The Effects of Adaptive Error Correcting Schemes on Satellite Optimized TCP

by

John Graumann

A Thesis presented to the University of Manitoba in
partial fulfillment of the requirements for the degree of
Master of Science
in the
Department of Electrical and Computer Engineering
Winnipeg, Manitoba

October 2006

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

The Effects of Adaptive Error Correcting Schemes on Satellite Optimized TCP

by

John Graumann

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

of

Master of Science

ABSTRACT

With the popularity of satellite networks as a means for Internet delivery, the performance of the Transmission Control Protocol (TCP) over satellite links has been well studied. Unfortunately, the protocol has shown problems with channel utilization in such environments, which is a key performance factor for satellite operators. Performance enhancing proxies are one solution to this problem, and have been shown to provide significant improvements in channel utilization. Recent research, however, has been focused on the use of adaptive error correction in satellite networks at the lower layers, resulting in the ratification of the DVB-S2 standard. While adaptive error correction can improve channel utilization when channel conditions are good, it also results in decreases in available user bitrate during noisy periods due to the addition of parity bits. The research performed in this thesis shows that the throughput of a performance enhancing proxy suffers when adaptive error correction is introduced, resulting in up to 20% of the bandwidth being wasted during poor channel conditions. A solution involving the addition of a simple bandwidth estimation scheme to the proxy is studied, and is successful in increasing the average channel utilization in poor conditions to almost 90%. Alternatively, it is shown that the introduction of a secondary TCP flow running in parallel with the performance enhancing proxy traffic can result in near 100% channel utilization when priority queues are introduced at the satellite gateway. Tightly integrating performance enhancing proxies with the satellite gateways that employ adaptive error correction is recommended as the best solution.

ACKNOWLEDGEMENTS

Many thanks to my family for their patience and support during my thesis, in particular to my wife for putting up with me over the last few years. Thanks also to my advisor Dr. Ken Ferens for his topic idea and for his help and advice.

This research has been supported in part by the Natural Sciences and Engineering Research Council of Canada.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF COPYRIGHTED MATERIAL FOR WHICH PERMISSION WAS OBTAINED

1. INTRODUCTION

Satellite networks have become an increasingly popular medium to deliver both broadcast and interactive services to users around the globe. One main reason for this is that satellite terminals can be easily deployed to remote locations where no terrestrial infrastructure exists. This makes satellite communications an attractive way to offer Internet access to remote users. The majority of Internet applications, such as file transfers and streaming video, use the Transmission Control Protocol (TCP) for reliable delivery. Since TCP was designed for low latency, low error-rate network links, satellite networks initially presented system integrators with the challenge of effective channel utilization when TCP was the protocol of choice. As a result, the use of satellites as a medium for Internet delivery has always been limited by the usefulness of TCP over those links.

Much research has been done in the area of improving TCP performance over satellite links, including modifications to the protocol itself. Increasing the maximum allowable receive window and allowing TCP to acknowledge out of order segments are two examples [AlGS99]. Other research has looked at link layer enhancements that hide the effects of errors in the network by using error correction and error recovery techniques [LiDa04]. These enhancements could either be deployed at the endpoints of the network near the TCP hosts, where they could use their knowledge of the protocol to further optimize the transmission, or they could reside at the endpoints of the satellite link itself. In either case, both techniques have the advantage of not requiring any changes to existing implementations of the TCP protocol.

Results of this research have led to the deployment of so-called performance enhancing proxies in satellite networks [PhFe02]. A performance enhancing proxy (PEP) is a device that sits between the endpoints of a TCP connection and enhances the connection, either explicitly with the knowledge of the endpoints, or transparently. If a PEP intercepts data packets from the TCP connection, it can forward them over the satellite link using either

an optimized version of TCP or a custom protocol specifically designed for the parameters of the channel. Commercial implementations of performance enhancing proxies have been deployed in existing networks and provide optimized TCP to users every day [Nett00], [Idirec]. The use of such systems is popular in Digital Video Broadcasting satellite networks (DVB-S) that deliver interactive services.

In addition to performance enhancing proxies, the performance of TCP can also be improved by lowering the bit error rate of the underlying channel. One way to achieve this is to apply error correcting bits to the data packets before they are sent over the satellite link. Traditionally, lower level protocols, such as those used in the DVB-S specification, provide a fixed amount of lower level error correction. As a result, the amount of error correction applied is tuned to the worst-case conditions expected on the underlying link.

Recent research in the area of error correction has introduced the concept of adaptive error correction [BaLK04], [CiGaRi], [LiGT02], [LiDa04]. Since the channel conditions in a satellite network vary over time due to factors such as inclement weather and other interference, it has been shown that varying the amount of error correction to match the prevailing state of the link can provide significant improvements in channel utilization, especially during favorable channel conditions. Adaptive error correction has been introduced in the recently ratified DVB-S2 specification, which provides the option for adaptive coding and modulation [MoMi04].

The integration of DVB-S2 equipment supporting adaptive error correction into an existing network could present a major problem, however. Since the performance enhancing proxies currently deployed were designed for older DVB-S systems, where the amount of error correction is static, they assume that the available user bitrate of the channel is constant during the course of the transmission. This is no longer the case with the introduction of adaptive coding methods. As more error correcting parity bits are added to compensate for poor channel conditions, the available user bitrate will decrease. This could potentially result in congestion, causing packets to be dropped before they are

sent over the satellite channel. While a TCP connection would correctly detect the congestion and reduce its sending rate accordingly, performance enhancing proxies are typically designed to attribute dropped packets to bit errors. Consequently, rather than backing off their output bitrate to ease congestion on the link, a performance enhancing proxy would continue to send data at the same rate, causing the link to remain congested until channel conditions improve. A decrease in channel utilization would result, appearing to counter any benefits that adaptive error correction introduces.

To study this phenomenon, a simple performance enhancing proxy is designed that demonstrates improvement over the performance of TCP in a traditional DVB-S network. Adaptive error correction based on the DVB-S2 specification is then introduced to the system, and the resulting effects of varying available user bitrate are observed.

This thesis is organized as follows: Chapter 2 provides background and related research on the Transmission Control Protocol, proposed enhancements for TCP over satellite networks, and an overview of the DVB-S2 specification; Chapter 3 details the design and implementation of the performance enhancing proxy developed in this thesis; Chapter 4 presents the simulation results and a discussion of those results; Chapter 5 provides conclusions based on the results and offers recommendations for future research.

2. BACKGROUND

2.1 Transmission Control Protocol

The Transmission Control Protocol (TCP) is a reliable network delivery protocol commonly used in Internet applications such as HTTP (web browsing), FTP (file transfer) and streaming video. A TCP connection is created between two endpoints, or hosts, using a three-way handshaking system built into the protocol. After a connection has been established, data is transferred between the two hosts using a stream of bytes, or octets. For this reason, TCP is also described as a stream delivery protocol. In practice, data is actually broken into pieces called TCP segments, the maximum size of which is dictated by the lower level delivery protocols. In a typical Ethernet network, such as those found in most homes and offices, the maximum segment size is 1460 bytes.

Most applications that use TCP do so because they require reliable, in-order delivery of data. For example, file transfers must be free of lost or corrupted data, while real-time applications such as streaming video require that data packets be delivered in sequence. Although packet corruption on Internet links attributable to bit errors is uncommon, segments on the Internet can be lost as a result of congestion at network routers. In addition, two segments may take different paths when traveling between hosts, resulting in out-of-order delivery.

Segments in TCP are identified using sequence numbers that correspond to their byte position in the stream. When data is received, the receiver sends positive acknowledgements to the sender indicating that segments have been received correctly. The acknowledgements (ACKs) are cumulative, meaning that if a particular segment is acknowledged, the sender can also assume that all segments prior to it were received correctly. Conversely, if segments are lost or received out of order, the receiver will not acknowledge new segments until all preceding ones have been received. Instead, it will send a duplicate acknowledgment for the last in-order segment that was correctly

received. This has the effect of telling the sender that, while data is still flowing through the network, one or more segments may have been lost and need to be retransmitted.

Rather than sending and acknowledging a single segment at a time, TCP uses what is called a sliding window mechanism. When the connection is set up, the receiver advertises a value called the receive window *(rwnd)*, which represents the maximum number of unacknowledged bytes that the sender can transmit at once. This typically corresponds to the amount of buffer space that the receiver has available. After the sender has transmitted as many bytes as allowed by the receive window, it must wait for acknowledgements to arrive before it can continue transmitting. When an acknowledgement finally arrives from the receiver, the sender can "slide" the window forward by the amount of acknowledged bytes, creating room in the window to send more data. This concept can be visualized by assuming the sender maintains three pointers in its window [Come00]. The left-most pointer holds the sequence number of the start of the window, which corresponds to the oldest unacknowledged segment previously sent. The middle pointer points to the most recently transmitted segment. Finally, the right-most pointer points to the end of the window, or the last allowable segment that the sender may transmit. As acknowledgments arrive, both the left and right pointer slide forward by the number of acknowledged bytes, creating space in the window for additional segments to be transmitted.

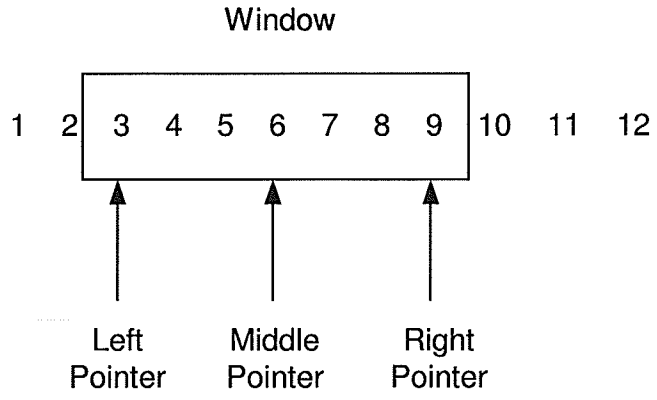

Figure 2.1 - TCP Sliding Window Illustration.

Since the advertised receive window in TCP is sent with every acknowledgement, its value can change over the course of the transmission. If the receiver cannot accept more data due to a lack of buffer space, it can reduce the window size by advertising a smaller window. This will halt or slow the transmission of data from the sender until the receiver is ready. This process is the method by which TCP avoids congestion at the receiving end of the connection.

While the *rwnd* parameter ensures that dropped segments never occur at the receiver, it does not provide protection against congestion at the buffers of intermediate devices on the network. Competing network traffic or a simple lack of buffer space on equipment in the network may cause congestion, resulting in dropped packets. To account for this, the actual window size used by the TCP sender is calculated as the lesser of the receive window and another window called the congestion window *(cwnd).* When a TCP connection is initiated, the congestion window starts at no larger than two segments and can grow from there as data is sent over the network using a process called *slow-start.* For every acknowledgment that returns from the receiver, the sender can increase the value of the congestion window by one segment. In this way, *cwnd* can double for every window of data sent. If the latency of the link is low, the window can grow very quickly. Therefore, to prevent congestion due to *cwnd* growing at too fast a rate, TCP will exit *slow-start* once the congestion window hits a predefined value called *ssthresh,* which is typically set to match the size of the receive window when the connection is initiated. Once *ssthresh* is reached, TCP enters a second phase called *congestion avoidance.* During *congestion avoidance* the congestion window can only be increased by one segment for every round trip time.

As long as segments are received and acknowledged without error, the congestion window will continue to grow. However, if the sender detects that segments have been lost, TCP assumes that congestion has occurred on the network and will take appropriate action. This assumption is based on the fact that bit errors on terrestrial links are extremely rare, and thus lost packets are most likely due to congestion. Losses are

detected through the use of a retransmission timer that is set for each transmitted segment. If the timer expires before a corresponding acknowledgement is received, the segment is assumed to be lost due to congestion and is retransmitted. When this happens, TCP will reduce its rate of transmission by setting *ssthresh* to half the current value of *cwnd,* and then setting *cwnd* back to a single segment. This causes TCP to re-enter *slow-start* as described above. It should be noted that the length of the retransmission timer must be large enough so that acknowledgments can arrive before it expires. To account for this, TCP adapts the value of the timer by using arriving acknowledgments to estimate the round-trip time of the link.

The congestion avoidance method described above is fairly aggressive in attempting to reduce congestion. To increase performance in the case where only a few segments are lost, TCP also uses a concept called *fast-retransmit.* If a single segment is lost on the network, the sender will receive a duplicate acknowledgement for each subsequent segment that reaches the receiver. If the sender receives three duplicate acknowledgements for a segment, it will assume that the corresponding segment was lost and will re-send the segment immediately without waiting for the retransmission timer to expire. Since the arrival of acknowledgments indicates that the sender is reasonably sure that the network is not overly congested, it will also employ a technique called *fast-recovery.* Rather than *cwnd* being set to a single segment, it will instead be divided in half. *Ssthresh* will also be set to the new value of *cwnd,* causing TCP to skip *slow-start* and go directly into *congestion avoidance.*

2.2 TCP and Satellite Networks

Satellite networks offer several characteristics that make them popular as a medium for content delivery. For example, a single satellite footprint can cover a very large geographical area that can service a large number of users [Stal02]. This enables the deployment of satellite terminals to remote areas or locations where the installation of terrestrial infrastructure, such as coax or fiber optics, is cost-prohibitive. The benefits of increased coverage over a terrestrial link can be obtained without sacrificing bandwidth,

as a single satellite transponder can provide a bitrate that is sufficient for transmitting files, video, and other data over the Internet. As a result, the use of satellite networks as a way to connect branch offices, remote populations or even single users to the Internet has many attractive benefits. Despite these advantages, though, certain characteristics of satellite channels present challenges to protocols such as TCP.

2.2.1 Propagation Delay

Unlike low latency terrestrial networks, satellite networks can exhibit a large propagation delay. While round trip times of packets on the Internet can be as low as 100 ms, the latency on a satellite channel is typically much higher. This is most evident in geostationary (GEO) satellite networks, with satellites that orbit the earth at approximately 36,000 km above the earth's equator [Stal02]. A packet leaving a ground station must travel this distance once from ground-to-satellite and again from satellite-to-ground, corresponding to a one way trip of approximately 250 ms. If the same network is used again on the return trip, the resulting round trip time is on the order 500 ms, which is well above the normal delay observed on a ground link.

This increased latency can be detrimental to the performance of TCP in a number of ways. As mentioned previously, TCP initiates its transmission using a congestion window of only one or two segments. The window then proceeds to grow by one segment for every ACK that is received during the *slow-start* phase. In terrestrial networks, the relatively low round trip time means that ACKs arrive quickly, allowing the window size to grow at a faster rate. However, when the latency is increased, the congestion window will grow at a much slower rate since acknowledgements take longer to arrive. That being said, the performance hit seen due to the slow growth of the window becomes negligible over the course of a lengthy transmission as long as there are no further impairments due to lost segments.

Another problem that the high latency of satellite channels presents is that of bandwidth utilization. As previously stated, the TCP sender's actual window size is taken as the

lesser of *rwnd* and *cwnd*. Even if the congestion window grows to a very large size, the sender cannot send more unacknowledged segments than the receive window allows, forcing it to wait for acknowledgements before transmitting more data. If the round trip time is sufficiently large, it is possible that only a single window of data will be allowed to leave the sender for each return trip. The maximum throughput of TCP and its relationship to the window size and the latency of the link can actually be defined as in Equation 2.1, where W is the window size and RTT is the round trip time of the network.

$$Throughput = \frac{W}{RTT} \qquad (2.1)$$

The equation shows that either an increase in the window size or a decrease in the round trip time will increase the maximum attainable throughput of the protocol. While the maximum receive window was originally specified in TCP as 64 KB, an option called window scaling was introduced to allow TCP to use higher windows and thus take better advantage of high latency links. However, network equipment and client computers running Microsoft Windows may not be optimized to support it. Therefore, while TCP contains provisions to account for larger round trip times, it is not realistic to assume that every client receiving data over a satellite network has been configured accordingly.

2.2.2 Increased Bit Error Rate

The second disadvantage of satellite networks is a higher bit error rate (BER) [AlGS99]. The higher bit error rate can be caused by a decrease in the signal-to-noise ratio of the received signal, which can be attributed to a number of factors. Loss of signal power over long distances, inclement weather such as rain or snow, and improperly aligned antennas will all cause the signal to be attenuated, resulting in a higher bit error rate. Additionally, interference from adjacent satellites or adjacent transponders on the same satellite can also increase the BER of the link. Even without taking into account impairments due to severe weather, satellite networks can see a BER on the order of 10^{-7} [AlGS99]. Although in some cases corrupt packets can be retransmitted using protocols like TCP, the higher latency of the link combined with this higher error rate reduces the efficiency of the

protocol greatly. The main reason for this is that TCP's philosophy is to assume that data losses are due to congestion and not the result of bit errors. This causes TCP to enter *slow-start*, as described above, which in turn reduces its rate of transmission. An analytical model of the performance of TCP has been developed that shows that, as the probability of a bit error is increased, the throughput of TCP decreases dramatically [PaFT98]. To counter this, most satellite networks provide some form of error correction at the link layers to reduce the bit error rate.

2.2.3 Bandwidth Asymmetry

In addition to the two factors described above, many satellite networks also exhibit asymmetry in the link characteristics seen on the forward and return link. Many satellite networks only use the satellite channel for transmitting data on the forward link, which carries packets from the server to the client host. The reason for this is that the required bandwidth from client to server is typically far less than that required in the other direction, since acknowledgements and connection requests represent the bulk of client traffic. As a result, while data from a server to a client machine may flow over the high bandwidth satellite link, return traffic may traverse a lower bitrate terrestrial link such as a dial-up connection or an ISDN line. Both the latency and the bandwidth of the return link can be much different than that of the forward link. If a low bitrate return link such as dial-up were used to service several clients behind a satellite gateway, the rate of returning TCP acknowledgements could exceed the capacity of the link, resulting in congestion and lost acknowledgements. The lost ACKs will again cause TCP to assume that congestion occurred, exacerbating the problems already discussed.

2.3 Improving TCP over Satellite Networks

All of the problems described above have been well studied and many solutions have been proposed, and in some cases implemented, that aim to improve the performance of TCP in satellite networks. Proposals include modifications to the protocol itself, link layer mechanisms aimed at minimizing the effects of errors in the network, and the use of performance enhancing proxies. These will all be discussed below.

2.3.1 Proposed Modifications to TCP

One modification to TCP is called TCP SACK (selective acknowledgements) [AlGS99]. TCP SACK allows the receiver to acknowledge specific segments even if they were received out of order. This means that in the instance that segments are lost on the network, the sending host will know which segments need to be retransmitted without waiting for timeouts to occur. Although *fast-recovery* and *fast-retransmit* allow TCP to retransmit a single segment immediately, subsequent lost packets cannot be detected until the acknowledgement for the first retransmission arrives or a timeout occurs. TCP SACK improves upon this by instead enabling the transmitting host to detect losses as they occur, in some cases avoiding a switch to *slow-start* altogether. TCP SACK has been shown to provide significant performance increases in instances where multiple segments are lost in a single window [AlHO97].

To address bandwidth asymmetry, techniques such as delayed acknowledgements have also been implemented. In this case, the receiver is allowed to wait until at most two segments have been received correctly before sending a cumulative acknowledgement. This has the effect of reducing return link traffic. However, the consequence is that acknowledgements arrive much less frequently to the sender, resulting in slower growth of *cwnd.*

Another modification that has been proposed is a TCP variant called TCP Westwood. TCP Westwood uses a bandwidth estimation technique to help the sender calculate more optimum settings for *cwnd* and *sstresh* when congestion occurs. This allows TCP Westwood to adjust its sending rate to correspond with the available bandwidth on the link when packet losses do occur, as opposed to traditional implementations that cut *cwnd* in half. TCP Westwood has been shown to provide a significant performance increase over current TCP implementations in high bit error rate environments such as satellite networks [GeSaWZ].

Lastly, while not a modification of TCP directly, a protocol called XFTP has also been proposed [AlKrOs]. XFTP works on the principle that while a single TCP connection

using a certain window size can achieve a fixed bandwidth over a link, two connections with the same parameters should be able to achieve roughly twice the performance. XFTP calls for modifications to the traditional file transfer protocol (FTP), where files would be split into pieces that could be downloaded in parallel by multiple connections. While this can dramatically improve performance, the benefits are reduced when the number of connections becomes so large that competition for available bandwidth among connections occurs, causing congestion.

While the methods discussed here have been shown to provide performance improvements in some cases, modifications to the TCP protocol are rarely implemented. The reason for this is that TCP is an end-to-end protocol that operates in an environment where connected hosts can be separated by a number of intermediate network devices. TCP was carefully designed with this environment in mind, where aggressive use of available bandwidth would decrease the usability of the Internet. Any modification that goes against TCP's design philosophy of being a "fair" protocol with respect to other Internet traffic is therefore not likely to be accepted into the standard.

2.3.2 Link Layer Mechanisms

In order to reduce the effects that bit errors have on TCP, link layer mechanisms for both error correction and error recovery are commonly used. These measures can reduce the effective bit error rate seen by the receiver, as well as recover erroneous packets before they are detected as losses by TCP.

2.3.2.1 Error Correction

Forward error correction (FEC) involves adding additional information called parity bits to a stream of data prior to transmission. In the event that an erroneous bit occurs in a packet, the receiver can use the redundant information to correct the error in the packet so it does not need to be retransmitted by the source. This is useful in networks where retransmissions are impossible due to the lack of a return link, or in environments such as satellite networks where the round trip delay makes retransmissions inefficient [LeWi00].

Since it is done at the link layer, error correction is transparent to upper layer protocols like TCP. The amount of redundancy can be tuned to the known characteristics of the link to provide the desired quality of service. While adding more parity bits to the stream generally increases the probability that an error can be corrected, the fixed bandwidth of the link means that less user or data bits can be sent. The tradeoff between error rate and bandwidth must therefore be weighed when designing an error correction scheme for a particular link.

A common class of error correcting schemes involves the use of block codes. Data is divided into blocks of size k bits by the FEC encoder at the sender. Each block is then encoded by adding a fixed number of parity bits, resulting in a transmitted block size of n bits. The strength of a block code is typically described using a value called the coding rate R_c, which is simply the ratio of data bits to the total number of transmitted bits. In the absence of compression, R_c is always less than or equal to one. The number of bit errors that can be corrected in a block depends largely on the design of the code. For example, a type of block code called a Hamming code is designed to correct only a single bit error in a block, whereas BCH codes can correct a number of errors that is directly proportional to the number of parity bits $(n-k)$ added to the block [Stal02].

$$R_c = \frac{k}{n} \quad (2.2)$$

A common example of a block code is the Reed-Solomon code, which is used extensively in correcting errors on storage devices such as compact disks. Reed-Solomon codes process data in equal sized chunks, or symbols, where each chunk is m bits in length. These codes are also used in satellite communications and are responsible for improving error performance in Digital Video Broadcasting over Satellite (DVB-S) networks.

