This is to certify that the

dissertation entitled

Noise Tolerant Compression Protocols for Wireless Environments

presented by

Stephen James Perkins

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Matt W. Mutka
Major professor

Date April 24, 1997

Noise Tolerant Compression Protocols for Wireless Environments

By

Stephen James Perkins

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1997

Abstract

Noise Tolerant Compression Protocols for Wireless Environments

By

Stephen James Perkins

Wide area wireless networks will have limited bandwidth. One standard approach for increasing the effective throughput on such links is to employ data compression. In such systems, the sender applies a compression algorithm to selected portions of a datagram and the receiver reverses the process to obtain a duplicate of the original. Implicit in this process is the notion that both endpoints maintain identical state information so that their work can remain synchronized. Errors in the data can cause a loss of synchronization that will eventually lead to incorrect results after decompression.

This work investigates the performance of two link level compression algorithms operating in the presence of noise. The initial sections explore system operation by providing a low–bandwidth traffic survey and a detailed analysis of the noise–free behavior of the compression algorithms. The remainder of the document investigates algorithm behavior in the wireless environment. In this environment, it is likely that noise corruption will cause synchronization loss and a corresponding reduction in system performance.

A new header compression protocol and a new payload compression protocol are presented. The modified algorithms allow the compressors to better tolerate errors. The new

header compression protocol is shown to behave comparably to the standard algorithm under noise–free conditions. It significantly outperforms the standard under noisy conditions. The new payload compression protocol is also shown to successfully reduce the effects of synchronization loss. However, its throughput is limited by the failure of the TCP implementation to adequately handle noise. The TCP problem is further investigated and references are presented that address the deficiency.

This work is dedicated to my parents. Their love and support kept me going.

Acknowledgments

I first and foremost want to thank my friends for helping me with the transition from a warm Southern climate to a cold Northern landscape. Y'all will remain the brightest part of my MSU memories! I also want to thank my professors for all their guidance and hard work. Y'all got me moving in the proper direction. A special thanks goes to TechSmith Corporation! I couldn't have finished without their financial and technical support. Thank you! Also, thanks go to Lighthouse Design for their superb academic software, NeXT Software, Inc. for their OS support, and Watershed Technologies for their software donations. Finally, I wish to thank all those who helped collect data using the PPP software. Chapter 4 would not exist without you folks!

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction

Wide–area wireless networks will play an increasingly important role as societies move toward ubiquitous connectivity. While wired networks are approaching data rates that make the use of many existing communications protocols inefficient, their wireless counterparts continue to be bandwidth limited. Enhancements to these wireless links will be critical for providing a robust and usable network infrastructure. This introduction chapter discusses how bandwidth constraints and wireless noise characteristics require special consideration in the design of low–level compression routines. It outlines the steps taken in this dissertation to develop new error tolerant data compression protocols. These new protocols provide superior performance in a noisy environment while maintaining good performance in the error free environment. The discussion will start by addressing wireless bandwidth.

1.1 The Wireless Bandwidth Constraint

Cost factors, spectrum allocation, and a variety of channel impairments will impose severe bandwidth limitations on wide–area networks [3, 21, 22, 27]. It is important to understand the significance of the bandwidth problems facing wireless systems. While wire

based systems may overcome bandwidth limitations via technology (e.g., analog phone systems being replace by ISDN, xDSL, and other higher bandwidth systems), wireless systems are faced with a fundamentally different obstacle, that of shared frequency spectrum. Since the electro–magnetic spectrum is a truly shared resource, strict laws govern its use. National and international regulatory agencies allocate most of the usable spectrum and devices are not allowed to arbitrarily increase their operating bandwidth. To make matters worse, wireless communication systems typically receive narrow spectrum allotments for their use.

Since bandwidth is directly related to the maximum signalling rate, narrow band allocations put an upper limit on the signalling speeds of a wireless transceiver. Roden [41] examines this limit and provides the mathematical relationship between bandwidth and signalling rate. Briefly, he shows that in order to increase the rate at which a signal changes (i.e., increase its baud rate), one must allow the signal to range over a larger set of frequencies (i.e., take up greater bandwidth). Given a fixed bandwidth allotment, and thus a maximum signalling rate, the only options for increasing the data rate are to use larger signalling constellations that allow more bits per baud to be transferred. This is the mechanism that has been used to provide V.32 and V.34 telephone modems. Unfortunately, this approach quickly becomes limited by noise.

