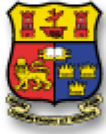

University College Cork **Computer Science** Department

Fourth Year Project 2005

A New Transport Protocol Module for the Network Simulator (NS-2)

Abstract

A network transport protocol is presented which incorporates aspects of both TCP and UDP. In short, the new protocol (RelUDP) is a reliable protocol like TCP, but which is a Datagram based protocol like UDP. The new protocol does not suffer from a phenomenon known as Head-of-the-Line (HOL) blocking which occurs in TCP and which uses an aggressive retransmission policy with minimal congestion control. The project was implemented as a new module for this written in C++ and added to the Network Simulator (NS-2), which is widely used in simulation experiments involving telecommunications network

Student Name:	Adrian Paul Duffy
Student ID:	101086481
Supervisor:	Dr John Vaughan
Second Reader:	Dr Richard Studdert

I confirm that, except where indicated through the proper use of citations and references, this is my own original work and that I have not submitted it for any other course or degree.

Signed **Adrian Duffy**

Date: / /

Table of contents

TABLE OF CONTENTS 2

ACKNOWLEDGEMENTS 4

1 CHAPTER 1 - INTRODUCTION 5

1.1 MOTIVATION 5
1.2 RESEARCH 7
1.2.1 General Network Research 7
1.2.2 The Network Simulator Research (NS-2) 8
1.2.3 Language Research (C++) 9
1.3 ORGANISATION OF PAPER 10

2 CHAPTER 2 – BACKGROUND RESEARCH 11

2.1 INTRODUCING THE NETWORK ARCHITECTURE 11
2.1.1 Internet Protocol (IP) 13
2.1.2 User Datagram Protocol (UDP) 15
2.1.3 Transmission Control Protocol (TCP) 16
2.1.3.1 Congestion Control 17
2.1.4 Comparison between UDP and TCP 19
2.1.5 Head-of-Line Blocking (HOL) 20
2.1.6 Session Initiation Protocol (SIP) 21
2.1.7 Voice over IP (VoIP) 22
2.2 LANGUAGES 22
2.3 NETWORK SIMULATOR (NS-2) 23
2.3.1 Choosing a Simulator 23
2.3.2 Basics 24
2.3.3 Event Driven Simulator 25

3 CHAPTER 3 – LOGICAL DESIGN OF NEW PROTOCOL 27

3.1 NETWORK SIMULATOR MODULAR DESIGN 27
3.1.1 Architecture Overview 28
3.1.2 Elements of Simulation 29
3.2 REQUIREMENTS 30
3.2.1 Reliability 30
3.2.1.1 Loss Detection 31
3.2.1.2 Loss Recovery 32
3.2.2 Retransmission 33
3.2.2.1 Deciding which packets to retransmit 33
3.2.3 Use Datagram to transport data 34
3.2.3.1 Header size 35
3.2.4 Connectionless and Stateless 36
3.2.4.1 No Connection Setup needed 36
3.2.4.2 No Connection State needed 36
3.2.5 Absent of HOL blocking 37
3.2.6 Minimal Congestion Control 38

4 CHAPTER 4 – PHYSICAL IMPLEMENTATION OF NEW PROTOCOL 40

4.1 INTRODUCTION 40
4.1.1 Naming the new protocol 41
4.2 FULFILLING REQUIREMENTS AND FUNCTIONALITY 41
4.2.1 Implementing a Basic Datagram protocol 42

4.2.1.1 Packets and Packet Headers 42
4.2.1.2 TclClass – Mirrored Hierarchy 44
4.2.1.3 oTcl – Linkage 46
4.2.1.4 sendmsg() method 47
4.2.1.5 recv() method 49
4.2.1.6 Summary 50
4.2.2 Implementing Reliability 50
4.2.2.1 Loss Detection 50
4.2.2.2 Loss Recovery 57
4.2.2.3 Enforcing a maximum Time Delay 60
4.2.2.4 Duplicate Acknowledgements 62
4.2.2.5 Retransmission Design issues 66
4.2.3 Hol blocking 68
4.2.4 No congestion control 68
4.3 FINAL VERSION 68
4.4 POTENTIAL ADDITIONS 71
4.4.1 Including a Random Element 71
4.4.2 Including Fragmentation 71
5 CHAPTER 5 –SIMULATION AND TESTING OF NEW PROTOCOL 73
5.1 INTRODUCTION 73
5.2 NETWORK ANIMATOR (NAM) 73
5.3 SIMULATIONS 74
5.3.1 Simulation 1 – A simple network 74
5.3.2 Simulation 2 – A simple network with Packet loss 75
5.3.3 Simulation 3 – A complex network with minimal congestion 76
5.3.4 Simulation 4 – A complex network with greater congestion 77
6 CHAPTER 6 - CONCLUSION 79
REFERENCES 81
REFERENCE FORMAT: 81
ABBREVIATIONS 83
A APPENDIX 84
A.1.1 RELUDP.H 84
A.1.2 RELUDP.CC 85

Acknowledgements

I would like to thank my supervisor Dr John Vaughan for his invaluable guidance and timely input provided throughout the duration of this project. His encouragement was a great source of motivation for me. I would especially like to thank him for his extensive comments on the project write up.

I would also like to thank Ph.D. student Meredith Lulling, for sharing his extensive knowledge and for his constant support during the course the project. His constant assistance and overwhelming interest have made this project both an interesting one and a successful one.

Adrian Duffy

20th March 2005

1 Chapter 1 - Introduction

The Internet began as an experiment in the late 1960s and has become a world-wide data network that is used for a wide variation of applications. As Internet is no longer owned by a single entity it is a conglomeration of tens of thousands of independently managed computer networks, some different than others. The Internet was designed to provide best effort service and does that to a large extent even now. The Internet attempts to route all packets as soon as possible but provides no guarantees regarding the Quality of Service provided to a user. Most Internet communications are based on data transfers over connections between pairs of hosts. These hosts communicate with each other by a standardised set of protocols. The basic protocols that are used in the Internet are TCP [1] and UDP [2]. Both of these protocols provide a very different service to applications.

1.1 Motivation

Recent advances in technologies and availability of less expensive computer processing power have led to a surge in Applications offering new and improved services online to users. There is a great increase in Multimedia applications such as videoconferencing and applications which use the network as a method to make voice calls (VoIP).
Unfortunately the two Transport protocols which are commonly used throughout the Internet do not suit the needs of the emerging Multimedia applications.

TCP provides a reliable connection and is used by the majority of current Internet applications. Reliable in the way that TCP promises to deliver any packets sent with its protocol. TCP, besides being responsible for error checking and correcting, is also responsible for controlling the speed at which this data is sent, for example congestion control. TCP is capable of detecting congestion in the network and will back off transmission speed when congestion occurs. TCP provides the functionality which is needed my Multimedia applications but with a price. TCP has high overhead with big packet sizes that cannot effectively provide the service needed by the new emerging applications.

UDP however provides no congestion control mechanisms or reliability. It simply sends the Packet and provides no attempt to detect packet loss/delay etc or overcome it. UDP uses a small packet called a Datagram, which can be sent quickly through a network. UDP is not suitable for Multimedia applications either as of its lack of functionality. Multimedia applications need some form of reliability since they are providing a service.

The aim of this project is to design a new Protocol that will be suitable for Multimedia applications. It should have the functionality of TCP with the small packet size of UDP.

One way to test and evaluate a protocol and its application benefits is to create a real world implementation for an arbitrary operating system and set up test environments consisting of links, switches routers and end hosts. However, initially it may be more economical to work with a simulator, as it is considerable cheaper and fast and easy to change the environment or the implementation and monitor the results. Thus the new Protocol was designed and implemented in the Network Simulator (NS 2) [3].

1.2 Research

The research carried out during this project can be categorised into different sections. These sections will be discussed in the following sections.

1.2.1 General Network Research

A broad understanding of computer networks was required in order to complete the project. The main book which was used during the project was called "Computer Networks A systems Approach" [3]. This is a detailed book which describes computer networks and all the necessary implementation issues concerning computer networks, invaluable when designing a new protocol. A great number of websites were useful while researching for the project. The first website [4] describes the IP protocol, internet routing, ICMP, TCP, UDP all in detail. This site was very useful at the beginning of the research stage. Another website which was quite useful was involving the Connectionless Transport UDP [5]. This site introduces UDP in some detail, discussing its advantages and disadvantages. TCP's congestion control, algorithms used and its variants are discussed thoroughly in the website [6]. A very useful comparative analysis between TCP and UDP is discussed in the website [7], which introduces all the implementation issues and the suitable uses of the two protocols depending on different network demands.

The Session Initiation Protocol (SIP [8]) was discussed in detail in the website [9]. It was necessary to research multimedia applications such as Voice over IP (VoIP), therefore a presentation by Peter Parnes provided invaluable. The presentation [10] describes VoIP in detail, including SIP, call setup, with worked examples. An article discussing Internet Telephony provided useful

[11]. It introduced the requirements and implementation of the area, discussing IP telephony in general, the challenges associated with IP, Quality of Service (QoS), Reliability and optical Ethernet. A potential reliable protocol and its features are examined in the website [12]. Another similar paper discusses "Reliable IP Telephony Applications with SIP" [13] and gave a good introduction to applications that need reliability, SIP and Internet Telephony in general. One paper which introduced SIP and TCP's variations and also more importantly examines the throughput variation for SIP traffic was quite useful [14]. The paper gave a good incite into the different variants of TCP, how they effect SIP's performance.

1.2.2 The Network Simulator Research (NS-2)

The Network simulator is quite difficult to use at first and even harder to master. There is a very steep learning curve with the Simulator itself, thus it takes a large amount of effort before a user could reach the required level to add a new agent in the Simulator. Therefore quite a lot of time was spent using tutorials and guides on the Internet while researching the Simulator.
The following tutorials were some of the material used in researching the Network Simulator.

- "Ns documentation" [15], this is the documentation that comes with the Network Simulator. It is quite useful in understanding the Simulator and how it works. The main problem with the documentation is that it is not regularly updated, and does not always correspond with the version of NS which is available.
- "Tutorial for the Network simulator ns" [16], is a quite detailed tutorial which discusses the basics of the Network Simulator, the implementation of a simple ping protocol and also

introduces the basic workings of simulations in NS using Tcl scripts.

- "How to add a new protocol in ns2" [17], detailed presentation illustrating how to add a simple ping protocol to the network simulator. Some very important fundamentals were learned from this.
- "Network simulation and protocol implementation using network simulator 2" [18], This is a basic paper introducing the protocols available in NS-2 with a light overview how they work
- "Network Simulator (ns) Tutorial 2002" [19], this website contains a number of tutorials dealing with the Network Simulator. It is a good preliminary guide to extending ns, discussing packet headers, memory allocation and debugging new code.
- "NS by example" [20], this is a very good tutorial, describes the workings of ns in depth, including the scheduler, Tcl scripts, linking variables etc. This is one of the best tutorials available for beginners/advanced users of the Network Simulator.
- "Ns-2 tutorial" [21], this is a simple presentation that discusses Tcl and oTcl in detail. It also introduces the Network Animator and variable linkage. The only downfall is that this presentation does not go into a lot of detail.
- "Ns-2 network simulator" [22], describes all necessary aspects of the simulator for moderately advanced users, dealing with simulation scripts and also the corresponding Tcl and C++ object hierarchy.

1.2.3 Language Research (C++)

In order to implement a new protocol in the Network simulator, advanced knowledge of C++ was needed. All the network objects

are implemented in C++, therefore quite a good understanding of C++ is needed to adjust or create new objects. The book C++ which was firstly used during this project was called "A Guide to C++ Programming" [23]. This book introduces C++ at a beginner level and guides the reader to moderately detailed understanding of C++. Unfortunately the book did not go into enough detail required for the project. To get the required understanding of the language, online tutorials were used [24]. This tutorial went into enough detail to complete the project successfully.

1.3 Organisation of Paper

This project is organised as follows

- Chapter 2 introduces all the background technologies involved in this project
- Chapter 3 discusses the logical design of the new protocol, discussing a variety of design issues and the consequences.
- Chapter 4 discusses the physical implementation of the protocol in the Network Simulator. This chapter is divided into stages of development, progressively getting more complicated and closer to the final product.
- Chapter 5 shows the various tests and simulations that were completed on the protocol using NAM [25].
- Chapter 6 finally concludes the Project.

2 Chapter 2 - Background Research

2.1 Introducing the Network Architecture

The basic network architecture that is used around the world to connect users and allow the exchange of information is simple but moderately effective. This collection of users connected over computer networks is commonly referred to as "The Internet". Firstly the Internet is a collection of various types of networks spread throughout the world. All these different types of networks can interact because they all use the same basic network model and network protocols. Basically what distinguishes the Internet is its use of a set of protocols called TCP (Transmission Control Protocol) and UDP (User Datagram Protocol), both of which sit on top of a basic protocol called IP (Internet Protocol).

Application	FTP	E-mail	WWW	DNS	VoIP	NFS
Transport	TCP			UDP		
Network	IP					
Physical	Ethernet		AAL-5		HDLC	

Figure 1 DoD Layered Network Architecture

To begin with, Data networks are basically telecommunication networks that are operated for data/information exchange between data communication devices such as computers. Figure 1 shows the layered architecture of such a network. This reference Model is known as the "DoD Four Layer model" as it was originally created

by the United States Department of Defence. It is now considered the primary architectural model for internetworking communications. Most of the network protocols used today have a structure generally based on this model.

This model is split up into four basic layers.

- Firstly the application layer provides services that software applications need and also provides the ability (an interface) for user applications to interact with the network.
- The transport layer is the layer that deals with providing the requirements needed by the application layer. Currently there are only two commonly used protocols in the Transport layer, these being TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).
- The network layer implements the Internet Protocol (IP) which is common to all applications run on a network. The more technical aspects of the network such as routing messages using the optimum path available and dealing with link failures etc. are dealt with in this layer.
- Finally the Physical layer is responsible for delivering data over the particular hardware medium in use.

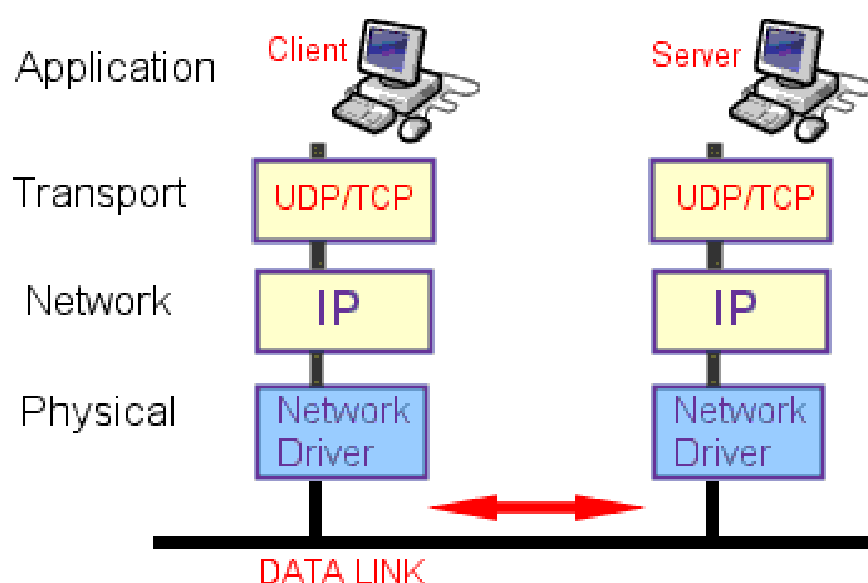

Figure 2 DoD Layered Network Architecture example

The layer we will be most interested in will be the Transport layer. As already mentioned this layer has two commonly used protocols, the advantages disadvantages of these will be further explained in the oncoming sections.

2.1.1 Internet Protocol (IP)

As already mentioned the IP protocol (Internet Protocol) operates on the network layer. The Internet Protocol uses small objects called datagrams to transfer data over a packet switched network. This protocol provides a best effort network layer service for connecting computers to form a computer network. Every node/computer on the network is identified by a globally unique address called an IP address. The containers that are used to carry the data being sent across the network are known as either "packets" or "datagrams". Each packet carries the IP address of the computer that is sending the packet and also the address of the intended recipient or recipients of the packet. Other management information is also included.

IP routers transmit datagrams between intermediate nodes on the IP network. The IP routers are kept quite simple, as no actual information is stored concerning the datagrams that are forwarded on the network link. The most difficult task IP routers must complete is concerned with determining the optimum link to use to reach the desired destination in a network. This process is known as "routing" and although this process is quite computationally intensive, it is only performed at periodic intervals.

An IP network typically uses a dynamic routing protocol to find alternate routes whenever a link fails and also to find the

shortest/fastest path to the destination. This provides considerable robustness from the failure of either links or routers. This does not guarantee reliable delivery, but more a "Best Effort" to transmit the data successfully. This "best effort" service is adequate for some applications and it is possible for these applications to use the simple transport protocol (UDP).

