identical, the connection will behave in different ways dependant on the state.

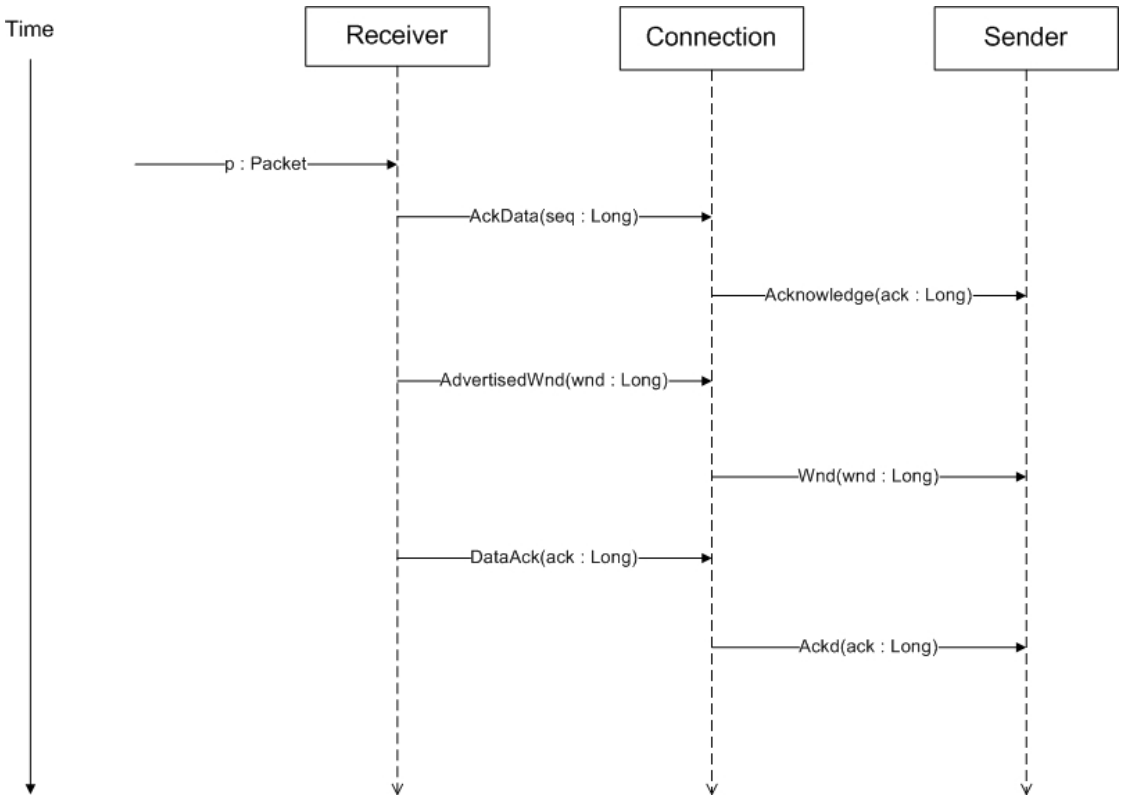


Figure 4.2: Sequence of messages between the Receiver, Connection and Sender objects.

It should also be noted from this example the ease with which the state and its transitions can be identified. A non-functional approach to this problem would require many different variables to track the state based on the combination of their values. This would require a large amount of checking of values using IF-statements which make the interpretation of code more difficult by forcing a reader to infer the variety of different values that those variables could hold.

The relationship between the `Sender` and the `Connection` is almost identical to that described for the `Receiver` with the communications flow being predominantly in the `Connection` → `Sender` direction. Also the `SendBuffer`, equivalent

in terms of communication to the `ReceiverBuffer` is significantly less complex in not having to deal with issues such as reordering. Contrastingly, the `Sender` is significantly more complex than the other two components and as such will be discussed further in the following section.

4.3.2 Sender

The `Sender` class describes all activities relating to the sending of data across the TCP connection. Data is provided by the application to the `Connection` via the `Socket` interface. `Connection` objects then arrange this data for its sender to commence transmission by passing the data to the `SendBuffer`.

A sender is able to request data up to a maximum sequence number based on either the window size or the MSS for the current connection. This communication is again between the two actors, sender and buffer, and so uses message passing to achieve this goal. Whenever the Sender does not have messages incoming and it has an open window, it will request data from the buffer up to the size of one segment. This data then arrives via a message, handled in the main match statement of the established function/state, this can be seen in Figure 4.3. The `Sender` match statement is among the larger of those found in the code base due to the high

```
def synReceived(isn : Long) {
     react {
         case ACKD(ack) =>
             if (ack == isn+1)
                  established(isn+1);
         case ...
     }
}

def established(seq : Long) {
     react {
         case ACKD(ack) =>
             tcpSender ! Acknowledged(ack);
             established(seq);

         case ...
     }
}
```

Figure 4.3: Demonstration of the use of match statements and case classes in the Connection object.

amount of work that the sender has to do in coordinating the arrival of segments at the remote host. Case classes are use wherever the message is triggered by the remote host, that is if a packet arrives acknowledging a particular group of sequence numbers the ending sequence number is passed to the sender via an `Ackd` case class.

Figure 4.4 demonstrates the few lines of code that are required in processing the large variety of different inputs a TCP connection can provide. Wherever possible the implementation has divided up any input into its smallest constituent parts to allow processing to continue asynchronously. For example a received segment may contain an acknowledgement, a window advertisement and some data. In this case the segment is dealt with by notifying the sender of these inputs individually, so that it may process them asynchronously and the amount of code required is dramatically reduced.