While error correcting codes at the link layer are usually provided at the bit-level, or the symbol level in the case of Reed-Solomon codes, block codes also exist that can provide

error correction on a block of packets. Since they allow for recovery of packets that have been lost from a stream, packet-level error correcting schemes are usually referred to as erasure codes. Similar to bit-level FEC, erasure codes work on a block of packets of size *N,* with *M* redundant packets added for a resulting block size *N+M.* Codes can be designed where, if *M* or fewer packets are lost in a block, they can be recovered without retransmission [Huit96]. While erasure codes are typically used in multicast applications, they can also be used in point-to-point applications. Erasure codes can be an effective way of recovering missing packets due to bit errors, but can also reduce the effects of lost packets due to congestion [Huit96]. Care should be taken when implementing them in practice, however, as the addition of redundant packets to the network could make congestion worse. For example, it has been shown that a simple selective automatic repeat-request protocol performs worse when using packet-level FEC in cases where the overall bitrate of the transmission is held constant [Kost02]. Conversely, end-to-end packet level FEC has been shown to improve the performance of TCP under certain conditions, particularly over links with high error rates and long round trip times [LuKa04].

2.3.2.2 Error Recovery

In addition to error correction, error recovery is necessary when bit errors that occur in data cannot be corrected. Error recovery in a network is typically done using an automatic repeat-request (ARQ) protocol. ARQ protocols require feedback from the receiver to notify the transmitting host when data has been lost or corrupted so that it can be retransmitted. A common example of such a protocol has already been discussed previously in the description of TCP. While error recovery is commonly used at higher layers in the network, research has also been done into its benefits at the link layer [JiZh00].

The simplest form of ARQ is called Stop-and-Wait. After the sender has transmitted each segment, it waits for either a positive acknowledgment (ACK), indicating that the packet was received correctly, or a negative acknowledgement (NAK), indicating that the packet

was received in error. Like TCP, a timer is set for each transmitted segment. If the timer expires before an ACK or NAK is received, the segment is retransmitted. Stop-and-Wait protocols can be looked at as having a window size of one segment, which Equation 2.1 shows will not allow for full utilization of links with large propagation delays.

A slightly more complex protocol is the Go-Back-N. In this scheme, the sender maintains a larger window and sends segments continuously as long as acknowledgements continue to arrive. If the receiver detects a segment that is out of sequence, a negative acknowledgment (NAK) is sent for the missing segment and all subsequent data is ignored. The NAK allows the sender to detect losses without waiting for the timer to expire. As the name of the protocol suggests, a loss detected at the sender causes it to go back the point where the loss occurred and retransmit all subsequent packets. In high BER links, it has been shown that this scheme can lose its effectiveness quite quickly [LiYu80].

A third common form of ARQ scheme is called Selective Repeat. In this instance, the receiver keeps a buffer that is larger than a single segment, as in Go-Back-N. When an out of sequence segment is received, a NAK is sent for the missing data, as before. At the same time, an acknowledgment is piggybacked on the NAK to tell the sender that the out of sequence segment was received correctly. Subsequent segments are also acknowledged, meaning that the sender need only retransmit the lost segment when it receives the NAK. The Transmission Control Protocol's error recovery scheme most closely resembles Selective Repeat ARQ.

ARQ schemes implemented at the link layer have been studied as a way to correct errors before they can be detected by TCP, preventing TCP from cutting its window size and entering *slow-start*. Although most of the research done in this area is focused on mobile applications, the same premise still applies to satellite networks. Link layer ARQ protocols require that the underlying equipment buffer segments as they are sent, so that they can be retransmitted in the case that losses occur. Since this is done at the link layer, a point-to-point communication link must exist between sending and receiving hardware.

One advantage to these types of schemes is that upper layer protocols need not be modified to take advantage of any performance increases, and in fact do not need to be aware that error recovery is taking place. Secondly, the time taken to detect and correct errors is greatly reduced when compared to end-to-end mechanisms, since the lower latency of a single hop link will allow for faster detection and recovery of errors [JiZh00].

Variations on the ARQ protocols discussed here have also been proposed, with the main goal of improving the throughput of these schemes in satellite environments. For example, a variation on the Go-Back-N algorithm was proposed and found to provide increased performance in satellite networks [LiYu80].

2.3.3 Performance Enhancing Proxies

An alternative way to increase the performance of TCP over satellite links is the use of performance enhancing proxies (PEP). A performance enhancing proxy is another device on the network that resides between the TCP endpoints and the satellite link. A proxy's main purpose is to hide the undesirable properties of the channel from the TCP endpoints. Proxies can be designed with specific knowledge of the properties of the underlying link, which is an advantage that an end-to-end protocol like TCP does not have.

A performance enhancing proxy will intercept traffic from a TCP connection, either transparently or with the knowledge of the sending application, and optimize it so that it performs better over the link. It does this using two techniques. First, a proxy can buffer traffic flowing from the transmitting host (the server) and acknowledge it even before it reaches the end destination. This has the effect of hiding errors on the satellite link from the server, allowing it to continue transmitting at its current rate, and in some cases freeing up resources on the server before the transmission is totally complete. This process is called *local acknowledgments*. When local acknowledgments are used, the proxy could also intercept and suppress acknowledgements as they return from the receiver, although this is not required since TCP will ignore any acknowledgments for segments outside its sending window. The other technique that is used is called *local*

retransmissions. This refers to the process of a performance enhancing proxy retransmitting segments when they are lost so that the TCP server does not have to. When *local acknowledgments* are used, *local retransmissions* must always also be implemented.

In practice, there are two major types of performance enhancing proxies: spoofing proxies and split-connection proxies. A spoofing proxy monitors TCP traffic as it passes through, and can transparently modify the behavior of the traffic to increase the performance of the connection. For example, transmitted segments can be cached at the proxy as they are sent and acknowledged locally. If the proxy detects that losses have occurred on the link, it can retransmit the segments before the TCP sender notices the error. This is best illustrated in Figure 2.2. The authors in [IsAl01] examined the performance of TCP spoofing over a high latency, congested link, and found that it offered significant performance increases in such an environment. A specific implementation of a transparent proxy is the TCP SNOOP protocol [BoKG01]. TCP SNOOP works by caching TCP segments as they travel from the sender to the receiver, however it does not acknowledge them locally. Rather, if duplicate acknowledgements are detected as they return from the receiver, lost segment are simply retransmitted directly from the proxy and the ACKs are suppressed to prevent them from reaching the server. This allows for the retransmission of segments without the TCP sender detecting the losses and entering congestion control.

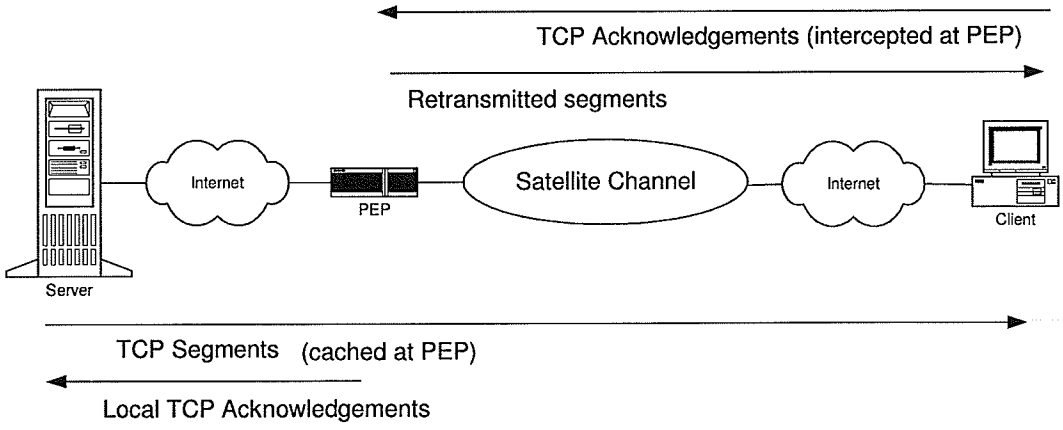

Figure 2.2 - TCP Spoofing.

The second type of performance enhancing proxy is a connection splitting proxy. Connection splitting proxies are distinguished from spoofing proxies in that they actually involve more than just a single end-to-end TCP connection. When a TCP connection is created, it is terminated at the PEP on the server side of the connection. A new connection is created from the proxy to the receiving host, effectively splitting the connection in two. The connection from the PEP to the client can be a modified form of TCP, or regular TCP that has had its window parameters tuned to take better advantage of the satellite link. This architecture, shown in Figure 2.3, enhances the connection at only a single point and is called an integrated performance enhancing proxy.

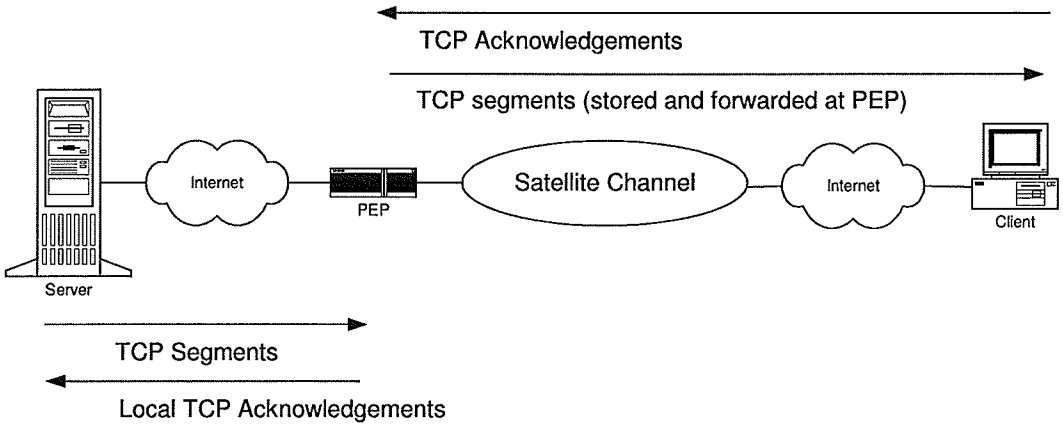

Figure 2.3 - Connection splitting (integrated).

Connection splitting can also be done in a distributed fashion. Distributed connection splitting proxies, shown in Figure 2.4, require a PEP at each end of the connection. In this case, three connections are actually created: one connection from the server to the PEP on the sending side, one between the two proxies over the satellite link, and one from the PEP at the receiving end to the client host. The major advantage here is that the protocol used on the middle connection need not be TCP. Instead, it can be a custom protocol that is designed specifically for the satellite link. Some implementations, for example, use a

custom protocol running on top of the User Datagram Protocol (UDP), which is connectionless and has no built in congestion control.

Distributed proxies offer several advantages. For example, data flowing through the PEP can be compressed to save bandwidth or encrypted for increased security. They can also use ARQ schemes for error recovery that are specifically chosen to perform well over satellite links, such as those that are tuned to high latency networks, or that reduce traffic on the return link.

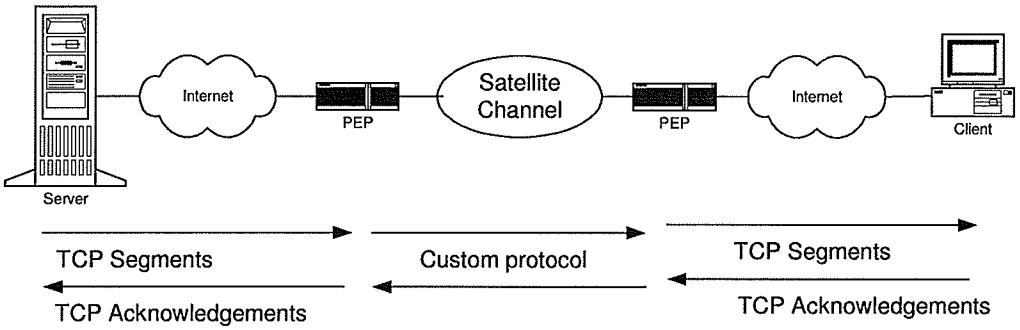

Figure 2.4 - Connection splitting (distributed).

Performance enhancing proxies have been studied extensively, and many implementations have been researched and evaluated. For example, a split-connection PEP that uses negative acknowledgements for error recovery was introduced, and was shown to improve the performance of TCP over single-hop satellite links [VeKM02]. This scheme throttles the bitrate of its connections with the help of link utilization measurements, guaranteeing that losses due to congestion do not occur. However, the proposed measurement relies on the fact that the total available link bandwidth is known ahead of time.

A UDP-based split connection scheme that relies on packet erasure codes for error correction has also been studied [Phil01]. While congestion control was implemented, it only took into account congestion at the receiver and did not account for congestion on

intermediate network devices. Although it employed erasure codes to account for losses, error recovery was left for future research.

The results of PEP research have culminated in commercial applications that have already been deployed in the field. For example, a split-connection approach using a connectionless protocol over the satellite link that is tuned specifically to the bandwidth of the satellite link [Nett00], and a transparent store and forward approach similar to the spoofing method described above [Idirec] have been deployed. Literature for these products shows that they can increase TCP performance in satellite environments by as much as 250% during web browsing sessions [Nett00].

2.4 Adaptive Forward Error Correction

While the combination of TCP enhancements, link level error control and the use of performance enhancing proxies can improve the performance of TCP over satellite links significantly, the protocol's performance is not the only consideration when designing satellite networks. Bandwidth utilization is an important metric for a service provider that wishes to get more performance out of a particular link, so that as many clients as possible can be serviced. In traditional satellite networks, the amount of error correction applied is usually fixed for the duration of the transmission. Network conditions, however, are time varying, and can change at any moment due to changes in position (in mobile terminals) or weather conditions. Since coding rates must be chosen so that they provide the desired level of error correction for the worst imagined channel conditions, bandwidth will be wasted due to the unnecessary transmission of parity bits when channel conditions are good. As a result, research has also been done extensively in the area of adaptive forward error correction, where the level of error correction applied during transmission is varied over time to match the channel conditions [Vuce91].

2.4.1 Related Research

The idea of adaptive error correction in both satellite networks and mobile wireless communications has been well researched. For example, a link layer protocol called TCP-

AFEC was proposed [LiGT02]. The authors implemented a link layer adaptive FEC protocol using Reed-Solomon block codes to improve the performance of TCP over mobile wireless channels. The protocol does not include link-layer retransmissions and relies solely on TCP to recover lost packets. A unique aspect of the proposed scheme is that it is considered to be TCP aware. The scheme uses lost packets to estimate the bit error rate of the channel, and the best tradeoff of bandwidth and bit error rate is calculated using the TCP model described earlier [PaFT98]. The authors found that over a range of symbol error rates, TCP-AFEC provided anywhere from a 7% to 78% increase in throughput over SNOOP and TCP-SACK, even when the latter schemes were complemented with some form of fixed error correction.

A packet level end-to-end adaptive FEC algorithm was also investigated [BaLK04], where varying levels of block erasure codes were used over a satellite link. Similar to TCP-AFEC, the algorithm is TCP aware in that it uses the same throughput model to find the optimum level of FEC required on the link. Using feedback from the receiver, the packet error rate and available channel bandwidth are estimated to aid in the calculation. The authors come to a similar conclusion as was found with TCP-AFEC. Specifically, they conclude that adaptive FEC at the packet level can be an effective tool when used over a long latency link that incurs losses due both to errors and congestion, but that the benefits are reduced when the round trip time is small.

2.4.2 DVB-S2

Adaptive forward error correction has gone from the realm of theory to practice with the ratification of DVB-S2, the second-generation satellite broadcasting specification, which was developed by the Digital Video Broadcasting group [ETSI05].

The original specification, DVB-S, was first introduced in 1994 and millions of terminals using the standard have since been deployed worldwide [TeMo04]. While initially designed for digital video broadcasting as the name suggests, DVB systems are also used as a method of distributing point-to-point IP multimedia content using a method known

as multi-protocol encapsulation (MPE). Essentially, MPE allows IP packets to be segmented and sent in MPEG-2 transport stream packets, the delivery unit used by DVB-S. As a result, DVB-S systems are currently deployed that offer IP services for a number of multimedia applications and services.

The DVB-S2 standard is intended to improve upon DVB-S by improving transmission efficiency, increasing implementation flexibility, and reducing modulator complexity [MoMi04]. The specification was designed for a number of intended services, including broadcast television service, Internet access, and data content distribution.

DVB-S2 offers increased spectrum utilization with its new coding and modulation schemes. The original specification (DVB-S) used Quadrature Phase Shift Keying (QPSK) modulation, and fixed length Reed-Solomon block codes for error correction. Reed-Solomon codes are replaced by low-density parity-check codes (LDPC) in DVB-S2. Extensive simulations showed that these codes were up to 35% more efficient in providing error correction than those used in DVB-S [MoMi04]. In addition, QPSK is no longer the only modulation scheme used. Three other options were added, including 8-PSK, 16-APSK and 32-APSK.

The combination of the chosen modulation and coding scheme is referred to in DVB-S2 as the protection level. The chart in Figure 2.5 shows the possible protection levels in DVB-S2, and their performance in comparison to those used in the original standard. The multitude of modulation and coding combinations means that DVB-S2 can provide quasi-error free operation (packet error rates on the order of 10^{-7}) from signal-to-noise (C/N) levels of 16 dB all the way down to –2.4 dB.

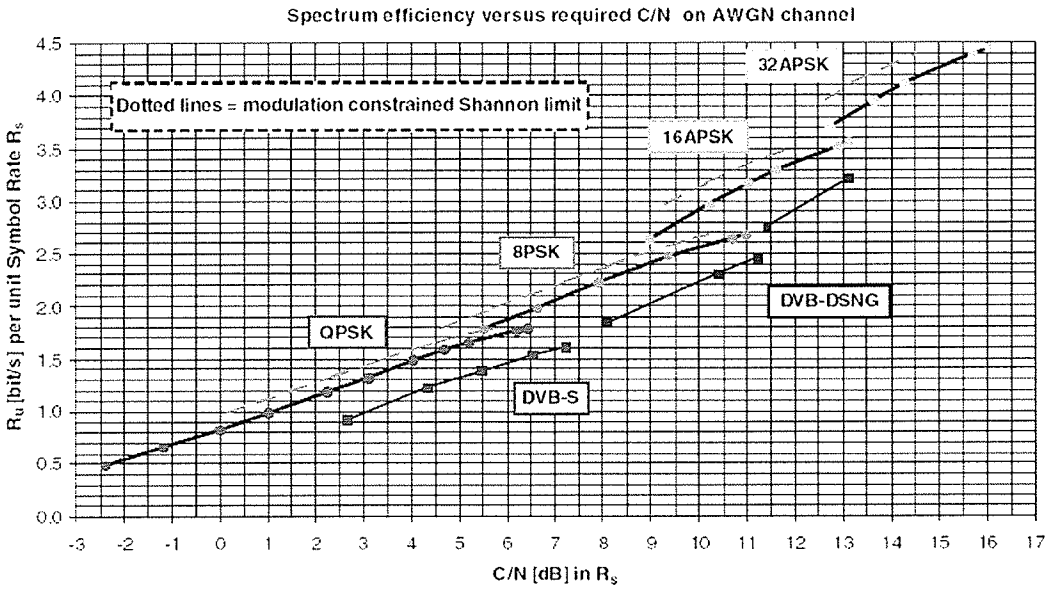

Figure 2.5 - DVB-S2 protection levels [MoRe04]. [1]

The DVB-S2 framing structure allows for different protection levels to be used for different client terminals. Data is segmented into fixed length physical layer frames with payloads of either 64,800 or 16,200 bits. Each frame contains a header that tells the receiver what modulation and coding method was used so that it can be correctly decoded. Figure 2.6 shows that frames are time-division multiplexed within the channel. While the header in each frame is coded with a low rate 7/64 block code, the payload in adjacent frames need not use the same modulation and coding scheme.

Since client terminals can be distributed over a large geographic area within the satellite footprint, it follows that they may be experiencing vastly different channel conditions. Local weather patterns and signal attenuation result in different bit error rates seen at each client terminal. In DVB-S, this meant that the coding scheme chosen would have to be selected to provide protection to the client experiencing the worst conditions, at the worst imagined time [MoRe04]. However, DVB-S2 allows the operator to choose the

[1] Source: "DVB-S2, the second generation standard for satellite broadcasting and unicasting", Alberto Morello and Ulrich Reimers. 2004. Copyright John Wiley & Sons Limited. Reproduced with permission.

optimum protection level for each terminal, and thus will result in better use of the channel.

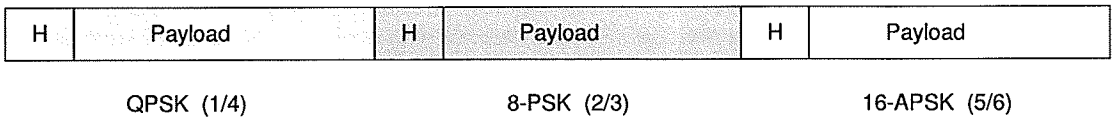

Figure 2.6 - DVB-S2 Physical Framing.

In addition, DVB-S2 can also employ adaptive error correction for each individual client. Since the protection level is specified in the header of each physical frame, it can change at any time without requiring configuration changes at the client terminal. This means DVB-S2 can provide not only variations in protection level over geographical space, but can also vary the chosen levels over time to match the prevailing state of the channel. In DVB-S2, this concept is called adaptive coding and modulation (ACM).

As seen in Figure 2.7, ACM uses feedback from client terminals via a return channel to allow it to decide which protection level to use. To reduce return traffic, client terminals may choose to send reports only when a change in protection level is needed [MoMi04]. The client terminal is responsible for estimating the state of the channel at any given time to give the sender as much information as possible so that it can select the correct protection level. As such, the ability to accurately estimate the state of the current channel conditions becomes an important factor.

To aid in channel estimation, DVB-S2 incorporates the use of pilot symbols that are embedded in the stream. These symbols can be used to determine the signal-to-noise ratio at the receiving terminal. It has been shown that by using these pilot symbols, the channel error performance can be estimated accurately enough to avoid packet loss during slow channel fades, while losing only 0.4% of packets during faster fades [CiGaRi]. The algorithm sacrifices some transmission efficiency by erring on the side of caution when choosing which protection level to use.