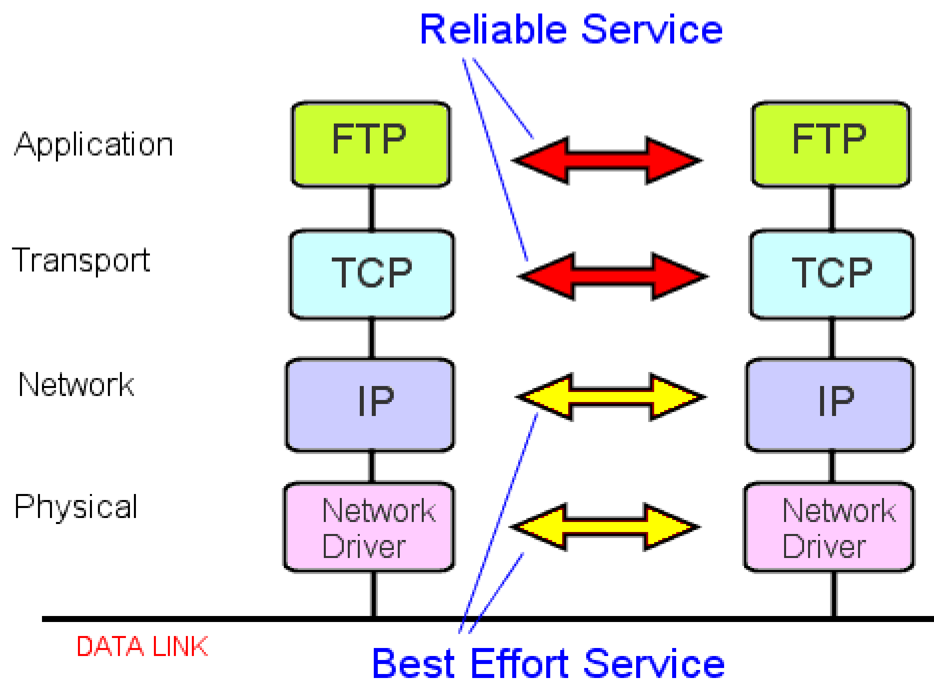

Figure 3 DoD Layered Network Architecture, illustrating reliable and unreliable layers

Some applications such as FTP, WWW etc need additional functions such as end-to-end error and sequence control to give a reliable service. This reliability is provided by the Transmission Control Protocol (TCP) which is widely used across the Internet.

2.1.2 User Datagram Protocol (UDP)

UDP stands for User Datagram Protocol that provides a connectionless host to host communication path. This protocol is connectionless meaning it does not require a prior connection setup as in TCP for example advertising, negotiation or preparation. UDP has minimal overhead; each packet on the network is composed of a small header and user data. It is called a UDP datagram. The UDP packet structure can be seen in figure 4.

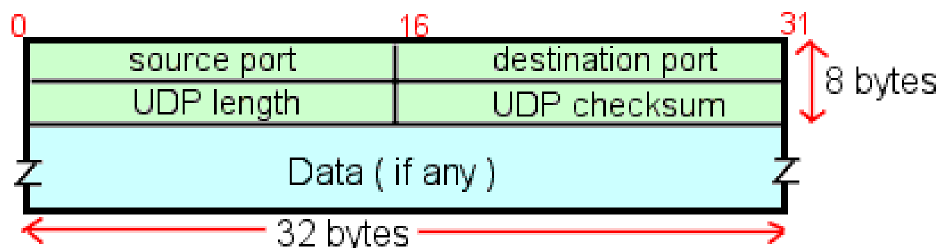

Figure 4 UDP Packet Structure

UDP preserves datagram boundaries between the sender and the receiver. This means that the receiver will receive an event for each datagram sent and the Receive method will return a complete datagram for each call. If the buffer is too small, the datagram will be truncated on the other hand if the buffer is too large, just one datagram is returned.

UDP is an unreliable protocol. Therefore there is absolutely no guarantee that the datagram will be delivered to the desired destination. Unless the bandwidth is full (high congestion) the failure rate is very low on the Internet and Local Area Networks.

Another point to note about UDP is that as well as the problem of datagrams not being delivered, it is also possible that they are delivered in an incorrect order. It means that if two packets were

sent straight after each other, it is possible to receive the second packet before the first one, as the packet sent first might have been delayed in the network. You can also receive duplicate packets.

The application that is running on UDP must be prepared to handle all those situations: missing datagram, duplicate datagram or datagram in the incorrect order. So if reliability and in order delivery is needed by the application it will need to be implemented in the application layer.

UDP is very effective where fast delivery is needed, but where reliability is not.

2.1.3 Transmission Control Protocol (TCP)

TCP stands for Transmission Control Protocol. TCP is a connection-oriented protocol that is responsible for reliable communication between two end processes/applications. Unlike UDP, TCP's unit of data transferred is called a stream, which is a sequence of bytes.

With TCP, before actually transmitting any data between the two end processes/applications, a connection between the two end points must be created. The data can be transferred in full duplex connection. A full duplex connection is one where data can be sent and received simultaneously on a single connection. To free system resources, the connection must be closed when the transfer is completed. Both end points know when the session is opened and closed. The data transfer cannot take place before both ends have agreed upon the connection. Either side can close the connection; the closing side informs the other side. Provision is made to close gracefully or just abort the connection.

As already mentioned TCP's unit of data transferred is called a stream (a sequence of bytes). Being stream oriented means that

the data is an anonymous sequence of bytes. There is nothing to make data boundaries apparent as there is in UDP. The receiver has no means of knowing how the data was actually transmitted across the network. The sender can send many small data chunks and the receiver receive only one big chunk, or vice versa. The only thing that is guaranteed is that all data sent will be received without any error and in the correct order. Should any error occur, it will automatically be corrected (the data will be retransmitted as needed) or the error will be notified if it can't be corrected.

When data is to be sent, it will be put it in a buffer until it can be actually transmitted. Eventually the data will be sent in the background and an event will be generated once the buffer is emptied. On receiving data, a program must wait until it receives the event, which tells it that data was received. Each time a data packet comes from the lower level, this event is triggered. The application must call the Receive method to actually get the data from the low-level buffers.
As the data is a stream of bytes, your application must be prepared to receive data as sent from the sender, fragmented in several chunks or merged in bigger chunks. What happens depends on traffic load, router algorithms, random errors and many other parameters that are uncontrollable.
The majority of TCP protocols use a special code as command delimiter to suggest the end of the data sent. Each client request is sent as is with this special code appended (CR/LF pairs). The server receives the data as it arrives, assembles it in a receive buffer, scans for the special code to extract commands from the received stream, and finally removes them from the receive buffer.

2.1.3.1 Congestion Control

Congestion control is a major issue when it comes to networks, and TCP has a variety of implementations to deal with this. The basic

problem is how to deal with high congestion in a network. If the network is highly congested a large number of packets will be delayed or lost, this will in turn cause a high rate of retransmission. A high rate of retransmission in a congested network will in fact congest the network even more so. There are a few variations of the TCP protocols, which deal with congestion differently.

TCP Tahoe is the base standard used in modern TCP implementations. Originally end system based traffic control functions were added which were designed to address congestion issues in a network. This was partially due to the threat of congestion collapse in the Internet as a whole. TCP Tahoe employs a reactive congestion control strategy. What this means is that the protocol has to experience loss in order to control its throughput. This can have a negative effect on the performance of the network as a result. In summary, TCP Tahoe repeatedly increases the rate at which packets are being sent in an effort to find the point at which congestion occurs, and then uses this point as a maximum send rate.

TCP Reno is basically the enhanced version of TCP Tahoe which includes the Fast Recovery algorithm. This is an extension of the Fast Retransmit algorithm, which optimises Reno for the case when a single packet is dropped from a window of data (buffer), therefore is more subtle than Tahoe. Reno also supports the use of Delayed Acknowledgements, which basically acknowledges every second packet instead of acknowledging every packet.

TCP Vegas takes a different approach and uses the measured Round Trip Time (RTT) to accurately calculate the amount of data packets that the sender can send to avoid congestion. The Round

Trip Time is the time it takes from the moment a packet is sent to the moment it is acknowledged. Vegas also implements a Slow Start mechanism. Vegas tries to predict when congestion is about to occur and adapts its window/send rate to compensate. This has a proactive approach as it attempts to reduce rate its sending rate before packets start being discarded by the network, being more congestion avoidance, instead of congestion control.

TCP Sack uses the TCP Reno algorithms and adds the feature of Selective Acknowledgement (Sack). The Selective Acknowledgement strategy was first implemented to address the problems presented by multiple packet loss from a window of data or multiple packet loss in a short period of time. Essentially the SAck version of TCP provides a mechanism whereby the data receiver can inform the sender of all packets successfully received rather than simply acknowledging the last packet that was received in order. This prevents the retransmission of packets that have been successfully received but not acknowledged.

2.1.4 Comparison between UDP and TCP

The two protocols described here, TCP and UDP are the most commonly used ones. The reason why there are only two commonly used transport protocols is mainly because they both are simple, extendable and both serve completely opposite needs and requirements. TCP provides a reliable connection and is used by the majority of current Internet applications. TCP, besides being responsible for error checking and correcting, is also responsible for controlling the speed at which this data is sent, for example congestion control. TCP is capable of detecting congestion in the network and will back off transmission speed when congestion

occurs. These features protect the network from congestion collapse. UDP however provides no congestion control mechanisms. A congested link that is only running TCP will be approximately fair to all users. When UDP data is introduced into this link, there is no requirement for the UDP data rates to back off, forcing the remaining TCP connections to back off even further. Thus UDP have been proved to dominate congested links.

TCP	UDP
· Connection-Oriented · Reliability in delivery of messages · Splitting messages into datagrams · Keep track of order (or sequence) · Use checksums for detecting errors	· Connectionless · No attempt to fragment messages · No reassembly and synchronization · In case of error, message is retransmitted · No acknowledgment
o Remote procedures are not idempotent o Reliability is a must o Messages exceed UDP packet size	o Remote procedures are idempotent o Server and client messages fit completely within a packet o The server handles multiple clients (UDP is stateless)

Table 1 UDP vs TCP

The table 1 illustrates the main differences of both protocols.

2.1.5 Head-of-Line Blocking (HOL)

Head-of-the-Line (HOL) blocking is a phenomenon that can occur when TCP is used as a transport protocol in Session Initiation Protocol (SIP) signalling. TCP assumes a single stream of data and ensures that the segments of that stream are delivered in the

sequence in which they are sent. In telephony call set-up most of the segments in the stream will not be inter-related. The majority will be independent from each other. The problem occurs when sending independent messages over an order-preserving TCP connection which in turn causes the delivery of messages sent later to be delayed within a receiver's lower layer buffers until an earlier lost/delayed message is retransmitted and arrives. Segments behind the recovering packet are delayed even though they may not be related to it. This can have undesirable effects on the performance of Applications using the protocol.

2.1.6 Session Initiation Protocol (SIP)

The Session Initiation Protocol (SIP) is a signalling protocol used for establishing sessions in an IP network. A session could be a simple telephone call between two parties or it could be a collaborative multimedia conference session. The ability to establish these sessions means that a large amount of innovative services become possible, such as voice enriched internet content, make a voice call from a link on a web page, Instant Messaging etc. The Voice over IP (VoIP) community has adopted SIP as its protocol of choice for signalling. At the moment SIP is still evolving and being extended as new technology arises.

When creating a session SIP specifies only what it needs to specify. This saves on the amount of data that needs to be sent, thus making the whole process faster. SIP was developed purely as a mechanism to establish sessions, it is not concerned about the details of a session, it just initiates, terminates and modifies sessions. This simplicity means that SIP is scaleable, extensible, and it sits comfortably in different deployment scenarios and architectures. SIP is a request response protocol that closely

resembles two other Internet protocols, HTTP and SMTP. Using SIP, telephony becomes another viable web application and integrates easily into other Internet services. Therefore the protocol is basically a simple toolkit that converged-voice and multimedia services can be created with.

2.1.7 Voice over IP (VoIP)

VoIP (voice over IP) is basically voice delivered using the Internet Protocol (IP). In general, this means sending voice information in digital form in individual packets rather than in the circuit-committed protocols of the public switched telephone network. A major advantage of VoIP and Internet telephony is that it is cheaper as it avoids the charges charged by local telephone services. To help ensure that packets get delivered in a timely fashion VoIP uses the real-time protocol (RTP). When using public networks, it is currently quite difficult to guarantee Quality of Service (Qos) in a voice call, for example to guarantee that the call will stay at a certain standard 100% of the time. A more suitable service is possible when provided by private networks specifically for VoIP that can guarantee a certain Quality of Service. One technique used to help ensure faster packet delivery is to ping all possible network Gateways that have access to the public network and choose the optimum path before establishing a TCP connection with the other end.

2.2 Languages

In order to make any progress in this project, two programming languages were vital. These languages are C++ and Tcl. The C++

code is discussed in greater detail in the chapter concerning the physical implementation of the protocol. The Tcl scripting can be seen in the chapter concerning the simulation of the protocol.

2.3 Network Simulator (NS-2)

The Network Simulator (version 2) is an object-oriented, discrete event driven network simulator that was originally developed at the University College of Berkeley. It is written in C++ and oTcl. NS is primarily useful for simulating local and wide area networks (LAN, WAN), and thus invaluable when designing a new Protocol. The disadvantage of the Network Simulator is that it is a considerably large system with a relatively steep initial learning curve. Initially it is quite difficult for a first time user to use or attempt to extend the Simulator, as there are few user-friendly manuals.

2.3.1 Choosing a Simulator

The NS-2 simulator was chosen over other available simulators for a number of reasons.

- NS-2 is the most widely used simulator for network simulations.
- The Simulator is an open source, freely downloadable piece of software, which runs on Linux platform and on windows platform under Cygwin.
- NS-2 is easily extensible once familiar with the software; any extensions to existing network protocols can be implemented with ease.
- Since NS-2 is widely used, there are many simulation results available, which can be used for comparison for new Agent/Protocol simulations.

2.3.2 Basics

The Network Simulator is capable of simulating a wide variety of IP networks. It implements network protocols such as TCP and UPD, traffic source behaviour such as FTP (File Transfer Protocol), Telnet, Web, CBR and VBR. It also is capable of simulating router queue management mechanisms such as Drop Tail, RED (Random Early Detection) and CBQ (Class Based Queuing), routing algorithms such as Dijkstra etc. Currently, NS is written in C++ and oTCL (Object-oriented Tcl script language).

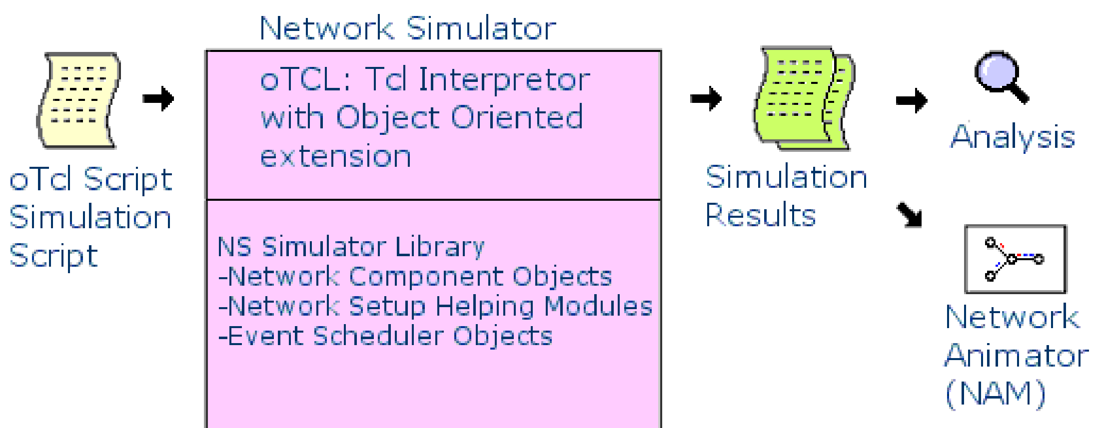

Figure 5 Simplified users view of the Network Simulator

Figure 5 depicts a simplified user's view. NS is fundamentally an Object-oriented Tcl (oTcl) script interpreter that has a simulation event scheduler and network component object libraries. In order to join these network objects so that they can interact NS has network setup (plumbing) module libraries (actually, plumbing modules are implemented as member functions of the base simulator object). In order to use NS, the OTcl script language is used. Examples of such scripts can be found in Chapter 5. To setup and run a simulation

network, a user should write an OTcl script that initiates an event scheduler, sets up the network topology using the network objects and the plumbing functions in the library, and tells traffic sources when to start and stop transmitting packets through the event scheduler. In the network setup, the possible data paths among network objects are connected by setting the 'neighbour' pointer of an object to the address of an appropriate object. When a new network object is required, it can be created either by writing a new object or by making a compound object from the existing object library, and connect the data path through the object. The plumbing or interconnecting OTcl modules simplify the process significantly. The power of NS comes from this interconnecting.