The interactions of the `Receiver`, `Connection` and `Sender` objects are essentially the triggers for the execution of the TCP logic. Although this implementation does not include some TCP features and options, these base objects and states are all that would be required for most of the major extensions. Each of these objects represents one major function of the system and is completely encapsulated from the others, this provides a neat separation of concerns that should make extension significantly easier.

4.4 Timers

Key to any implementation of TCP is the use of timers for many different functions. Scala, and in particular the actors model of concurrency, allows the use of timers to be more seamlessly integrated with other code in the implementation. Furthermore when a timer is in use it is more visible and it is much easier to reason about the impact that it has on the code running.

The majority of the execution of the TCP implementation takes place in one of the many case statements that appear in the `Connection`, `Sender` or `Receiver` classes. These case statements match on messages being received from other actors, e.g. the `Receiver` object sends a `Ackd` message to the `Connection` object whenever an acknowledgement is received from the remote host allowing for that acknowledgment to be logged and the relevant segment(s) to be removed from the retransmission queue. In developing the timers to interact with these objects it is beneficial to create them in such a way as a timeout is not a special case and is viewed as being normal operation. As such all timer timeouts are dealt with as being messages sent from actors so that the objects which initiated the timer deals with them in an identical way.

```
def established(wnd : Window, ack : Long, rttSeq : Long)

...

react {
   case (data : Array[Byte]) =>
      //Send the data
      if (rttSeq == 0)
         rttTimer ! "START";
      send(wnd.getLeftEdge, data, ack);
      val newWnd : Window = wnd.dataSent(seq+data.size);
      established(newWnd, 0, if (rttSeq == 0) seq else rttSeq);
   case Wnd(update : Long) =>
      //Move to state established with new advertised window
      established(wnd.updateReceived(update), ack, rttSeq);
   case Ackd(ackd : Long) =>
      //Remove relevant segments from retransmit queue and open window
      //If RTT covers this segment stop timer
      retransmitQueue ! Ackd(ackd);
      if (ackd >= rttSeq)
         rttTimer ! "STOP";
      established(wnd.updateWindow(ack), ack, if(ackd>=rttSeq)0 else rttSeq);
   case Retransmit(seq, segment) =>
      //Readjust window and resend segment
      resend(segment);
      established(window.retransmit(seq), ack, rttSeq);
   case Ack(seq) =>
      //Update the ACK to be sent in the next outgoing segment
      established(wnd, seq, rttSeq);
   case "DelayAck" =>
      //Acknowledgment timer has expired, send segment with ACK
      if (ack != 0)
         acknowledge(ack);
      established(wnd, 0, rttSeq);
}
```

Figure 4.4: Partial code listing for the match statement in the established state of the Sender class.

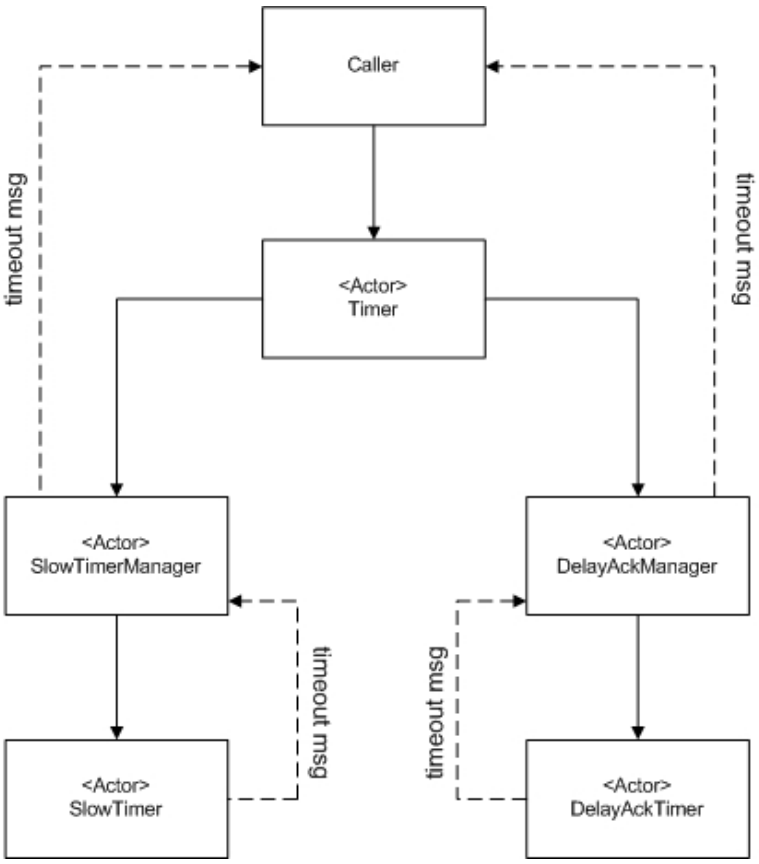


Figure 4.5: High-level design of timers.

4.4.1 ISN Generator

The simplest of timers in this implementation is the initial sequence number generator. As discussed in section 2.2.1 each connection requires a different ISN to allow it to differentiate between packets sent on the current connection and those sent on a connection with the same major characteristics (source and destination IPs and port numbers) that has expired. To facilitate this an `ISNGenerator` object controls the current ISN for the whole system, when a new connection is created it requests this object send it a new ISN.