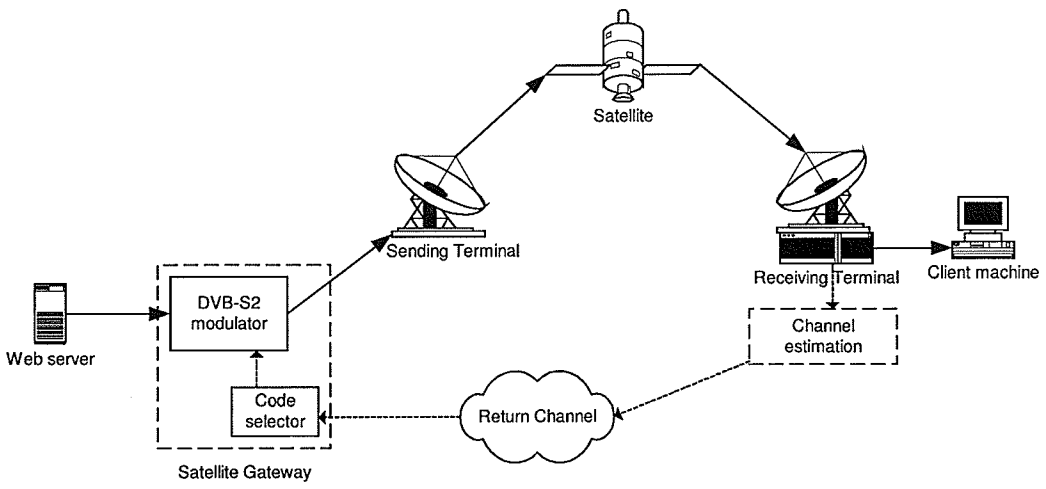

Figure 2.7 - DVB-S2 ACM Operation.

In addition to channel estimation accuracy, the time it takes for messages to reach the sender is also important. This is referred to as the loop delay. Studies of satellite channels have shown that rain fades typically do not cause the channel to change by more than 0.5 – 1.0 dB per second [RiVM04]. Given this, most networks with reasonable round trip times should be able to provide channel estimates in enough time so that losses are minimized.

Although the implementation of a DVB-S2 satellite gateway is operator specific, a number of possible architectures have been proposed for IP services [RiVM04], the selection of which depends on a number of factors such as the type of traffic offered and the quality of service required. Common to many of the schemes is the use of what is called an ACM router. Implemented as part of the satellite gateway, the ACM router is responsible for two things. First, it accepts the channel estimates from the many client terminals and selects the proper protection level for each client. Second, it accepts incoming client traffic and buffers it before it is sent to the DVB-S2 modulator.

Traffic can be sent to the modulator in one of two ways. The first method has traffic entering the modulator using a single input stream. An additional input is used to specify the protection level that data should be encoded with. The second method uses multiple

inputs to the modulator, where each input corresponds to a single protection level. These architectures are shown in Figure 2.8 and Figure 2.9 respectively [RiVM04].

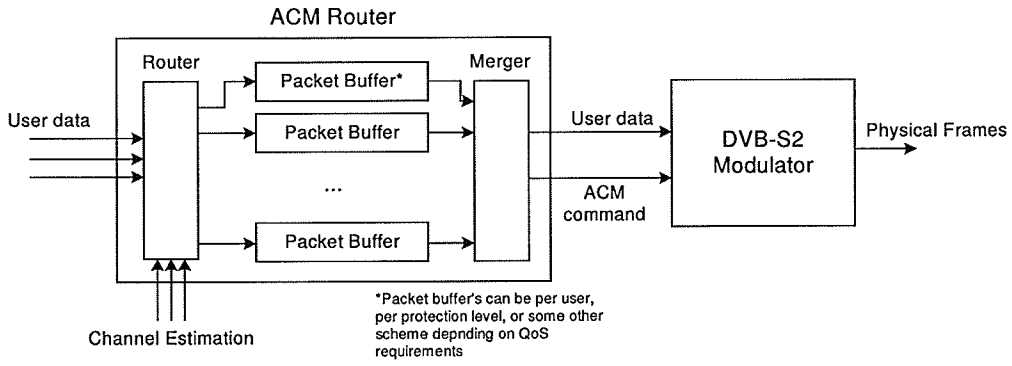

Figure 2.8 - ACM Router with Single Modulator Input.

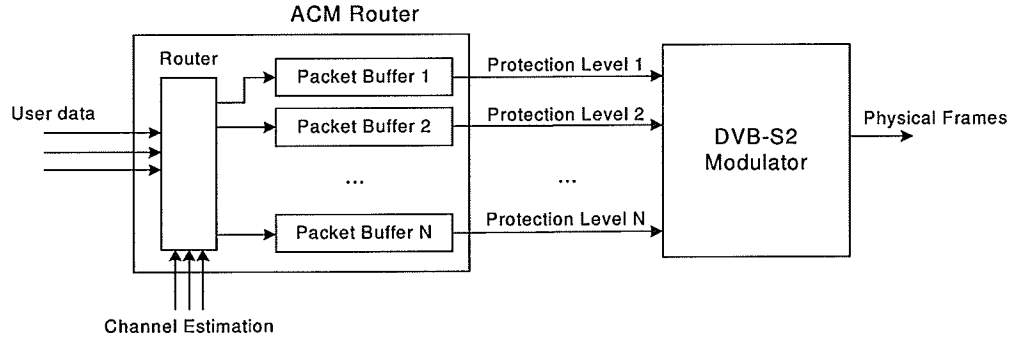

Figure 2.9 - ACM Router with Multiple Modulator Inputs.

The ACM router must ensure that the offered traffic at its output does not exceed the capacity of the link. It is therefore responsible for buffering the incoming IP data as it arrives. As shown in Figure 2.8, possible implementations could include a single input queue per client host, a single queue per protection level, or a combination of both. In either case, since queue space must be finite, the ACM router will drop packets if the offered traffic becomes too high for the modulator to handle.

2.4.3 Potential Problems with Adaptive Error Correction

While the bandwidth savings achieved by DVB-S2's adaptive error correction scheme could potentially reduce the cost of offering Internet access over satellite by a factor of 10 [RiVM04], there are potential problems with simply adding DVB-S2 equipment to existing networks. The main side effect of using adaptive coding and modulation schemes in a satellite environment arises from the fact that the symbol rate of the transponder must remain fixed for the duration of the transmission. This means that a change in the modulation scheme or coding rate for a particular client terminal actually changes the bitrate available to transmit user data. As a result, care must be taken that sending hosts do not overflow the satellite gateway's buffers when the available user bitrate decreases as a result of changing protection levels. While the overall output bitrate of the system may stay relatively constant when averaging the protection levels of thousands of client terminals, individual clients may experience drastic changes in bitrate depending on the quality of service schemes implemented at the gateway.

Flow control could be implemented to prevent congestion, but in many cases client and server machines are quite a distance away from the satellite gateway. This means that communication between sending hosts and DVB-S2 equipment is not always possible. Research into DVB-S2 makes note of this, and touches on some high level solutions to handle congestion. Priority queuing schemes and large buffers to handle temporary peaks are among these [RiVM04]. Protocols such as TCP would be considered ideal candidates for transmission over channels with changing bandwidth, since the protocol can adapt its transmissions rate when congestion occurs.

In many current DVB-S system implementations, however, the use of pure TCP over these channels may no longer be reality. As discussed, performance enhancing proxies are currently deployed in the field that use custom transmission protocols over the satellite link as a way to increase channel utilization. In some cases, congestion control is not implemented since a fixed user bitrate is assumed for the duration of the transmission. As a result, the reduction of available channel bitrate in poor states may have a detrimental effect on these proxies, causing a large amount of congestion at the satellite

gateway. This would decrease channel utilization, seemingly countering the goal of introducing adaptive error correction to the system in the first place. Simply swapping existing DVB-S equipment with newer DVB-S2 equipment is therefore not possible without first studying the effects that this would have on equipment optimized for the older standard.

3. SIMULATION DESIGN AND IMPLEMENTATION

The previous discussions have described how performance enhancing proxies are currently deployed in existing systems to enhance the performance of the TCP protocol over satellite links. With the introduction of ACM in the DVB-S2 standard, proxy equipment designed for DVB-S-based systems may not perform optimally due to reduced user bitrate resulting from adapting the coding rate to the prevailing channel conditions. To study this, a simple performance enhancing proxy is simulated and its resulting behavior is observed and measured when adaptive error correction is introduced. The performance enhancing proxy designed in this thesis is called SimplePEP.

3.1 SimplePEP

SimplePEP is a connection splitting proxy similar in concept to those described in 2.3.3. SimplePEP is symmetric, with a proxy sitting at each end of the satellite link. The proxy sitting at the sending side of the satellite channel is called the *server proxy,* and the one at the client side of the channel the *client proxy.* By terminating the TCP connection at either end of the satellite link, SimplePEP can take advantage of local acknowledgments and retransmissions and sends data reliably over the link using its own custom protocol. A simple network incorporating SimplePEP is shown in Figure 3.1.

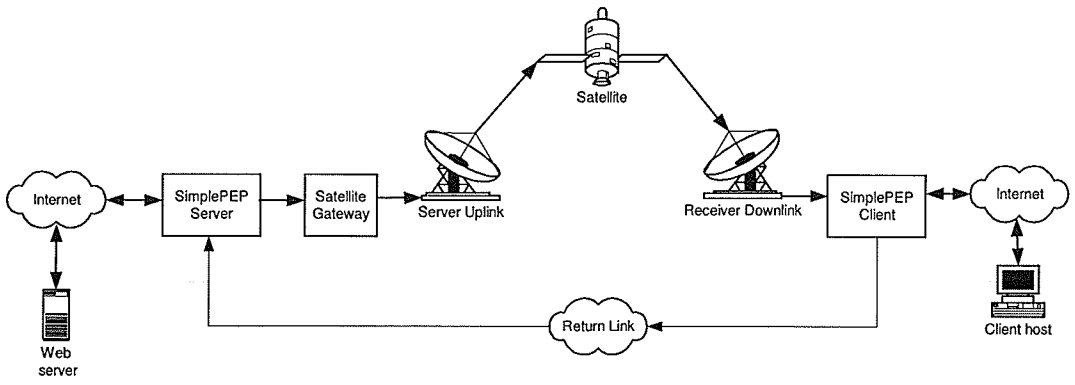

Figure 3.1 - SimplePEP Network Configuration.

When a TCP connection is established between a client host and a web server on opposite sides of the satellite link, SimplePEP intercepts that TCP connection and creates three separate connections instead. One TCP connection is established between the server proxy and the web server, while another is setup between the client proxy and the client host machine. The final connection is created between the two proxies themselves, and is responsible for marshalling transmitted data over the satellite link. As segments flow from the web server over the TCP connection, they are buffered and stored at the server proxy. The server proxy then forwards them over the satellite using a datagram protocol similar to UDP. When segments are received at the client proxy, they are forwarded on to the client host using the second established TCP connection. In practice, a PEP is designed to handle more than a single TCP connection, so segments traveling between the proxies are encapsulated into SimplePEP packets by adding a header that contains information identifying which connection the data belongs to. The contents of this header are discussed in detail in Table 3.1.

In order to ensure the reliable delivery of data, SimplePEP uses an ARQ protocol for error recovery based on the Selective Repeat algorithm described earlier. As Figure 3.1 shows, a feedback channel exists between the client and server proxy over which acknowledgments are sent by the client proxy. The error recovery scheme in SimplePEP was designed to mitigate the effects of both losses due to bit errors and the high round trip time of the link. While another common design goal of performance enhancing proxies is to reduce return traffic, the assumption is made for simplicity that the return link used is a high-bandwidth satellite link, where lost acknowledgements will not be an issue.

Individual packets in SimplePEP are identified using sequence numbers that are unique to each connection. Upon starting the transmission, the server proxy can send as many unacknowledged packets as it wishes up to the pre-defined window size. Once it reaches the end of its window, it must wait for acknowledgements to return before it can slide its window forward and continue transmitting. Since SimplePEP is designed to run over a

satellite link, the window size can be preset according to Equation 2.1 to allow for full utilization of the link.

If the server proxy reaches the end of its window, it will not have any buffer space available to accept additional TCP segments from the web server. To prevent congestion at the server, flow control is accomplished by simply setting the receive window size on the "web server"-"server proxy" TCP connection to zero. This stops further data from arriving from the web server. Once data is emptied from the server proxy's buffers, it can again advertise a larger window to the web server allowing transmission to resume. Since this method prevents the web server's TCP agent from entering slow start, it can resume transmitting at its previous speed. In addition to avoiding congestion at the server proxy, losses at the client proxy are avoided simply by configuring the receiver's buffers to match the size of the server's sending window.

An acknowledgement is sent from the client proxy whenever a data packet is received. ACKs are identified specifically in the SimplePEP header as shown in Table 3.1. When an acknowledgement is sent, two separate pieces of information are sent to the server proxy. The first, *acked_seqno,* is the sequence number that corresponds to the packet that was just received. This tells the server proxy that the packet was transmitted without error, and the server can release it from its buffer. The second field in the acknowledgment is the *seq_no* field. This field identifies to the sender the sequence number of the packet at the left-most side of the receive window. In other words, it is the sequence number of the next packet that must be received before data can be delivered to the client host. If no packets have been lost on the channel, these two numbers will match. However, if a prior packet was lost or received in error, the acknowledged segment cannot be delivered to the client host until all previous segments have also been received.

Table 3.1 - SimplePEP Packet Header.

Name	Size	Description
packet_type	1 byte	Indicates to the proxy what type of packet this is. Possible values are: 0 – DATA 1 – ACK 2 – NAK
seq_no	4 bytes	The sequence number of the packet. For a data packet, this is the sequence number of the segment being sent. For an ACK or NAK packet, this is the sequence number of the next expected segment in the window.
acked_seqno	4 bytes	The sequence number of the segment being acknowledged.
id	4 bytes	The connection identifier.

The SimplePEP server proxy uses two mechanisms to detect when losses have occurred. The first is the use of retransmission timers, similar to those described in TCP. A single retransmission timer is immediately set for each packet that is sent. If an acknowledgment for a packet is not received before the timer expires, the packet is assumed to be lost and it is retransmitted. The second mechanism that is used is negative acknowledgments, or NAKs. If the client proxy receives a packet out of order, it will send a NAK for each packet preceding it that it considers lost to the network. Upon receiving a NAK for a particular packet, the server proxy will retransmit it immediately without waiting for the timer to expire. This is similar in concept to TCP's *fast-retransmit* algorithm, but explicitly allows the server proxy to detect drops without requiring it to

wait for duplicate acknowledgements. The entire process, using a window size of only three packets for illustration, is shown in Figure 3.2.

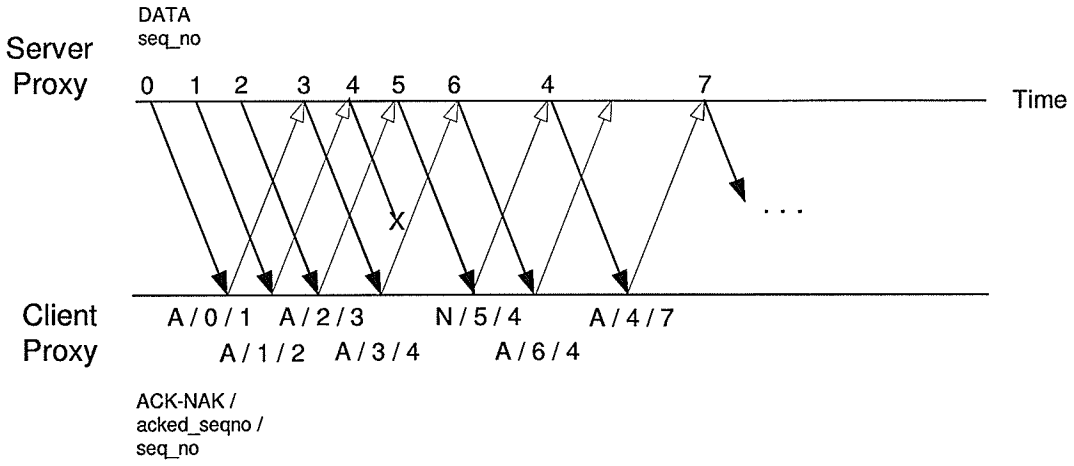

Figure 3.2 - SimplePEP ARQ scheme.

Although SimplePEP uses an ARQ scheme similar in concept to TCP, it has a number of advantages that allow it to perform better in a satellite network. First, since the server and client proxies reside very close to the satellite uplink, the round-trip time and required length of the retransmission timers can be determined ahead of time. This eliminates the need to estimate the latency as TCP does. Since SimplePEP does not perform this estimation, it is important to choose a value that is both short enough to prevent unnecessary idle periods when losses occur, and long enough so that timers do not expire prematurely. Variables such as maximum queuing delay, propagation delay and any other operations that could delay ACKs from reaching the server proxy must all be taken into account when setting the round trip time.

The second advantage SimplePEP has over TCP is that, as mentioned previously, the window size can be pre-set to match the latency of the link, allowing for full utilization of the channel. While TCP grows its window slowly during the *slow-start* and *congestion avoidance* phases, SimplePEP instead begins sending data using the pre-configured window size and does not adjust it during transmission. Since performance enhancing proxies are typically deployed on dedicated networking equipment, there is little concern

that the window size chosen will result in buffer sizes that exceed the available resources of the equipment.

Lastly, SimplePEP has the advantage of being able to assume that losses are not due to congestion on the network. While this may seem unreasonable, many DVB-S systems currently deployed use quality of service measures that split up the available bandwidth on the link into channels, where each channel can guarantee a constant bitrate to the services that run over top of it. It is this aspect of SimplePEP, and indeed other performance enhancing proxies, that could prove problematic when introducing adaptive error correction to the system. For the simulations performed in this thesis, it is assumed that a fixed pipe of 1.5 Mbps has been allocated to SimplePEP traffic.

3.2 Simulation System Description

To study the effects of adaptive error correction, an implementation of SimplePEP that is designed for operation in a DVB-S system is integrated into a DVB-S2-based network. A high-level block diagram of the forward link traffic flow through the simulated DVB-S2 network is shown in Figure 3.3. As TCP segments are sent from the web server through the server proxy they are encapsulated into SimplePEP packets and sent to the satellite gateway. At the satellite gateway, packets are assembled into DVB-S2 physical frames and error correcting bits are applied according to the current protection level. The modulator then translates the frames into symbols based on the current modulation scheme, and sends them over the satellite link, which is assumed to have a delay of 250 ms. At the receiving end, the reverse process is followed until TCP segments are ultimately sent all the way through to the client host. During the transmission, channel estimation information is sent continuously from the receiving satellite gateway over a low bitrate return channel so that the protection level can be adjusted at the transmitting gateway according to the prevailing channel conditions. Lastly, although not shown in Figure 3.3, there also exists a return channel over which acknowledgments from the SimplePEP client proxy are sent. As mentioned earlier, this return link is assumed to be a

high-bandwidth satellite link with enough capacity to handle all returning acknowledgments without congestion occurring.

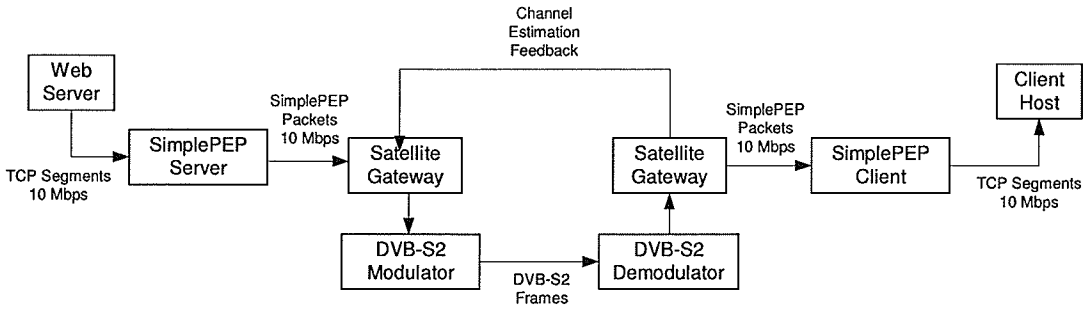

Figure 3.3 - Forward link traffic flow.

3.2.1 Error Model

Packet errors over the satellite link are simulated through the introduction of an error model to the system. In order to study the effects of adaptive error correction, it is necessary that the model vary the bit error rate over time.

While several models have been proposed and studied for satellite channels, the simplest approximation of a time-varying channel is called the Gilbert-Elliot model. The Gilbert-Elliot model simplifies a time-varying channel using a 2-state Markov model [Vuce91]. The link is therefore presumed to be in one of two distinct states at any given point in time: the "good" state, or the "bad" state. The good state is one where the bit error rate is fairly low, while the bad state is used to represent periods where the probability of errors in the received data is higher, due to factors such as inclement weather. While each state can be modeled using a variety of methods, the simplest way is to assign each state a fixed bit-error probability. Transitions from the system's current state to the other state are evaluated at fixed time intervals, where the chance of leaving and entering the bad state when the system is currently in the good state is given fixed probability p, while transitions from the bad to the good states occur with probability q. The state diagram is illustrated in Figure 3.4.

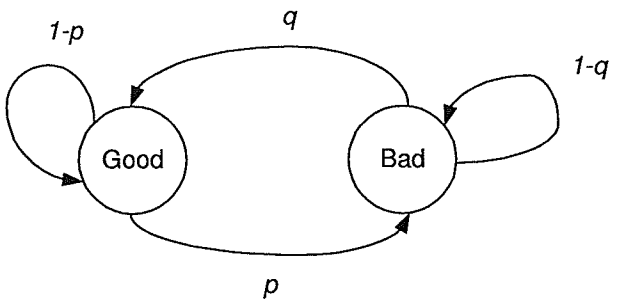

Figure 3.4 - Gilbert-Elliot Error Model.

While this model has been used in various papers to simplify the modeling of a satellite channel, it has been shown that in some cases an n-state Markov model is more suitable [Swee99]. In this thesis, a three-state Markov model has been developed that allows for the channel to vary between three distinct states. Rather than assigning each state a fixed bit-error rate, the states will instead be defined by the received signal level seen at the receiver. The model is not intended to accurately model a particular satellite environment, but allows for sufficient study of the effects of changing protection levels on the SimplePEP protocol.

For each state in the model, it is assumed that a DVB-S2 protection level exists that can provide quasi-error free performance (defined in the DVB-S2 specification as having a bit-error rate of 10^{-7}). Therefore, in addition to a received signal level, each state also represents a particular DVB-S2 protection level, and thus a fixed available user bitrate. The model used here, with its associated transition probabilities, is shown in Figure 3.5. The transition probabilities were chosen so that the channel spends the most time in the good state, where the error rate is low and the available user bitrate is at its maximum. The second, or middle state is considered a transition state, where the channel fades from good to bad. As a result, the time spent in this state is considerably lower. When in the intermediate state, equal probability is assigned to the channel transitioning either back to the good state, or into the bad state.

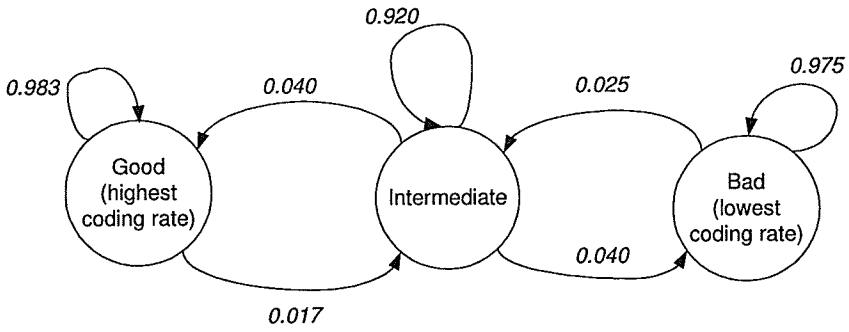

Figure 3.5 - Time Varying Channel Model.