2.3.3 Event Driven Simulator

Another major component of NS beside network objects is the event scheduler. An event in NS is a unique packet ID which has a scheduled time and the pointer to an object that handles the event. It is the event scheduler that keeps track of simulation time and runs all the events in the event queue scheduled for the current time by invoking appropriate network components. It is usually these network components that issued the events, and let them do the appropriate action associated with packet pointed by the event. Network components communicate with one another passing packets between each other. This passing of packets however does not consume actual simulation time. If on the other hand a network components needs to simulate time handling a packet it simply uses the event scheduler by issuing an event for that packet. It then waits for the event to be fired to itself before doing further action handling the packet. Another use of an event scheduler is the timer. For example, the Transmission Control Protocol uses a timer

to keep track of a packet transmission time out. Timers use event schedulers in a similar manner that delay does. The only difference is that the timer measures a time value associated with a packet and does an appropriate action related to that packet after a certain time goes by, and does not simulate a delay.

3 Chapter 3 - Logical Design of New Protocol

3.1 Network Simulator Modular Design

The design of the Network Simulator (NS-2) uses a paradigm called shared object design. This means that the system is programmed in two languages. In both languages there is a corresponding hierarchy of network objects, but the objects in one are accessible in the other. There are also objects accessible to only one part of the system. NS is written in Object Oriented Tcl and C++. This is mainly for an efficiency reason. NS separates the data path implementation from control path implementations. In order to reduce packet and event processing time, the event scheduler and the basic network component objects in the data path are written and compiled using C++.

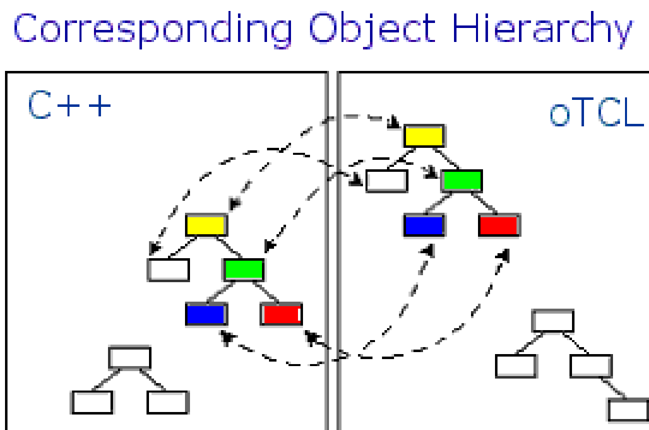

Figure 6 Corresponding Object Hierarchy between C++ and oTCL

These compiled objects are made available to the OTcl interpreter through an OTcl linkage. This OTcl linkage creates a matching OTcl object for each of the C++ objects and also makes the configurable control variables specified by the C++ object act as member functions and variables of the corresponding OTcl object. In this way, the controls of the C++ objects are given to OTcl. This makes it possible to change linked C++ variables from a TCL script. It is also possible to add member functions and variables to a C++ linked OTcl object. Of course some objects in C++ that do not need to be controlled in a simulation or internally used by another object do not need to be linked to OTcl. Figure 6 shows an object hierarchy example in C++ and OTcl. One thing to note is that for C++ objects that have an OTcl linkage forming a hierarchy, there is a matching OTcl object hierarchy very similar to that of C++.

3.1.1 Architecture Overview

Figure 7 shows the general architecture of NS. A general user (not an NS developer) can be thought of standing at the left bottom corner, designing and running simulations in Tcl using the simulator objects in the OTcl library. As you move from this corner to the right hand top corner more knowledge and understanding of NS as a whole is needed. The event schedulers and most of the network components are implemented in C++ and

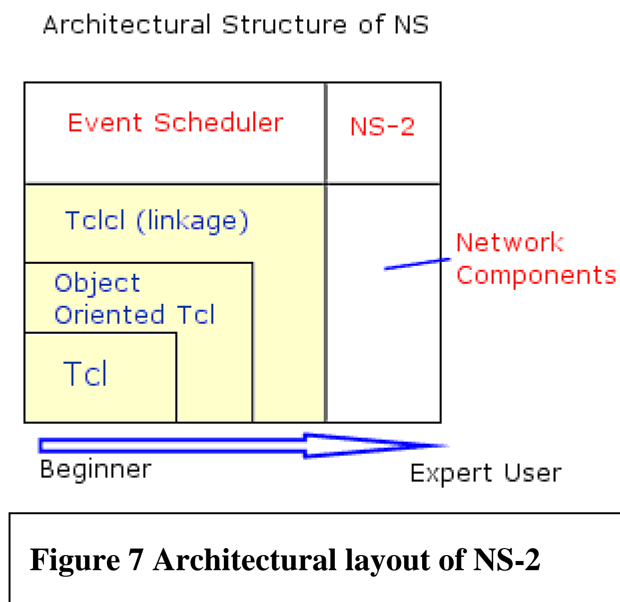

Figure 7 Architectural layout of NS-2

available to OTcl through an OTcl linkage that is implemented using tclcl. The whole diagram together makes NS, which is an Object Oriented extended Tcl interpreter with network simulator libraries.

The network can be freely designed in a scenario and it is written in OTcl. The scenario contains descriptions for some or all of the elements of simulation described below.

3.1.2 Elements of Simulation

In this section the elements of simulation in ns are described.

- Nodes in ns handle packet forwarding and can be both receivers and routers at the same time. There are both unicast and multicast nodes available in the simulator.
- Links basically interconnect two nodes.
- Packets work as events in the event driven simulation. Receiving or sending a packet is essentially an event. There are functions for creating, accessing and changing both the packets and the headers of the packets as will be discussed later.
- Queues can be connected to links. The type of dropping/forwarding procedure can be chosen from several different types.
- Error models are used to introduce packet losses in the simulation. Thus simulating link failures/congestion etc. This is useful as it creates a controllable aspect of introducing packet losses or congestion simulation.
- Agents represent the endpoints for network layer packets in the simulation. They are responsible for creation and destruction of these types of packets. There are several agents already implemented in NS, for example UDP, TCP etc.

It is the goal of this project to add an agent to the NS hierarchy.

- Applications are above the agents. There are two types of applications; traffic generators and simulated applications. The traffic generators generate traffic in a predefined pattern. Simulated applications are for example FTP and telnet where the generated traffic is made as similar to the real applications traffic as possible.

3.2 Requirements

In this section the logical requirements of the new protocol that is being added will be discussed. The basic requirements are as follows.

- Reliability
- Retransmission
- Use Datagram to transport data
- HOL blocking absent
- Minimal Congestion Control

3.2.1 Reliability

When creating a protocol which will be aimed for VoIP signalling or possibly multimedia applications such as video conferencing one of the most basic functionality that's expected is reliable transport. For example if two users were using a multimedia application on the network such as a voice call or a videoconference it is vital that the data sent gets delivered successfully. If the data being sent does not arrive at its desired location the quality of the call/conference is deeply impacted for example voices become unrecognisable or video stalls. This is because the application on the receiving end is

missing vital information that was lost over the network. This loss of data can be due to various reasons that are common in the Internet. It should also be noted that the receiver might have got the data but when it checked the data and found that the data has been corrupted (may be due to transmission errors) the receiver sent a retransmission request.

Providing data reliability consists of two basic components.

- Loss detection
- Loss Recovery.

3.2.1.1 Loss Detection

One of the fundamental issues to be dealt with when designing a reliable protocol is how does the receiver get to know that it has lost a packet that it should have received. Similarly how does the sender know that a packet it sent did not arrive at the destination? There are many approaches to handle this. Now let's try to understand the various techniques which illustrate how the loss could be detected.

If the data from the sender is a set of packets with sequence numbers then the loss of a packet will be detected when the receiver receives a packet whose sequence numbers are not contiguous. Say the receiver received a packet with sequence number 1 and another with the sequence number of 3, its pretty clear that the receiver has missed packet with number 2. But one thing to note here is that we are assuming in order delivery of packets, which we do not want to enforce. In the case of out of order delivery of packets, it could very well be the case that the packet with the sequence number 6 was not lost but is on its way to the receiver. So in cases like these the receiver cannot simply ask

the sender to retransmit just because the packet has not arrived in order.

One method would be to take the workload off the receiver and for the sender to figure out if a packet was lost in the network. If all packets sent included a sequence number this could be possible. When the receiver receives a packet of sequence number X it sends a small packet called an acknowledgement also with sequence number X back to the sender, informing the sender that the packet has not been lost. This may seem like a potential solution but it does not cater with all the possible events. If the acknowledgement was lost in the network, the sender would believe that packet also to be lost in the network. If the sender could record the packets sent in a buffer. On receiving an acknowledgement the sender could then remove the acknowledged packet from the buffer. Thus leaving the buffer full of packets that were lost in the network or packets whose acknowledgements were lost in the network.

3.2.1.2 Loss Recovery

As the name suggests this function deals with how the sender responds when it has detected a packet loss. Basically when the receiver receives a packet it sends an acknowledgement back to the sender, at this point the sender knows if a packet or the corresponding Acknowledgement has been lost.

Figure 8 illustrates the various possible situations where either the Datagram packet or the Acknowledgement gets lost in the network. As already mentioned the sender has a buffer of sent packets, which is really a buffer of unacknowledged packets. Thus the sender can use this buffer to judge which packets need to be retransmitted. The issue of retransmission is dealt with in the next section.

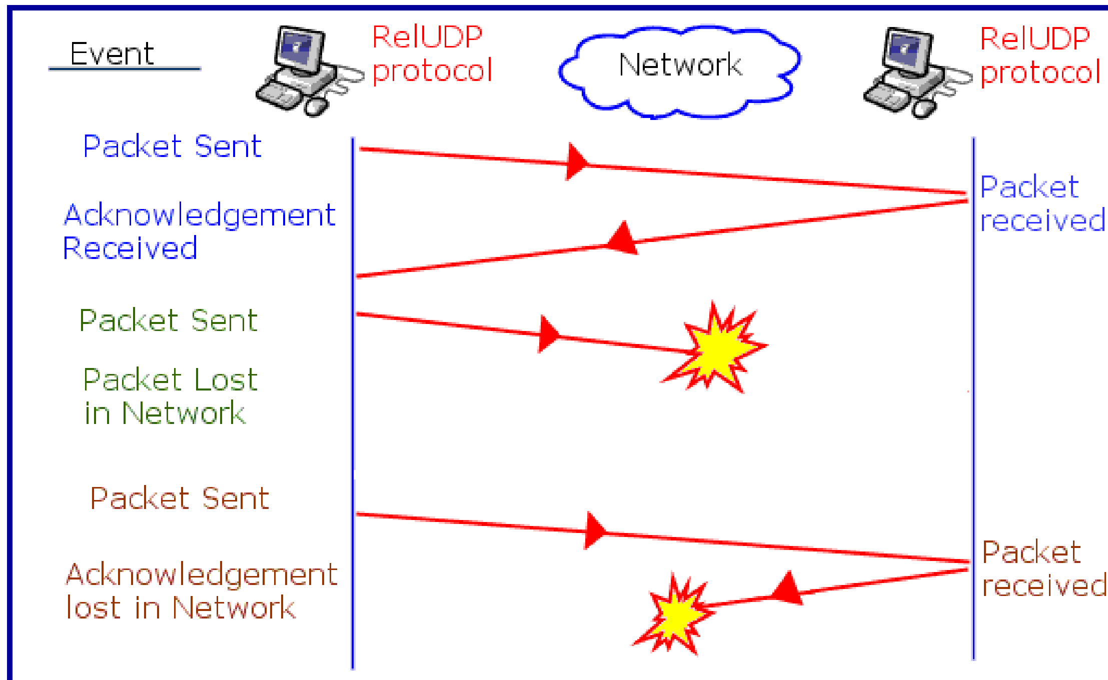

Figure 8 RelUDP protocol performance, illustrating when a Packet gets sucessfully Acknowledged, when the Packet gets lost, and when the Acknowledgement Packet gets lost.

3.2.2 Retransmission

In order for this protocol to be reliable, it must retransmit packets that have not been acknowledged successfully.

3.2.2.1 Deciding which packets to retransmit

Basically as well as packets that have not been acknowledged successfully there is another circumstance that would require retransmission of a packet. Given the nature of the protocol, if the packets were delayed an excessive amount of time on the network before being received it could have negative effect on the applications/programs using the protocol. For instance, in a videoconference it is very important to receive the information as fast as possible, so that the video is visually comprehensible and audible. If some information for this conference was to arrive late it would have adverse effects on the conference, causing what is known as a jitter (late information basically disrupting the

performance of some application). Therefore it is also vital to retransmit packets that have been delayed excessively.

The next issue is how to determine whether a packet is late or not. What variable of time would determine this? A static variable would only work under a certain situations, for example the delay from sending a packet from Ireland to the USA will be significantly larger than the delay sending a packet across the college campus. Having an adaptive variable which adapts to the networks current delay is a possibility, but seemed inadequate as networks can come congested quite quickly, leaving the protocol with a variable time which is too low, and thus resending quite a large amount of packets. In this protocol a fixed maximum delay time was defined (the maximum RTT), which would represent the requirements of the application using the protocol. This simply means that the application would only accept packets if they had a delay time less than half (RTT/2) this fixed variable.

A further possibility would be to have a random element choosing the maximum delay time, between two values. This would introduce a real life random delay on the packets. When groups of delayed packets are resent together (due to a node failure etc), they tend to congest the network even more so. Therefore the random element would deal with this problem.

Thus concluding that the new protocol should retransmit packets if

- The packet got dropped/lost in the network
- The packet got delayed excessively in the network

3.2.3 Use Datagram to transport data

One very important requirement of this protocol is fast delivery of packets. This might seem like an uncontrollable requirement at first

but with greater inspection there are various techniques to produce this. When the functionality and performance of the two most used protocols are studied (UDP and TCP), the desired functionality of our protocol becomes clearer. It has already been noted that the reliability of TCP was desired, but unfortunately TCP has large overhead in dealing with data (sending and receiving), it requires a connection which takes time to setup which is undesirable in the new protocol. UDP on the other hand uses small packets "datagrams" that would be ideal for the new protocol. They are small, therefore they can be sent faster. Datagrams also are mostly independent of each other, so can be treated individually.

3.2.3.1 Header size

If the application that is using our reliable protocol deals with messages of large sizes, then the amount of time that it will take for the application to retransmit the message will be large and in that case it is very likely that loss is due to congestion. The retransmission of large packets into a congested network will in fact make the situation worse by congesting the network even more so. Consequently it is essential that the packet size used in our protocol is as small as possible. Comparing header sizes in UDP and TCP illustrates that UDP's header size is significantly smaller than TCP's header size.

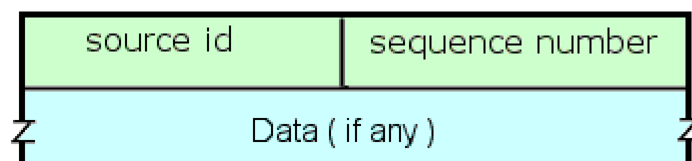

Figure 9 RelUDP Packet Structure

The proposed header of the new protocol is shown in Figure 9. The source id is needed so that protocols will know where the packet is coming from. The sequence number is the identifier to tell packets apart from each other. The datagram packet size will be small, similar to the packet used by UDP.

3.2.4 Connectionless and Stateless

As already mentioned in Chapter 2 one of the main differences between TCP and UDP is whether a connection needs to be created at the start of a connection, and also whether a State has to be maintained. Both of these features are not required in the new protocol.

3.2.4.1 No Connection Setup needed

Before it starts to transfer data TCP uses a three-way handshake to set up a connection. UDP on the other hand just sends data without any formal preliminaries. Thus UDP is free from the delay required to establish a connection. If a connection was needed to be created before the transfer of data, the resulting delay would be unsuitable for applications where speed is a must. Therefore the new protocol will be connectionless

3.2.4.2 No Connection State needed

Some protocols like TCP maintain a connection state in the end systems. This connection state involves keeping track of the logical state of the protocol, for example whether the protocol is connected, closed, waiting for acknowledgement etc. UDP does not maintain connection state. Applications can process quite a lot more packets/ information if it does not have to process extra data associated with the state. Therefore the new protocol will be designed to be Stateless similar to UDP.

3.2.5 Absent of HOL blocking

As already mentioned the phenomenon known as Head of the Line (HOL) blocking occurs when sending independent messages over an order preserving TCP connection. This in turn causes the delivery of messages sent later to be delayed within a receiver's transport layer buffers until an earlier lost message are retransmitted and arrive.