Since signalling rates are fixed by the allocation of spectrum, and the number bits per baud is limited by the attainable constellation size given the noise, other mechanisms must be employed to further increase the throughput of these systems. The following section outlines several general approaches that can be used to increase the performance of these systems.

1.2 Approaches to Performance Enhancement

Communications systems are large and contain many complex interacting components. Enhancements to such systems can be performed in many independent areas and with independent optimization goals. Figure 1.1 provides a way of looking at optimization areas. While this is certainly not the only way of grouping optimizations, it is useful for discussing where this work fits with respect to other performance enhancement endeavors.

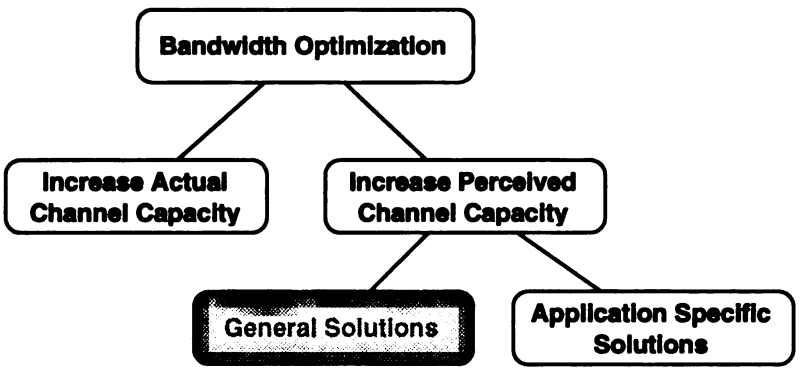

Figure 1.1: Areas of System Optimization

1.2.1 Increasing Actual Channel Capacity

Enhancement techniques that fall into this category attempt to increase the actual channel capacity of a system. There are several ways of doing this:

- Increase the bit rate of the channel. This can be done by increasing the bits per baud on analog systems. For example, moving from V.32bis to V.34 on modem based systems keeps the same baud rate but adds one more bit per baud (doubling the number of values that can be decoded).

- Provide better source encoding by matching the source encoding to the particular operating environment chosen. Error detection and error correction schemes have a

tradeoff between performance and overhead.

- Reduce the noise on the channel. As confirmed by Shannon's channel capacity theorem [42], reducing the noise present on a channel will increase the capacity of the channel. Indeed, if it was not for noise, any channel (regardless of its bandwidth) could have infinite capacity. Examples of noise reduction would be:
  - Line conditioning for analog telephone lines.
  - Interference reduction in wireless media, either through CDMA techniques, MAC techniques, or by moving to a less crowded area of the spectrum.
- Move to a higher bandwidth media. For example, moving from twisted pair wire to coaxial cable or from coaxial cable to fiber–optics.

It should be apparent that most of this work is actually performed at the link and physical layers. Further, it is typically a technological solution. That is to say that increases in performance are typically gained by employing better technology. However, many of these techniques reach a point of diminishing returns as one approaches the maximum theoretical limits of a channel. At some point, it may not be worth the cost (either in terms of dollars or computing power) to increase the bit rate of a particular system configuration.

1.2.2 Increasing Perceived Channel Capacity

Enhancement techniques that fall into this category attempt to increase the *effective* bandwidth of the system. These techniques are designed to reduce the latency and *perceived* wait time that users experience while using the system. These methods can again be broken into two separate categories:

Application Specific These enhancements are designed to increase the performance of one particular application. Users will perceive an increase in performance only as long as they are using that specific application. Examples of such optimizations would be custom compression techniques for certain data types (e.g., JPEG for images) or optimization of specific data transfers (e.g., Protocol Assist from TechSmith Corporation or XRemote for X–Window sessions). The effectiveness of these techniques may be reduced when multiple applications share the use of a single channel.

General These enhancements increase the system performance independent of specific applications. The mechanisms may take the form of better support for multiple access, quality of service mechanisms for shared channels, or data compression of low–level multiplexed data streams.