The signal-to-noise in each state is separated by approximately 1 dB from adjacent states. Based on literature stating that rain fades occur at no more than 1 dB per second [RiVM04], transitions in the model are evaluated at one second intervals. Using the associated probabilities, the average length of time in seconds that the system spends in each state before leaving is shown in Equations 3.1, 3.2 and 3.3.

$$L_{good} = \left(\frac{1}{1-0.983}\right) = 58.8 \qquad (3.1)$$

$$L_{intermediate} = \left(\frac{1}{1-0.92}\right) = 12.5 \qquad (3.2)$$

$$L_{bad} = \left(\frac{1}{1-0.975}\right) = 40 \qquad (3.3)$$

In addition to the average length of time the model is in each state before a transition occurs, the steady-state probabilities can also be calculated. To calculate this, Markov model theory is used. A Markov model can be expressed using a transition matrix, where each row represents a state in the model, and each column represents the probability that the system transitions to each of the other states. The state transition matrix for the model in this thesis is shown in Equation 3.4.

$$P = \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \\ P_{31} & P_{32} & P_{33} \end{bmatrix} = \begin{bmatrix} 0.983 & 0.017 & 0 \\ 0.04 & 0.92 & 0.04 \\ 0 & 0.025 & 0.975 \end{bmatrix} \quad (3.4)$$

The steady state of a Markov model gives the probability that the system is in a particular state, independent of the previous state of the system. This relationship is shown in Equation 3.5, where *a1*, *a2*, and *a3* are the probabilities that the system is in the good, intermediate and bad state respectively. When the system is in steady state, Equation 3.5 shows that the values of *a1*, *a2*, and *a3* do not change, even after a transition has occurred.

$$(a_1, a_2, a_3) = (a_1, a_2, a_3) * P \qquad (3.5)$$

$$a_1 + a_2 + a_3 = 1 \quad (3.6)$$

Expanding Equation 3.5 across the matrix gives us a system of three equations that, when combined with Equation 3.6, can be solved to obtain the steady state probabilities.

$$a_1 = P_{11}a_1 + P_{21}a_2 + P_{31}a_3 \quad (3.7)$$

$$a_2 = P_{12}a_1 + P_{22}a_2 + P_{23}a_3 \quad (3.8)$$

$$a_3 = P_{13}a_1 + P_{23}a_2 + P_{33}a_3 \quad (3.9)$$

Using the values for the state transitions in Figure 3.5, the steady state probabilities of a_1=*0.475*, a_2=*0.202,* and a_3=*0.323* are obtained. These values can also be thought of as the percentage of time, on average, that the system is in each state. This is used later to calculate the average throughput possible through the system.

3.2.2 Adaptive Coding and Modulation

As discussed in 2.4.2, DVB-S2 has the ability to use pilot symbols to measure the signal-to-noise level at the receiving terminal. This information is then fed back to the sending gateway, where the appropriate protection level is selected. Since measuring the SNR is not possible in a network simulator, it is assumed that some accurate method of measuring the signal-to-noise ratio at the receiver exists. Further, the assumption is made that changes in the received signal level can be detected fast enough so that the protection level can be changed before bit errors occur. This means that when ACM is enabled in the system, the actual packet error rate in each state of the error model will be 10^{-7}, or what the DVB-S2 specification calls quasi-error free.

As mentioned, for each state in the Markov model there exists a corresponding protection level that protects the system from bit errors. The signal levels and the corresponding coding and modulation rates for each state are selected from the DVB-S2 specification, and are detailed below in Table 3.2. Using the spectral efficiency numbers taken directly from the DVB-S2 specification [ETSI05], the corresponding user bitrate for each state is also calculated. For the system implemented in this thesis, it is assumed that the 1.5 Mbps channel is guaranteed only when the system is in the good state. Thus, the available user bitrate begins at the full rate of 1.5 Mbps and decreases from there according to the spectral efficiency of each protection level.

Table 3.2 - Selected Protection Levels.

State	E_s/N_o (dB)	Protection Level	Spectral Efficiency	Spectral Efficiency compared to *good* state	Available User Bitrate
good	10.21	16-APSK 3/4	2.966728	1	1.50 Mbps
intermediate	9.35	8-PSK 5/6	2.478562	0.835 ≈ 5/6	1.25 Mbps
bad	7.91	8-PSK 3/4	2.228124	0.751 ≈ 3/4	1.125 Mbps

One final bit of information can be derived using what is known of both the selected protection levels and the error model. Since the probability of the system being in each state has been calculated, and the available user bitrate in each state is known, the overall steady-state user bitrate of the system can be calculated using Equation 3.10.

$$AverageBitrate = Bitrate_{good}a_1 + Bitrate_{\text{int}ermediate}a_2 + Bitrate_{bad}a_3 \quad (3.10)$$

Using the derived probabilities from the previous section, the steady state throughput of the system, or the long-term average available user bitrate available to SimplePEP traffic, is 1.33 Mbps.

3.3 ns-2 Simulator

The results in this thesis are obtained through simulation by making use of the ns-2 simulator. The ns-2 simulator is a discrete-time, event driven network simulator. Its open source nature has made the simulator a popular choice for network research, and it is constantly growing due to contributions from researchers. Numerous modules have been developed for the simulator, implementing a variety of different networking protocols such as TCP, routing protocols and wireless networks. The ns-2 simulator is used extensively in many of the papers cited in this thesis. The simulator can be downloaded freely and compiled for a variety of platforms [NS2Sim].

The simulator operates by using two separate programming technologies: C++ and Object Tcl (OTcl). The first portion of ns-2 is written in the C++ programming language. Since this code is optimized and compiled for the target platform, it can run very quickly. As a result, it forms the core of the simulator, and is where packet and queuing operations are calculated. New protocols that are added to the simulator are typically implemented in C++. The second part of the simulator makes use of OTcl, a common scripting language. Scripted programs are not compiled ahead of time, but are instead interpreted as they are run. The ns-2 simulator takes advantage of this feature by using OTcl scripts as an interface whereby the user can setup the various parameters of the simulation. Anything

from the topology of the network to the operating parameters of the protocols used can be configured using an OTcl script. When a simulation is performed, the ns-2 simulator will read and interpret the OTcl script to generate the results.

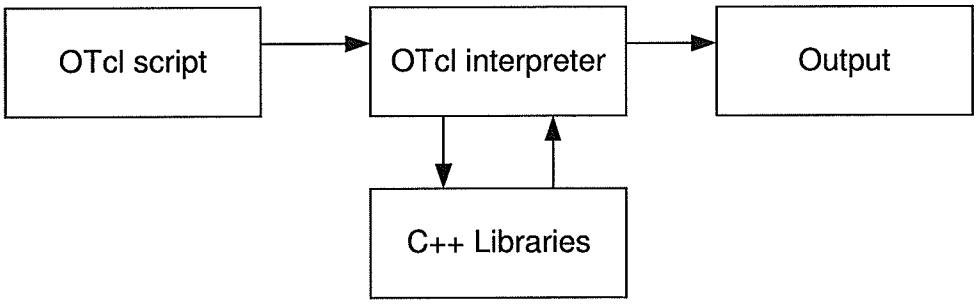

Figure 3.6 - ns-2 simulator.

A typical simulation topology in ns-2 consists of endpoints, or nodes, that are connected via links. A node could represent a host or router, while a link models the network path between them. The bandwidth, propagation delay, queue size and error rate of a link can be specified using OTcl commands. At each node, one or more agents are attached. An agent typically represents a particular protocol or application that is being simulated. An example network topology is shown in Figure 3.7 to illustrate this more clearly. When a simulation is performed, detailed statistics such as dropped packets and average queue sizes can be obtained using the built-in tracing and monitoring interfaces.

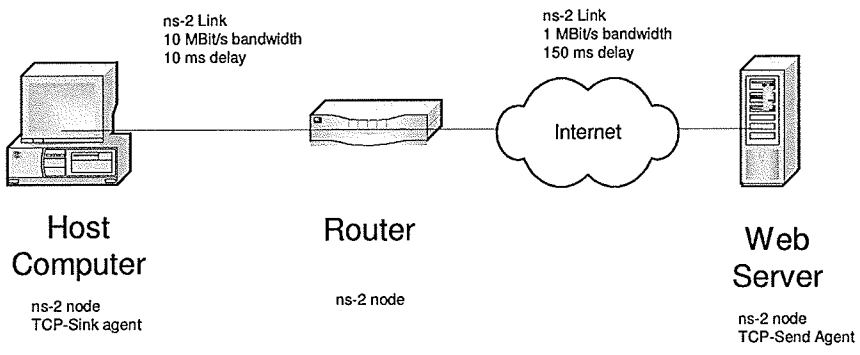

Figure 3.7 - Simple network topology in ns-2.

3.3.1 Implementation Considerations

The network topology simulated in this thesis is similar to the traffic flow diagram shown in Figure 3.3. In order to simulate the SimplePEP protocol, a new ns-2 agent was created in C++. The listing for this agent is included as Appendix 7.2. The source code for the SimplePEP agent includes the time-varying error model described earlier. When the simulation is run, the Markov error model is evaluated every second to simulate the transitions into different channel states. As packets are sent over the link from the SimplePEP server, they are passed to the error-model module where error correction is added according to the current state of the channel. Packets are then sent through the link that connects the SimplePEP server and client proxies. The link itself is given a 250 ms one-way delay and assigned a bandwidth of 1.5 Mbps, according to the system description outlined in section 3.1. Congestion at the satellite gateway (the ACM router) is simulated using a simple Drop-Tail queue attached to the link at the server side of the connection. In a Drop-Tail queue, the last packet added to the queue is dropped when the buffer becomes full. Since the SimplePEP server will be set up with a constant output bitrate of 1.5 Mbps to match the capacity of the channel, congestion will result when the error correction added to outgoing packets decreases the available user bitrate according to Table 3.2.

The only portions of Figure 3.3 that are not physically included in the simulation are the TCP client and server. The TCP implementation in ns-2 does not allow for dynamic advertised receive window sizes, instead simplifying things by using a fixed receive window that is specified once at run time. This means that the SimplePEP server cannot stop traffic from flowing from the TCP server by setting *rwnd* to zero in the case that its buffers become full. This would result in congestion on the TCP connection, causing the TCP server to employ its congestion control measures. As a result, rather than the TCP server being included in the simulation, the SimplePEP server implementation assumes that there is a constant flow of data available from the web server.

Through bindings between the C++ code and OTcl scripts, SimplePEP allows for a number of parameters to be configured before a simulation is run. These are summarized below in Table 3.3.

Table 3.3 – Configurable SimplePEP Parameters

sendrate	The rate at which the SimplePEP server should transmit packets over the link. This is always set to 1.5 Mbps in this thesis. Rather than sending packets out in short bursts, SimplePEP will attempt to space packets as evenly as possible according to the desired sending rate.
blocksize	The unit packet size without error correction. This is fixed at 1500 bytes in this thesis.
packetlimit	The number of packets to send before terminating the transmission. This can also be thought of as the file size that is sent over the link.
windowsize	The sending and receiving window size, in packets. This must be set to the same value for both the server and client agent.

4. PERFORMANCE EVALUATION IN A DVB-S2 NETWORK

This section describes each of the simulations run using the model previously described, and provides analysis of the results obtained. As mentioned earlier, individual simulations are configured and performed using OTcl scripts, a sample of which is included in Appendix 7.3.

Each simulation involves the transmission of a single 50,000-packet file across the link, where each packet is 1500 bytes in length. To benchmark the results, the total time taken to transfer the file is measured so that the throughput can be calculated. The throughput is defined simply as the total file size divided by the total transfer time. This calculation omits retransmitted packets and overhead from error correcting bits, and represents the actual performance of the connection as it relates to the transport of user data. Since the error model changes from state to state randomly, the system will experience varying conditions across simulations trials. To account for this, results obtained are actually the average of multiple trials. In addition, a metric called the *transmission efficiency* is used to compare the results of various simulations. The transmission efficiency is simply the resulting throughput divided by the actual available user bitrate during the length of the transmission. In other words, the transmission efficiency indicates what percentage of the available bitrate was actually used for user data, versus bandwidth that was either left idle or used for retransmitting lost packets.

4.1 Comparison of SimplePEP to TCP

Before SimplePEP can be evaluated in an adaptive error correction environment, its performance must first be evaluated against that of TCP to ensure that it actually provides improvement over TCP in a satellite network. Since SimplePEP is designed for use in a DVB-S system where there is no adaptive error correction, the adaptive model described in section 3.2.1 is not used in this benchmark test. Instead, a simple uniform packet-error model already built into the ns-2 simulator is used to simulate losses resulting from

packet errors on the network. The average throughput of the two protocols is compared at varying packet error rates and is evaluated using two different maximum window sizes. The first window size used is 40 packets, corresponding to 64 KB of data, which is the largest setting possible in TCP without using the window scaling option described in section 2.2.1. The second window size used is 63 packets, allowing for full utilization of the link according to Equation 2.1. The results obtained from transmitting a file across the simulated satellite link at varying packet error rates are shown in Figure 4.1.

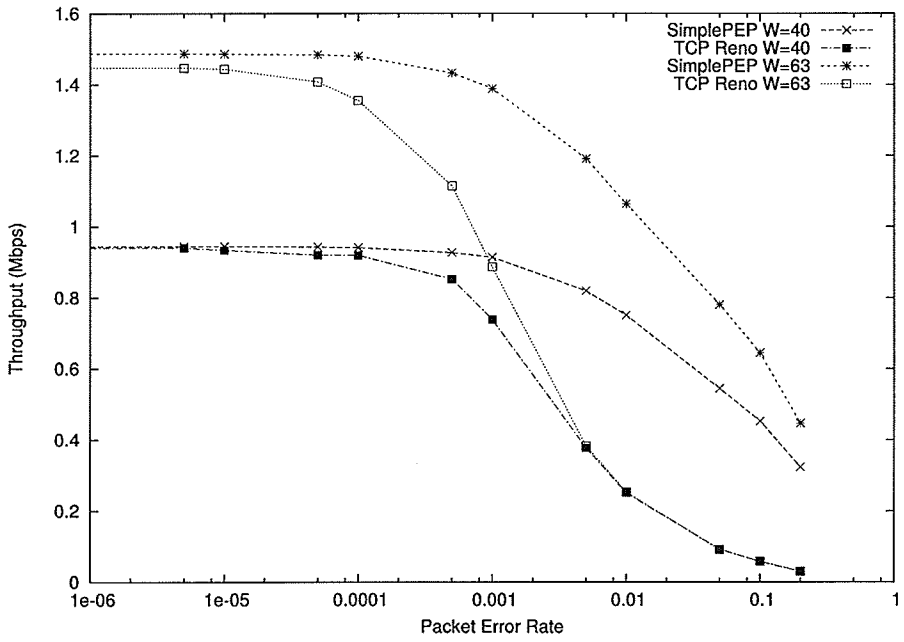

Figure 4.1 - SimplePEP and TCP comparison.

From Figure 4.1, it can be seen that, when the bit error rate is low, SimplePEP and TCP perform quite similarly. Although SimplePEP does not need to grow its window at the start of the transmission as TCP does, the resulting performance gain is made insignificant over the transmission of a long file. However, when the error rate begins to grow, SimplePEP outperforms TCP considerably. For example, when comparing the two protocols with a 63-packet window size, SimplePEP exhibits a throughput that is over 1.5 times that of TCP when the error rate of the channel is 10^{-3}, and over 4 times that of TCP when the error rate is 10^{-2}. When the sender detects errors, the TCP agent assumes that

congestion has occurred and drops its rate of transmission. Conversely, although the SimplePEP server proxy must also wait for packets to be retransmitted, it does not reduce its window size and can instead resume sending at its pre-configured output bitrate. Once the error rate begins to approach 1 in every 10 packets, SimplePEP is still capable of providing some amount of throughput while TCP almost comes to a halt entirely.

4.1.1 Choosing the SimplePEP Window Size

It is interesting to note that SimplePEP appears to perform much better with a 63-packet window size than it does when using only 40 packets. While this can be largely attributed to the poor utilization of the channel as shown in Equation 2.1, it is worth further study to see if an even larger window size can improve the performance of the protocol. The same file is transmitted over the channel, as above. However, this time the performance using a window size of 75, 150 and 200 packets is compared to the 63-packet window size. The results are again compared at different packet error rates and are shown in Figure 4.2.

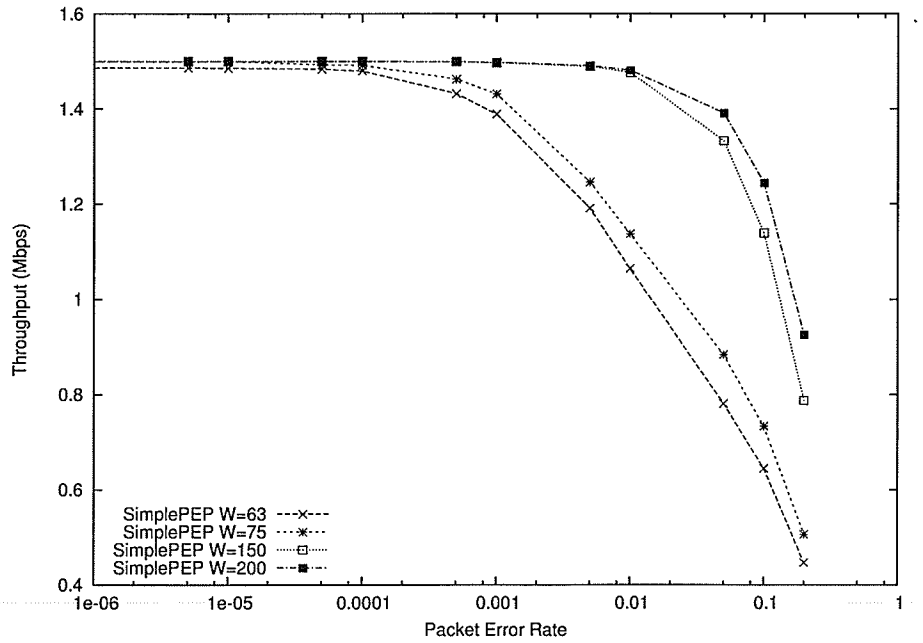

Figure 4.2 - Choosing SimplePEP Window Size.

When the error rate is low, it is evident that window sizes above 63 do not provide any performance advantage. However, as larger amounts of packet errors begin to occur, a larger window size appears to improve the throughput of the protocol. This can be explained by taking a close look at the ARQ protocol used by SimplePEP. When a packet error occurs, the SimplePEP server must retransmit it and wait for an acknowledgement before it can slide its transmission window forward. In the case that the window size is small, the server could reach the end of its window before an acknowledgement for the retransmitted packet arrives, forcing it to wait before it can continue to send packets. On the other hand, larger window sizes allow the server to continue transmitting while it waits for the acknowledgement, rather than sitting idle. For windows sizes of 150 and 200 packets, dropped packets have little to no effect on the throughput until the packet error rate reaches 10^{-2}. Based on this finding, a window size of 150 packets is used for the remaining simulations.

4.2 SimplePEP and Adaptive Error Correction

The adaptive error correcting model described in 3.2.2 is now introduced to the system to study what effect, if any, the model has on the throughput of the SimplePEP protocol. As discussed in section 3.2.2, the packet error rate in each state is fixed at 10^{-7}, which corresponds to the quasi-error free performance as specified in the DVB-S2 standard. This means that almost all lost packets in the system will be a result of congestion at the satellite gateway, and very few packets will be lost to bit errors. The same 50,000-packet file is sent over the channel from the SimplePEP server proxy to the client proxy. Since the adaptive error correcting model has been enabled, packets sent through the system are no longer only 1500 bytes in length. Rather, they are increased in size according to the current protection level of the error model.

The results of ten simulations and their averages are shown in Table 4.1. To calculate the transmission efficiency of each trial, the available user bitrate over the course of the transmission is calculated. This is done by simply recording the percentage of time that the system spends in each state during the length of the simulation, and multiplying it by

the corresponding available user bitrate in that state according to Table 3.2. Note that the observed values are similar to the theoretical value of 1.33 Mbps calculated earlier.

Table 4.1 - Efficiency of SimplePEP with Adaptive FEC

	Time (s)	Throughput (Mbps)	Available User Bitrate (Mbps)	Efficiency
1	485.42	1.24	1.34	0.92
2	468.17	1.28	1.38	0.93
3	465.52	1.29	1.40	0.92
4	490.17	1.22	1.36	0.90
5	474.09	1.27	1.38	0.92
6	479.44	1.25	1.36	0.92
7	550.18	1.09	1.27	0.86
8	497.60	1.21	1.35	0.89
9	443.48	1.35	1.43	0.95
10	610.02	0.98	1.21	0.81
Avg.	**496.41**	**1.22**	**1.35**	**0.90**

As the results indicate, SimplePEP fails to take full advantage of the link capacity when adaptive error correction is introduced. While the average efficiency of the 10 trials was 90%, some trials showed channel utilization as low as 81%. To better understand what is causing the lower utilization, a plot of the average instantaneous throughput of SimplePEP versus time is shown in Figure 4.3. Also shown on the same graph is the user bitrate that should be attainable according to the state of the channel at that point in time. As Figure 4.3 shows, when the channel is in the good state, SimplePEP uses the channel efficiently, sending at the configured value of 1.5 Mbps. However, once the available bitrate of the channel drops due to the addition of parity bits in the poor channel states, the performance of the protocol suffers greatly as congestion begins to occur at the satellite gateway. Close inspection of the graph shows that during the bad state of the channel, the average throughput of the protocol is on average only 0.85 Mbps, even

though the channel could, in theory, accommodate a constant transmission rate of 1.125 Mbps.