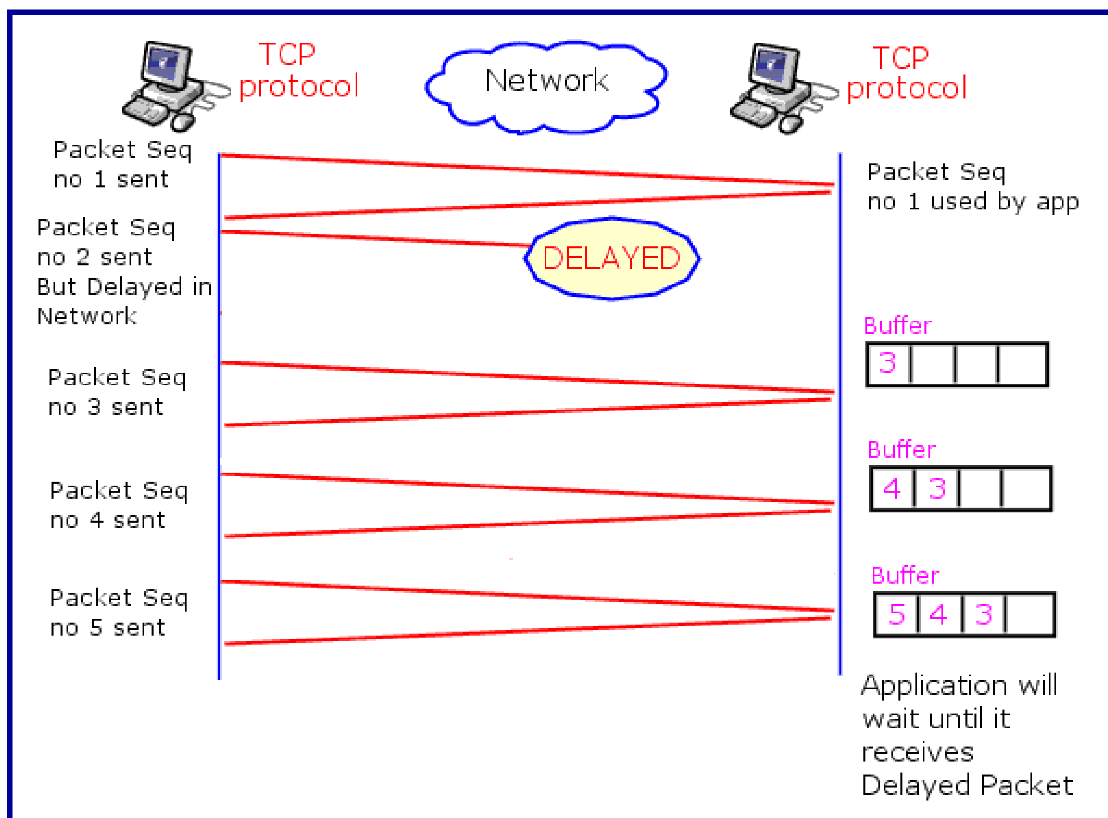

Figure 10 Illustrating HOL Blocking

Figure 10 shows how the phenomenon occurs. This diagram shows that Packet 2 was delayed in the Network, therefore the Packets 3,4 and 5 will stay in the Transport buffer until the delayed Packet 2 arrives. The application will not receive Packet 3,4 or 5 until the

delayed packet arrives, which will result in the applications service being effected.
This can have undesirable effects on the connection and is definitely undesirable with the new protocol as a will be aimed for VoIP signalling or possibly multimedia applications. Therefore when designing the new protocol, order preserving will not implemented. If packets get slightly delayed, and arrive out of order the receiver accepts the packets and will not wait until the packets are in order.

3.2.6 Minimal Congestion Control

Congestion is basically when there are very high amounts of traffic on the network, which in turn effects the performance of applications running on the network by creating a greater delay and a larger amount of packets getting dropped or lost throughout the network. One of the main reasons why the receiver might have lost the message from the sender and the reason for retransmission is because the network is congested.
In some situations congestion management and control is imperative in order to allow the network to recover from congestion and operate in a state of low delay and high throughput (number of packets being sent). In situations like this TCP would be an ideal protocol to use as it uses various congestion control mechanisms. The basic TCP congestion control (Tahoe) causes TCP to reduce its sending rate when congestion is encountered along the network path as evidenced by dropped packets.
On the other hand there are situations where congestion control would not suit the applications running on the network and prove to be counterproductive. For some applications, retransmitted data may be outdated by the time it is received so that retransmissions would not be effective. Furthermore, the majority of congestion control mechanisms reduce the sending rate by a large amount in

response to a single lost packet. This is a good way to avoid or reduce congestion in the network, but some applications may not be able to cope with such an abrupt rate reduction. Applications like audio (VoIP) and video transmissions. As the new protocol is aimed for VoIP signalling or possibly multimedia applications congestion control would be counterproductive. Thus no congestion control will be implemented in the new protocol.

4 Chapter 4 - Physical Implementation of New Protocol

4.1 Introduction

This chapter will deal with the physical implementation of the protocol. The stages of implementation will be illustrated in each section below, a different section for each extra functionality or requirement added.

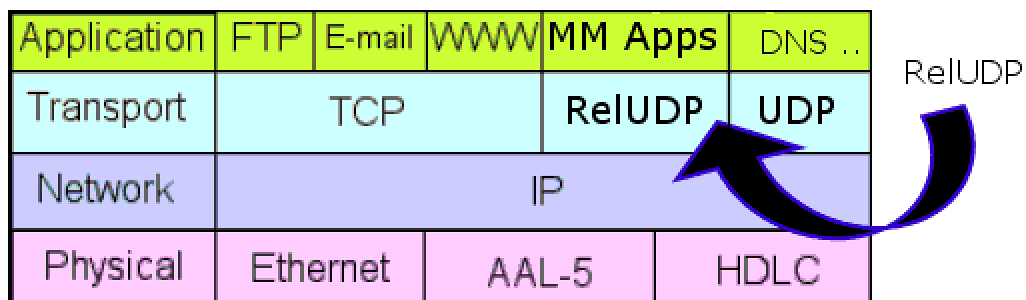

Figure 11 Illustrating where the new protocol will sit in the DoD four layer model

The first question is at which network layer will the protocol sit on? Figure 11 shows an updated diagram of the DoD four layer network model that includes the new protocol. The protocol will be on the same level as TCP and UDP as it too is a transport protocol.

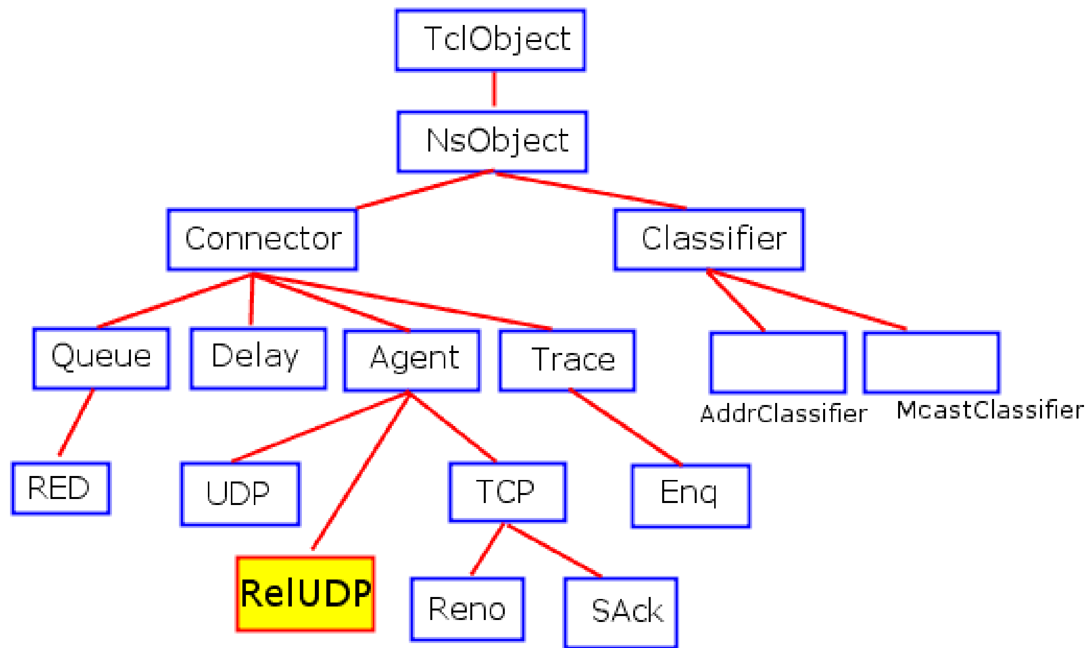

Figure 12 Object Hierarchy of the Network Simulator

Figure 12 shows the object Hierarchy in NS. As the new agent that is being added is in the transport layer it is also on the same level (Transport Layer) as UDP and TCP in the Object hierarchy.

4.1.1 Naming the new protocol

The protocol was called "RelUDP", as it is similar to UDP in the way it uses small packets/Datagrams, but has added functionality such as reliability. Thus RelUDP basically stands for Reliable User Datagram Protocol.

4.2 Fulfilling Requirements and Functionality

The requirements for the new RelUDP protocol were discussed in the last chapter. Here is a list reviewing these requirements

- Reliability
- Retransmission

- Use Datagram to transport data
- HOL blocking absent
- Minimal Congestion Control

These requirements were not implemented all at once. The process of adding functionality to the protocol was a long and tedious one. Each function or requirement was first designed, then implemented and finally tested and possibly redesigned if necessary. Therefore the whole Development can be seen as a series of stages.

4.2.1 Implementing a Basic Datagram protocol

The first goal or stage of implementation was to get a basic Datagram protocol coded and working, similar to UDP. The protocol was implemented as a two-way agent. A two-way agent is symmetric in the sense that it represents both a sender and receiver.
The first thing done was to create a packet header. This is basically a structure in C++ that represents the packet object that the protocol uses.

4.2.1.1 Packets and Packet Headers

The fundamental unit of exchange between protocols in simulations is an object called a Packet. This packet object that is written in C++ provides enough information to link a packet on to a queue, figure out where the packet is being sent, where it is coming from etc. When designing a new protocol it is good practice to define a new packet header or maybe extend existing headers with additional fields to suit the protocols needs.
New packet headers are created by

- Defining a new C++ structure with the needed fields.

- Defining a static class to provide OTcl linkage.
- Modifying the simulator initialisation code to assign a byte offset that points to where the new header is to be located in the packet relative to others.

The next task is to design a new packet header that will be used with the new protocol "RelUDP".

```
struct  hdr_RelUDP  {
u_int32_t srcid_;
int seqno_;

/* per-field member functions */
u_int32_t& srcid() { return (srcid_); }
int& seqno() { return (seqno_); }

/* Packet header access functions */
static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_RelUDP* access(const Packet* p) {
return (hdr_RelUDP*) p->access(offset_);}
};
```

Figure 13 Packet Header of the RelUDP Protocol

Figure 13 shows the Packet header designed for the new protocol. The RelUDP header will require a source identifier field and a sequence number field. This structure shown, hdr_RelUDP defines the basic layout of the RelUDP packet header. It defines the fields are needed and how big they are. The compiler uses this structure to compute byte offsets of the fields. One important thing to note is that there is no objects of this structure type are ever directly allocated. The member functions shown provide a layer of

abstraction that is used by other network objects wishing to read or modify the header fields of packets. Another important thing to note regarding the Packet object is that when a packet is created it has every possible packet type (including the packet type's header fields) in the object. Thus the static class variable offset_ is used to locate the byte offset at which the new header is located. Two methods are provided to utilise this variable to access this header in any packet. These methods are offset() and access().
offset() is used by the packet header management class and should seldom be used.
access() is used to access a particular header in a packet; for example hdr_RelUDP::access(p) accesses the packet header of the RelUDP header.

Now that the new header is defined the next step is to create the basic functionality that the protocol needs which will be implemented in the RelUDP protocol file (RelUDP.cc).
The fundamental functionality of a Basic Datagram Protocol would include the ability to send a packet of a certain set size, segmenting if necessary and also to receive the packet on the other end. This basic functionality is implemented in the functions sendmsg() and recv() respectively. Before the implementation of these two methods is described, the TclClass Hierarchy and the oTCL linkage must be explained.

4.2.1.2 TclClass – Mirrored Hierarchy

The base class called TclClass is a pure virtual class. Any classes derived from this base class provide the following two functions:

1. They construct the interpreted class hierarchy to mirror the compiled class hierarchy.
2. They provide methods to instantiate new TclObjects.

The mirrored Object hierarchy has already been discussed in earlier chapters.

Each such derived class is associated with a particular compiled class in the compiled class hierarchy, and can instantiate new objects in the associated class. For the new protocol, a static class derived from the base TclClass is needed to create this mirrored hierarchy.

```
static class RelUDPAgentClass : public TclClass {
public:
        RelUDPAgentClass() : TclClass("Agent/RelUDP") {}
        TclObject* create(int, const char*const*) {
                return (new RelUDPAgent());
        }
} class_RelUDP_agent;
```

Figure 14 Creation of the RelUDPAgentClass

Figure 14 illustrates the creation of this static class "RelUDPAgentClass", and is associated with the class RelUDPAgent. This class will instantiate new objects in the class RelUDPAgent.

Figure 15 shows the compiled class hierarchy for RelUDPAgent is that it derives from Agent that in turn roughly derives from TclObject. Concentrating on Figure 14 again, this shows the static class "RelUDPAgentClass". This class defines the constructor,

```
RelUDPAgentClass() : TclClass("Agent/RelUDP") {}
```

Figure 15 Tcl Class Hierarchy

and one additional method, to create instances of the associated TclObject.

```
        TclObject* create(int, const char*const*) {
                return (new RelUDPAgent());
```

Figure 16 The Tcl Object create() returning Tcl objects in the class RelUDPAgent

When the Simulator is first started, it will execute the RelUDPAgentClass constructor for the static variable class_RelUDP_agent, which in turn sets up the appropriate methods and the interpreted class hierarchy. This class is associated with the class RelUDPAgent, thus it creates new objects in this associated class. Another thing to note is that the RelUDPAgentClass::create() method returns TclObjects in the class RelUDPAgent. Therefore when a user is writing a Tcl script to run a simulation includes the new protocol (new Agent/RelUDP), the method RelUDPAgentClass::create() is invoked.

4.2.1.3 oTcl – Linkage

As already mentioned in a previous chapter, it is possible to link variables between the two languages used (C++ and oTcl). This allows C++ variables to be accessible through Tcl during the simulation, which allows greater flexibility and control of the simulation. This is accomplished in the class's constructor:

```
RelUDPAgent::RelUDPAgent() : Agent(PT_RelUDP), seqno_(-1)
{
        bind("packetSize_", &size_);
}
```

Figure 17 Variable Linkage

As Figure 17 depicts the linking of C++ variable size_ and OTcl instance variables packetSize_, this grants the user the ability to adjust the packet size during simulation.

4.2.1.4 sendmsg() method

This method is the most complicated method of the simple Datagram protocol that is being implemented. The main aim of this method is to send a Datagram to the destination. sendmsg() takes in three parameters.

- int nbytes specifies the number of bytes that is being sent to the destination
- AppData* data allows the user to attach user application data in the packet.
- const char* flags allows the user to include optional string flags that could be used by an application

```
void UdpAgent::sendmsg(int nbytes, AppData* data, const char* flags)
{
	Packet *p;
	int n;
	if (size_)
		n = nbytes / size_;
	else
		printf("Error: RelUDP size = 0\n");

	// If they are sending data, then it must fit within a single packet.
	if (data && nbytes > size_)
	{
		printf("Error: data greater than maximum RelUDP packet size\n");
		return;
	}
	while (n-- > 0) {
		p = allocpkt();
```

```
                hdr_cmn::access(p)->ptype() = PT_RelUDP;
                hdr_cmn::access(p)->size() = size_;
                hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(p);
                relUDPHdr->seqno() = ++seqno_;
                target_->recv(p);
        }
        n = nbytes % size_;
        if (n > 0) {
                p = allocpkt();
                hdr_cmn::access(p)->ptype() = PT_RelUDP;
                hdr_cmn::access(p)->size() = n;
                hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(p);
                relUDPHdr->seqno() = ++seqno_;
                target_->recv(p);
        }
        idle();
}
```

Figure 18 RelUDP sendmsg() method

Figure 18 shows the implementation of this method. Firstly the method performs some simple error checking at the start of the method. The size_ variable is the current size of the packet, i.e. the number of bytes that the packet can hold. This is checked to be greater than zero, or basically if the variable has been set. If this variable has been set, it calculates how many segments if any that the data must be sent in. It then checks if the user has included data that it will fit in the packet. Segmentation is dealt with here if necessary. If the data must be segmented, the while loop sends n packets, n being the number of segments needed. The actual process of sending a packet is quite simple. First a packet p, which has already been declared at the start of the method, is allocated space and created. Then the packet type is selected. This is where we are using the packet header that was created earlier. The size of the packet is set, the sequence number is incremented and finally

the packet is sent. It is the target_->recv(p); which actually simulates the sending of a packet. What it is actually doing is calling the recv method of target_. The target_ variable is a pointer to the destination protocol/node. After any segmentation is needed, the remainder of the data (if any) is sent exactly the same way described above.

4.2.1.5 recv() method

As already mentioned the most complicated method in the Basic Datagram protocol that is being implemented was the sendmsg() method. The recv() method is quite simple. Firstly it takes in two parameters;

- Packet* pkt, the packet that is being received
- Handler* the handler which handles the event

```
void UdpAgent::recv(Packet* pkt, Handler*)
{
        if (app_ )
        {
                // If an application is attached, pass the data to the app
                hdr_cmn* h = hdr_cmn::access(pkt);
                app_->process_data(h->size(), pkt->userdata());
        }
        Packet::free(pkt);
}
```

Figure 19 RelUDP recv() method

Figure 19 shows the implementation of the recv() method. Firstly the method checks whether the user added user application data in the packet, and if so it passes it to the application to process the

data. If on the other hand the user did not add any application data to the packet, the packet is freed from memory and dropped.