This dissertation focuses on increasing the *perceived* bandwidth of a wireless system by enhancing data link level compression algorithms that operate in the noisy wireless environment. By concentrating on compressors at the data link layer, all higher layer traffic will benefit from the compression algorithms. Data link layer compressors are "general" solutions and fall into the shaded "General Solutions" box of Figure 1.1. The compressors are further described in the following section.

1.3 Data Compression

Since wide–area wireless links have such low bit–rates, the sending system has enough time to execute algorithms that compress the data prior to transmission. If the receiver decompresses the data before handoff to the receiving process, then that process will appear to see a link that has higher capacity. While such compression mechanisms can exist

at several layers of a communications stack, this dissertation will focus on compression algorithms that exist at the data link layer. As such, the input to the compressor will be datagrams that are passed down from the network layer.

1.3.1 Header and Payload Compression

The datagram structure naturally leads to a class of compressors that focus on reducing protocol header size and a class of compressors that focus on reducing packet payload size. *Protocol header* compression algorithms are important because they can significantly reduce the overhead of transport protocols for interactive sessions. These compressors provide noticeable benefits for systems that require a small maximum transmission unit (MTU). Unfortunately, header compression algorithms are usually tied closely to the structure of a protocol header and cannot be extended to the datagram payload. Instead, they provide a small but fixed gain using few CPU cycles.

Algorithms that are applied to the *payload* of a datagram can significantly reduce the datagram size. While these algorithms may not provide ratios as high as header compressors when specifically applied over header data, they can be applied over the more general payload data. Payload compression, especially when measured on bulk data transfers, can give byte reductions that far exceed the effects of header compression. This is of utmost importance for wireless and other low–bandwidth interfaces.

1.3.2 Compression Dictionaries

The datagram compression process requires the maintenance of some type of state table or dictionary. This dictionary directs the encoding and decoding process. The earliest compression algorithms maintained a fixed dictionary that was created and initialized

identically in both the compressor and decompressor before operations began. This hard-coded dictionary did not change during operation and removed the need for the compressor to explicitly communicate its dictionary to the decompressor. Since the encoding was fixed, all compressed packets were encoded in an identical fashion and did not depend on each other. The drawback of the encoding was that the dictionary had to be designed to give the best "overall" performance and could not adapt to variations in the input. Therefore, the compression ratios provided by this type of compression were not very good.

More complex compression algorithms used an adaptive dictionary. In such systems, the encoding dictionary was built according to the characteristics of the incoming data. Custom built dictionaries allowed these algorithms to provide better ratios than the fixed dictionaries described earlier. The problem with adaptive strategies was that the dictionary had to be communicated to the peer. This could be done one of two ways. Explicitly, meaning the dictionary was sent to the decompressor before the sequence began, or implicitly, meaning the dictionary was recreated at the peer according to the data received.

Explicit dictionary communication has some unfortunate drawbacks for use in a packet based environment. First, it requires a large transmission overhead. This is not acceptable in a low–bandwidth environment. It also makes the assumption that the input is entirely available and already compressed. This is not the case for packet based compressors. In these systems, input can be considered as an infinite stream of datagrams passed down from higher layers. To address these concerns, implicit transmission is used.

The implicit transmission approach requires that the compressor and decompressor start out with identical (possibly empty) dictionaries. As the compressor encodes the input, it updates its dictionary and sends enough information to the peer to allow it to decode the input and create a dictionary entry. In this way, the compressor and decompressor dynamically

build up identical copies of the dictionary based on the characteristics of the transmitted data. Proper implementation of this approach can allow the implicit transmission of a dictionary with almost no transmission overhead.

1.3.3 Synchronization Requirements

Algorithms that build dictionaries through implicit transmission must maintain precise synchronization between the endpoints. Since the decompressor's dictionary is derived from the input stream, that stream must precisely match the stream generated as the output of the compressor. For file based compressors, this is typically not a problem since the path from the compressor to disk to decompressor is typically error free. However, for wireless channels, this is not the case. A single lost or corrupted packet can cause the receiver's input data stream to become unsynchronized with the compressor's output stream. Continued operation will cause the dictionaries to diverge and will produce incorrect decompressed output.