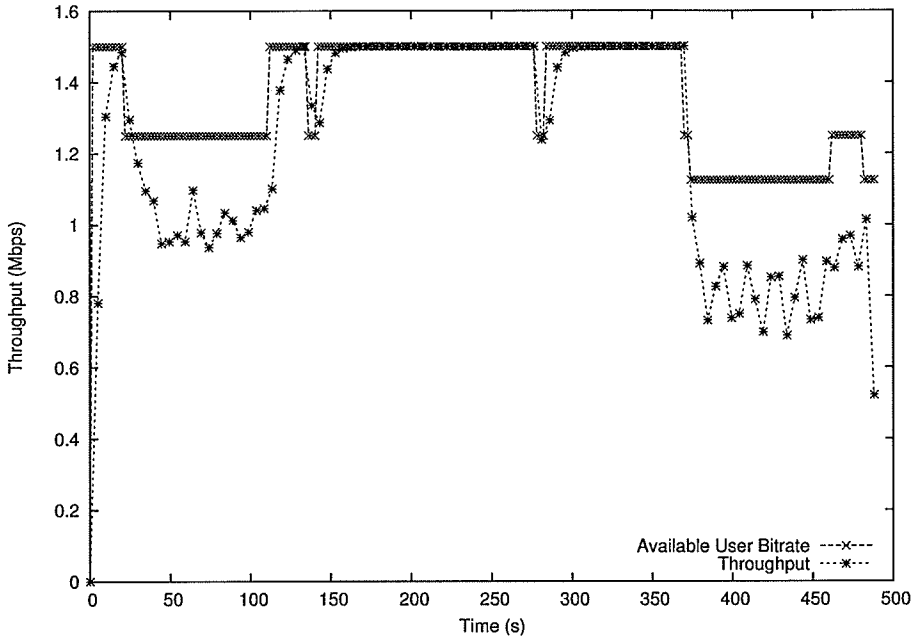

Figure 4.3 - SimplePEP with Adaptive FEC Throughput vs. Time.

4.3 Possible Improvements to the Protocol

4.3.1 Varying the Output Bitrate

The performance in each of the intermediate and bad channel states could be improved by lowering the server's output bitrate to the point where congestion no longer occurs. For example, if the server was configured to send data at 1.125 Mbps, it could maintain this rate steadily throughout the transmission, regardless of the channel's current state. However, setting the bitrate of the server to a lower value means that channel utilization will be poor when in the good state, since in theory the average available bitrate on the link is 1.33 Mbps. For shorter transmissions, it may be desirable to take this approach, particularly if the entire transmission were to take place during times of poor channel conditions. However, if the long-term average utilization of the system is more important than the short-term transmission rate, it may be possible to find an optimum output bitrate

where lower utilization in good channel conditions is offset by reduced congestion in the poor states. To study this, the throughput of the transmission is observed using a variety of different output server bitrates, and the results are plotted in Figure 4.4.

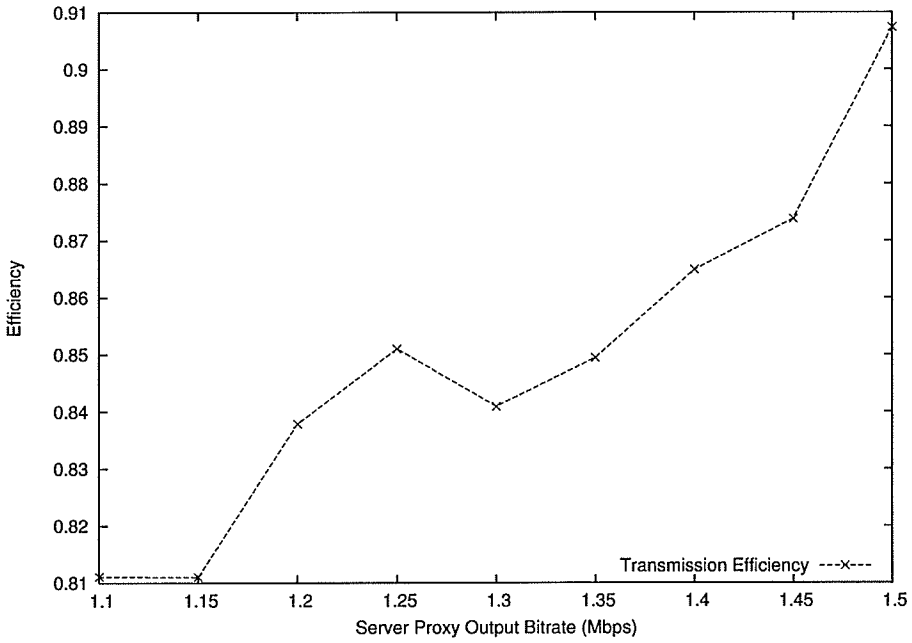

Figure 4.4 - Varying the Output Bitrate.

The plot shows that reducing the server's bitrate appears to provide no benefit to the average channel utilization when transferring a large file. The increase in throughput during the good channel states seems to more than make up for the decrease in efficiency that occurs during the two noisy states. The only exception to this rule appears to be when the output bitrate at the server is set 1.25 Mbps, which is the point where congestion in the intermediate state no longer occurs. While this observation is true for the model implemented here, it may not be so for a systems where the channel spends more time in poor channel conditions.

4.3.2 Satellite Gateway Queue Size

In addition to the output bitrate of the performance enhancing proxy, satellite operators may have control over the queue size at the satellite gateway. In this case, the queues referred to here are located in the packet buffers at the ACM router. As mentioned in section 2.4.2, the ACM router buffers packets prior to the application of error correction by the modulator. When the protection level changes, packets are emptied from these queues at a rate that corresponds to the available user bitrate resulting from the selected protection level. If the queues are emptied too slowly, congestion occurs and packets are dropped from the back of the queue. In previous simulations, a default queue size of 10 SimplePEP packets was used. However, since only one full window of packets can be outstanding at any given time, a queue size large enough to hold an entire window of packets should be sufficient to prevent congestion. Unfortunately, this may not always be possible if resources on the gateway equipment are limited. Benefits could result, though, from an increased queue size that is somewhere between these two extremes. To study this, the output bitrate of the SimplePEP server is fixed at 1.5 Mbps, and the transmission efficiency, or channel utilization, is compared over a range of queue sizes.

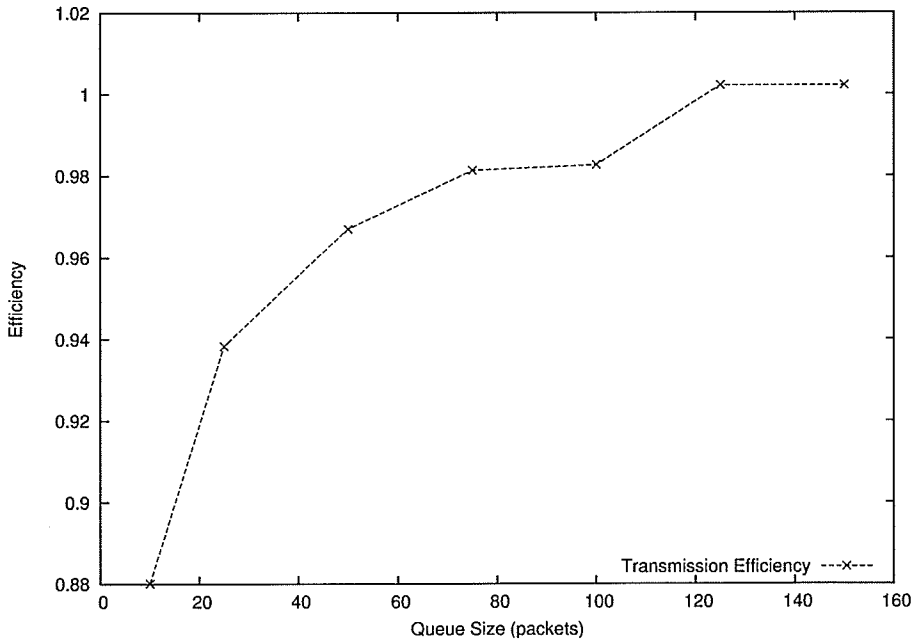

Figure 4.5 - Efficiency Versus Queue Size.

Figure 4.5 shows a steady increase in efficiency as the queue size is increased. This can be directly attributed to the fact that a larger queue will take longer to overflow during periods of poor channel conditions, thus reducing the number of dropped packets detected by the SimplePEP server. As expected, 100% efficiency occurs when the queue size is set to 150 packets, or the server's sending window size. It is interesting to note however, that 100% efficiency also appears to be achieved at some value less than that.

It turns out that it is possible to calculate the ideal queue size to use so that congestion does not occur. To study this, a number of variables are first defined. The average input rate of packets entering the queue is the rate at which the server proxy is sending packets. This will be called R_{in}, which in this case is 1.5 Mbps, or 125 packets/second. The rate at which packets exit the queue will be called R_{out}. Since the goal is to size the queue to avoid dropped packets during worst-case channel conditions, R_{out} will be set to the rate at which packets exit the queue in the bad channel state, which according to Table 3.2 is 1.125 Mbps, or 93.75 packets/second. Therefore, the rate at which the queue grows can be determined using Equation 4.1. Using the numbers provided, the resulting rate at which the queue fills up is 31.25 packets per second.

$$R_{queue} = R_{in} - R_{out} \quad (4.1)$$

Assuming that no other delays are imposed on acknowledgments as they return from the client proxy, the rate of ACKs returning to the server proxy should be equal to the rate that packets are exiting the queue (R_{out}). If the server proxy is forced to wait for an acknowledgement to arrive before it can send the next data packet, new packets will be clocked out at the same rate that ACKs are arriving, essentially adapting the transmission rate to the output rate of the queue. Since the sending window effectively shrinks and grows as packets leave the server and new acknowledgements arrive, the point in time when the server is forced to wait will occur when the window size shrinks to zero.

At the start of transmission, the available window size, W, is 150 packets. For every packet that is sent by the sender, W is reduced by one packet. Conversely, every arriving

acknowledgement allows the available window size to increase by 1. Since data is initially sent at 125 packets/second, it follows that the rate at which the window decreases is equal to R_{in}. Similarly, the rate at which the window is increased matches that of arriving acknowledgments, or R_{out}. Using this information, a formula for the window size as a function of time can be derived. This is shown in Equation 4.1, where RTT corresponds to a single round-trip time, or the point at which the first acknowledgement arrives from the receiver. As discussed, the RTT in the system in this thesis is 500 ms.

$$W(t) = 150 - R_{in}*t + R_{out}*(t-RTT) \quad (4.2)$$

Setting the window size in Equation 4.2 to 0, and plugging in the remaining values results in Equation 4.3.

$$W(t) = 0 = 150 - 125*t + 93.75*(t-0.5) \quad (4.3)$$

Solving the equation, it is found that the transmission rate of the SimplePEP server begins to be affected by arriving acknowledgments after 3.3 seconds. Given the rate at which the queue is filled according to Equation 4.1, the number of packets that will be in the queue after 3.3 seconds can be found.

$$Q(t) = R_{queue} * t = (31.25)*(3.3) = 104 \text{ packets} \quad (4.4)$$

Rounding the obtained value to the nearest packet shows that a queue size of 104 packets should eliminate congestion, even if the output bitrate of the server is set to its maximum rate of 1.5 Mbps. In practice, simulation shows that the actual value is 105 packets. The discrepancy can be attributed to rounding errors that occur when calculating the transmission rate at the SimplePEP server.

As Figure 4.6 shows, when the channel bitrate changes due to additional error correcting bits being added, a queue size of 105 packets allows the SimplePEP server to adapt its

sending rate based on the rate of returning acknowledgements, resulting in no dropped packets at the satellite router.

Note that while a window size of 150 packets is used here, this is actually not necessary if the queue size is adjusted properly. As shown in Figure 4.2, when the packet error rate is low, a large window size provides no performance improvement over a window size that is tuned to simply take advantage of the long delay of the link. Therefore, if it is assumed that no dropped packets occur due to congestion, the window size can be adjusted to 63 packets without a corresponding drop in performance. Using Equation 4.2, 100% efficiency can then be achieved in the system with a queue size of only 17 packets.

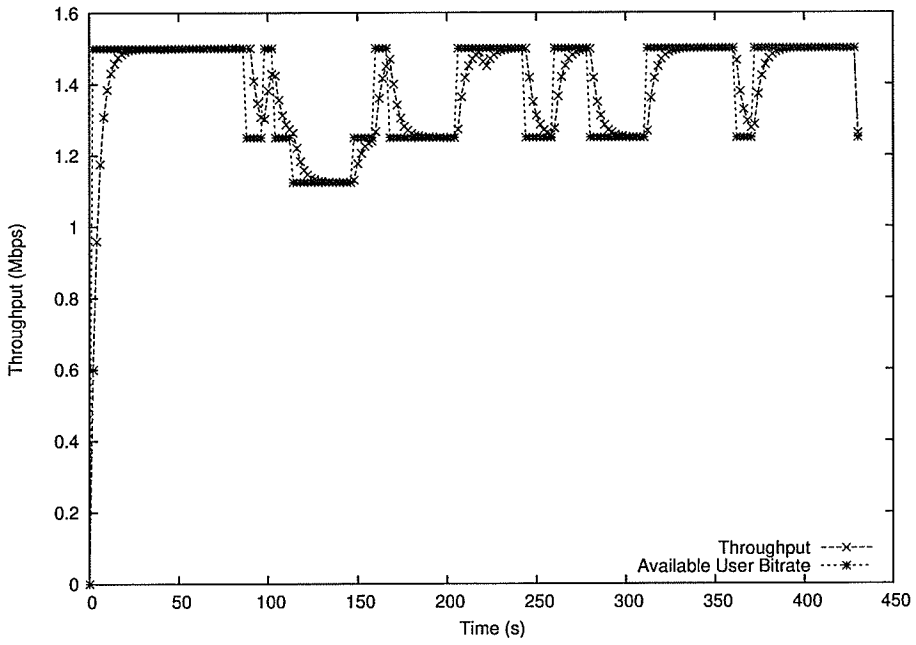

Figure 4.6 - Queue Size 105 Throughput vs. Time.

Although modifying the queue size is relatively straightforward, other factors may make this method unfeasible. First, if there are thousands of clients sending traffic through the satellite gateway, it would most likely not be possible to size each queue large enough to guarantee that no packets are lost when peak traffic is offered as a result of the limited available resources. Secondly, care must be taken when sizing queues to ensure that the

queuing delay does not add to the loop delay seen by the adaptive error correction process. If packets are stored in queues on a per protection level basis, as is the case in Figure 2.9, it is possible that in the time it takes a user packet to exit a queue, the channel state may have changed so that the chosen protection is longer adequate, resulting in packet errors at the client terminal.

4.3.3 Parallel TCP Flows

If it is not possible to modify the queue size, due to either limited resources on the satellite gateway or simply because of the inability to modify the DVB-S2 equipment parameters, there is another way to take advantage of the changes in bandwidth. As discussed in section 2.1, TCP adapts to changing bandwidth conditions through its congestion control algorithms. The option of introducing a secondary TCP flow to the system running in parallel to the SimplePEP traffic is therefore proposed. The TCP flow could be referred to as opportunistic. If extra bandwidth is available due to the channel being in the good state, the extra TCP flow can take advantage by filling the vacancy. Conversely, if the channel conditions are poor and SimplePEP is using up all available resources, the transmission rate of the secondary connection can be reduced accordingly. While SimplePEP traffic itself would need to be sent at a lower rate for the duration of the transmission, the aggregate channel utilization between the two flows may provide better efficiency than what SimplePEP alone could provide.

To simulate this scenario, SimplePEP is setup to transmit at an output bitrate of only 1.125 Mbps, corresponding to the available bitrate during the bad state. A TCP sender and receiving pair are also added to the system to simulate the opportunistic flow. The resulting throughput of the two flows and the aggregate throughput are shown in Figure 4.7.

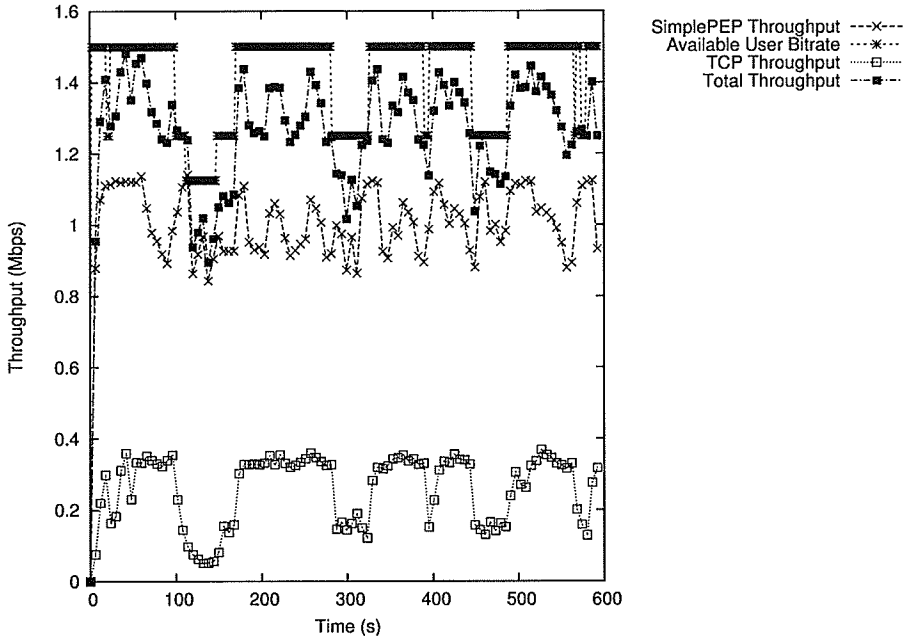

Figure 4.7 - TCP Opportunistic Throughput vs. Time.

The results show that while the TCP flow increases and decreases its bitrate according to the state of the channel, the performance of SimplePEP seems to fluctuate as well. It appears as though SimplePEP is still experiencing dropped packets as a result of congestion, and is not able to transmit at a steady rate of 1.125 Mbps. Instead, it is achieving an average throughput of less than 1 Mbps. This can be explained by the fact that a simple Drop-tail queue is being used at the satellite gateway, where packets are dropped with equal probability with no consideration given to the source of the traffic. As TCP grows its window during *slow-start* and *congestion avoidance*, it eventually creates congestion on the link, resulting in SimplePEP packets being dropped at the gateway. Analysis shows that the idea of introducing a competing TCP flow actually reduced overall transmission efficiency to 89%, below what was even achievable using only an unmodified SimplePEP connection.

Increased performance could likely be achieved if the queue were configured to drop TCP traffic with a higher probability than SimplePEP traffic. This would allow the SimplePEP flow to achieve a constant 1.125 Mbps for the duration of the transmission,

while only dropping the lower priority TCP segments if congestion occurred. To test this theory a simple priority queuing scheme in ns-2 called PriQueue is used. The PriQueue concept is similar in principle to a Drop-Tail queue, but with one major exception. If a packet arrives on a flow that is considered higher priority, it is inserted at the front of the queue. As a result, when the queue overflows, the lower priority traffic at the back of the queue will be dropped first. Thus, SimplePEP traffic is labeled in the simulation as high priority. The results of the simulations are compared to a lone SimplePEP connection, and the Drop-Tail implementation, and are presented in Table 4.2.

Table 4.2 – SimplePEP with Opportunistic TCP Comparison.

	SimplePEP	**SimplePEP w/ TCP**	**SimplePEP w/ TCP and PriQueue**
Average Transfer Time	496.41 s	600.56 s	533.22 s
SimplePEP Throughput	1.22 Mbps	1.00 Mbps	1.125 Mbps
TCP Throughput	-	0.22 Mbps	0.21 Mbps
Total Throughput	1.22 Mbps	1.22 Mbps	1.335 Mbps
Available User Bitrate	1.35 Mbps	1.37 Mbps	1.35 Mbps
Efficiency	90%	89 %	99 %

As the numbers indicate, the priority queuing scheme produces the desired effect. In fact, the combination of a single TCP flow and the lower rate SimplePEP traffic is able to produce a transmission efficiency greater than 99%. To visualize this result, a plot of the throughput of both connections, as well as the aggregate throughput, is shown in Figure 4.8. As the figure indicates, SimplePEP transmits at a constant bitrate of 1.125 Mbps for the duration of the transfer, and only the lower priority TCP traffic is dropped at the queue. The TCP throughput correctly adapts to the available user bitrate of the channel when the system enters the different states of the error model.

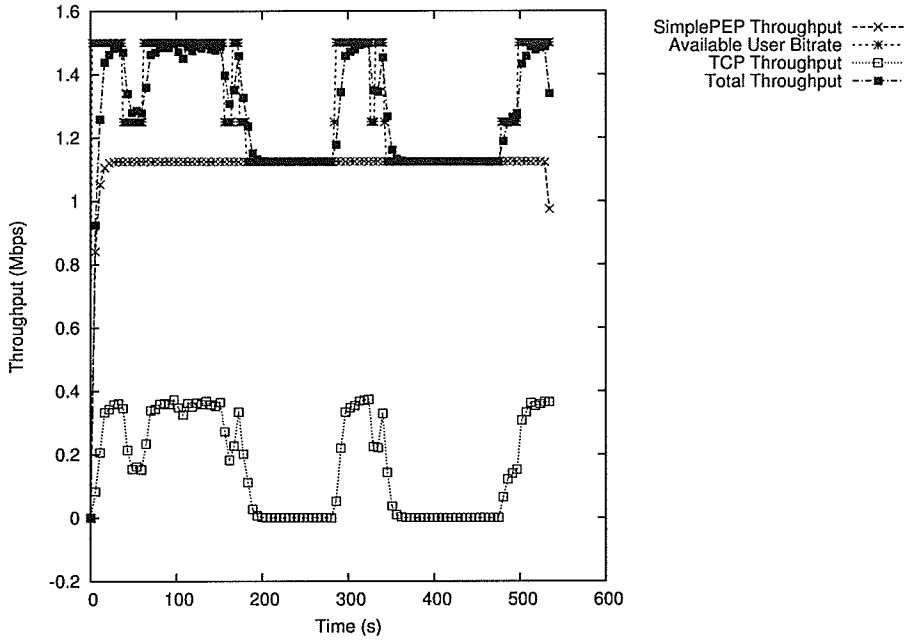

Figure 4.8 - TCP Opportunistic with Priority Queuing Throughput vs. Time.

4.3.4 Bandwidth Adaptation

The reason that TCP can take better advantage of the changing channel bandwidth is due to the adaptation built into the protocol. It follows that better performance from SimplePEP could also be obtained if it too adapted to decreases in available user bitrate. Since SimplePEP is a custom protocol designed and configured specifically for a satellite link, the bandwidth adaptation scheme could be implemented more intelligently than TCP's. For example, since it is assumed that congestion due to competing traffic flows is not an issue, reductions in bitrate of SimplePEP traffic need not be as aggressive as TCP's congestion control.

A simple bandwidth adaptation scheme is introduced to the proxy that is based loosely on the Packet-Pair algorithm [CaFM04]. The Packet-Pair algorithm is a simple technique used to measure the available bandwidth on a channel. Its main principle is as follows: packets are sent in succession from the server as close together as possible. The assumption is made that these packets are placed in the queue at the bottleneck one after

the other. If this assumption holds for the returning acknowledgments, it follows that the queuing delay on the network can be estimated by measuring the time difference between arriving acknowledgments. The channel bandwidth can then be obtained by dividing the packet size by the queuing delay. This concept is best illustrated in Figure 4.9. This technique breaks down under a number of scenarios, such as when a packet takes separate paths through routers on the network, but is an excellent candidate for a single hop satellite link.