4.2.1.6 Summary

The majority of the basic Datagram protocol was explained above. The functionality of the protocol was very basic. It simply sends data in Datagram packet and segments the data if necessary. The next goal is to add reliability to the protocol implementation.

4.2.2 Implementing Reliability

The basic Datagram Protocol described in the last section is a simple but effective one. As it is it is lacking functionality that is needed in the new Protocol. One of the basic functions or services that must be added is reliability. The importance of having a reliable Protocol is described in the last chapter.

Providing data reliability consists of two basic components.

- Loss detection
- Loss Recovery

The oncoming sections deal with these two basic components in more detail.

4.2.2.1 Loss Detection

In the basic Datagram Protocol there is no way of knowing whether a Datagram packet actually got delivered to the destination node. The Protocol basically sends the Datagram Packets and hopes for the best. The first aim is to detect a Packet loss.

In order to do this a new Packet must be created. This Packet is called an Acknowledgement packet. Is basic purpose is to inform the sender that a certain Packet has been delivered successfully. The size of an Acknowledgement is considerably smaller than a

regular Datagram Packet, thus Acknowledgements have little effect on Congestion. The first goal is to design a new Packet Header similar to the Datagram Packet header that was designed earlier.

```
struct hdr_RelUDPAck {
u_int32_t srcid_;
int seqno_;

/* per-field member functions */
u_int32_t& srcid() { return (srcid_); }
int& seqno() { return (seqno_); }

/* Packet header access functions */
static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_RelUDPAck* access(const Packet* p) {
return (hdr_RelUDPAck*) p->access(offset_);}
};
```

Figure 20 RelUDP Acknowledgement Packet Structure

Figure 20 shows the Acknowledgement Packet header designed for the new protocol. This header will be very similar to the RelUDP header that was designed earlier. It also will require a source identifier field and a sequence number field. This sequence number field will be used to match a Packet sent to an Acknowledgement Packet received.

As the Acknowledgement Packet Header is now designed the next goal is to modify the revc() method in the Basic Datagram Protocol so that when it receives a packet, it sends an Acknowledgement Packet back to the original sender, thus acknowledging the Packet. The modifications to the recv() method must enable the protocol to receive both regular RelUDP Packets and RelUDPAck Packets and to differentiate between them.

```
void RelUDPAgent::recv(Packet* pkt, Handler *h)
{
        hdr_cmn* cho = hdr_cmn::access(pkt);
        if((cho->ptype() == PT_RelUDP)) // if it is a normal RelUDP packet
        {
        //We want to create a new RelUDPAck packet and
        //send it acknowledging the packet
                Packet *np = allocpkt();
        //access the common header of the new packet
                hdr_cmn* nch = hdr_cmn::access(np);
        //sets the size of the new packet to 12 as it is an Ack
                nch->size()=12;

        //access the IP header fields of the packet being received and
        //the new packet being created
                hdr_ip * iph = hdr_ip::access(pkt);
                hdr_ip * newiph = hdr_ip::access(np);

        //switch the Destination and source of the packet that was
        //received and set these values to the new packet
                newiph->dst()=iph->src();
                newiph->flowid()=iph->flowid();
                newiph->prio()=iph->prio();

                hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(pkt);
                nch->ptype() = PT_RelUDPAck;

                hdr_RelUDPAck* relUDPAckHdr = hdr_RelUDPAck::access(np);
                relUDPAckHdr->srcid()=addr();
                relUDPAckHdr->seqno()=relUDPHdr->seqno();

                target_->recv(np);
                Packet::free(pkt);
        }...
```

Figure 21 Revised RelUDP recv() method

Figure 21 shows the modifications needed to the recv() method so that it can handle basic RelUDP Packets. Basically what is needed in this recv() method is the ability to reply by sending an Acknowledgement Packet, when a RelUDP Datagram Packet is received. The first thing that is done in this method is to check the type of the incoming Packet. If the packet type is RelUDP, then the method must send an Acknowledgement back to the node that sent the RelUDP Packet. This is done as follows. First a new Packet is created (np) and its size is set to 12 bytes. Note that this is considerably smaller than a regular Datagram Packet size. The IP header fields are then accessed to figure out where the Packet has come from. Then the new Packets IP header destination address is set to the incoming packets source address, thus switching the direction of the Packet. The packet type is set to RelUDPAck as it is an Acknowledgement and finally the sequence number of the incoming packet is copied to the sequence number of the new Acknowledgement Packet. The acknowledgement is sent.

At the moment the protocol will send a Datagram, and on receiving a Datagram will send an acknowledgement back to the sender. The issue of Packet loss detection is quite not fully implemented. The next step is to implement a mechanism that records the packets sent in a buffer. Thus this buffer could be used with the incoming Acknowledgements to figure out which Packets indeed were lost in the network and require retransmission.

```
void RelUDPAgent::recordPacket(Packet* pkt)
{
        packetsSent[packetIndex_]=pkt->copy();
        packetIndex_++;
}
```

Figure 22 RelUDP recordPacket() method

Figure 22 shows the implementation of the recordPacket() method. The basic aim of this method is to record the Packet pkt in the buffer. The buffer is actually a packet array called packetsSent. One thing that must be explained is the packetIndex_ variable. This variable keeps count of the amount of Packets in the buffer. It is a public class variable, thus each instance of the protocol that is created contains this variable. The copy() method is used to copy the contents of the packet into a Packet array (buffer). This method is called when a RelUDP packet is being sent. Thus all Datagram packets sent are recorded in the buffer. It is with the help of this method that the protocol can detect packet loss.

As all packets being sent are recorded in the buffer, one possibility of detecting Packet loss would be to remove the corresponding (same sequence number) Packet when an Acknowledgement arrives. If the protocol was implemented in this manner, it is deducible the Packets left in the buffer are the Packets that were lost or the packets who's Acknowledgements were lost.

In order to implement loss detection a number of methods have to be added to the protocol. The basic theory is to remove a Packet from the buffer if the Packet has been acknowledged successfully. This theory seems viable at first but on closer inspection it lacks efficiency. Removing a Packet from the buffer would require copying every other Packet in the old array into a new array, which would be computationally expensive. A slightly different approach was taken which produces the same outcome without the efficiency problem. Instead of removing the acknowledged packet altogether, the packet is marked for deletion. Then when the buffer fills up, a method is called which basically removes the Packets that are

market for removal, and copies the remainder into a new buffer. Two implementation issues must be discussed here.

Firstly the Packets are marked for deletion by setting the sequence number of the Packet to –2. Thus any Packets in the buffer that do not have a sequence number of –2 must be retransmitted.

```
else if(cho->ptype() == PT_RelUDPAck)
{
  hdr_RelUDPAck* relUDPAckHdr = hdr_RelUDPAck::access(pkt);
  int sequenceNumber =relUDPAckHdr->seqno();
  int pIndex=-1;

  for(int index=0; index<copy ;index++ )
  {
  hdr_RelUDP* buffer_relUDPHdr= hdr_RelUDP::access(packetsSent[index]);
       if(buffer_relUDPHdr->seqno()==sequenceNumber)
       {
              pIndex=index;
       }
  }
  if(pIndex!=-1)
  {
  hdr_RelUDP* old_RelUDPHdr = hdr_RelUDP::access(packetsSent[pIndex]);
  old_RelUDPHdr->seqno()=-2;
  }
  Packet::free(pkt);
}
```

Figure 23 Revised RelUDP recv() method

Figure 23 shows the rest of the recv method, which deals with receiving acknowledgements. The first loop simply searches through the buffer for a matching Packet with the same sequence number. Then if a Packet is found it is marked for deletion by simply setting its sequence number to –2, as explained previously.

Secondly the implementation of the method that removes the Acknowledged Packets must be explained.

```
void RelUDPAgent::removeAckdPackets(int newBufferSize)
{
   Packet *tempPacketarray = new Packet[newBufferSize];

   int numberPacketsAdded=0;
   for(int index=0; index<packetIndex_;index++ )
   {
     hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(packetsSent[index]);
     if(relUDPHdr->seqno()!= -2)
     {
        tempPacketarray[numberPacketsAdded]=*packetsSent[index];
        numberPacketsAdded++;
     }
    }

    for(int index=0; index<numberPacketsAdded ;index++ )
    {
        *packetsSent[index]=tempPacketarray[index];
    }

    delete[] tempPacketarray;
    packetIndex_=numberPacketsAdded;
    BufferSize=newBufferSize;
}
```

Figure 24 RelUDP removeAckdPackets() method

Figure 24 shows the implementation of the removeAckdPackets() method. This method takes in a parameter newBufferSize, which is simply the size of the new buffer that is going to be produced by this method. This new buffer will contain all the packets in the previous buffer that weren't marked for removal/deletion. The method first creates a temporary Packet array of newBufferSize

size, then it simply searches through the original buffer for any Packets who's sequence number is not –2, or in other words who have not been marked for removal. If it does find such Packets, it copies them into the temporary array. It then copies the Packets from the temporary array into the new buffer array, and deletes the temporary array. This method is quite efficient, as it gives the opportunity of resizing the buffer array only when necessary, thus keeping memory consumption at a minimum.

Therefore the protocol now is capable of detecting Packet loss. The next issue to deal with is Loss Recovery, which basically will implement functionality to recover form the lost Packet, or retransmit the Packet in other words.

4.2.2.2 Loss Recovery

As the protocol can now detect lost Packets a method is needed to retransmit these Packets.

```
void RelUDPAgent::checkBufferLoop()
{
  for(int index=0; index<packetIndex_ -4;index++ )
  {
        hdr_RelUDP* origRelUDPHdr = hdr_RelUDP::access(packetsSent[index]);
        //if the packet is not marked for deletion already
        if(origRelUDPHdr->seqno()!= -2)
        {
        Packet *p;
        p = allocpkt();
        p=packetsSent[index]->copy();
        hdr_RelUDP* newRelUDPHdr = hdr_RelUDP::access(p);
        recordPacket(p);
        origRelUDPHdr->seqno()=-2;
        Scheduler::instance().schedule(target_, p, 0);
        }
  }
}
```

Figure 25 RelUDP checkBufferLoop()

Figure 25 shows the implementation of the method checkBufferLoop() which is used to retransmit lost Packets. The method simply searches through the buffer for any packets that are not marked for removal, and resends the packet. When it resends the packet it sets the original packet in the buffer for removal, and records a new packet in the buffer. There are two important things to note about this implementation. The method does not search through the last four elements in the array, as they are most likely the packets that are currently being sent (an improved version will be discussed later). Secondly the main issue is from where to call this method. It should be called regularly to ensure that lost packets would be retransmitted quickly. The reliability of the

protocol is effected depending on from where this method is called. If it is called form the

- sendmsg() method then there is a possibility that a Packet could get lost after this method was called for the last time in the simulation. For example is a Packet was sent, and the checkBufferLoop() method resends any Packets needed at that time but if the Packet gets lost after that it will never be resent. This would make the protocol unreliable.
- recv() method a similar problem would arise. If the last packet being sent got lost in the Network, the final recv() method would never be called because no Acknowledgement would have been sent. Thus this Packet would never be resent, which also transforms the protocol to an unreliable one.

Therefore it is clear that a new mechanism is needed so that this method can be called regularly to ensure reliability. Luckily the Network Simulator provides a TimeHandler class which solves this problem.

In the Network Simulator Timers may be implemented in C++ or OTcl. The new protocol will need a timer implemented in C++, which is based on an abstract base class (defined in timer-handler.h). The TimerHandler gives the ability to create, schedule, reschedule or cancel a timer. When a timer is scheduled (by using the sched(delay) method) to expire delay seconds in the future. When the timer expires an event is created, which can be handled. Then handling of this event can call the checkBufferLoop() method. This will ensure that the method is called regularly, thus making the protocol reliable.

```
class CheckBufferTimer : public TimerHandler {
public:
        CheckBufferTimer(RelUDPAgent* t) : TimerHandler(), t_(t) {}
        inline virtual void        expire(Event*);
protected:
    RelUDPAgent* t_;
};
```

Figure 26 Defination of the CheckBufferTimer() method

Figure 26 shows the definition of the timer used in the new protocol. The method void CheckBufferTimer::expire(Event*) will be called when the timer expires. The timer must be rescheduled at the end of the checkBufferLoop() method by the code timername.resched(next_time).
At this point the protocol can be stated to be reliable. The protocol sends Datagram packets, replies with Acknowledgement packets, records the packets it sends, and resends the packets that do not get acknowledged. There is still the issue of retransmission of delayed packets. This is discussed in the next section.

4.2.2.3 Enforcing a maximum Time Delay

The importance of Packets arriving below a certain delay is discussed in detail in the last chapter. The basic reason is that if the packets are arriving too late, the information might be out of date, and consequently disrupt the performance of multimedia applications. The new protocol is designed with aim of using it in multimedia applications and related areas, therefore it is important to enforce this in the new protocol. The next goal is to implement added functionality to the existing protocol that would retransmit packets if their RTT (Round Trip Time) is greater than a fixed variable or if they have been in the buffer longer than a fixed

variable. To recap the Round Trip Time is the amount of time taken between the sending of a Packet and the receiving of an Acknowledgement for that Packet. A number of adjustments need to be made to the protocol to implement this.
Firstly the buffer must be extended.

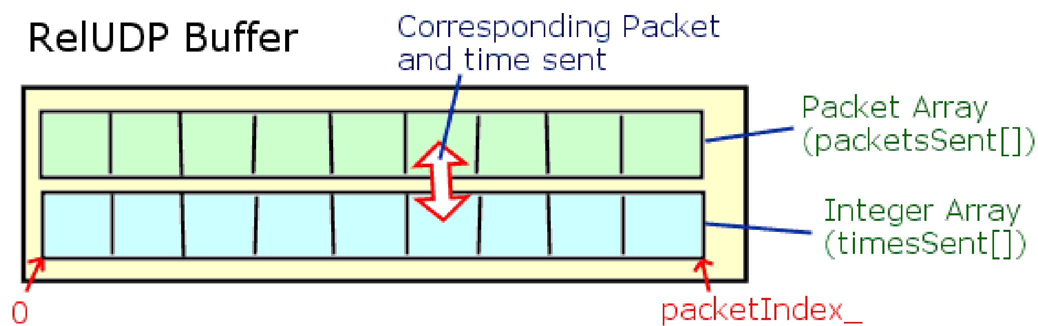

Figure 27 RelUDP Protocol Buffer

As Figure 27 illustrates, the concept of the buffer must hold a packet array that holds the actual packets, and a corresponding integer array that holds the time that the corresponding Packet was sent at.
The next adjustment to the existing protocol is to the methods recordPacket() and removeAckdPackets(). The changes to these methods are as expected and can be seen in the final version of the protocol. The method which requires substantial change would be the checkBufferLoop()

```
void RelUDPAgent::checkBufferLoop()
{
        double sendtime;
        double local_time = Scheduler::instance().clock();
        for(int index=0; index<packetIndex_;index++ )
        {
        hdr_RelUDP* origRelUDPHdr = hdr_RelUDP::access(packetsSent[index]);
        if(origRelUDPHdr->seqno()!= -2)
```

```
        {
                if((local_time - timesSent[index])>=MAXDELAY)
                {
                        sendtime = Scheduler::instance().clock();
                        Packet *p;
                        p = allocpkt();
                        p=packetsSent[index]->copy();
                        hdr_RelUDP* newRelUDPHdr = hdr_RelUDP::access(p);
                        recordPacket(p,sendtime);
                        origRelUDPHdr->seqno()=-2;
                        Scheduler::instance().schedule(target_, p, senddelay);
                }
        }
        }
        double next_time =CHECK_BUFFER_INT;
        if(next_time >0)
        {
                cb_timer.resched(next_time);
        }
}
```

Figure 28 Revised RelUDP checkBufferLoop() method

Figure 28 shows the changes made to the method checkBufferLoop. Note the timer code is also shown. This method now searches through the buffer for packets that have not been marked for removal and that have not been in the buffer greater than a variable (MAXDELAY) amount of time. If it finds any packets that satisfy these requirements, it resends those packets.

4.2.2.4 Duplicate Acknowledgements

This is a problem that can occur in the Protocol design as it is currently designed. The basic problem is that at any time it is

possible to have duplicate acknowledgement Packets (acknowledging the same Packet) in the Network.