In order to maintain dictionary coherence, techniques are employed to monitor the datastream and detect loss of synchronization. Upon such detection, the algorithms enter a resynchronization state. While in this state, all received packets are discarded until synchronization is regained. Upon resynchronization, normal operation begins again. It is then the responsibility of a higher layer (the TCP transport layer in this case) to detect any discarded packets and retransmit them.

The resynchronization mechanism can cause severe reductions in system performance. A naive resynchronization implementation can give acceptable performance when there are not many errors. However, in the wireless environment, the higher probably of error can cause a naive implementation to reduce the performance of the compressor to below that of

not using compression. More sophisticated resynchronization procedures are needed when operating in such environments.

1.4 Problem Statement – Synchronization Loss in Wireless Environments

It should be clear that wide–area wireless networks are bandwidth constrained and noise limited. Maximizing the capacity of such links is critical for providing a robust and usable environment. Current packet compression algorithms do not work well in this wireless environment. Noise corrupted packets tend to invoke expensive resynchronization procedures. Excessive packet loss and retransmissions then significantly reduce the usable capacity of the link. By making these compression algorithms more tolerant to noise, one could reduce the number of discarded and retransmitted packets. This would increase the perceived capacity of a link by removing redundant transmissions.

In order to write noise tolerant algorithms effectively, one must understand many aspects of the system operation. Specifically, one needs to:

- Understand the channel. Wide–area wireless links are often noisy and error prone. To account for the errors, one must understand their characteristics.
- Identify the types of application level traffic being generated, taking into account the low–bandwidth nature of the underlying channel. This traffic must be carefully examined since it forms the input to the compression algorithms.
- Identify what traffic is actually flowing over the link. On top of application level traffic types, one must account for the effects of protocol overhead and the reductions provided by data compression.

- Identify the shortcomings of current compression algorithms in noisy environments. How do current algorithms perform when moved to the hostile wireless environment?

- Enhance selected algorithms so that they demonstrate reasonable performance over noisy links.

This dissertation addresses all these areas by examining a working system and analyzing its behavior when wireless noise characteristics are added. It is asserted and shown that new noise tolerant compression protocols can reduce the synchronization requirements between the compressor and decompressor. Such protocols will work in a noise free environment with little noticeable degradation while their performance in a noisy environment can far exceed the original protocols.

1.5 Contributions of this Work

This work will make several contributions. A traffic survey and analysis yields information on traffic mixes produced by users on restricted bandwidth systems. The results presented are based on end–point collections rather than the standard network backbone collections. This survey provides hard evidence about what applications people choose to use when they are on bandwidth restricted systems. The second contribution is an analysis of exactly how compression affects packets on a link. Typical compression statistics provide an average compression ratio. This work goes beyond simple ratio statistics and investigates the effectiveness of compression by looking at individual application level protocols at the packet level of granularity. Such information provides important details that can be used to enhance specific application level and link level compressors. Finally, this dissertation presents two noise tolerant protocols for use with compression algorithms operating

in wireless environments. Using the top application level traffic types found in the traffic survey, the new compression protocols are shown to outperform their original counterparts when used in Gaussian and dependent error environments.

1.6 Document Organization

This dissertation is presented in a bottom up fashion. Following the review of previous work in Chapter 2, Chapter 3 addresses the environment used for all experiments. It specifically combines the results in several fields of communication to present an error model that generates burst noise characteristics similar to those found in wide–area wireless networks. Chapter 4 reassesses the assumptions made in previous traffic analysis experiments and then describes a traffic survey conducted under conditions similar to a wide–area wireless network. Unlike previous surveys, this work takes into account the low–bandwidth nature of the channel and then analyzes application level traffic on a *per user basis*. The *per user* results provide the data for the analysis in Chapter 5. Chapter 5 looks at the effectiveness of current compression protocols over noise–free channels and provides a base of comparison for later enhancements. Using the noise–free channel as a base, Chapters 6 and 7 show how current protocols fail to operate successfully in a noisy environment. Each of these chapters then describes a new *noise tolerant* method for communicating compressed packets over noisy channels. Chapter 8 summarizes the entire document and describes areas for future work.