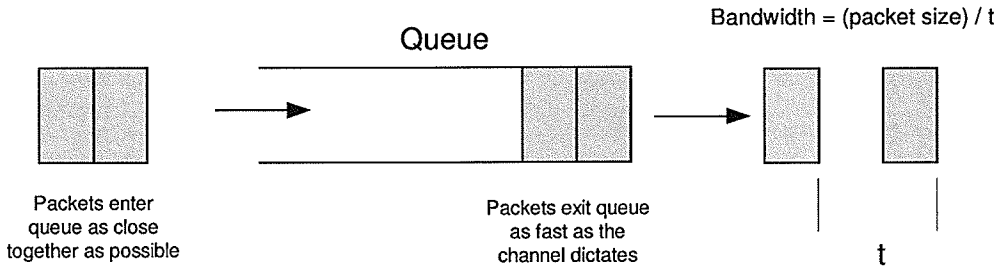

Figure 4.9 - Packet Pair Algorithm.

To illustrate this point, a simple bandwidth estimation technique is added to SimplePEP. As acknowledgments arrive at the sender, the estimated channel bandwidth is calculated. If the estimated bandwidth is lower than the current rate of transmission, the sender will adjust its rate to the match. Since packets are sent out of the server proxy according to the set output bitrate, and not in bursts as required by the Packet-Pair algorithm, an alternate method to detect increases in available bandwidth is used. If the estimated bandwidth is equal to the current output bitrate, the server proxy will probe for increases in available user bitrate by increasing its sending rate by ten percent. This adjustment is done once for every round trip time. If losses occur as a result of the new rate, the bitrate will be reduced. However, if returning acknowledgments from the new set of packets indicate that the channel bitrate has increased, this process will continue until the maximum rate is found. To avoid congestion at the upper ceiling of the channel, the rate is never increased beyond the pre-set output bitrate of 1.5 Mbps. While this procedure would be considered too aggressive for a "fair" networking protocol, the assumption is made that congestion is self-induced, and does not affect other clients offering traffic through the satellite

gateway. The same 50,000-packet file is again transmitted using the proposed bandwidth estimation scheme, and the results are shown in Table 4.3. A plot of the system throughput versus time is also shown in Figure 4.10.

Table 4.3 - SimplePEP versus Bandwidth Adaptive SimplePEP

	SimplePEP	SimplePEP w/ Bandwidth Adaptation
Average Transfer Time	496.41 s	463.70 s
Average Throughput	1.22 Mbps	1.30 Mbps
Available User Bitrate	1.35 Mbps	1.37 Mbps
Efficiency	90%	95 %

It is clear from the illustration that the throughput of the transmission comes much closer to the available channel bandwidth than was possible without bandwidth estimation. In fact, even with the simple scheme proposed here, the overall transmission efficiency is increased to 95% when using bandwidth estimation.

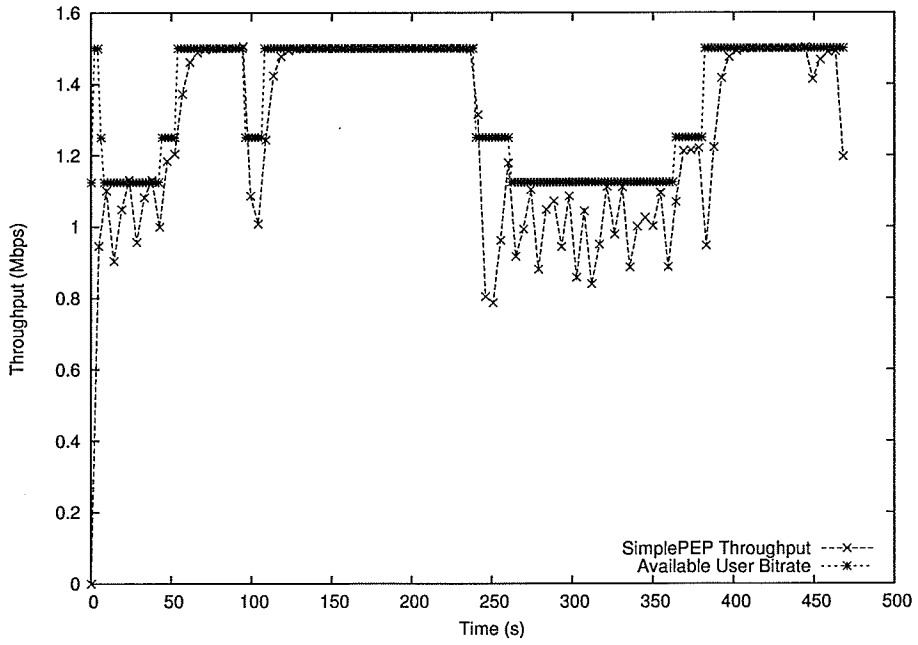

Figure 4.10 - Bandwidth Estimation Throughput vs. Time.

4.3.5 DVB-S2 Aware Proxies

While a more intelligent bandwidth estimation scheme could be implemented, the most desirable scenario may involve a performance enhancing proxy that knows exactly the available user bitrate at any given time. If SimplePEP were able to communicate with the satellite gateway, the gateway could send information about the current user protection level, allowing SimplePEP to adjust its bitrate to match the available bitrate on the channel. To illustrate this, feedback between the DVB-S2 gateway and SimplePEP is introduced to the simulation. Whenever the protection level is changed, the SimplePEP server is made aware so that it can adjust its output bitrate to match. As can be seen from the results in Figure 4.11, channel utilization jumps to 100% since packets are no longer lost due to congestion. Although similar results were achieved by increasing the queue size in the satellite gateway, larger buffer sizes may be required depending on the protection levels offered by the system and the number of clients sending traffic through the gateway. Conversely, the results obtained here using the feedback mechanism use a queue size at the gateway of only 10 packets. Unfortunately, as with the bandwidth estimation scheme, modifications to an operational PEP would be necessary.

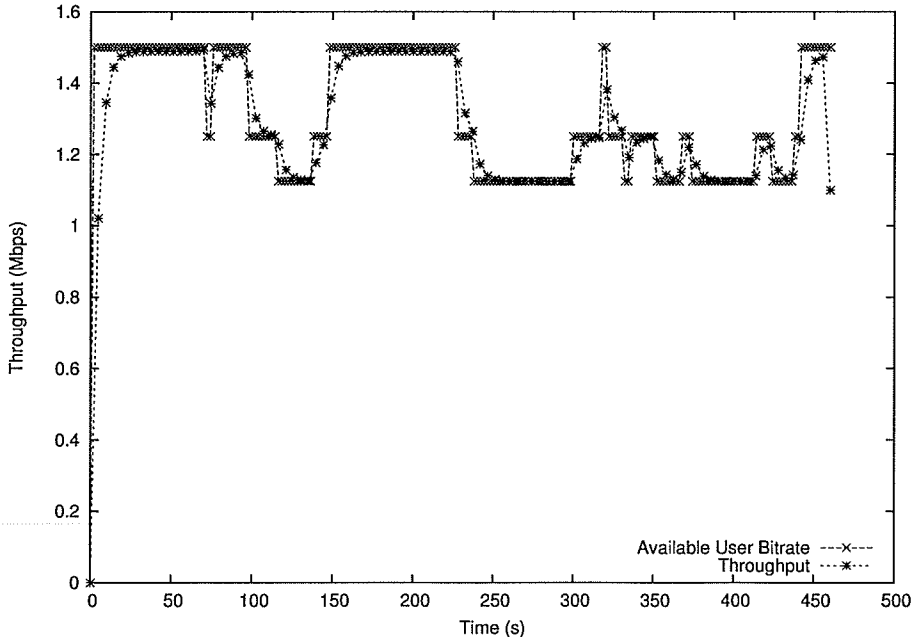

Figure 4.11- DVB Aware Throughput vs. Time.

5. CONCLUSIONS AND FUTURE WORK

The motivation of this thesis was to investigate the effects of adaptive error correction on systems using performance enhancing proxies. As a result of the ratification of the DVB-S2 standard, many network operators using performance enhancing proxies in first generation DVB-S systems may soon be faced with link performance issues if they choose to employ existing proxy equipment in new, second generation networks.

Consistent with prior research in the field, it has been shown here that a properly designed performance enhancing proxy can yield dramatic performance improvements over satellite channels when compared to TCP, even if the TCP parameters are optimized to take better advantage of the satellite link. Since the performance enhancing proxy in this thesis (SimplePEP) was designed for a DVB-S system, it assumes that a fixed bandwidth is available for the duration of the file transfer. However, when adaptive error correction is introduced to the system, with the goal of providing improved packet-error rates, the available bitrate to the performance enhancing proxy no longer remains constant. This results in congestion at the satellite gateway as a result of a decrease in available user bitrate when the channel conditions are poor. While channel utilization remains high when the link conditions are good, bandwidth is wasted during poor channel conditions, as SimplePEP is forced to retransmit packets that were lost to congestion. In fact, SimplePEP was only able to utilize the link with an overall 90% efficiency over the transmission of a large file.

SimplePEP can provide better long-term performance by sacrificing dropped packets in poor channel conditions for a higher bitrate in good channel conditions. In other words, the performance gains achieved by fully utilizing the link when the channel is good, outweigh the effects of congestion when the channel is bad. Whether this applies to a real-world application depends greatly on the characteristics of the link, such as the percentage of time the channel spends in the good state. Using such a method is not beneficial when transferring small files when the system is entirely in one of the high

BER channel states, since the transmission efficiency in those states degrades considerably.

It was also demonstrated that adjusting the queue size at the satellite gateway (ACM router) can result in increased performance to the point where no packets are dropped at all, allowing for full utilization of the channel. The optimum queue size can be calculated based on the parameters of the link and the ARQ protocol used by the performance enhancing proxy. However, the usability of this method depends largely on the parameters of the system as a whole, since queue sizes cannot be set arbitrarily high when a large number of clients are stressing the system. As well, large queuing delays could increase the loop delay seen by the adaptive error correcting scheme. If a packet is assigned to a protection level, and is subsequently forced to wait in the queue for a large period of time, the state of the channel may have changed such that the selected protection level is no longer sufficient to transmit the packet without error.

If changes to queue sizes on the DVB-S2 equipment are not possible, better overall transmission efficiency can be obtained by introducing a secondary TCP flow. Since a TCP connection adapts its rate of transmission to the available user bitrate, it could be used to carry lower priority traffic while the SimplePEP server could provide guaranteed, higher quality of service traffic. However, this scheme only provides benefits when a priority queuing scheme that favors SimplePEP packets over those transmitted by TCP is introduced at the satellite gateway. If SimplePEP and TCP packets are dropped with equal probability, the congestion that occurs as TCP attempts to increase its rate of transmission during *slow-start* results in less than full utilization of the channel. The concept of introducing a secondary TCP connection has the advantage over other schemes in that the performance enhancing proxy can continue to operate without modification.

If modifications to the performance enhancing proxy itself are possible, however, bandwidth estimation could be an effective way to improve performance. Even a simple scheme based on the packet-pair algorithm, as proposed in this thesis, is sufficient to

provide SimplePEP with adequate channel estimates, thus increasing the overall utilization of the channel from 90% to over 95%. The bandwidth estimation scheme used was based on the assumption that SimplePEP could increase its rate of transmission without affecting other traffic running though the satellite gateway.

It is also concluded that the ideal solution to improve channel utilization is to either provide feedback to the performance enhancing proxy from the satellite gateway, providing information about the available user bitrate, or to integrate the two systems into a single network device. The knowledge of the exact available bitrate at any given time results in few, if any, dropped packets, allowing the channel to be used to full capacity. While the DVB-S2 literature notes that flow control between remote Internet servers and the satellite gateway may not be feasible, due both to the number of network devices and their respective distances from the gateway, the use of performance enhancing proxies means that communication would only need to be provided between the gateway and a single proxy. As a result, it is recommended that future systems provide tight integration between the PEP equipment and DVB-S2 equipment when possible.

5.1 Contributions

The main contributions to the field as a result of the research conducted in this thesis are outlined below:

i) Development of a simple connection splitting performance enhancing proxy in C++ for the ns-2 network simulator. The proxy was shown to improve the performance of TCP in satellite networks by up to 4 times. The source code for the proxy is included in Appendix 7.2, and can be used for further study relating to satellite networks.

ii) Identification of the effects of adaptive error correction on performance enhancing proxies. It was shown that the variation in bandwidth resulting from changing protection levels causes less than optimal utilization of the channel. This runs

contrary to the goal of introducing adaptive error correction to the system in the first place.

iii) Proposal of several techniques to improve channel utilization, including introducing bandwidth estimation to the performance enhancing proxy, introducing secondary TCP flows to take advantage of available bandwidth, and integrating a new proxy implementation with the satellite gateway.

5.2 Recommendations for Future Work

This section provides recommendations for future research based on the findings in this thesis. These recommendations are itemized below:

i) While it was shown that a simple priority queuing scheme is sufficient when introducing competing TCP traffic to the system, it may be desirable to investigate other schemes when different quality of service requirements exist. Queuing priorities based on the type of traffic, rather than just the source, could be implemented. For example, different client traffic running through SimplePEP could be given higher or lower priority, allowing individual connections to offer different levels of service. Users transmitting real-time video or audio could be favored over users performing simple file transfers. Alternate priority schemes at the satellite router should be a future point of investigation.

ii) The bandwidth estimation scheme proposed was very simple. It is expected that, given the simplicity and known parameters of the link between a pair of proxies, a more efficient algorithm could be devised that would provide more accurate bandwidth estimations, resulting in closer to optimum transmission efficiency. For example, in the system proposed in this thesis, using returning NAKs as an indication of congestion may improve the estimated values.

iii) Iintegration between a performance enhancing proxy and DVB-S2 equipment would supply the ideal environment with respect to channel utilization. If this is not possible, a standard mechanism by which DVB-S2 equipment can communicate changes in coding rates to other network equipment could be developed. Such a protocol should be studied to see if it has merit.

iv) The effects of competing traffic are not studied here. It is conceivable that a DVB-S2 gateway could be implemented in such a way that changes in protection levels for other clients on the link affect the bitrate available to SimplePEP traffic. Possible ways to account for occasional dropped packets due to temporary peaks as the gateway switches protection levels should be studied. For example, introducing erasure codes to SimplePEP traffic, allowing it to recover from minor congestion, may be beneficial.

v) A reliable return link with no ACK congestion was assumed for the simulations performed in this thesis. The effects of a lower bitrate return channel, where returning acknowledgments may be lost, should be studied. For example, lost acknowledgments may make bandwidth estimation more difficult.

6. REFERENCES

[AlGS99] Allman, M., Glover, M., and Sanchez, L., "Enhancing TCP over satellite channels using standard mechanisms", January 1999. RFC 2488.

[AlHO97] Allman, M., Hayes, C., Kruse, H., and Ostermann, S., "TCP performance over satellite links", in *Proceedings of the 5th International Conference on Telecommunication Systems*, Nashville, TN, March 1997.

[AlKrOs] Allman, M., Kruse, H., and Ostermann, S. "An application-level solution to TCP's satellite inefficiencies", in *Proceedings of the ACTS Results Conference*, NASA Lewis Research Center, September 1995.

[AlPS01] Allman, M., Paxson, V., and Stevens, W., "TCP congestion control", 2001. RFC 2581.

[BaLK04] Baldantoni, L., Lundqvist, H., and Karlsson, G., "Adaptive End-to-End FEC for Improving TCP Performance over Wireless Links", Master's thesis, KTH Royal Institute of Technology, 2004.

[BoKG01] Border, J., Kojo, M., Griner, J., Montenegro, G., and Shelby, Z., "Performance enhancing proxies intended to mitigate link-related degradations", RFC 3135, June 2001.

[CaFM04] Capone, A., Fratta, L., and Martignon, F., "Bandwidth estimation schemes for TCP over wireless networks", *IEEE Transactions on Mobile*, vol. 3, no. 2, pp. 129-143, April 2004.

[CaGG04] Casini, E., De Gaudenzin, R., and Ginesi, A., "DVB-S2 modem algorithms design and performance over typical satellite channels",

International Journal of Satellite Communications and Networking, vol. 22, issue 3, pp. 281–318, 2004.

[ChJK05] Choi, E., Jung, J., Kim N., and Oh, D., "Complexity-reduced algorithms for LDPC decoder for DVB-S2 systems", *ETRI Journal*, vol. 27, no. 5, pp. 639-642, Oct. 2005.

[CiGaRi] Cioni, S., Toffano, V., Gaudenzi, R., and Rinaldo, R., "Adaptive coding and modulation for broadband satellite networks: part I", http://www.cost280.rl.ac.uk/documents/WS2%20Proceedings/documents/pm-5-011a.pdf.

[Come00] Comer, D., *Internetworking with TCP/IP: Principles, Protocols, and Architectures*, New Jersey, Prentice Hall, 2000.

[DuZh02] Dutta, D., and Zhang, Y., "An early bandwidth notification (EBN) architecture for dynamic bandwidth environments," *IEEE International Conference on Communications*, vol. 4, pp. 2602-2606, April 2002.

[EhLR03] Ehsan, N., Liu, M., and Ragland, R., "Evaluation of performance enhancing proxies in Internet over satellite", *International Journal of Satellite Communications*, vol. 16, issue 6, pp. 513-534, May 2003.

[ErSL04] Erozn, M., Sun, F., and Lee, L., "DVB-S2 low density parity check codes with near Shannon limit performance", *International Journal of Satellite Communication Networks*, vol. 22, issue 3, pp. 269-279, June 2004.

[ETSI05] ETSI EN 302 307 V1.1.1 (2005-03), Digital Video Broadcasting (DVB): Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications.

[GeSaWZ] Gerla, M., Sanadidi, M. Y., Wang, R., and Zanella, A., "TCP Westwood: congestion window control using bandwidth estimation", *Global Telecommunications Conference*, vol.3, pp. 1698-1702, 2001.

[Huit96] Huitema, C., "The case for packet level FEC.", *IFIP Conference Proceedings*, vol. 73, pp. 109-120, October 1996.

[Idirec] "The iDirect Series 1000 – Network Accelerators", iDirect Technologies, http://idirect.qorvis.com/page.ww?section=News&name=White+Papers.

[IsAl01] Ishac, J., and Allman, M., "On the performance of TCP spoofing in satellite networks", *IEEE Milcom*, vol. 1, pp. 700-704, October 2001.

[JaBB92] Jacobson, V., Braden, R., and Boman, D., "TCP extension for high performance", RFC 1323, May 1992.

[JiZh00] Jing, W., and Zhisheng, N., "A reliable TCP-Aware link layer retransmission for wireless networks", *International Conference on Communication Technology*, vol. 1, pp. 900-905, 2000.

[Kost02] Kostas, T., "Packet erasure FEC on ARQ protocols", in *Proceedings of SPIE*, vol. 4866, pp. 126-137, June 2002.

[KoMa03] Kota, S., and Marchese, M., "Quality of service for satellite IP networks: a survey", *International Journal of Satellite Communications and Networking*, vol. 21, issue 4-5, pp.303-249, July 2003.

[LeWi00] Leon-Garcia, A., and Widjaja, I., *Communication Networks: Fundamental Concepts and Key Architectures*, McGraw-Hill, 2000.

[LiCo00] Lin, S., Costello Jr., D., *Error Control Coding: Fundamentals and Applications*, New Jersey, Prentice-Hall, 1983.

[LiDa04] Lin, H., and Das, S. K., "ARLP: an adaptive link layer protocol to improve TCP performance over wireless fading channels", *Wireless Communications and Mobile Computing*, vol. 4, issue 6, pp. 655–668, September 2004.

[LiGT02] Liu, B., Goeckel, D. L., and Towsley, D., "TCP-cognizant adaptive forward error correction in wireless networks," *Global Telecommunications Conference*, vol.3, pp. 2128-2132, November 2002.

[LuKa04] Lundqvist, H., and Karlsson, G., "TCP with End-to-End Forward Error Correction", in *Proc. of International Zurich Seminar on Communications*, IZS 2004, February 2004.

[LiYu80] Lin, S. and Yu, P. S., "An effective error control scheme for satellite communications," *IEEE Transactions on Communications*, vol. 28, pp. 395-401, March 1980.

[MoMi04] Morello, A., and Mignone, V., "DVB-S2: Ready for Liftoff", EBU Technical Review, October 2004, http://www.ebu.ch/en/technical/trev/trev_index-dvb.html.

[More02] Morelos-Zaragoza, R., *The Art of Error Correcting Coding*, West Sussex, England, John Wiley & Sons, 2002.

[MoRe04] Morello, A., and Reimers, U., "DVB-S2, the second generation standard for satellite broadcasting and unicasting", *International Journal of Satellite Communications and Networking*, vol. 22, issue 3, pp.249-268, June 2004.

[Nett00] "NettGain: How to maximize throughput on VSAT Networks", Flash Networks Inc., 2000, http://www.flashnetworks.com.

[NS2Sim] The Network Simulator – ns-2, http://www.isi.edu/nsnam/ns/.

[PaFT98] Padhye, J., Firoiu, V., Towsley, D., and Kurose, J., "Modeling TCP throughput: a simple model and its empirical validation", in *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 303-314, 1998.

[Phil01] Philopoulos, S., "Improving the Performance of TCP over Satellite Channels", Master's thesis, University of Manitoba, June 2001.

[PhFe02] Philopoulos, S., and Ferens, K., "Proxy-based connection splitting architectures for improving TCP performance over satellite channels", in *Proceedings of the 2002 IEEE Canadian Conference on Electrical and Computer Engineering*, vol. 3, pp. 1430-1435, 2002.

[PoCe03] Potorti, F. and Celandroni, N., "Maximizing single connection TCP goodput by trading bandwidth for BER", *International Journal of Communication Systems*, vol.16, no.1, pp. 63-79, February 2003.

[Rapp96] Rappaport, T., *Wireless Communications Principles and Practices*, New Jersey, Prentice-Hall, 1996.

[RiVM04] Rinaldo, R., Vazquez-Castro, M. A., and Morello, A., "DVB-S2 ACM modes for IP and MPEG unicast applications", *International Journal of Satellite Communications and Networking*, vol. 22, issue 3, pp. 367–399, June 2004.

[Sanc04] Sancho, I., "On Adaptive Forward Error correction for Real-Time Traffic", Masters thesis, University of Zaragoza, Spain, 2004.

[Silv05] "Taking Forward Error Correction (FEC) To The Next Level", Silver Peak Systems Inc., White Paper, 2005.

[Stal02] Stallings, W., *Wireless Communications and Networks*, New Jersey, Prentice Hall, 2002.

[Swee99] Sweeney, P., "Channel modeling and error control strategies for the LEO satellite channel", presented at International Symposium on Communications Theory and Applications, Ambleside, UK, July 1999.

[TeMo04] Tew, B., and Morello, A., "DVB-S2: Most advanced satellite delivery possible", White Paper, April 2004, http://www.dvb.org.

[TiSK05] Tickoo, O., Subramanian, V., Kalyanaraman, S., and Ramakrishnan, K.K., "LT-TCP: end-to-end framework to improve TCP performance over networks with lossy channels", in *Proceedings of the International Workshop on QoS*, Passau, June 2005.