Major implementations of TCP use a 4ms interval to increment the ISN for the system[26], this means that for ISNs to cycle (and therefore for a connection to be potentially confused by old packets) a time of 4 hours and 55 minutes will pass, given a 32-bit sequence number, making it highly improbable that a closed connection will still have packets on the network. To facilitate this timeout an `ISNTimer` object is instantiated by the generator which periodically (every 4ms) sends a message back to the generator to indicate that the ISN should be incremented.

The generator therefore is subject to two different types of message, one for timeout and the other for ISN request. Each of the three players in this communication is an actor, a connection actor a generator actor and a timer actor. Upon creation the connection sends a message to the ISN generator (instantiated in the initial setup of ScalaTCP) containing itself, to allow the generator to return the sequence number. The ISN generator uses a recursive function to carry out its behaviour. It is aware that there are only two types of message which it can receive, a message indicating a timeout or an actor indicating a request for an ISN. A case statement is used to deal with the two messages as demonstrated in figure 4.6. The timer itself is a simple looping function which makes use of the Java `Thread.sleep` function, notifying the generator every time it returns from its sleep.

```
def manage(isn : Long) {
     react {
         case (requestor: Actor) =>
             requestor ! isn;
             manage(isn+1)
         case msg =>
             \\TIMEOUT received
             manage(isn+1);
     }
}
```

Figure 4.6: Code snippet showing the behaviour of the ISNGenerator

The ISN generator timer demonstrates the way in which a timer can be implemented in a clear way in Scala, however it is relatively simple in that it is aware both of the timeout time and its single client at compile time. More complex timers will not only require differing timeouts but they will also be required to work with multiple clients.

4.4.2 Delay Acknowledgment Timer

The basic operation of TCP requires that all segments received are acknowledged. Furthermore there is a time constraint on the acknowledgment of segments as if the acknowledgment arrives late then the segment will be retransmitted and the sender will enter some mode of congestion control. With these concerns noted TCP still attempts to reduce the amount of sending it is required to do by grouping segments together for acknowledgment. This means that if two segments are received in quick succession then a single segment can be sent to acknowledge them both (by setting the acknowledgment field of the header to the greatest sequence number

of all bytes received plus one, assuming the segments arrived in order relative to each other and relative to all other unacknowledged segments). This reduces the amount of sending required of the receiving host to half its sending otherwise. This implementation uses a delay of 200ms to allow for multiple segments to be acknowledged in one ACK.

The 200ms timer (fast timer) is implemented in exactly the same way the 4ms timer was in the ISN generation. More interestingly the manager for the fast timer is able to provide the timeout message to multiple clients. The manager performs two functions, firstly it stores all actors which have subscribed to the timer and secondly it notifies all its clients when the timer has timed out. Note that this differs from the ISN generator where a single actor was notified of the timeout repeatedly. Instead the manager must store those actors which wish to be notified temporarily and it must continue to accumulate those actors until the timeout occurs. Such behaviour is easily dealt with using recursive function calling and asynchronous message passing.

Again a case statement is used to differentiate between the two types of message that the manager can receive. These are a message containing a timeout message or a message containing an actor which wishes to subscribe to the timer. Functional programming languages are best suited to dealing with lists as a means of storage and so each of the subscribers is contained in a Scala `List`. When the local host has a large number of TCPs which are receiving segments this list will accumulate a significant number of actors before it times out, at which time it then iterates through the list of actors and notifies each in the order in which they subscribed, i.e. first in first out. Again a tail-recursive function is used to allow for reduced clutter code and to make the state transition more visible to readers. An outline of this code is provided in figure 4.7.

```
def manage(subscribers : List[Actor]){
    react {
        case (subscriber: Actor) =>
            manage(subscribers ::: List(subscriber));
        case msg =>
            for (subscriber <- subscribers)
                subscriber ! timeoutMsg;
    }
}
```

Figure 4.7: Code snippet showing how subscription and timeout is handled in ScalaTCP.

The clients of this timer are, perhaps counter intuitively, `Sender` objects. Although the receiver for a TCP is responsible for the arrival of segments, which subsequently

require acknowledgement, the abstraction technique utilised in this implementation is to allow the receiver to understand that all segments are acknowledged with no further information. The justification for this abstraction lies in the fact that the sender is the component of the system which benefits from the cost saving activity of delaying acknowledgements, as it potentially is required to send less segments. Thus, this implementation has taken the approach of allowing the sender to control its own rate of sending, rather than have it dictated to it by the receiver (via the `Connection` object).

The fast timer demonstrates how Scala has been utilised to develop timers for TCP where multiple clients need to be notified of a prearranged (i.e. compile time defined) timeout. The last remaining timer type, where the timeout is decided at run-time for multiple clients is known as the slow timer and is outlined in the next section.

4.4.3 Slow Timer

The slow timer is the final of the timers created to be made available to this implementation of TCP. It provides connections with a more versatile timer, allowing them to specify the length of time that passes before the timer reports that it has expired. Furthermore it allows the calling object to select the timeout message that they receive upon expiration, thus allowing the calling object to differentiate between different timer timeouts.

The slow timer is again governed by a manager class which operates in much the same way as that seen for the delayed acknowledgement timer. However, instead of maintaining just a list of subscribers it also maintains the desired message for that particular subscriber and the timeout which they expect to be notified after. It would be highly inefficient to maintain a timer for each of these instances and so instead the implementation maintains a single timer which expires every 500ms. Thus, the actual timeout given to the calling object is a multiple of this 500ms timer, it is this multiple which the manager stores.