Chapter 2

Previous Work

Chapter 1 motivates investigations in several diverse fields including error modeling, network traffic analysis, data compression, and network protocols. This chapter outlines relevant works in those areas. Because the areas are somewhat dissimilar, a section will be devoted to each with the understanding that later chapters will draw from these previous works. Since noise characterization is a basic requirement for developing and testing noise tolerant protocols, noise modeling techniques will be addressed first. Following that is a review of network traffic collection endeavors. The review includes a rationale for why previous works are inadequate for this investigation and identifies why the traffic analysis presented in Chapter 4 is unique. Finally, data compression methodologies and their communications protocols are reviewed. This last section includes reviews of advanced ARQ protocols that are designed to enhance the performance of TCP connections. While the ARQ works do not seem immediately relevant, these protocols will become important for understanding the results presented in Chapter 7.

2.1 Wireless Network Error Characteristics

Shannon's pioneering work on information theory ([42]) describes channel capacities in the presence of noise. It shows noise as being the fundamental limitation in any communications channel. Because of its importance, any investigation of channel performance must start with a solid characterization of noise. This section will discuss models for describing burst noise channels and provides a justification for the noise model chosen.

2.1.1 Channels with Memory

Noise models are tied closely with the medium over which they occur. Wide–area wireless channels can be characterized as having numerous impairments including multi–path fading, power constraints, spectrum constraints, environmental characteristics (e.g., rain or snow), terrain impairments (e.g., hills and foliage), and co–channel transmission interference. It is universally acknowledged that errors encountered in such transmission systems occur in bursts [28]. With burst errors, there is a statistical dependence that is not captured using a Gaussian error distribution. Channels exhibiting these statistical dependencies are said to have *memory*.

Kanal et al. [26], mathematically describe the motivation to investigate channels having dependent error characteristics (memory). Briefly, they show that a channel model known as the binary *discrete memoryless channel (DMC)* gives maximum entropy. Assigning this maximum entropy value the name H_o, they further observe that channels having memory (a dependence between observed errors) have an upper bound entropy that is less than H_o. Let μ be the difference in entropy for a channel having memory and the DMC having the same bit error rate. The quantity μH_o describes the unused capacity of a channel having memory versus the DMC. Unlocking this extra capacity though an understanding of dependent error

characteristics is the motivating factor for channel model investigations.

Two approaches have been pursued for modeling dependent channels. The first approach attempts to physically model the propagation characteristics of the channel. These models focus on very specific environments and attempt to estimate received signal strength. They can give highly accurate error predictions. The second approach attempts to analytically model the channel. Analytic models are usually less computationally expensive but require the use of simplifying assumptions. Both approaches will be reviewed.

2.1.2 Propagation Models

Propagation models are mathematical representations that attempt to define the statistical characteristics of received signal strength under adverse (signal attenuating) conditions. Estimating the received strength allows one to calculate the signal–to–noise ratio and to determine the probability of receiving bit errors. The field of propagation modeling is well established and contains a significant amount of theory. This section describes several useful models that were evaluated for this investigation. It is not intended to be a comprehensive review of the field of propagation modeling.

Received Signal Strength

The pre–detection average signal power arriving at the output of the antenna of a receiver can be modeled by:

$$S = \frac{EIRPG_r}{L_s L_o} \tag{2.1}$$

S is the average signal power. $EIRP$ is the *effective radiated power* in watts. G_r is the gain of the receiving antenna, L_s is the space loss and L_o are other losses that occur in the

system (such as the small scale fading described later). Very simple models set:

$$L_s = \left(\frac{4\pi d}{\lambda}\right)^2 \tag{2.2}$$

d is the distance between the transmitter and receiver and λ is the wavelength of the signal. The loss is then proportional to the square of the distance. These models are known as free–space loss models.

Log–Distance Path Loss Models

The simplest extension to the previous model is based on the observation that the path loss at a particular distance d from a known location is log–normally distributed about the mean distance–dependent value. That is to say that if one traveled in a circle around a source (e.g., always maintained a fixed distance from that source), the signal strength would vary log–normally about the mean. If one represents signal strength using equation 2.1, along with the expanded representation for L_s found in equation 2.2, then the average received strength can be represented as:

$$S = \frac{EIRPG_r\lambda^2}{(4\pi)^2 d^2 L_o} + X_\sigma \tag{2.3}$$

X_σ is a zero–mean log–normally distributed random variable with standard deviation σ.