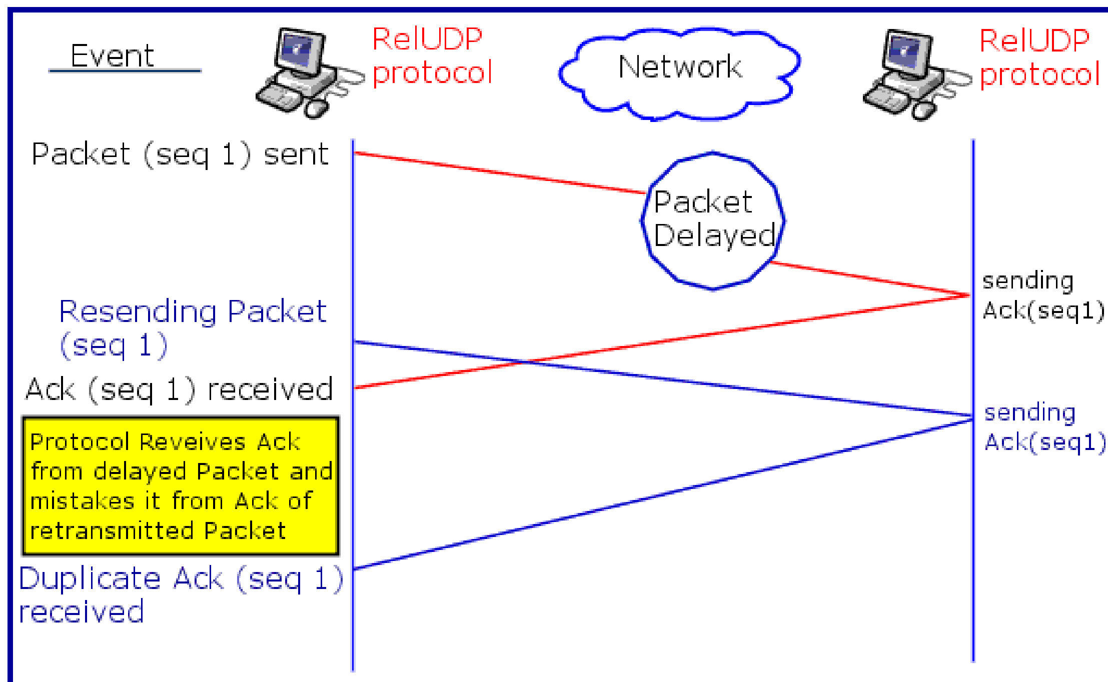

Figure 29 Duplicate Acknowledgement problem

This happens as follows, (see Figure 29) first when a Packet (sequence number 1) is sent by the Protocol and is successfully delivered at its destination, an acknowledgement Packet (sequence number 1) is sent in response. If the network is congested and the acknowledgement has not returned in the MAXDELAY amount of time, the protocol resends Packet 1 as it believes the Packet was lost or delayed in the network. Shortly afterwards the delayed acknowledgement arrives, consequently the protocol believes this to be the acknowledgement of the second retransmitted Packet. Therefore the Protocol will mark a delayed Packet off the buffer that should have been retransmitted.

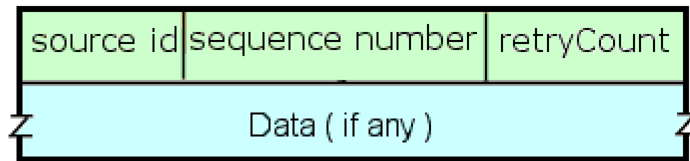

Figure 30 Revised RelUDP Packet Structure

To overcome this problem, a secondary sequence number is needed in both the RelUDP and the RelUDPAck Packet headers. This will make it possible for the Protocol to differentiate between different retransmissions of the same packet. Figure 30 shows the revised RelUDP/RelUDPAck Packet header.

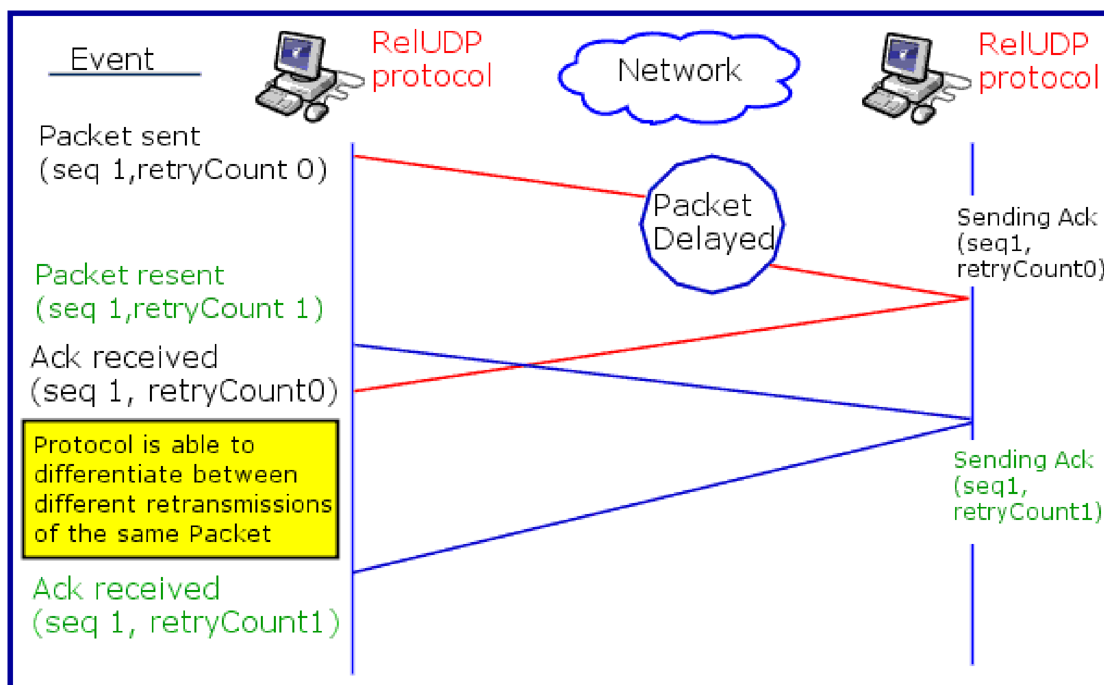

Figure 31 Overcoming Duplicate Acknowledgements with revised RelUDP header.

Figure 31 shows how the new header field overcomes this problem. The protocol is able to differentiate between different retransmissions of the same packet.

```
struct  hdr_RelUDP  {
u_int32_t srcid_;
int seqno_;
int retryCount_;

/* per-field member functions */
u_int32_t& srcid() { return (srcid_); }
int& seqno() { return (seqno_); }
int& retryCount() { return (retryCount_); }

/* Packet header access functions */
...
```

Figure 32 Modifications made to RelUDP header

Figure 32 illustrates the modifications made to the header field of the RelUDP packet header. Note the same changes are needed in the RelUDPAck header. The secondary sequence number variable is called retryCount_. A member function is also implemented to access this variable.

Slight modifications must be made to the Protocol to implement this new secondary sequence number scheme. When a Packet is retransmitted, the new recorded Packet's retryCount_ Header field is incremented. Also when an acknowledgement arrives it must satisfy all three conditions to remove the corresponding Packet form the buffer/to prove that the Packet has arrived on time.

1. The acknowledgements sequence number must be the same as the corresponding Packets sequence number in the buffer.
2. The acknowledgement's retryCount_ value must be the same as the corresponding Packet's retryCount_ value in the buffer.
3. The Packets RTT must be below the MAXDELAY variable time.

4.2.2.5 Retransmission Design issues

When implementing the extra functionality on to the Protocol, one final design issue remains. This issue arises with the option of Packet retransmission. Two different designs were created which retransmit Packets at different times and through different methods.

The first design will be discussed here, and the second design is the design chosen for the final end version of the Protocol. The fundamental difference between the two designs is basically when to retransmit lost or delayed packets.

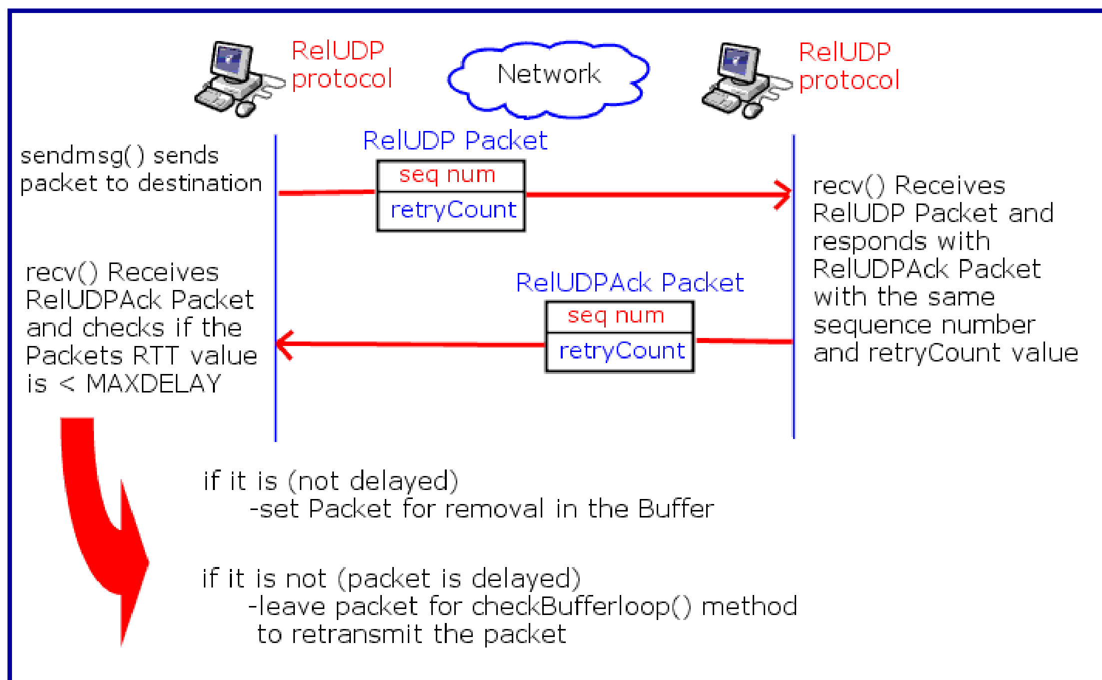

Figure 33 Functionality of the RelUDP Protocol Version 1

The first version employs the functionality shown in figure 33. First when a packet is sent by the protocol that arrives successfully at the destination an acknowledgement is sent in return. The main difference is what the protocol does when an acknowledgement is received. It first checks through the buffer for a Packet matching

the sequence number and the retryCount value. If it finds such a Packet, it checks the Packets Round Trip Time to be less than the MAXDELAY static variable.

- If it less than the MAXDELAY value is it sets that packet for removal in the buffer
- If it is not less than the MAXDELAY value it simply leaves it up to the checkBufferLoop() method to retransmit the packet. This method is called continuously by the timer ever 25miliseconds.

The checkBufferLoop() method checks through the buffer for Packets that have not been marked for removal and that have been in the buffer greater than MAXDELAY amount of time. Finding any Packets that satisfy these conditions, the method will send a copy of the Packet, record it in the buffer and finally set the original packet for removal.

The main reasons against this version were both logical and physical ones. Firstly the new Protocol should retransmit lost/delayed Packets as quickly as possible, to avoid the performance of the application using the protocol to be hindered. Therefore when the acknowledgement Packet is received, it should retransmit the Packet then rather than leaving it for the checkBufferLoop method to retransmit the Packet later. Another point is that this protocol will retransmit Packets in groups which is generally undesirable as it could possibly worsen congestion in the Network.

The physical issues concern the Network simulator itself. When testing this version of the protocol under highly congested networks, it was found that as the checkBufferLoop method is running at the same time as the protocol. This was seen to produce undesirable results or segmentation faults as the two processes were accessing the same information, for example Packets in the buffer.

The final version of the Protocol deals with these issues and is discussed later in this chapter.

4.2.3 Hol blocking

The protocol provides reliability, but does not enforce in order delivery. By avoiding this functionality, HOL blocking is avoided.

4.2.4 No congestion control

As the new protocol is aimed for VoIP signalling or possibly multimedia applications congestion control would be counterproductive. Thus no congestion control was be implemented in the new protocol. Thus if the network does get highly congested, the protocol does not adjust in any way to deal with the congestion; it simply keeps sending Datagram at the same rate.

4.3 Final Version

The protocol was designed from a series of stages, each stage adding new functionality bring the Protocol closer to the final result. As quite a lot of the details have already been discussed in previous stages, this section will give a brief overview of the workings of the Final version of the Protocol explaining any changes as they appear.

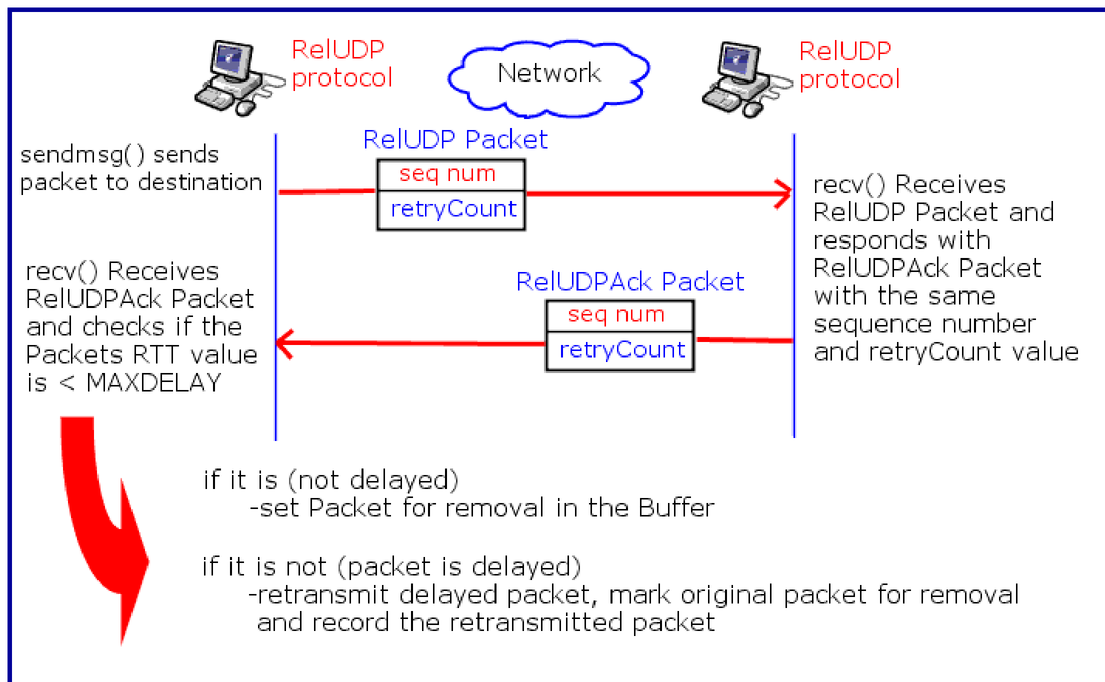

Figure 34 Functionality of the RelUDP Protocol Final Version

Figure 34 shows the functionality and the workings of the Protocol. There are two basic improvements that the Final version of the Protocol makes.

```
If ( delay >= MAXDELAY )
{
   double local_time = Scheduler::instance().clock();
    hdr_RelUDP* old_RelUDPHdr =   hdr_RelUDP::access(packetsSent[pIndex]);
    Packet *newpkt;
    newpkt = allocpkt();
    newpkt=packetsSent[pIndex]->copy();
    hdr_RelUDP* newRelUDPHdr = hdr_RelUDP::access(newpkt);
    newRelUDPHdr->retryCount()=(relUDPAckHdr->retryCount()) +1;
    target_->recv(newpkt);
    recordPacket(newpkt,local_time);
    old_RelUDPHdr->seqno()=-3;
}
else
{
    hdr_RelUDP* old_RelUDPHdr = hdr_RelUDP::access(packetsSent[pIndex]);
    old_RelUDPHdr->seqno()=-2;
}
checkBufferLoop();
...
```

Figure 35 Additions to recv() method

The first is what the Protocol does when an acknowledgement is received. Unlike the previous version of the Protocol this version retransmits the Packet as soon as it figures out that it is delayed. Figure 35 illustrates this point. It shows a fragment of the code from the recv() method.

The second improvement is that the timer does not call checkBufferLoop every 25miliseconds. Instead it is called in the receive method, which eliminates the problem of segmentation faults. Every time the method is called it reschedules itself to be called 600 milliseconds later. This in fact only calls the method

when the Protocol has finished sending Packets. Thus providing guaranteed reliability, even to the last packet sent.

4.4 Potential Additions

4.4.1 Including a Random Element

One possible addition that could be added to the Protocol would be to change the static variable MAXDELAY to a random variable between 450 and 500 milliseconds. When a node (router) or link fails, quite a large amount of packets are lost or delayed. This in turn causes these packets to be resent at the same time, which congests the network even more so. Adding this random element will lessen the effect of this problem.

4.4.2 Including Fragmentation

The protocol as it is does not implement fragmentation. Fragmentation is simply a process to overcome the problem of heterogeneous Maximum Transmission Units. Nodes on the Network store the Packets/Datagrams to be sent in memory; when sending the Datagram it is put in an IP frame suitable for the network. Each hardware technology specifies the maximum amount of data that a frame can carry. For example some links on the Network will only be able to transfer a certain amount of data; this is called the Maximum Transmission Unit (MTU).