[UbKl03] Ubik, S., and Klaban, J., "Experience with using simulations for congestion control research", CESNET Technical Report, December 2003, http://www.cesnet.cz/doc/techzpravy/2003.html.

[VeKM02] Velenis, D., Kalogeras, D., and Maglaris, B., "SaTPEP: a TCP performance enhancing proxy for satellite links", in *Proceedings of 2nd International IFIP-TC6 Networking Conference*, Pisa Italy, May 2002.

[Vuce91] Vucetic, B., "An Adaptive Coding Scheme for Time-Varying Channels", *IEEE Transactions on Communications*, vol. 39, issue 5, pp. 653-663, May 1991.

[ZhRo04] Zhu, J., and Roy, S., "Improving link layer performance on satellite channels with shadowing via delayed two-copy selective repeat ARQ", *IEEE Journal on Selected Areas in Communications*, vol. 22, issue 3, pp. 472-481, April 2004.

7. APPENDIX

7.1 SimplePEP.h

```
//
// simplepep.h
//
// SimplePEP client and server agent for ns-2, header file.
// See simplepep.cc for more detailed description of classes.
//

#ifndef _simplepep_h
#define _simplepep_h

#include "agent.h"
#include "tclcl.h"
#include "packet.h"
#include "address.h"
#include "ip.h"
#include "timer-handler.h"
#include "tcp.h"
#include "tcp-sink.h"

// maximum window size
#define MAX_WINDOW_SIZE     (200)

// SimplePEP packet types
#define SIMPLEPEP_TYPE_DATA (0)
#define SIMPLEPEP_TYPE_ACK  (1)
#define SIMPLEPEP_TYPE_NAK  (2)

// SimplePEP packet header definition
struct hdr_simplepep {
  unsigned char packet_type;     // 0=DATA, 1=ACK, 2=NAK
  unsigned int  seqno;           // start of receive window
  unsigned int  acked_seqno;     // acked/naked sequence number

  unsigned char fec_level;       // DVB-S2 header, protection level

  unsigned int  blocknumber;     // erasure codes (packetlevel FEC)
  unsigned int  blocksize;
  unsigned int  fecredundancy;
  bool          redundancy;

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

```
class SimplePEPAgent;
class DVBErrorModel;

// ns-2 timer definitions

// Sending timer
class SimplePEPTimer : public TimerHandler {
public:
  SimplePEPTimer(SimplePEPAgent *a) : TimerHandler() { a_ = a; }
protected:
  virtual void expire(Event*);
  SimplePEPAgent* a_;
};

// Retransmission timer
class SimplePEP_TXTimer : public TimerHandler {
public:
  SimplePEP_TXTimer(SimplePEPAgent *a) : TimerHandler() { a_ = a; }
protected:
  virtual void expire(Event*);
  SimplePEPAgent* a_;

};

// FEC adaptation timer
class SimplePEP_FECTimer : public TimerHandler {
public:
  SimplePEP_FECTimer(SimplePEPAgent *a) : TimerHandler() { a_ = a; }
  int  level;
protected:
  virtual void expire(Event*);
  SimplePEPAgent* a_;
};

// DVB Error Model Timer
class DVBErrorModelTimer : public TimerHandler {
public:
  DVBErrorModelTimer(DVBErrorModel *emodel) : TimerHandler() { emodel_
= emodel; }
protected:
  virtual void expire(Event*);
  DVBErrorModel* emodel_;

};


// DVB Error model
#define MAX_STATES     (3)
#define MAX_PROTECTION (3)
class DVBErrorModel
{
  friend class DVBErrorModelTimer;

  public:
    DVBErrorModel();
    bool corrupt(int protection_level);
```

```
    void init();
    void stop();
    int  currentsnrlevel();

    double state_time[MAX_STATES];
    double starttime;

  protected:
    DVBErrorModelTimer markovtimer;
    void update();

  protected:
    int    currentstate;
    int    numstates;
    double state_per[MAX_STATES][MAX_PROTECTION];
    double state_transition[MAX_STATES][MAX_STATES];
};

// SimplePEP agent class
class SimplePEPAgent : public Agent {

public:
  SimplePEPAgent();
  int command(int argc, const char*const* argv);
  void recv(Packet*, Handler*);

  void timeout();
  void sendpkt();
  void send_data_packet(unsigned int sequence_number, unsigned int
blocknumber, bool retransmit=false, bool redundancy=false);
  void schedule_packet(unsigned int sequence_number);
  bool is_packet_available(unsigned int *seqno, unsigned int
*blocknumber, bool *redundancy);
  void send_ack_packet(unsigned int seqno, unsigned char ack_nak,
unsigned int acked_seqno, bool redundancy);
  void corruptPacket();
  void setfec(int level);
  void adapt(int  level);
  void results();

 protected:
  void start();          // start sending at the desired rate
  void startfixed();     // start sending at the desired rate
  void stop();           // stop sending
  void reschedule_timeout();
  void reschedule_transmittimeout();
  void recalctxrate();

  DVBErrorModel errormodel;  // instance of a DVB error model
  double overhead[256];      // hold overhead for each protection level

  Agent     *Transmitter_;   // pointer to server so client can report
                             // changes in FEC levels

  SimplePEPTimer     sendtimer;
  SimplePEP_TXTimer  transmit_timeout;
  SimplePEP_FECTimer fec_timer;
```

```
// INTERNAL VARIABLES AND COUNTERS
unsigned int   rx_seqno;        // next expected receive_seqno
unsigned int   tx_seqno;        // next seq_no to transmit

unsigned int   tx_last;         // last packet transmitted
unsigned int   rx_next;         // next packet we expect to receive

double    last_packetsent;      // time last packet/ack sent/received
double    last_ackreceived;
double    last_acksent;

unsigned int   tx_block_last;   // variables for erasure code blocks
unsigned int   rx_block_next;
unsigned int   rx_blocknumber;

unsigned int   blocksent;
unsigned int   redcount;

double         interval;        // interval to output packets
double         lastincrease;    // time sendrate_ was last increased
double         bandwidth_estimate; // current BW estimate

// store information on current packets in the window
double         tx_timers[MAX_WINDOW_SIZE];
bool           tx_window[MAX_WINDOW_SIZE];
bool           tx_acked[MAX_WINDOW_SIZE];
unsigned int   tx_blocksize[MAX_WINDOW_SIZE];
bool           rx_window[MAX_WINDOW_SIZE];
bool           rx_nak[MAX_WINDOW_SIZE];
int            tx_blocknumber[MAX_WINDOW_SIZE];
unsigned int   rx_blockpacketcount[MAX_WINDOW_SIZE];

// STATISTICS (ACCESS VIA OTCL SCRIPTS)
unsigned int   packetssent_;
unsigned int   bytessent_;
unsigned int   bytesfecsent_;
unsigned int   bytespacketlevelfec_;
unsigned int   bytesretransmitted_;
unsigned int   packetsdropped_;
unsigned int   packetsretransmitted_;
unsigned int   packetserrors_;
double         available_bitrate_;

// SETUP VARIABLES (CHANGE VIA OTCL SCRIPTS)
int     currentfeclevel_;  // set starting protection level
int     adapt_;            // enable ACM
int     dvbaware_;         // enable DVB aware functions
int     enableacks_;       // for debugging (do not disable)
int     errormodel_;       // enable internal DVB error model
int     tracedebug_;       // enable debug tracing

int     WINDOW_SIZE_;      // window size (in packets)
double  RTT_;              // one-way round trip time of the channel
double  sendrate_;         // the desired sending rate (Mb/s)
int     blocksize_;        // the desired packet size (bytes)
int     selectiveack_;     // selective ACK (do not disable)
```

```
    int     usenak_;            // NAKs (do not disable)
    int     delayack_;          // enable delayed ACKs (not used)
    int     bwadapt_;           // enable bandwidth adaptation

    int     packetlevelfec_;    // enable packet-level erasure codes
    int     fecblocksize_;      // total block size (k)
    int     fecredundancy_;     // redundancy (n-k)

    int     packetlimit_;       // number of pkts to send before stopping
};

#endif
```

7.2 SimplePEP.cc

```
//
// simplepep.cc
//
// SimplePEP client and server agent for ns-2.
//

#include "simplepep.h"
#include "ranvar.h"

#define min(a,b) (((a)<(b))?(a):(b))
#define max(a,b) (((a)<(b))?(b):(a))

//=============================================================
// TCL Binding functions
//=============================================================

static class SimplePEPClass : public TclClass {
public:
   SimplePEPClass() : TclClass("Agent/SimplePEP") {}
   TclObject* create(int, const char*const*) {
      return (new SimplePEPAgent());
   }
protected:
} class_SimplePEP;


//=============================================================
// Packet format for SimplePEP traffic (see "simplepep.h")
//=============================================================

int hdr_simplepep::offset_;
static class SimplePEPHeaderClass : public PacketHeaderClass {
public:
   SimplePEPHeaderClass() : PacketHeaderClass("PacketHeader/SimplePEP",
sizeof(hdr_simplepep)) {
   bind_offset(&hdr_simplepep::offset_);
   }
} class_simplepephdr;


//=============================================================
// DVBErrorModel class:
//   Markov error model and adaptive error correction.
//   Error probability in each state is based on the
//   modulation and coding level of the encoded packets.
//=============================================================

DVBErrorModel::DVBErrorModel() : markovtimer(this)
{
   int i = 0;

   // set initial state
```

```
  currentstate = 0;
  numstates = MAX_STATES;

  // state transition probabilities
  state_transition[0][0] = 0.983;
  state_transition[0][1] = 0.017;
  state_transition[0][2] = 0.0;

  state_transition[1][0] = 0.04;
  state_transition[1][1] = 0.92;
  state_transition[1][2] = 0.04;

  state_transition[2][0] = 0.0;
  state_transition[2][1] = 0.025;
  state_transition[2][2] = 0.975;

  // state PER, 10^-7 in most cases
  state_per[0][0] = 0.0000001;
  state_per[1][0] = 0.0000001;
  state_per[2][0] = 0.001;

  state_per[0][1] = 0.0000001;
  state_per[1][1] = 0.0000001;
  state_per[2][1] = 0.0000001;

  state_per[0][2] = 0.0000001;
  state_per[1][2] = 0.0000001;
  state_per[2][2] = 0.0000001;

  // keep track of length of time in each state for stats
  state_time[0] = 0;
  state_time[1] = 0;
  state_time[2] = 0;
}

////////////////////////////////////////////////////
// DVBErrorModel::corrupt
//   Called by the Corrupt packet (packet error rate based on
//   correction level.
//
bool DVBErrorModel::corrupt(int protection_level)
{
  double u = Random::uniform();
  double p = state_per[currentstate][protection_level];

  if (u<=p) { return(false); }  // drop packet
  return(true);
}

////////////////////////////////////////////////////
// DVBErrorModel::update
//   Called every interval to determine state transitions.
//
void DVBErrorModel::update()
{
  double  currenttime = Scheduler::instance().clock();
  int     i = 0;
```

```
   double  u = Random::uniform();
   double  p = 0.0;

   for (i=0; i<numstates; i++)
   {
     p += state_transition[currentstate][i];
     if (u<=p) break;
   }
   state_time[currentstate]+=(currenttime-starttime);
   starttime = currenttime;
   if (currentstate!=i && i<numstates)
   {
     currentstate=i;
   }
   markovtimer.resched(1.0);
}

////////////////////////////////////////////////////
// DVBErrorModel::init
//   Start the timer to begin the error model transitions
//
void DVBErrorModel::init()
{
   double currenttime = Scheduler::instance().clock();

   starttime = currenttime;
   markovtimer.resched(1.0);  // set timer for 1 second
}

////////////////////////////////////////////////////
// DVBErrorModel::stop
//   Stops the error model
//
void DVBErrorModel::stop()
{
   if (markovtimer.status()==TIMER_PENDING) markovtimer.cancel();

   // log the time in each state to a file for analysis
   FILE *f = fopen("channel-state.txt", "a+");
   fprintf(f, "%0.2f %0.2f %0.2f\n", state_time[0], state_time[1],
state_time[2]);
   fclose(f);
}

////////////////////////////////////////////////////
// DVBErrorModel::currentsnrlevel
//   Used to return the current state of the channel
//
int DVBErrorModel::currentsnrlevel()
{
   return currentstate;
}


//============================================================
// DVBErrorModelTimer class:
//   Calls DVBErrorModel::update upon expiry to transition the model
```

```
//=============================================================

void DVBErrorModelTimer::expire(Event *e)
{
  emodel_->update();
}

//=============================================================
// SimplePEPTimer, SimplePEP_TXTimer, SimplePEP_FECTimer:
//   Timer objects for SimplePEPAgent
//=============================================================

// Send Timer, used to throttle output bitrate
void SimplePEPTimer::expire(Event *e)
{
  a_->sendpkt();
}

// Retransmit timer for sent packets
void SimplePEP_TXTimer::expire(Event *e)
{
  a_->timeout();
}

// FEC timer, simulates delay in adapting to channel conditions
void SimplePEP_FECTimer::expire(Event *e)
{
  a_->setfec(level);
}

//=============================================================
// SimplePEPAgent:
//   Transmitter/Receiver for a simple TCP performance enhancing proxy.
//=============================================================

SimplePEPAgent::SimplePEPAgent() : Agent(PT_SIMPLEPEP),
sendtimer(this), transmit_timeout(this), Transmitter_(0),
fec_timer(this)
{

  // clear SimplePEP internal variables
  tx_block_last      =0;
  rx_block_next      =0;
  rx_blocknumber     =0;
  blocksent          =0;
  redcount           =0;
  tx_seqno           =0;
  rx_seqno           =0;
  tx_last            =0;
  rx_next            =0;
  last_packetsent    =0;
  bandwidth_estimate =0.0;
  last_ackreceived   =0.0;
  last_acksent       =0.0;
  lastincrease       =0;

  // initialize external variables (can be changed by TCL scripts)
```

```
sendrate_          =1.0;
blocksize_         =100;
enableacks_        =1;
WINDOW_SIZE_       =10;
RTT_               =0.05;
packetlimit_       =0;
adapt_             =true;
dvbaware_          =false;
currentfeclevel_   =2;
packetlevelfec_    =0;
fecblocksize_      =5;
fecredundancy_     =1;
tracedebug_        =0;
selectiveack_      =0;
errormodel_        =0;
usenak_            =0;
delayack_          =0;
bwadapt_           =0;

// reset statistics
packetssent_           =0;
bytessent_             =0;
bytesfecsent_          =0;
bytespacketlevelfec_   =0;
bytesretransmitted_    =0;
packetsdropped_        =0;
packetsretransmitted_  =0;
packetserrors_         =0;
available_bitrate_     =0.0;

for (int i=0; i<MAX_WINDOW_SIZE; i++) {
   tx_window[i]             =false;
   tx_acked[i]              =false;
   rx_window[i]             =false;
   rx_nak[i]                =false;
   tx_timers[i]             =0;
   tx_blocknumber[i]        =0;
   rx_blockpacketcount[i]   =0;
}

// set the default FEC levels
overhead[0xFF]   = 1.0;       // fec disabled
overhead[2]      = 4.0/3.0;   // most overhead
overhead[1]      = 6.0/5.0;
overhead[0]      = 1.0;       // least overhead

// bind internal variables to TCL variables
bind("sendrate_",               &sendrate_);
bind("fec_",                    &currentfeclevel_);
bind("adapt_",                  &adapt_);
bind("dvbaware_",               &dvbaware_);
bind("blocksize_",              &blocksize_);
bind("enableacks_",             &enableacks_);
bind("WINDOW_SIZE_",            &WINDOW_SIZE_);
bind("RTT_",                    &RTT_);
bind("packetlimit_",            &packetlimit_);
bind("selectiveack_",           &selectiveack_);
```

```
  bind("packetlevelfec_",         &packetlevelfec_);
  bind("fecblocksize_",           &fecblocksize_);
  bind("fecredundancy_",          &fecredundancy_);
  bind("tracedebug_",             &tracedebug_);
  bind("errormodel_",             &errormodel_);
  bind("usenak_",                 &usenak_);
  bind("delayack_",               &delayack_);
  bind("bwadapt_",                &bwadapt_);
  bind("Transmitter_",            (TclObject **)(&Transmitter_));

  bind("bytessent_",              &bytessent_);
  bind("packetssent_",            &packetssent_);
  bind("bytesfecsent_",           &bytesfecsent_);
  bind("bytesretransmitted_",     &bytesretransmitted_);
  bind("packetsdropped_",         &packetsdropped_);
  bind("packetsretransmitted_",   &packetsretransmitted_);
  bind("packetserrors_",          &packetserrors_);
  bind("last_packetsent",         &last_packetsent);
  bind("available_bitrate_",      &available_bitrate_);
}


//////////////////////////////////////////////////////
// SimplePEPAgent::command
//   TCL command interface for SimplePEP agent
//
int SimplePEPAgent::command(int argc, const char*const* argv)
{
  if (argc == 2) {

    // Send a single packet
    if (strcmp(argv[1], "send") == 0) {
      sendpkt();
      return (TCL_OK);
    }

    // Start sending packets
    if (strcmp(argv[1], "start") == 0) {
      start();
      return (TCL_OK);
    }

    // Stop sending
    if (strcmp(argv[1], "stop") == 0) {
      stop();
      return (TCL_OK);
    }

    // Print statistics
    if (strcmp(argv[1], "printresults") == 0) {
      results();
      return (TCL_OK);
    }

  }
  // If the command hasn't been processed by PingAgent()::command,
  // call the command() function for the base class
```

```
  return (Agent::command(argc, argv));
}

//////////////////////////////////////////////////////
// SimplePEPAgent::start
//   Start sending packets
//
void SimplePEPAgent::start()
{
  recalctxrate();
  if (packetlevelfec_) {
    packetlimit_ -= (packetlimit_ % fecblocksize_);
  }

  printf("Scheduling packet every %f seconds\n", interval);
  printf("Send rate: %f Mbps  Block size: %d packets\n", sendrate_,
blocksize_);
  printf("Acking is %s\n", enableacks_?"enabled":"disabled");
  printf("Packet limit: %d\n", packetlimit_);
  fflush(stdout);

  reschedule_transmittimeout();       // start the send timer
  if (errormodel_) errormodel.init(); // start the error model
}

//////////////////////////////////////////////////////
// SimplePEPAgent::start
//   Stop sending packets
//
void SimplePEPAgent::stop()
{
  if (transmit_timeout.status()==TIMER_PENDING)
    transmit_timeout.cancel();
  if (sendtimer.status()==TIMER_PENDING)
    sendtimer.cancel();
  if (errormodel_) errormodel.stop();
  results();
}


//////////////////////////////////////////////////////
// SimplePEPAgent::results
//   Print results when simulation ends
//
void SimplePEPAgent::results()
{
  printf("\nSIMULATION RESULTS\n");
  printf("==================\n");
  printf("Current time: %f\n", Scheduler::instance().clock());
  printf("Total bytes sent: %d\n", bytessent_ + bytesretransmitted_ +
bytesfecsent_);
  printf("Total data bytes sent: %d\n", bytessent_);
  printf("Total fec bytes sent: %d\n", bytesfecsent_);
  printf("Total retransmitted bytes: %d\n", bytesretransmitted_);
  printf("Total overhead bytes: %d\n", bytesfecsent_ +
bytesretransmitted_);
}
```

```
//////////////////////////////////////////////////////
// SimplePEPAgent::recalctxrate
//   Schedule the transmission timer based on the desired
//   sending rate
//
void SimplePEPAgent::recalctxrate()
{
  interval = blocksize_ / (sendrate_ * 1000000.0 / 8.0);

  // adjust interval if we are "DVB aware"
  if (dvbaware_)        { interval*=(overhead[currentfeclevel_]); }
  reschedule_transmittimeout();
}

//////////////////////////////////////////////////////
// SimplePEPAgent::setfec
//   Called by FECTimer when timer expires.
//
void SimplePEPAgent::setfec(int feclevel)
{
  if (feclevel!=currentfeclevel_){
    currentfeclevel_=feclevel;
  }
  available_bitrate_ = sendrate_/overhead[currentfeclevel_];
  recalctxrate();
}

//////////////////////////////////////////////////////
// SimplePEPAgent::adapt
//   Schedule the timer to set the FEC level to the new state.
//
void SimplePEPAgent::adapt(int feclevel)
{
  if (fec_timer.status()!=TIMER_PENDING)
  {
    fec_timer.level = feclevel;
    fec_timer.resched(0.1); // 100 ms delay on out of band channel
  }
}

//////////////////////////////////////////////////////
// SimplePEPAgent::reschedule_timeout
//   Reschedules the retransmission timer based on the oldest
//   sent packet.
//
void SimplePEPAgent::reschedule_timeout()
{
  int    i, ptr;
  double currenttime = Scheduler::instance().clock();
  double mintime     = currenttime;
  bool   outstanding = false;

  for (i=0; i<WINDOW_SIZE_; i++)
  {
    ptr=(tx_last+i)%WINDOW_SIZE_;
    if (tx_window[ptr] && (selectiveack_==0 || !tx_acked[ptr]))
```

```
      {
        outstanding = true;
        if (tx_timers[ptr]<mintime) mintime=tx_timers[ptr];
      }
    }
    if (outstanding)
    {
      double timeoutvalue = (RTT_*2.0)-(currenttime-mintime);
      if (timeoutvalue<0.0) timeoutvalue=0;
      transmit_timeout.resched(timeoutvalue);
    }
}

////////////////////////////////////////////////////
// SimplePEPAgent::reschedule_transmittimeout()
//   Reschedule the transmit timeout to send a packet
//    at the appropriate output bitrate.
//
void SimplePEPAgent::reschedule_transmittimeout()
{
   double currenttime      = Scheduler::instance().clock();
   double nextpackettime   = (last_packetsent + interval);
   double nexttime         = 0;

   if (nextpackettime>currenttime)
   {
     nexttime = (nextpackettime-currenttime);
   }
   if (nexttime<0.0) nexttime=0;
   sendtimer.resched(nexttime); // reschedule next send event
}

////////////////////////////////////////////////////
// SimplePEPAgent::timeout()
//
//   Called when the retransmission timer expires. Re-sends
//   a packet immediately if its timer has expired.
//
void SimplePEPAgent::timeout()
{
   int    i, ptr;
   double currenttime = Scheduler::instance().clock();

   // Re-send a packet if it has timed out
   for (i=0; i<WINDOW_SIZE_; i++)
   {
     ptr=(tx_last+i)%WINDOW_SIZE_;
     if (tx_window[ptr] && (selectiveack_==0 || !tx_acked[ptr]) &&
   (currenttime-tx_timers[ptr])>=RTT_*1.99) {
       tx_timers[ptr]=currenttime;
       send_data_packet(tx_seqno+i, tx_blocknumber[ptr], true);
     }
   }
   reschedule_timeout();
}
```