A second extension to the model is to replace the power of two in equation 2.2 by n. The number two represents free–space loss while higher values represent greater loss in more hostile environments. Values less than two have been used for in-building *line of sight* systems while values as high as six are reported for shadowed urban cellular radio systems [39]. Further details for such models may be found in [2].

Other Path Loss Models

The Log–Distance Path Loss model varies an exponent to change the space loss parameter L_s. Several other models have been used to derive L_s. The Okumura Model [34] is widely used for urban propagation studies. It is based on a set of frequency vs. distance curves that give a free space median attenuation correction in an urban environment. The path loss is calculated by determining the free space loss at the point of interest. Then a set of attenuation correction factors are derived from the curves and added to the free space loss. These correction factors attempt to account for variable terrain parameters and antenna heights.

The Okumura model is considered highly accurate but has complexity drawbacks. Hata [19] developed a model that attempts to create an empirical version of the Okumura curves. While it does not have all the path specific corrections present in the Okumura model, its predictions compare favorably with that model. A detailed description of both models are presented in [39].

2.1.3 Fading Models

The previous models estimate path loss caused by signal attenuation due to the separation distance between the transmitter and receiver. Such loss can be termed as "large–scale" fading since the distance d is usually large. To adequately model error characteristics, these large scale fading models must be combined with "small–scale" fading models that attempt to model the variations in signal strength that occur when a receiver moves distances of only a few feet around a point.

Small–scale fading can cause large variances in received signal strength and is usually caused by multi–path propagation. It is independent of the separation distance d and highly

dependent on factors such as mobile unit speed, signal bandwidth, time–delay spread of the received signal, and rate of change of the channel. Small–scale fading can be classified as either *flat fading* or *frequency selective* fading. Flat fading means that the system exhibits a constant gain and linear phase response over the bandwidth of the transmitted signal. If this is not the case, then the system exhibits frequency selective fading. Flat fading models are appropriate for this investigation.

There are two common flat fading models in wide spread use. They are based on the Rayleigh and Rician probability density functions [39]. The choice of which one to use depends on whether there is a dominant signal component. Rician is typically used when there is line–of–site propagation while Rayleigh is used when a transmitter or receiver is located in a cluttered environment. The Rayleigh environment is well suited for the environment under consideration. It has the following probability distribution function:

$$p(r) = \frac{r\, e^{-\left(\frac{r^2}{2\sigma^2}\right)}}{\sigma^2} \qquad 0 \leq r \leq \infty \tag{2.4}$$

σ^2 is the variance of the received signal r. Figure 2.1 shows this distribution for several values of σ. Values drawn from this distribution modify the large–scale model by affecting the value of L_o in Equation 2.1.

2.1.4 Analytical Models

This subsection addresses analytical models. The pioneering work on this type of dependent channel modeling was performed by Gilbert [17]. He modeled errors using a finite state Markov chain that consists of a *good* state, in which no errors occur, and a *bad* state in which errors may or may not occur according to some probability. In order to simulate

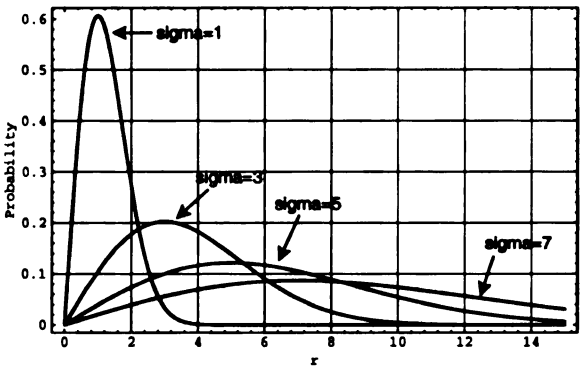

Figure 2.1: The Rayleigh PDF

the *burstiness* of errors, the state transition values are assigned such that when you enter either of the states, you are likely to stay there for awhile. This channel model is known as the *Gilbert Channel* and can be represented with the finite state diagram shown in figure 2.2.