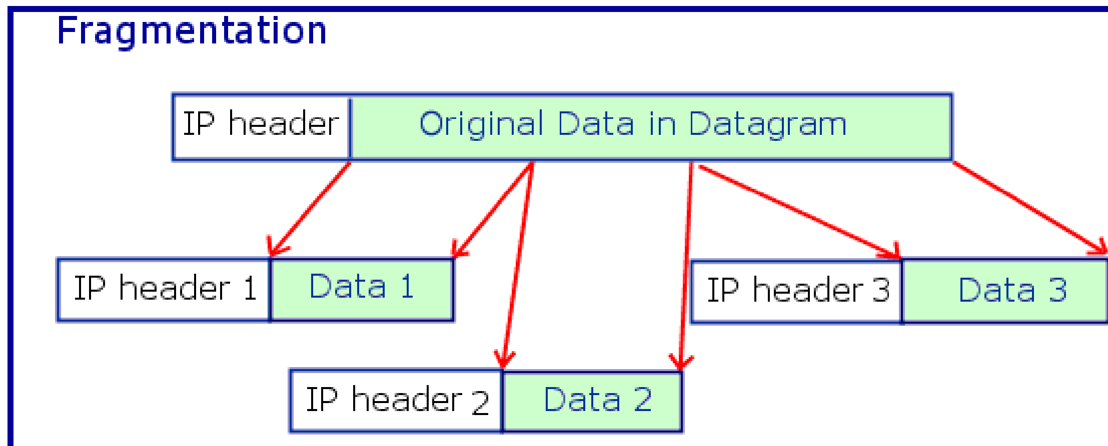

Figure 36 Fragmentation

The problem occurs when a Datagram that is larger than the MTU of the network is sent over the network; the solution is that the Datagram is divided into smaller "fragments" which are each sent separately. This is called Fragmentation. Figure 36 illustrates this.

5 Chapter 5 -Simulation and testing of New Protocol

5.1 Introduction

As the RelUDP Protocol was being developed, it was continuously tested in the Network Simulator. It is possible to insert print statements to make the output more understandable but there is a program that comes with NS which creates a graphical interface that can be used to study the protocols performance.

5.2 Network Animator (NAM)

NAM is a Tcl based animation tool for viewing network simulation traces. It supports topology layout and packet level animation

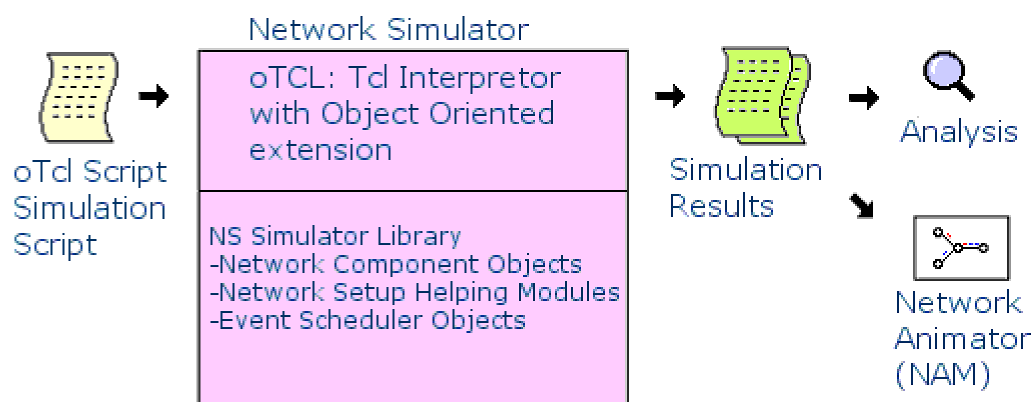

Figure 37 Users view of the Network simulator

When the Network Simulator finishes the simulation it is capable of producing one or more text-based output files that contain detailed simulation data. The Tcl scripts that actually run the simulation determine this. The data can be used for simulation analysis or as an input to a graphical simulation display tool called Network Animator (NAM). NAM has a useful graphical user interface that

allows the user to play, fast forward, rewind or pause the simulation.

5.3 Simulations

It was vital to test the Protocol during development to ensure that the functions work as expected. It is also useful to compare the protocol against other developed protocols to see the results. The various simulations will be described in the oncoming sections.

5.3.1 Simulation 1 - A simple network

The first simulation consists of a simple two-node topology as shown in figure 38. This simulation is simple but illustrates some basic functionality of the RelUDP protocol. It illustrates the protocol sending a packet (figure 39) and replying with an acknowledgement packet (figure 40).

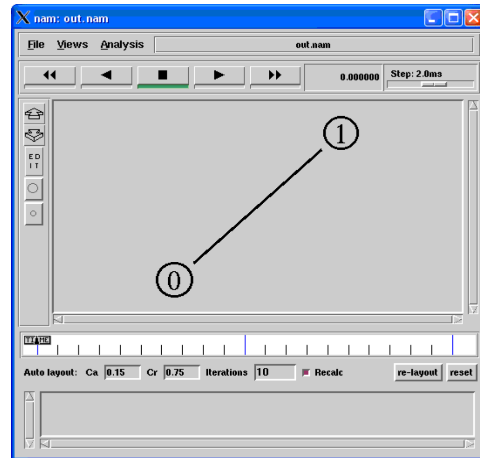

Figure 38 Simulation 1 Topology

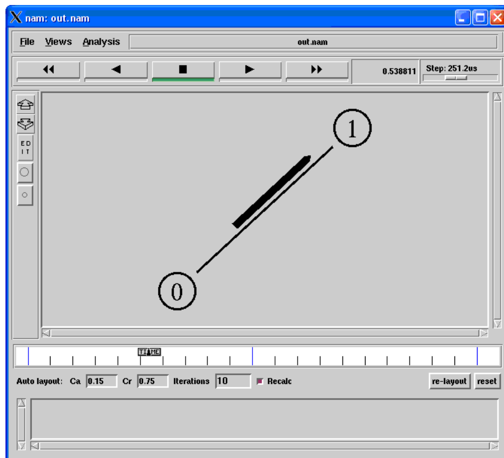

Figure 39 Simulation 1 - Packet Sending

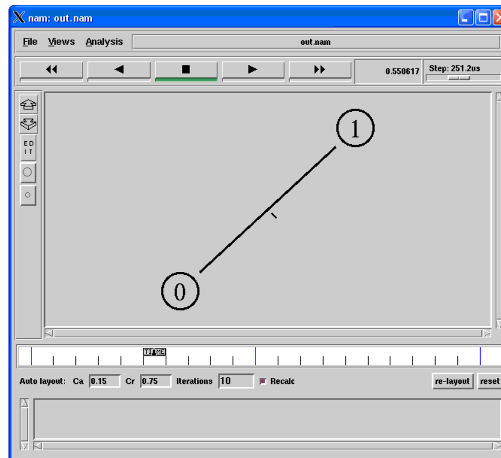

Figure 40 Simulation 1 – Sending Ack

5.3.2 Simulation 2 - A simple network with Packet loss

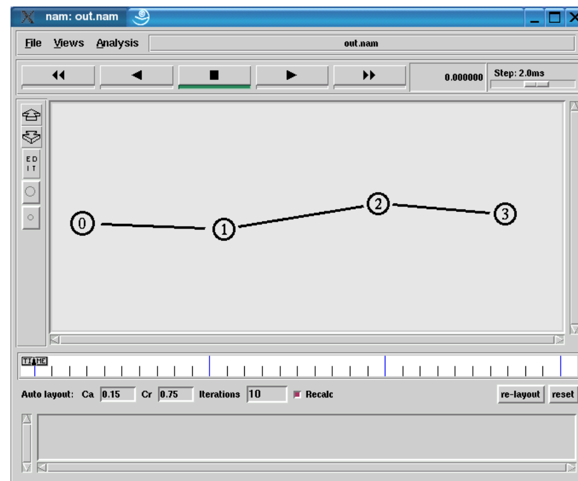

Figure 41 Simulation 2 Network Topology

The second simulation consists of a similar topology used in Simulation 1. This topology shown in figure 41 adds one aspect to simulation 1. This simulation introduces more functionality of the RelUDP protocol. By introducing an error model between the two nodes, random packets are dropped. This will test the protocol dealing with dropped packets in the network. The protocol can be seen to retransmit the dropped packets in this simulation illustrating its reliability. Figure 42 shows a screenshot of the Protocol running this simulation, notice the packet being dropped.

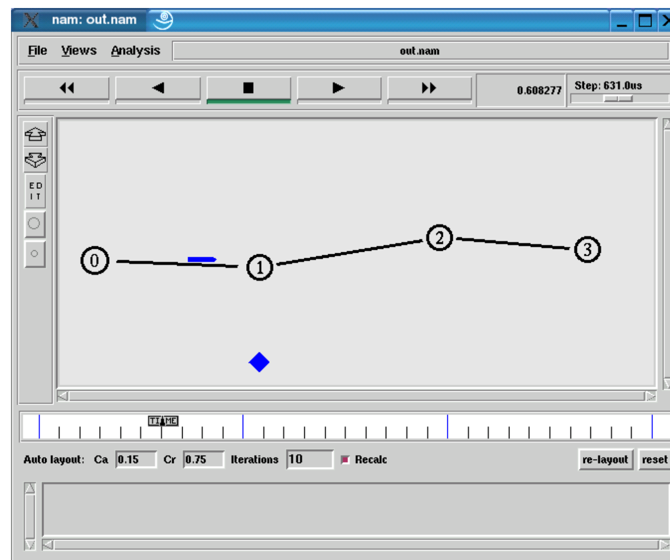

Figure 42 Simulation 2 running

5.3.3 Simulation 3 - A complex network with minimal congestion

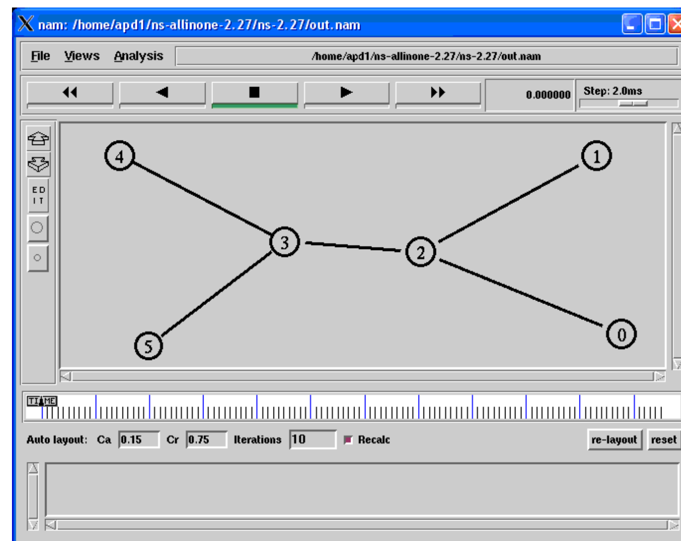

Figure 43 Simulation 3 Network Topology

The third simulation consists of a more complex network. Firstly it consists of four end point nodes, two of which are using the regular UDP protocol, and two of which are using the new Protocol RelUDP. This topology is shown in figure 43.

The RelUDP protocol will first start transmitting data across the bottleneck network. Then the UDP will also start using the same bottleneck link causing congestion. As the congestion builds packets get both delayed and dropped on busy links (This can be seen in figure 44). Notice the queue on the router 2, packets will be both lost and delayed form this point.

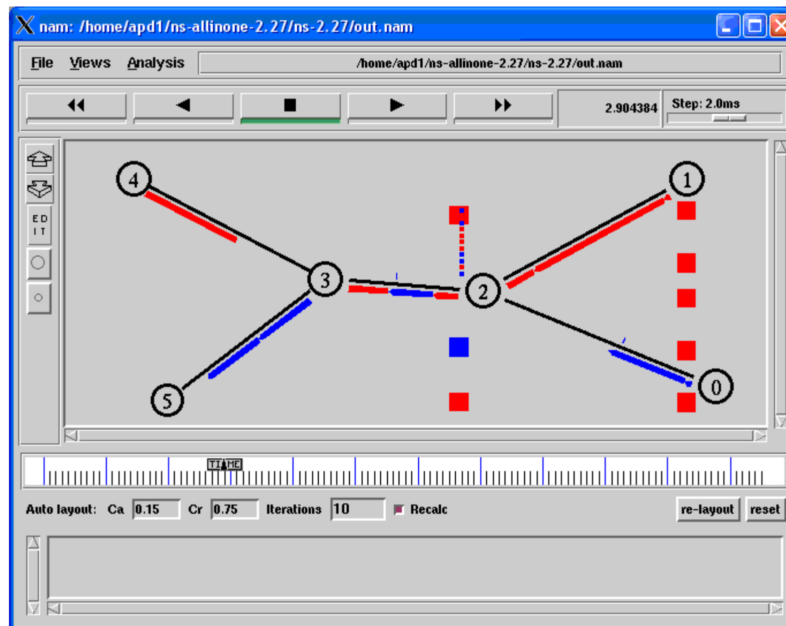

Figure 44 Simulation 3 running (Network is congested)

The UDP protocol then stops transmitting packets, and the new RelUDP protocol can be seen resending the delayed or dropped Packets until the Network resumes as it was when it started.

5.3.4 Simulation 4 - A complex network with greater congestion

The fourth simulation consists of exactly the same network topology as in Simulation 3, but it introduces very high congestion by setting the rate at which the UDP protocol sends packets to be very high. The network becomes highly congested but eventually the RelUDP resends all necessary packets and the Network returns to normal. Figure 45 shows a screenshot of the Protocol running this simulation, at this point the protocol is recovering form the high congestion of the network.

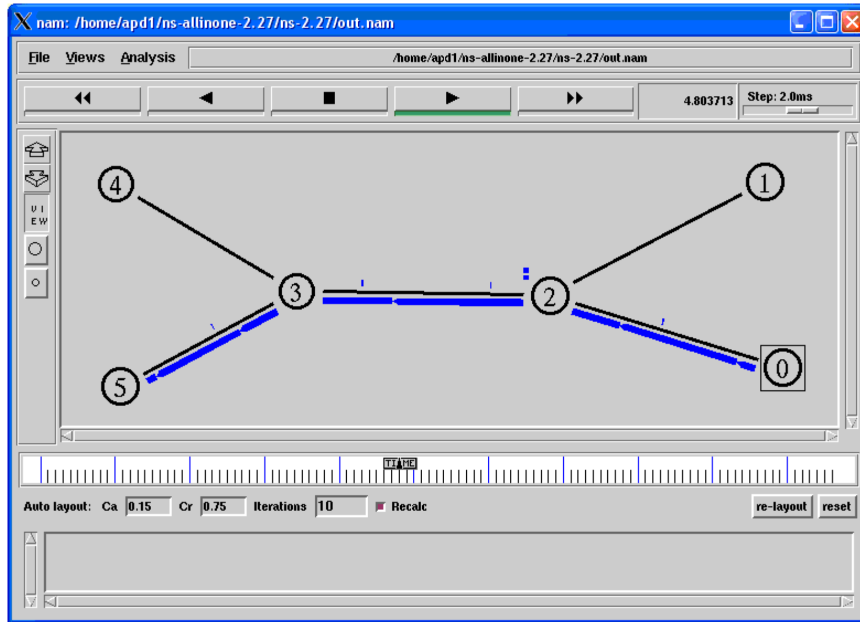

Figure 45 Simulation 4 running (recovering from high congestion)

6 Chapter 6 - Conclusion

The overall conclusion that can be made from completing this project is that the initial goal of the project was achieved. A new Protocol was first designed and then implemented in the Network Simulator.

The Protocol was designed for VoIP signalling or possibly multimedia applications such as video conferencing etc. The basic need for a new Transport protocol is fundamentally because the transport protocols that are commonly used today do not suit the needs of multimedia applications.
Therefore the new RelUDP protocol that was designed and implemented incorporates the desired functionality of TCP and also the small packet size of UDP.

The basic requirements of the protocol are defined as follows;

- Reliability – Since the protocol is designed for Multimedia Applications reliability is a must. The new protocol must be able to guarantee that all packets sent will arrive at the desired destination. If reliability was not implemented in the protocol congested networks would cause large packet delays/losses which would effect multimedia applications.
- Retransmission – In congested networks, the protocol must be able to retransmit packets if they are lost or heavily delayed on the network. This will maintain a level of service to the Application from which it can function properly.
- Use Datagram to transport data – The RelUDP protocol, like UDP will use datagrams to transfer data across the network.

This will allow the protocol to send the data quickly, as the packets are small.

- HOL blocking absent – The new protocol will avoid the phenomenon known as HOL blocking. This problem only occurs when the protocol implements in-order packet delivery. Consequently the new protocol does not implement in-order delivery.
- Minimal Congestion Control – For multimedia applications, congestion control mechanisms similar to those used in TCP may effect the performance of the application. This is fundamentally because congestion control mechanisms tend to decrease the rate at which the protocol is sending packets if the network is congested. Consequently the application ends up with delays in the incoming data, which effects the performance.

The protocol was tested in the Network simulator and simulated using the Network Animator. The results of these tests prove that the protocol was successful.