The manager maintains a list of the relevant values for each Actor which is subscribed to it. This list is made up of a reference to the Actor itself, the message which that Actor expects to receive upon timeout and the number of 500ms timeouts that need to occur before the timeout message should be send. Upon receipt of a new timeout from the slow timer the manager iterates over the list. Where the count for any given subscriber has reached zero the timeout message is sent to the subscribing Actor and that subscription is removed from the list, otherwise the count for the subscriber is decremented.

Figure 4.8 shows an outline of the code for handling the slow timer timeouts and notifying the relevant Actors. The `manager` function demonstrates a use of Scala's

tail recursion where the function being recalled is simply repeated with new parameters. This means that not only does this function use a small amount of memory, the equivalent of the function being called only once, but it also removes the overhead associated with repeated function calls. Furthermore, the `decrement` function demonstrates the use of pattern matching over Lists. As with most functional programming languages Scala is optimised to operate over lists and the use of a `match` statement in iterating over the List recursively reduces the amount of code required to perform this function as well as allowing a reader to identify the normal and special cases which the function deals with more easily.

```
def manage(waiters : List[(Int, String, Actor)], timers : List[(String,Actor)]) {
  react {
     case "SLOW_TIMER_TIMEOUT" =>
        for (timer <- timers)
           timer._2 ! timer._1;
        manage(decrement(waiters), timers);

     case (timeout : Int, msg : String, actor : Actor) =>
        manage(waiters ::: List((timeout, msg, actor)), timers);

     case (msg : String, actor : Actor) =>
        manage(waiters, timers ::: List((msg,actor)));
  }
}

def decrement(w : List[(Int, String, Actor)]): List[(Int,String,Actor)]={
  w match {
    case x :: xs =>
      if (x._1 == 1){
        x._3 ! x._2;
      return decrement(xs);
    }
    else
      return List((x._1-1, x._2, x._3)) ::: decrement(xs);

    case List() =>
      return List();
  }
}
```

Figure 4.8: Code snippet showing the use of tail recursion as well as pattern matching over lists in ScalaTCP.

The implementation of TCP as described above is extremely modular, highly concurrent and, it could be argued, very readable. The implementation has also been designed to have logic well encapsulated to enable easier extension and maintenance. This modularity is facilitated by using either Actors or classes, thus making use of the two paradigms on offer by Scala. Furthermore, in the extensive use of the features described in 3.1 Scala has been utilised to its full potential to produce an interesting implementation of this protocol.

Chapter 5

Evaluation

This chapter intends to evaluate the implementation of TCP in Scala in terms of software quality. Focus is on the readability and modularity of the code. The chapter begins by indicating how the implementation has met some of the criteria for the protocol. Once this has been established it moves on to compare the implementation to that found in the Linux kernel, to establish if the implementation provides an improved software quality.

5.1 Proof of Correctness

This section intends to show that this implementation of TCP successfully exhibits some of the properties that a TCP must have. It begins by showing that a connection can be established via three-way handshake and then moves on to show that data can be transferred and that data can be successfully acknowledged. These claims are backed up by analysing network traffic using TCP dump to show that the relevant packets, in the correct format have been moved about the network.

5.1.1 Three-way Handshake

As described in section 2.2.1 two TCPs form a connection by way of a three-way handshake. This means that the active host (the host instigating the connection) starts by sending a segment with the SYN flag set to the passive host. The passive host then returns a segment containing an ACK for the sequence number of the initial segment as well as having the SYN flag set. The final act of this connection establishment is for the active host to acknowledge the passive hosts response.

Figure 5.1 shows the output from TCP dump during the three way handshake. The TCP dump output is formatted in the following way:

timestamp **IP** (*details_of_ip_packet*) *sending_host* > *receiving_host*: *flags*,
chksum *checksum_value* (*checksum_correctness*), [*starting_sequence*:*ending_sequence*(*data_size*)]
[**ack** *relative_ack_number*] **win** *advertised_window*

Thus, in the first segment it can be seen that it was sent at time approximately 09:59:06. The active host is named filicudi and the passive host is named unsst. The SYN flag is set, hence the 'S' in the flags field. Furthermore the ISN for filicudi is 11 and it is advertising a window size of 1.

The output continues to show that in the second segment, sent in the reverse direction, the SYN flag is again set and an acknowledgement is contained, indicating that host unsst has received the segment with sequence 11 and is awaiting the segment with sequence 12. This is the second stage of the three-way handshake.

The final stage is for the active host, fillicudi, to acknowledge receipt of the SYN from unsst. This occurs and is represented in the final line of the TCP dump output in figure 5.1. Note that the flag value is simply '.' indicating no flags set. This does not include the ACK flag which will be set but is implied by the presence of the 'ack' and the ACK value further along. Also worth noting is that from the third segment onwards (i.e. the non-SYN segments) the acknowledgement number is relative, so an ACK of 1 indicates a real ACK of 10 in the raw segment.

09:59:06.003903 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: S, cksum 0x73d8 (correct), 11:11(0) win 1

09:59:06.130635 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) unsst.dcs.gla.ac.uk.webmin > filicudi.dcs.gla.ac.uk.10001: S, cksum 0x73be (correct), 9:9(0) ack 12 win 1

09:59:06.203395 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: ., cksum 0x73bf (correct), ack 1 win 1

Figure 5.1: TCP dump output for Scala TCP three-way handshake.