```
//////////////////////////////////////////////////////
// SimplePEPAgent::send_data_packet()
//
//   Sends a DATA packet with the desired sequence number.
//   Reschedules the retransmit and send timers.
//
void SimplePEPAgent::send_data_packet(unsigned int sequence_number,
unsigned int blocknumber, bool retransmit, bool redundancy)
{
     Packet* pkt = allocpkt();
     hdr_simplepep* pephdr=hdr_simplepep::access(pkt);

     pephdr->seqno           = sequence_number;
     pephdr->packet_type   = SIMPLEPEP_TYPE_DATA;
     pephdr->fec_level       = currentfeclevel_;
     pephdr->blocknumber   = blocknumber;
     pephdr->redundancy    = redundancy;
     pephdr->blocksize       = fecblocksize_;
     pephdr->fecredundancy = fecredundancy_;

     // add overhead based on current protection level
     hdr_cmn::access(pkt)->size() =
   (int)(blocksize_*(overhead[currentfeclevel_]));

     // update statistics
     if (!retransmit && !redundancy)
     {
       packetssent_++;
       bytessent_+=blocksize_;
     }
     else if (redundancy)
     {
       bytespacketlevelfec_+=blocksize_;
     }
     else
     {
       packetsretransmitted_++;
       bytesretransmitted_+=blocksize_;
     }
     bytesfecsent_ += hdr_cmn::access(pkt)->size() - blocksize_;

     // output stats to tracefile if enabled
     if (tracedebug_) {
       FILE* f = fopen("trace.out", "a+");
       fprintf(f, "%f\tsend\t%d\t%d\t%s\t%s\t%d\t%f\n",
     Scheduler::instance().clock(), sequence_number, blocknumber,
     redundancy?"red":"-", retransmit?"rtx":"-", hdr_cmn::access(pkt)-
     size(), Scheduler::instance().clock()-last_packetsent);
       fclose(f);
     }

     send(pkt, 0);           // send one packet

     last_packetsent = Scheduler::instance().clock();

     reschedule_timeout();          // reschedule the send timeouts
     reschedule_transmittimeout();  // reschedule the send timer
```

```
}

//////////////////////////////////////////////////
// SimplePEPAgent::schedule_packet()
//
//   Schedules the sending of the packet to send.
//   Currently not used in favor of the send timer.
//
void SimplePEPAgent::schedule_packet(unsigned int sequence_number)
{
}

//////////////////////////////////////////////////
// SimplePEPAgent::is_packet_available()
//
//   Checks to see if we have room in the window to send
//   another packet, returning seqno of next one to send.
//
bool SimplePEPAgent::is_packet_available(unsigned int *seqno, unsigned
int *blocknumber, bool *redundancy)
{
  int    ptr;
  int    i;

  // check to see if we can move to next block for packet-level FEC
  // NOTE: packet level FEC is currently not used in simulations
  if (packetlevelfec_)
  {
    if (blocksent==fecblocksize_)
    {
      if (redcount<fecredundancy_)
      {
        *seqno=redcount++;
        *blocknumber=tx_block_last;
        *redundancy=true;
        return(true);
      }
      blocksent=0;
      tx_block_last++;
      redcount=0;
    }
  }

  // allocate and initialize a new packet to send
  for (i=0; i<WINDOW_SIZE_; i++)
  {
    ptr=(tx_last+i)%WINDOW_SIZE_;
    if (!tx_window[ptr]) {
      tx_window[ptr]=true;
      tx_acked[ptr]=false;
      tx_blocknumber[ptr]=tx_block_last;
      tx_timers[ptr]=Scheduler::instance().clock();
      *seqno=tx_seqno+i;
      *blocknumber=tx_block_last;
      *redundancy=false;
      blocksent++;
      return(true);
```

```
      }
    }
    return(false);
}

////////////////////////////////////////////////////
// SimplePEPAgent::sendpkt()
//
//   Called when the send timer expires. Sends a single packet,
//   if available, and reschedules the send timer.
//
void SimplePEPAgent::sendpkt()
{
    bool          packet_available   = false;
    unsigned int  sequence_number    = 0;
    unsigned int  blocknumber        = 0;
    bool          redundancy         = false;

    if (packetlimit_!=0 && packetssent_>=packetlimit_)
    {
      for (int i=0;  i<WINDOW_SIZE_; i++)
      {
        if (tx_window[i]) return;  // exit if there are unacked packets
      }
      stop(); // stop transmisson when packet limit has been reached
      return;
    }
    if (is_packet_available(&sequence_number, &blocknumber, &redundancy))
    {
      packet_available = true;
    }
    if (packet_available) {
      send_data_packet(sequence_number, blocknumber, false, redundancy);
    }
}

////////////////////////////////////////////////////
// SimplePEPAgent::SendACK()
//
//   Send a simplepep ACK or NAK packet to the transmitter
//
void SimplePEPAgent::send_ack_packet(unsigned int seqno, unsigned char
ack_nak, unsigned int acked_seqno, bool redundancy)
{
    Packet* pkt = allocpkt();
    hdr_simplepep* pephdr=hdr_simplepep::access(pkt);
    hdr_cmn::access(pkt)->size() = 10;

    pephdr->seqno=seqno;
    pephdr->acked_seqno=acked_seqno;
    pephdr->packet_type=ack_nak;
    pephdr->redundancy=redundancy;

    send(pkt, 0); // send one packet
}
```

```
//////////////////////////////////////////////////
// SimplePEPAgent::recv()
//
//   Overrides Agent::recv()
//   Called when the SimplePEPAgent receives a packet
//
void SimplePEPAgent::recv(Packet* pkt, Handler*)
{
  hdr_simplepep* pephdr=hdr_simplepep::access(pkt);
  double  currenttime = Scheduler::instance().clock();
  int          ptr;
  unsigned int   i;
  int     fec_level;
  int     currentblockcount=0;
  int           ackcount=0;
  unsigned int   block_start_seqno;
  unsigned int   block_packet_count;

   switch(pephdr->packet_type) {

   // Handle reception of a data packet (receiver)
   case SIMPLEPEP_TYPE_DATA:

      if (Transmitter_ && errormodel_)
      {
        SimplePEPAgent* txagent = (SimplePEPAgent*)Transmitter_;
        fec_level=(int)pephdr->fec_level;

        // Adapt to changing channel conditions
        if (adapt_)
        {
          txagent->adapt(txagent->errormodel.currentsnrlevel());
        }

        // Drop packet based on error correction and current SNR
        if (!txagent->errormodel.corrupt(fec_level))
        {
          packetserrors_++;
          Packet::free(pkt);
          return;
        }
      }

      if (!pephdr->redundancy)
      {
        if (pephdr->seqno<rx_seqno) {

          // send periodic ACK if we are using delayed ACK
          // NOTE: we do not use this in our simulations
          if (delayack_ || (currenttime-last_acksent)>0.1) {
            send_ack_packet(rx_seqno, SIMPLEPEP_TYPE_ACK, pephdr-
          >seqno, pephdr->redundancy);
            last_acksent=currenttime;
          }

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

```
    }

    if (pephdr->seqno-rx_seqno >= WINDOW_SIZE_) {
      printf("ERROR: Received packet outside allowable window.\n");
    Packet::free(pkt);
      return;
    }

   ptr=(rx_next+(pephdr->seqno-rx_seqno))%WINDOW_SIZE_;
   rx_window[ptr] = true;
}

// Handle erasure codes
// NOTE: packetlevel FEC not used in this simulation
if (packetlevelfec_)
{
   block_start_seqno  = rx_seqno + (pephdr->blocknumber-
rx_blocknumber)*fecblocksize_;
   block_packet_count = 0;

   if (pephdr->blocknumber < rx_blocknumber)
     { Packet::free(pkt); return; }

   if (pephdr->blocknumber-rx_blocknumber >= MAX_WINDOW_SIZE)
   {
     printf("ERROR: Received block outside allowable window.\n");
     Packet::free(pkt);
     return;
   }
   if (block_start_seqno-rx_seqno >= WINDOW_SIZE_)
   {
     printf("ERROR: This block number is invalid.\n");
   Packet::free(pkt);
     return;
   }

   ptr = (rx_block_next+(pephdr->blocknumber-
rx_blocknumber))%MAX_WINDOW_SIZE;

   if (pephdr->redundancy) rx_blockpacketcount[ptr]++;
   block_packet_count = rx_blockpacketcount[ptr];

   // see if we have enough packets to reconstruct the block
   for (i=block_start_seqno; i<(rx_seqno+WINDOW_SIZE_); i++)
   {
     ptr=(rx_next+(i-rx_seqno))%WINDOW_SIZE_;
     if (rx_window[ptr]) block_packet_count++;
   }

   // mark whole block as received
   if (block_packet_count>=fecblocksize_)
   {
     for (i=block_start_seqno; i<(rx_seqno+fecblocksize_); i++)
     {
       ptr=(rx_next+(i-rx_seqno))%WINDOW_SIZE_;
       rx_window[ptr]=true;
     }
```

```
    }
  }

  // Advance receive window for all in-order packets.
  // When using erasure codes, cannot advance until whole
  // block has been receieved.
  i=0;
  while (rx_window[(rx_next+i)%WINDOW_SIZE_] && i<WINDOW_SIZE_){
    if (packetlevelfec_)
    {
      currentblockcount++;
      if (currentblockcount==fecblocksize_)
      {
        ackcount+=fecblocksize_;
        currentblockcount=0;
        rx_blocknumber++;
        rx_blockpacketcount[rx_block_next]=0;
        rx_block_next=(rx_block_next+1)%MAX_WINDOW_SIZE;
      }
    } else {
      ackcount++;
    }
    i++;
  }

  for (i=0; i<ackcount; i++)
  {
    ptr=(rx_seqno+i)%WINDOW_SIZE_;
    rx_window[ptr]=false;
    rx_nak[ptr]=false;
  }
  rx_seqno+=ackcount;
  rx_next=(rx_next+ackcount)%WINDOW_SIZE_;

  // send acknowledgement
  if (!delayack_ || (currenttime-last_acksent)>0.1) {
    send_ack_packet(rx_seqno, SIMPLEPEP_TYPE_ACK, pephdr->seqno,
  pephdr->redundancy);
    last_acksent=currenttime;
  }

  // send NAK for all non-received packets from start of window to
  // current received packet. NAK is sent once for each packet.
  if (usenak_)
  {
    for (i=rx_seqno; i<pephdr->seqno; i++) {
      ptr=(rx_next+(i-rx_seqno))%WINDOW_SIZE_;
      if (!rx_nak[ptr] && !rx_window[ptr]) {
        send_ack_packet(rx_seqno, SIMPLEPEP_TYPE_NAK, i, pephdr-
      >redundancy);
        rx_nak[ptr]=true;
      }
    }
  }
  break;

// Handle reception of an ACK or NAK packet (transmitter)
```

```
case SIMPLEPEP_TYPE_NAK:
case SIMPLEPEP_TYPE_ACK:

  // selective ack the packet
  if (pephdr->packet_type==SIMPLEPEP_TYPE_ACK && !pephdr->redundancy)
  {
    ptr = pephdr->acked_seqno-tx_seqno;
    if (ptr>0 && ptr<=WINDOW_SIZE_) {
      i=(tx_last+ptr)%WINDOW_SIZE_;
      if (tx_window[i]) tx_acked[i]=true;
    }
  }

  // if this is a new ACK, slide window forward to seqno+1
  ptr = pephdr->seqno-tx_seqno;
  if (ptr>0 && ptr<=WINDOW_SIZE_) {
    for (i=0; i<ptr; i++) {
      tx_window[(tx_last+i)%WINDOW_SIZE_]=false;
      tx_acked[(tx_last+i)%WINDOW_SIZE_] =false;
    }

    // advance transmit window
    tx_last   = (tx_last+ptr)%WINDOW_SIZE_;
    tx_seqno  = pephdr->seqno;
  }

  // re-send the packet immediately if NAK'ed
  if (pephdr->packet_type==SIMPLEPEP_TYPE_NAK) {
    ptr = (tx_last+(pephdr->acked_seqno-tx_seqno))%WINDOW_SIZE_;
    send_data_packet(pephdr->acked_seqno, tx_blocknumber[ptr], true);
  }

  // Calculate bandwidth estimate
  if (bwadapt_ && pephdr->packet_type==SIMPLEPEP_TYPE_ACK &&
last_ackreceived>0.1)
  {
    bandwidth_estimate = 0.9*bandwidth_estimate +
  0.1*blocksize_*8.0/(currenttime-last_ackreceived)/1000000;
    last_ackreceived = currenttime;

    if (bandwidth_estimate < (sendrate_*0.95)) {
      sendrate_ = bandwidth_estimate;
      lastincrease=0;
    }
    else if (bandwidth_estimate >=(sendrate_/0.95)) {
      sendrate_ = min(1.50, bandwidth_estimate);
      lastincrease=currenttime;
    }
    else if ((currenttime-lastincrease)>=1.5 && bandwidth_estimate
  >= (sendrate_*0.99))
    {
      sendrate_ = min(1.50, sendrate_/0.95);
      lastincrease=0;
    }
    recalctxrate();
  }
  if (pephdr->packet_type==SIMPLEPEP_TYPE_ACK)
```

```
        last_ackreceived=currenttime;

      if (tracedebug_)
      {
        FILE* f = fopen("trace.out", "a+");
        fprintf(f, "%f\tack\t%d\t%d\t%d\t%d\n",
      Scheduler::instance().clock(), pephdr->seqno, pephdr->acked_seqno,
      tx_last, tx_seqno);
        fclose(f);
      }

      reschedule_timeout();
      reschedule_transmittimeout();

      break;
    }

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

7.3 SimplePEP OTcl Sample Script

```
#
# simplepep-script.tcl
#

#######################################################
# Simulation variables
# Change these to modify the simulation parameters
#

# Error rate for PER model used for TCP comparisons. Usually disabled.
set errorrate          0.0

# Output bitrate of the SimplePEP server in Mb/s
set sendrate           1.5

# Enable/disable DVB aware functionality
set dvbaware           0

# Enable/disable Packet-level FEC. Disabled in this paper.
set packetlevelfec     0
set fecredundancy      1
set fecblocksize       5

# Window size in packets
set windowsize         63

# Queue size of server gateway, in packets
set queuesize          17

# Packet size of SimplePEP packets without FEC
set blocksize          1500

# One-way round trip time (multiplied by 2 inside simplepep.cc)
set roundtrip          0.5

# Current FEC protection level (0, 1, or 2)
set fec                2

# Number of packets to send before quitting (file size)
set packetlimit        50000

# Enable/disable selective ACKs. Always enable.
set selectiveack       1

# Enable/disable internal error model. Disable when running TCP
comparison test.
set errormodel         1

# Enable adaptive error correction. Always enable.
set adapt              1

# Enable/Disable bandwidth adaptation.
set bwadapt            1
```

```
# Output file for tracing
set resultsfile       results.txt


# Enable/disable NAKs (always enable)
set usenak            1

#########################################################
# Finish procedure. Called to log results when the simulation ends.
#
proc finish {} {
   global ns errorrate bytessent  p0 resultsfile sendrate

   # calculate average bitrate and total sent bytes
   set totaltime [expr [$p0 set last_packetsent] - 0.1]
   set bitrate [expr [expr [$p0 set bytessent_] * 8 / 1000000] / [expr
   ($totaltime)] ]
   set totalbytessent [expr [$p0 set bytessent_] + [$p0 set
bytesfecsent_] + [$p0 set bytesretransmitted_]]

   # output bitrate to console
   puts "bitrate at server: $bitrate"

   # save stats to results file
   set rf [open $resultsfile a]
   puts $rf "$sendrate $bitrate $totaltime [$p0 set bytessent_] [$p0 set
bytesfecsent_] [$p0 set bytesretransmitted_] $totalbytessent"
   close $rf

   # exit the simulation
   exit 0
}

#########################################################
# Statistics reporting procedure.
# Called every 2 seconds to log running stats.
#
proc reportstatistics {} {
   global ns p0 p1 oldbytes oldfec bitrate fecbitrate oldretx
retxbitrate packetlimit availablebitrate

   set interval 2.0
   set lambda   0.6

   # obtain current time
   set now [$ns now]

   # open output files
   set statsfile           [open simplepepstats-bitrate.txt a+]
   set fecstatsfile        [open simplepepstats-fecbitrate.txt a+]
   set retxstatsfile       [open simplepepstats-retxbitrate.txt a+]
   set availablestatsfile  [open simplepepstats-availablebitrate.txt a+]
   set totalstatsfile      [open simplepepstats-totalbitrate.txt a+]

   # get current stats from SimplePEP server
   set  bytessent            [$p0 set bytessent_]
   set  packetssent          [$p0 set packetssent_]
```

```
    set  bytesfecsent          [$p0 set bytesfecsent_]
    set  bytesretransmitted    [$p0 set bytesretransmitted_]
    set  packetsdropped        [$p0 set packetsdropped_]
    set  packetsretransmitted  [$p0 set packetsretransmitted_]
    set  packetserrors         [$p0 set packetserrors_]
    set  currentfeclevel       [$p0 set fec_]
    set availablebitrate       [$p0 set available_bitrate_]

    # calculate bitrates
    set bitrate [expr ($bitrate*($lambda))+($bytessent-
$oldbytes)/$interval/1000000*8*(1-$lambda)]
    set fecbitrate [expr ($bytesfecsent-$oldfec)/$interval/1000000*8]
    set retxbitrate [expr ($retxbitrate*($lambda))+($bytesretransmitted-
$oldretx)/$interval/1000000*(1-$lambda)]
    set totalbitrate [expr ($bitrate + $retxbitrate)]

    # save stats for next iteration
    set oldbytes $bytessent
    set oldfec $bytesfecsent
    set oldretx $bytesretransmitted

    # save stats in output files
    puts $statsfile "$now $bitrate"
    puts $fecstatsfile "$now $fecbitrate"
    puts $retxstatsfile "$now $retxbitrate"
    puts $availablestatsfile "$now $availablebitrate"
    puts $totalstatsfile "$now $totalbitrate"

    # close output files
    close $statsfile
    close $fecstatsfile
    close $retxstatsfile
    close $availablestatsfile
    close $totalstatsfile

    if { $packetssent < $packetlimit } {
      $ns at [expr $now+$interval] "reportstatistics"
    }
}

#####################################################
# Simulation script
#

# Create an ns-2 simulator object
set ns [new Simulator]

# Initialize stats and output files
file delete simplepepstats-bitrate.txt
file delete simplepepstats-fecbitrate.txt
file delete simplepepstats-retxbitrate.txt
file delete simplepepstats-availablebitrate.txt
file delete simplepepstats-totalbitrate.txt
file delete trace.out
set oldbytes 0
set oldfec 0
set oldretx 0
```

```
set bitrate 0
set fecbitrate 0
set retxbitrate 0

# Seed the internal random number generator
global defaultRNG
$defaultRNG seed 0

# Create the server proxy node
set n0 [$ns node]

# Create the client proxy node
set n2 [$ns node]

# Connect the nodes with simplex links
$ns simplex-link $n0 $n2 1.50Mb 250ms DropTail
$ns simplex-link $n2 $n0 1.50Mb 250ms DropTail

# Create the PER error model
set loss_module [new ErrorModel]
$loss_module set rate_ $errorrate
$loss_module unit pkt
$loss_module ranvar [new RandomVariable/Uniform]
$loss_module drop-target [new Agent/Null]

# Attach the PER model to the link from server to client proxy
$ns lossmodel $loss_module $n0 $n2

# Create SimplePEP agents and attach to server and client nodes
set p0 [new Agent/SimplePEP]
$ns attach-agent $n0 $p0

set p1 [new Agent/SimplePEP]
$ns attach-agent $n2 $p1

# Connect the two agents
$ns connect $p0 $p1

# Set the queue size at the server gateway
$ns queue-limit $n0 $n2 $queuesize

# Setup the SimplePEP server
$p0 set blocksize_      $blocksize
$p0 set sendrate_       $sendrate
$p0 set WINDOW_SIZE_    $windowsize
$p0 set RTT_            $roundtrip
$p0 set fec_            $fec
$p0 set packetlimit_    $packetlimit
$p0 set selectiveack_   $selectiveack
$p0 set dvbaware_       $dvbaware
$p0 set packetlevelfec_ $packetlevelfec
$p0 set fecredundancy_  $fecredundancy
$p0 set fecblocksize_   $fecblocksize
$p0 set errormodel_     $errormodel
$p0 set bwadapt_        $bwadapt
```

```
# Setup the SimplePEP client
$p1 set WINDOW_SIZE_     $windowsize
$p1 set enableacks_      1
$p1 set adapt_           $adapt
$p1 set Transmitter_     $p0
$p1 set packetlevelfec_ $packetlevelfec
$p1 set fecredundancy_   $fecredundancy
$p1 set fecblocksize_    $fecblocksize
$p1 set errormodel_      $errormodel
$p1 set usenak_          $usenak

# Schedule simulator events (start simulation, report stats, finish)
$ns at 0.1 "$p0 start"
$ns at 0.1 "reportstatistics"
$ns at 10000.0 "finish"

# Run the simulation
$ns run
```

7.4 Adding SimplePEP to ns-2

This section will describe the steps necessary to add the SimplePEP agent to a build of the ns-2 simulator so that the results obtained in this thesis can be reproduced. The steps below assume that a working installation of ns-2 exists on the target system. Instructions for obtaining and building the source code can be obtained from the ns-2 website [NS2Sim].

i) Add the SimplePEP packet type to the simulator by modifying the file "common/packet.h" in the ns-2 source tree and adding `PT_SIMPLEPEP` to the packet type enumeration.

```
enum packet_t {
   PT_TCP,
   PT_UDP,

   ......

   // insert new packet types here
   PT_SIMPLEPEP,
   PT_NTYPE // This MUST be the LAST one
};
```

ii) Add the SimplePEP packet name ("SimplePEP") to the `p_info` structure in the same file.

```
class p_info {
public:
   p_info() {
      name_[PT_TCP]= "tcp";
      name_[PT_UDP]= "udp";

      ...........

      name_[PT_ SIMPLEPEP]="SimplePEP";
      name_[PT_NTYPE]= "undefined";
   }
   .....
 }
```

iii) Add SimplePEP to the OTcl script responsible for handling ns-2 packet headers. To do this, add SimplePEP to the list of packet headers located in "tcl/lib/ns-packet.tcl".

```
foreach prot {
   AODV
   ARP

   ....

   SimplePEP
} {
   add-packet-header $prot
}
```

iv) Add SimplePEP to the ns-2 Makefile, so that it will be built into the simulator.

```
OBJ_CC = \
   tools/random.o tools/rng.o tools/ranvar.o common/misc.o
   common/timer-handler.o \
   common/scheduler.o common/object.o common/packet.o \

   ....

   simplepep.o \

   ....
```

v) Finally, rebuild the simulator using the tools required by the target platform. For a Linux machine, simply type "make depend" and "make" to complete the installation.