References

Reference Format:

Reference Number, Author(s) [Surname, First Name (& for multiple authors) OR Company Name], Reference Location [Internet URL OR Book Title OR Both], Date Last Visited

[1] TCP RFC, http://www.faqs.org/rfcs/rfc793.html, 12/03/05

[2] UDP RFC, http://www.faqs.org/rfcs/rfc768.html, 12/03/05

[3] Larry L.Peterson & Bruce S.Davie, Computer Networks A systems Approach second edition, 12/03/05

[4] Internet Protocols (IP) http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ip.htm 12/03/05

[5] Connectionless Transport UDP http://www-net.cs.umass.edu/kurose/transport/UDP.html, 12/03/05

[6] M. Chiang, TCP congestion control, 12/03/05

[7] Comparative analysis – TCP/UDP http://www.laynetworks.com/, 12/03/05

[8] SIP RFC, http://www.faqs.org/rfcs/rfc2543.html, 12/03/05

[9] SIP Session Initiation Protocol http://www.javvin.com/protocolSIP.html, 12/03/05

[10] Peter Parnes, Voice over IP (VoIP) lecture 7, http://www.cdt.luth.se/~peppar/kurs/smd151/lekt/7/voip-6.pdf, 12/03/05

[11] Internet Telephony inside networking http://www.tmcnet.com/it/0302/0302in.htm, 12/03/05

[12] RUDP Reliable UDP, http://www.javvin.com/protocolRUDP.html, 12/03/05

[13] Phillip T.Conrad, Reliable IP Telephony Applications with SIP using RserPool,

[14] Meredith Lulling, John Vaughan, investigation of TCP throughput variation for SIP traffic, 12/03/05

[15] Network Simulator (NS-2) Documentation http://www.isi.edu/nsnam/ns/ns-documentation.html, 12/03/05

[16] Marc Greis/VINT group, Tutorial for the Network simulator "ns", http://www.isi.edu/nsnam/ns/tutorial/, 12/03/05

[17] Xu Leiming, How to add a new protocol in ns2, http://www.cs.tcd.ie/~htewari/papers/ns-extend-xlming.ppt, 12/03/05

[18] Keijo Harju and susanna korventausta, Network simulation and protocol implementation using network simulator 2, 12/03/05

[19] Network Simulator (ns) Tutorial 2002, http://www.isi.edu/nsnam/ns/ns-tutorial/tutorial-02/, 12/03/05

[20] Jae Chung and Mark Claypool, NS by example, http://nile.wpi.edu/NS/, 12/03/05

[21] Ryan Dixon, Ns-2 tutorial, 12/03/05

[22] Su Wen, Ns-2 network simulator, http://protocols.netlab.uky.edu/~suwen/paper/ns-short.ppt, 12/03/05

[23] Paul Kelly, A Guide to C++ Programming, 12/03/05

[24] C++ Tutorial, http://www.cplusplus.com, 12/03/05

[25] Network Animator (NAM), http://www.isi.edu/nsnam/nam/, 12/03/05

Abbreviations

CBQ	Class-Based Queuing
HOL	Head of Lone blocking
IP	Internet Protocol
LAN	Local Area Networks
MTU	Maximum Transmission Unit
NAM	Network Animator
NS	Network Simulator
oTCL	Object oriented TCL
RED	Random Early Detection
RelUDP	new developed protocol
RTT	Round Trip Time
SAck	Selective Acknowledgement
SIP	Session Initiation Protocol
TCL	Scripting language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VoIP	Voice over IP
WAN	Wide Area Networks

A Appendix

A.1.1 RelUDP.h

```
#ifndef ns_RelUDP_h
#define ns_RelUDP_h

#include "agent.h"
#include "trafgen.h"
#include "packet.h"
#include "scheduler.h"
#include "timer-handler.h"

//"rtp timestamp" needs the samplerate
#define SAMPLERATE 8000
#define RTP_M 0x0080 // marker for significant events

struct  hdr_RelUDPAck  {
u_int32_t srcid_;
int seqno_;
int retryCount_;
/* per-field member functions */
u_int32_t& srcid() { return (srcid_); }
int& seqno() { return (seqno_); }
int& retryCount() { return (retryCount_); }
/* Packet header access functions */
static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_RelUDPAck* access(const Packet* p) {
return (hdr_RelUDPAck*) p->access(offset_);}
};

struct  hdr_RelUDP  {
u_int32_t srcid_;
int seqno_;
int retryCount_;
/* per-field member functions */
u_int32_t& srcid() { return (srcid_); }
int& seqno() { return (seqno_); }
int& retryCount() { return (retryCount_); }

/* Packet header access functions */
static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_RelUDP* access(const Packet* p) {
return (hdr_RelUDP*) p->access(offset_);}
};

class RelUDPAgent;

class CheckBufferTimer : public TimerHandler {
```

```
public:
        CheckBufferTimer(RelUDPAgent* t) : TimerHandler(), t_(t) {}
        inline virtual void  expire(Event*);
protected:
    RelUDPAgent* t_;
};

class RelUDPAgent : public Agent {
public:

  friend class CheckBufferTimer;
  RelUDPAgent();
  RelUDPAgent(packet_t);
  virtual void sendmsg(int nbytes, const char *flags = 0)
  {
        sendmsg(nbytes, NULL, flags);
  }
  virtual void sendmsg(int nbytes, AppData* data, const char *flags = 0);
  virtual void recv(Packet* pkt, Handler*);
  virtual int command(int argc, const char*const* argv);
  virtual void recordPacket(Packet* pkt, double timesent);
  void removeAckdPackets(int buffersize);
  virtual void checkBufferLoop();
  int packetIndex_;
  int BufferSize;

private:
  int seqno_;
  CheckBufferTimer cb_timer;
  Packet *packetsSent[4000];
  double timesSent[4000];

protected:
  double CHECK_BUFFER_INT;
  double MAXDELAY;
};

#endif
```

A.1.2 RelUDP.cc

```
#include "RelUDP.h"
#include "rtp.h"
#include "random.h"
#include "address.h"
#include "ip.h"

int hdr_RelUDPAck::offset_;
int hdr_RelUDP::offset_;


//Header class for the Acknowledgement Packet
class RelUDPAckHeaderClass : public PacketHeaderClass {
public:
```

```
        RelUDPAckHeaderClass() : PacketHeaderClass("PacketHeader/RelUDPAck",
                                     sizeof(hdr_RelUDPAck)) {
        bind_offset(&hdr_RelUDPAck::offset_);
  }
} class_RelUDPAckhdr;

//Agent class - RelUDP agent
static class RelUDPAgentClass : public TclClass {
public:
 RelUDPAgentClass() : TclClass("Agent/RelUDP") {}
 TclObject* create(int, const char*const*) {
        return (new RelUDPAgent());
  }
} class_RelUDP_agent;

/*--------------------------------------------------------------------
 *Constructors
 *
 */

RelUDPAgent::RelUDPAgent() : Agent(PT_RelUDP), seqno_(-1), cb_timer(this)
{
 bind("packetSize_", &size_);
 packetIndex_=0;
 BufferSize=4;
 MAXDELAY=.1;
 CHECK_BUFFER_INT==.7;
}

RelUDPAgent::RelUDPAgent(packet_t type) : Agent(type), cb_timer(this)
{
 bind("packetSize_", &size_);
 packetIndex_=0;
 BufferSize=4;
 MAXDELAY=.1;
 CHECK_BUFFER_INT==.7;

}

//This function deals with the event when the timer expires
void CheckBufferTimer::expire(Event*)
{
  t_->checkBufferLoop();

}

//This function adds the Packet "pkt" to the buffer "packetsSent"
void RelUDPAgent::recordPacket(Packet* pkt, double timesent)
{   //if the buffer is full, expand it

 if(packetIndex_==(BufferSize-1))
 {
        int newsize=BufferSize*2;
        removeAckdPackets(newsize);
 }
 //after every 10 packets sent, remove the Acknowledged packets
 if(((packetIndex_ % 10)==0) && (packetIndex_ != 0))
 {
        removeAckdPackets(BufferSize);
 }
 packetsSent[packetIndex_]=pkt->copy();
```

```
  timesSent[packetIndex_]=timesent;
  packetIndex_++;
}

//This function performs two tasks:
// (1)- It expands the buffer "packetsSent" to allow "buffersize" elements
//      it does this by copying the contents into a new array of "buffersize" size
// (2)- It removes the Packets which have been acknowledged in the required time
//            these have their sequence number set to -2
void RelUDPAgent::removeAckdPackets(int newBufferSize)
{
  Packet *tempPacketarray = new Packet[newBufferSize];
  double *tempTimeArray = new double[newBufferSize];

        int numberPacketsAdded=0;
        for(int index=0; index<packetIndex_;index++ )
        {
hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(packetsSent[index]);
              if((relUDPHdr->seqno()!= -2) && (relUDPHdr->seqno()!= -3))
              {
                    tempPacketarray[numberPacketsAdded]=*packetsSent[index];
                    tempTimeArray[numberPacketsAdded]=timesSent[index];
                    numberPacketsAdded++;
              }
        }

        for(int index=0; index<numberPacketsAdded ;index++ )
        {
              *packetsSent[index]=tempPacketarray[index];
              timesSent[index]=tempTimeArray[index];
        }

  delete[] tempTimeArray;
  delete[] tempPacketarray;

  packetIndex_=numberPacketsAdded;
  BufferSize=newBufferSize;
}



void RelUDPAgent::checkBufferLoop()
{
  double sendtime;
  double local_time = Scheduler::instance().clock();

  double senddelay=.0000001;
  for(int index=0; index<packetIndex_;index++ )
  {
        hdr_RelUDP* origRelUDPHdr = hdr_RelUDP::access(packetsSent[index]);
        //if the packet is not marked for deletion already
        if((origRelUDPHdr->seqno()!= -2) && (origRelUDPHdr->seqno()!= -3))
        {
              if((local_time - timesSent[index])>=MAXDELAY)
              {
                    sendtime = Scheduler::instance().clock();
                    Packet *p;
                    p = allocpkt();
                    p=packetsSent[index]->copy();
                    hdr_RelUDP* newRelUDPHdr = hdr_RelUDP::access(p);
                    newRelUDPHdr->retryCount()=(origRelUDPHdr->retryCount()) +1;
```

```
                    recordPacket(p,(sendtime+senddelay));
                    origRelUDPHdr->seqno()=-3;
                    Scheduler::instance().schedule(target_, p, senddelay);
                    senddelay=senddelay+0000001;
                }
        }
 }

 double next_time =CHECK_BUFFER_INT;
 if(next_time >0)
 {
        if (cb_timer.status() == TIMER_IDLE)
        {     cb_timer.sched(next_time);}
        if (cb_timer.status() == TIMER_PENDING)
        {     cb_timer.cancel();}
        cb_timer.resched(next_time);
 }
}

// put in timestamp and sequence number, even though RelUDP doesn't usually
// have one.
void RelUDPAgent::sendmsg(int nbytes, AppData* data, const char* flags)
{
 double sendtime;
 Packet *p;
 int n;
 if (size_)
        n = nbytes / size_;
 else
        printf("Error: RelUDP size = 0\n");

 if (nbytes == -1) {
        printf("Error:  sendmsg() for RelUDP should not be -1\n");
        return;
 }

 // If they are sending data, then it must fit within a single packet.
 if (data && nbytes > size_) {
        printf("Error: data greater than maximum RelUDP packet size\n");
        return;
 }

 double local_time = Scheduler::instance().clock();
 while (n-- > 0) {
        p = allocpkt();
        hdr_cmn::access(p)->ptype() = PT_RelUDP;
        hdr_cmn::access(p)->size() = size_;
        hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(p);
        relUDPHdr->seqno() = ++seqno_;
        hdr_cmn* cho = hdr_cmn::access(p);
        relUDPHdr->retryCount()=0;
        hdr_ip* iph = hdr_ip::access(p);
        const char* name = packet_info.name(cho->ptype());
        hdr_cmn::access(p)->timestamp() =
            (u_int32_t)(SAMPLERATE*local_time);
        p->setdata(data);
        target_->recv(p);
        sendtime = Scheduler::instance().clock();
        recordPacket(p,sendtime);
 }
 n = nbytes % size_;
```

```
  if (n > 0) {
        p = allocpkt();
        hdr_cmn::access(p)->ptype() = PT_RelUDP;
        hdr_cmn::access(p)->size() = n;
        hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(p);
        relUDPHdr->seqno() = ++seqno_;
        relUDPHdr->retryCount()=0;
        hdr_cmn* cho = hdr_cmn::access(p);
        hdr_ip* iph = hdr_ip::access(p);
        const char* name = packet_info.name(cho->ptype());

        hdr_cmn::access(p)->timestamp() =
            (u_int32_t)(SAMPLERATE*local_time);
        p->setdata(data);
        target_->recv(p);
        sendtime = Scheduler::instance().clock();
        recordPacket(p,sendtime);
  }
  idle();
}

// This function is called when a packet is received
void RelUDPAgent::recv(Packet* pkt, Handler *h)
{
  double recvtime= Scheduler::instance().clock();
     //Opens the common header of the packet being received
  hdr_cmn* cho = hdr_cmn::access(pkt);
  const char* packetname = packet_info.name(cho->ptype()); //The textual name of the
packet being received

  if((cho->ptype() == PT_RelUDP)) // if it is a normal RelUDP packet
  {


        double local_time = Scheduler::instance().clock();
        //We want to create a new RelUDPAck packet and send it acknowledging the
packet
        Packet *np = allocpkt();
        hdr_cmn* nch = hdr_cmn::access(np);  //access the common header of the new
packet
        nch->size()=12;          //sets the sixe of the new packet to 40 as it is an
Ack

        //access the IP header fields of the packet being received and the new packet
being created
        hdr_ip * iph = hdr_ip::access(pkt);
        hdr_ip * newiph = hdr_ip::access(np);

        //switch the Destination and source of the packet that was received and set
these valuse to the new packet
        newiph->dst()=iph->src();
        newiph->flowid()=iph->flowid();
        newiph->prio()=iph->prio();

        hdr_RelUDP* relUDPHdr = hdr_RelUDP::access(pkt);
        nch->ptype() = PT_RelUDPAck;

        hdr_RelUDPAck* relUDPAckHdr = hdr_RelUDPAck::access(np);
        relUDPAckHdr->srcid()=addr();
        relUDPAckHdr->seqno()=relUDPHdr->seqno();
        hdr_RelUDP* rch6 = hdr_RelUDP::access(np);
```

```
        relUDPAckHdr->retryCount()=relUDPHdr->retryCount();
        target_->recv(np);
        if (app_ ) {
              // If an application is attached, pass the data to the app
              hdr_cmn* h = hdr_cmn::access(pkt);
              app_->process_data(h->size(), pkt->userdata());
        }
        Packet::free(pkt);
  }
  else if(cho->ptype() == PT_RelUDPAck)
         {
              hdr_RelUDPAck* relUDPAckHdr = hdr_RelUDPAck::access(pkt);
              int sequenceNumber =relUDPAckHdr->seqno();
              int pIndex=-1;
              double sendtime;

              for(int index=0; index< packetIndex_ ;index++ )
              {
              hdr_RelUDP* buffer_relUDPHdr= hdr_RelUDP::access(packetsSent[index]);
              if((buffer_relUDPHdr->seqno()==sequenceNumber) &&
              (buffer_relUDPHdr->retryCount()==relUDPAckHdr->retryCount()))
              {
                    sendtime =  timesSent[index];
                    pIndex=index;
              }
              }

              double delay = recvtime-sendtime;
              if(pIndex!=-1)
              {
                    if(( delay >= MAXDELAY ))
                    {     double local_time = Scheduler::instance().clock();
                          hdr_RelUDP* old_RelUDPHdr =
                          hdr_RelUDP::access(packetsSent[pIndex]);
                          Packet *newpkt;
                          newpkt = allocpkt();
                          newpkt=packetsSent[pIndex]->copy();
                          hdr_RelUDP* newRelUDPHdr = hdr_RelUDP::access(newpkt);
                          newRelUDPHdr->retryCount()=(relUDPAckHdr->retryCount()) +1;
                          target_->recv(newpkt);
                          recordPacket(newpkt,local_time);
                          old_RelUDPHdr->seqno()=-3;

                    }
                    else
                    {
                          hdr_RelUDP* old_RelUDPHdr =
                          hdr_RelUDP::access(packetsSent[pIndex]);
                          old_RelUDPHdr->seqno()=-2;
                    }
                    checkBufferLoop();
              }
        Packet::free(pkt);
        }
        else
        {
              if (app_ ) {
              // If an application is attached, pass the data to the app
                    hdr_cmn* h = hdr_cmn::access(pkt);
                    app_->process_data(h->size(), pkt->userdata());
```

```
                }
                Packet::free(pkt);
        }
}

int RelUDPAgent::command(int argc, const char*const* argv)
{
  if (argc == 4) {
        if (strcmp(argv[1], "send") == 0) {
                PacketData* data = new PacketData(1 + strlen(argv[3]));
                strcpy((char*)data->data(), argv[3]);
                sendmsg(atoi(argv[2]), data);
                return (TCL_OK);
        }

  } else if (argc == 5) {
        if (strcmp(argv[1], "sendmsg") == 0) {
                PacketData* data = new PacketData(1 + strlen(argv[3]));
                strcpy((char*)data->data(), argv[3]);
                sendmsg(atoi(argv[2]), data, argv[4]);
                return (TCL_OK);
        }
  }
  return (Agent::command(argc, argv));
}
```