This TCP dump output confirms that the Scala implementation is able to perform the three-way handshake successfully. At each stage throughout this process the state of the connection at either host is changing, the active host moves through the following states: Closed → SYN sent → Established. The passive host moves through states: Listening → SYN received → Established.

The handshake is one of the more complex processes in TCP as where data transfer relies on existing information, the handshake builds a connection with very little known facts, including for the passive host, the source of the incoming connection. As such the handshake makes full use of the underlying code to strip down the headers and retrieve the necessary information. The principles by which the connection establishment is achieved are the same as those required for data transfer.

5.1.2 Data Transfer and Acknowledgement

In the same way that TCP dump has been used to demonstrate the success of the three-way handshake, it can be used to demonstrate successful data transfer. In the output shown in figure 5.2 data transfer is shown from an early stage in the connection. Small window sizes have been used to make the output more readable, for example the acknowledgement numbers increase in increments of 1 making them easier to track.

The TCP dump output shows 7 bytes of data being transferred across the network. The first segment contains two bytes and all subsequent segments contain a single byte of data. The advertised window for the connection is 1, as seen as the final entry in each line of output. Thus, it can be seen that the sending host is throttling its transfer speed based on the window being requested by the receiving host, another requirement of a TCP implementation.

The receipt of data is confirmed by the fact that each and every byte sent is subsequently acknowledged. Acknowledging segments are present in every second line, due to the small window constraint in this particular connection.

09:59:06.226759 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 42) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: ., cksum 0x72d5 (correct), 12:13(1) win 1

09:59:06.380684 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) unsst.dcs.gla.ac.uk.webmin > filicudi.dcs.gla.ac.uk.10001: ., cksum 0x73bd (correct), ack 3 win 1

09:59:06.390570 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 41) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: ., cksum 0x70d6 (correct), 14:15(1) win 1

09:59:06.578930 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) unsst.dcs.gla.ac.uk.webmin > filicudi.dcs.gla.ac.uk.10001: ., cksum 0x73bc (correct), ack 4 win 1

09:59:06.583376 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 41) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: ., cksum 0x6fd5 (correct), 15:16(1) win 1

09:59:06.783192 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) unsst.dcs.gla.ac.uk.webmin > filicudi.dcs.gla.ac.uk.10001: ., cksum 0x73bb (correct), ack 5 win 1

09:59:06.787000 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 41) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: ., cksum 0x6ed4 (correct), 16:17(1) win 1

09:59:06.985037 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) unsst.dcs.gla.ac.uk.webmin > filicudi.dcs.gla.ac.uk.10001: ., cksum 0x73ba (correct), ack 6 win 1

09:59:06.989826 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 41) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: ., cksum 0x6dd3 (correct), 17:18(1) win 1
09:59:07.190554 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) unsst.dcs.gla.ac.uk.webmin > filicudi.dcs.gla.ac.uk.10001: ., cksum 0x73b9 (correct), ack 7 win 1

09:59:07.194051 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 41) filicudi.dcs.gla.ac.uk.10001 > unsst.dcs.gla.ac.uk.webmin: ., cksum 0x6cd2 (correct), 18:19(1) win 1

09:59:07.390504 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40) unsst.dcs.gla.ac.uk.webmin > filicudi.dcs.gla.ac.uk.10001: ., cksum 0x73b8 (correct), ack 8 win 1

Figure 5.2: TCP dump output for Scala TCP showing data transfer.

Additional Observations from TCP dump

The TCP dump output in 5.2 can also be used to confirm the success of two other features of this implementation. Firstly, notice that the `cksum` value on each line is given (in hexadecimal form) and is listed as being "`(correct)`". This indicates that the checksum operation described in 4.1.1 has been successful for all of the segments being transmitted.

Furthermore, notice that the size of data being transmitted, in this case 1 byte per segment, is aligned with the advertised window of the receiver. This indicates that the sending host is correctly receiving notification of the window size as part of its execution and can adjust the send rate to suit. Additionally, this indicates that the method of communication, several asynchronous messages from the receiver to the sender via the connection, is successful as the sender is adapting to the conditions based on information that the receiver is providing.

5.2 Software Quality Evaluation

This section aims to establish if there is any difference in software quality between the Scala implementation of TCP and a C implementation. The C implementation used for this analysis will be that found in the Linux kernel v2.6.29.1 or the FreeBSD implementation described in [26]. At all times it must be taken into consideration that the Scala implementation is incomplete and does not offer many of the options which are available in the Linux equivalent. Despite this, comparisons can be successfully made between the two implementations to consider the impact on software quality.

Further to this, the Linux implementation of TCP has been through many versions and extensive testing as well as many revisions to improve code quality and efficiency. Due to the time constraints and the research nature of this project the Scala implementation has gone through only a small amount of refactoring.

5.2.1 Software Size and Modularity

The first consideration for software quality is the size of the implementations. Of course, given that the Scala implementation is incomplete and does not offer options the exact line count of the implementations is insufficient. [26] describes the FreeBSD implementation of TCP as being 4500 lines of code over 28 functions and in 6 source files. Using a similar listing of attributes for the Scala implementation, it consists of approximately 1500 lines of code over 130 functions and in 30 source files. Given the differences between the implementations it is impossible to say that the Scala implementation occupies less space than the FreeBSD equivalent, however, some conclusions can be drawn.

By taking these values as ratios of the total source code lines it is possible to draw a comparison between the two implementations. Firstly, the average number of lines per function in the FreeBSD implementation is approximately 161. This compares to an average of 11 lines per function in the Scala implementation. If we assume that this ratio is relatively consistent regardless of the amount of the protocol implemented then it is clear that an implementation in Scala, following the design principles outlined in this document produces and implementation which is more modular. There is no reason to suspect that this modularity will not continue as the code grows, the total number of functions in FreeBSD is 28 compared to 130 in the Scala implementation meaning that even if no more functions were required, which is highly improbable, the Scala implementation would still contain over 100 functions more than its C counterpart. Thus, for a definition of modularity that specifies that more functions means more modular code the Scala implementation would still be distinctly more modular.

However, the value of this modularity can be brought into question. It would be reasonable to suggest that a Scala implementation could make excessive use of functions to the point of reducing the quality of the code to increase modularity. Contrastingly, the use of a high degree of modularity could prove to provide a high degree of data encapsulation, as is the case in the Scala TCP implementation. It should be clear from chapter 4 that great lengths have been gone to to achieve a modular design which encapsulates components. Take the `Header` class, for example, which provides complete encapsulation for the segment header, including providing all checksumming functionality. At no stage in the remainder the code is an awareness of the structure of a TCP header required. Not only does this reduce the concerns of the programmer, it also allows for better collaboration as other areas of code can be completed in advance of the `Header` class by simply programming to the interface it presents. Furthermore, by modularising in this manner the ability to unit test becomes available to the programmer, this undoubtedly can lead to a higher quality of software.

The other ratio that can be considered by counting lines is the average numbers of lines per source file. With only 6 source files the FreeBSD implementation has an average of 750 lines per file, Scala TCP has a much lower average of 50. Once again this suggests that the modularity of the latter is better than that of the former. However, it also the case that the Scala TCP implementation could again be exhibiting modularity to the point of reducing software quality. The mark of correct modularity is that each component has a distinct purpose separate from the concerns of other components. It should be clear from the design and implementation chapters that this is the case in this project. C-based implementations do not go as far to separate concerns as has been achieved here, for example rather than encapsulating the header into a type many C implementations simply create a struct data structure that is visible to all areas of the system. Once again it is worth considering that although the Scala TCP is less complete than the FreeBSD equivalent, even if the number of source files did not increase in further development Scala TCP would have five times the number of source files

and therefore, by this measure, be more modular.

It is impossible to guarantee that the Scala implementation would consist of fewer lines than that of FreeBSD. However, it can be argued on two points. Firstly, Scala as a language requires less code to achieve the same goals. For example, as Scala has automatic memory management and a garbage collector it is unnecessary for the programmer to introduce code to control memory usage. Given that this is a requirement in the C implementation there are many lines of code which can be eliminated just by changing language. Many features in Scala will result in reducing code in this way compared to C. However, it is worth noting that the design approach taken here will increase the number of lines required. The high number of source files and functions lead to additional lines of code that would not be present in the "less modular" C implementation. These lines are class declaration statements, function headers and the requirement to repeat import statements in several different source files, of course it is also the case that there will be more import statements as there are more classes and objects to be imported in the first place.

The second argument for the Scala implementation having fewer lines is that the current implementation lays down a large portion of the "base" required for TCP operation. The largest portions of code in this implementation are those which handle state in the sender, receiver and connection. However, each feature of the implementation represents only maximum 20 lines of code in among this state code and has at most one supporting class which is specific to it. This is due to the high level of modularity in this implementation, for example the creation of segments is handled within the `Header` object and it consists of many lines. In the C implementation these lines may feature many times in the code, in the Scala implementation they are required once and called many times. If the maximum lines of code for a feature is 20 among the state code and each feature has at most one supporting class, and the average class size is 50 (from above), then each feature can be implemented with an average of 70 lines. This equates to over 40 addtional features before the source lines total surpasses that of the FreeBSD implementation. Furthermore, the largest source files are those containing the state, i.e. `Sender`, `Receiver` and `Connection`, had these been eliminated from the total the average class file size would be significantly lower, approximately 30 lines, thus further reducing the number of lines that a new feature would contribute.

5.2.2 Readability & Understandability

One of the major enhancements which this project presents is an increase in the readability of the TCP implementation. Readability is improved by better syntax, less code and a more structured design.

The Scala syntax is similar in style to that of Java and indeed C. However, it eliminates some of the more awkward syntax that can be found in C, for example

that dealing with pointers etc.

Understandability is improved by reducing the burden on the reader. The modular structure of the Scala TCP implementation leads to confined components of code which stand alone and require little understanding of the system as a whole. Within these components the use of case classes to identify messages leads to a very clean method of using the Actors model of concurrency. In using functions to represent particular states in TCP the order of state execution and the transitions between those states are very clear.

Within each state the use of `match` statements leads to further clarity in the code. By identifying the possible messages that are incident on any given state, the receipt of data in the ESTABLISHED state for example, a very clear view of the workings of that entire state becomes available.

For the most part the understandability of the system is improved by a stronger design which abstracts away the complexities of certain components. For example, the most complex component is likely to be the `Header` class, but by managing this separate from the remaining code a reader can either ignore the inner workings of the TCP header, or consider it in isolation, separate from the other components of the system.

5.2.3 Security

In using Scala for this development, and therefore compiling down to Java bytecode, the system in inherently more secure. The JVM prevents much of the access that has plagued C implementations of network protocols. Furthermore, by abstracting over memory management, issues such as buffer overflows are eliminated as memory is allocated randomly and in an unpredictable manner.

Overall, the development of network protocols would seem to be significantly improved in simply using a high-level language. The only area which would appear to be affected negatively is the efficiency. Efficiency is a major concern, especially when considering network traffic and as such this software is never likely to lead directly to a commonly used implementation. However, it is clear that Scala provides some features which would be very beneficial for network protocol development.

Chapter 6

Conclusions

Ultimately this project has shown that it is possible to implement a network protocol in a high-level language and in particular, Scala. It has been shown that such an implementation, with appropriate design, will be better structured and more readable. These facts alone make a high-level TCP more understandable and therefore more maintainable and extensible. The main draw back in this development is that a large cost is likely to be found in the efficiency of the protocol.

The Scala programming language has been used very successfully in this project and has shown to be very useful for this kind of development. However, it was noted throughout this project that Scala suffers from some problems. Firstly, in lacking support for unsigned types Scala makes any low-level programming problematic and the mechanisms for dealing with this, by assigning values to larger types, impact efficiency and reduce readability due to the amount of bit shifting that is require to move between types. Furthermore, Scala, being a relatively new language lacks the support that would be desirable of a high-level language. For example, most of this development has proceeded using text editors rather than a powerful IDE that would be available for more established languages like Java. The Scala community is also very small, meaning that the amount of information about Scala is reduced and difficult to find.

On the plus side however, Scala has shown that many of its features are excellent for this type of development. The Actors model of concurrency has shown to increase modularity of the software and, as computers continue to increase the number of CPUs on chip, this model will better support the concurrent operation of the protocol leading to a reduction in the efficiency gap. The most striking difference in using Scala is the modularity of the system produced, with 100 more functions and 24 more source files than the FreeBSD equivalent. Scala's ability to model systems as a mixture of classes and Actors means that a level of encapsulation is possible that would not be in many other languages.

This project presents a strong argument that high-level languages can be useful in network protocol design. In using a high-level language that is portable and

packed with features it takes the concept to extremes to highlight the benefits such development can provide.

6.1 Further Work

It would be useful to take the software developed here and continue towards a complete implementation of the Transmission Control Protocol. This project has developed a good base from which the protocol could be quickly developed and the resulting analysis would provide yet stronger results in favour of developing network protocols in high-level languages. Furthermore, a full analysis of efficiency could then be conducted to consider the impact developing in Scala has had on the speed of the protocol.

Another point of interest would be to consider developing some of the TCP options, perhaps Selective Acknowledgements (see section 2.2.2). Scala is intended as a language which scales well and the overall design of the system would seem to lend itself to being extended due to the highly modular way in which it operates.

It is possible that some of the lessons learned and discussed in this document could be used to begin to construct a language or framework around which network protocols of the future can be developed. Such a language would clearly place more emphasis on efficiency than this project has, but by singling out the features from Scala it can still begin to remove some of the difficulties found in developing these protocols in languages such as C.

Bibliography

[1] Jpcap. http://netresearch.ics.uci.edu/kfujii/jpcap/doc/.

[2] Rocksaw. Savarese.org, http://www.savarese.org/software/rocksaw/index.html.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[4] Edoardo Biagioni. A structured TCP in standard ML. *SIGCOMM Comput. Commun. Rev.*, 24(4):36–45, 1994.

[5] Edoardo S. Biagioni. A Structured TCP in Standard ML. Technical report, Pittsburgh, PA, USA, 1994.

[6] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), April 2003.

[7] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130, New York, NY, USA, 2008. ACM.

[8] Colin Myers, Chris Clack, Ellen Poon. *Programming with Standard ML*. Prentice Hall, 1993.

[9] Herb Derby. The performance of FoxNet 2.0. Technical report, 1999.

[10] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614 (Informational), September 2006.

[11] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[12] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 116–128, New York, NY, USA, 2005. ACM.

[13] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM.

[14] Mike Williams Joe Armstrong, Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[15] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006.

[16] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 3–13, New York, NY, USA, 1999. ACM.

[17] Martin Odersky, Lex Spoon, Bill Venners. *Programming in Scala*. Artima, 2008.

[18] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[19] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Trans. Comput. Syst.*, 10(2):110–143, 1992.

[20] Javier Paris, Victor Gulias, and Alberto Valderruten. A high performance Erlang TCP/IP stack. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 52–61, New York, NY, USA, 2005. ACM.

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

[22] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[23] Jonathan Shapiro. Programming language challenges in systems codes: why systems programmers still use C, and what to do about it. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems*, page 9, New York, NY, USA, 2006. ACM.

[24] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Weasley, 1999.

[25] W. Richard Stevens. *TCP/IP Illustrated Volume 1: The Protocols*, volume 1. Addison-Wesley, 1994.

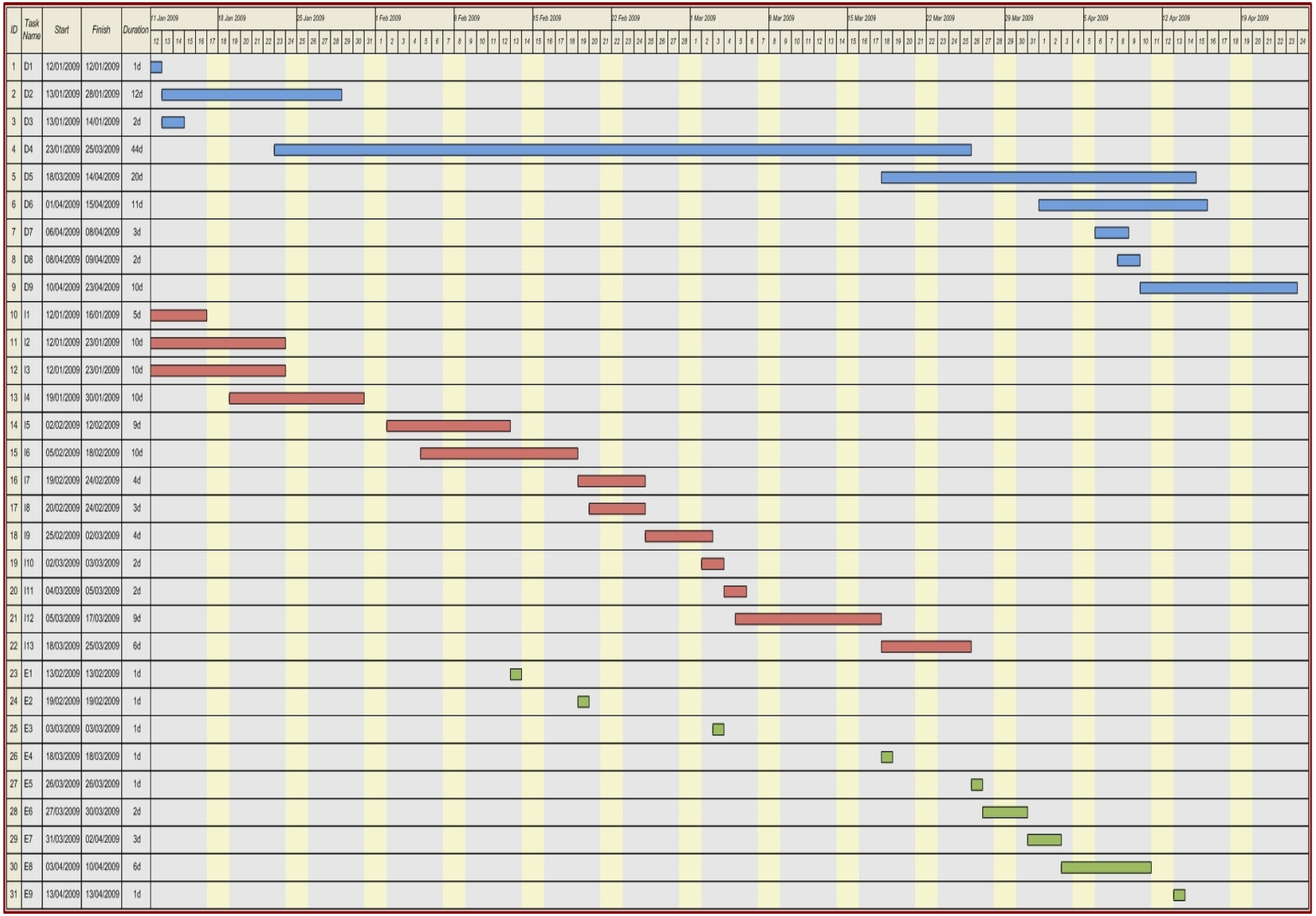
[26] W. Richard Stevens. *TCP/IP Illustrated Volume 2: The Implementation*, volume 1. Addison-Wesley, 1994.

Appendix A

Gantt Chart

The Gantt chart on the following page shows the timeline upon which the project was originally proposed.

Although initially progress was in line with this diagram the time became increasingly constrained due to difficulty with utilising the libraries for accessing the IP layer (RockSaw[2] and Jpcap[1]) as well as hardware setup problems and a general lack of familiarity with the Scala programming language. Ultimately many of the features proposed to be implemented could not be included within this timescale.



ID	Task Name	Start	Finish	Duration
1	D1	12/01/2009	12/01/2009	1d
2	D2	13/01/2009	28/01/2009	12d
3	D3	13/01/2009	14/01/2009	2d
4	D4	23/01/2009	25/03/2009	44d
5	D5	18/03/2009	14/04/2009	20d
6	D6	01/04/2009	15/04/2009	11d
7	D7	06/04/2009	08/04/2009	3d
8	D8	08/04/2009	09/04/2009	2d
9	D9	10/04/2009	23/04/2009	10d
10	I1	12/01/2009	16/01/2009	5d
11	I2	12/01/2009	23/01/2009	10d
12	I3	12/01/2009	23/01/2009	10d
13	I4	19/01/2009	30/01/2009	10d
14	I5	02/02/2009	12/02/2009	9d
15	I6	05/02/2009	18/02/2009	10d
16	I7	19/02/2009	24/02/2009	4d
17	I8	20/02/2009	24/02/2009	3d
18	I9	25/02/2009	02/03/2009	4d
19	I10	02/03/2009	03/03/2009	2d
20	I11	04/03/2009	05/03/2009	2d
21	I12	05/03/2009	17/03/2009	9d
22	I13	18/03/2009	25/03/2009	6d
23	E1	13/02/2009	13/02/2009	1d
24	E2	19/02/2009	19/02/2009	1d
25	E3	03/03/2009	03/03/2009	1d
26	E4	18/03/2009	18/03/2009	1d
27	E5	26/03/2009	26/03/2009	1d
28	E6	27/03/2009	30/03/2009	2d
29	E7	31/03/2009	02/04/2009	3d
30	E8	03/04/2009	10/04/2009	6d
31	E9	13/04/2009	13/04/2009	1d