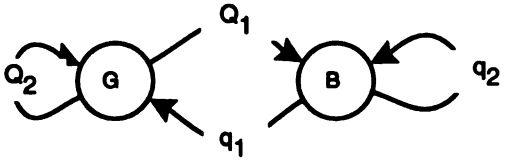

Figure 2.2: The Gilbert Model

The Gilbert Channel has some problems. Analytically, it is difficult to determine the entropy of a data stream because the model does not have the desired property of being *unifilar*. This is the property of being able to derive the state sequence of the Markov model by observing the output of the model. Since the bad (error) state may or may not generate an error, one is unable to determine if a state transition to the bad state occurred and then did not generate any errors. While the non–unifilar aspect of the process is not important when the model is just used to generate an error sequence, there is a more serious problem.

There are statistical dependencies between the lengths of good sequences and error bits. The Gilbert Channel is unable to model these dependencies because it is a *renewal* process. In other words, when a state transition occurs out of the bad state, the system starts over fresh in the good state. There can be no dependence between these transitions.

While the Gilbert Channel has some problems, it sparked a flurry of activity in the modeling of channels with memory. Elliot [12] later made a simple modification to the Gilbert Channel model that allowed errors to also occur in the good state (with some probability k). This channel model, known as the Gilbert–Elliott channel was widely used. Kanal et al. [26] provide an excellent review of many of the other modifications. They describe work on infinite–state Markov models, higher order Markov models, and gap Markov models.

A particularly important model to this dissertation was proposed by Fritchman [16]. The Fritchman model was developed to try to bridge some of the previous models. Fritchman observed both finite and infinite state models. He noted that the finite–state Markov models were inflexible while the infinite–state models were too difficult to handle in a tractable manner. Fritchman proposed a finite state Markov chain that was partitioned into two groups. States in the *Good* partition were error free states while states in the *Bad* partition were error generating states. This model is attractive but again has the drawback of being a non–unifilar model. By adding the constraint that the *Bad* partition only contain one state, the model becomes unifilar. This constrained model has the strength that its parameters can be easily generated by computing some simple statistics from observed channel error data.

2.1.5 Model Selection

The objective for this investigation is not to derive an extremely accurate channel error model. Rather it is to provide a reasonable error model that can be used to investigate the performance of new noise tolerant data compression protocols. Therefore, an approximate characterization of the channel is sufficient.

There were two noise models that were used for this investigation. The initial system used the Hata loss model (as presented on page 16) with Rayleigh fading channels. The strength of this approach is that the output of the model is defined in terms of the *energy-per-bit/noise-power-spectral-density* ($\frac{E_b}{N_o}$). This value is the standard metric used to define the expected bit error rate performance of a modulation scheme. Thus, the model could be used to derive error characteristics for a variety of modulation schemes. However, there were several drawbacks with the model. The first is that the model was computationally expensive. Changes in selected parameters meant costly recomputation time. Second, there were many parameters involved. Small changes in these parameters could lead to large changes in the resulting error characterization. While it would be possible to perform sensitivity analyses on these parameters, it was deemed that other models could provide adequate error streams with fewer parameters, less calculation, and simpler verification and validation of correctness. For these reasons, the propagation model was dropped in favor of an analytic model. Specifically, a Simplified Fritchman Partitioned Markov Model. This model is well understood, gives accurate results, and is more computationally feasible than a propagation model.

The choice of the Fritchman model was influenced by an investigation performed by Swarts and Ferreira [44]. Their work specifically investigates the Markov characterization of fading mobile channels. The authors, using dedicated hardware, generated actual error

streams for a slow moving mobile unit in an urban environment and a fast moving mobile unit in a highway environment. Analysis of the resultant data allowed the authors to derive parameters for both four and five state Fritchman models. These models were found to generate favorable approximations to the observed errors. For each of the slow and fast moving conditions, the authors also investigated four different modulation schemes. This lead to parameters for eight Fritchman models[1]. This dissertation uses a subset of these reported parameters to derive a set of error streams that are then applied to our channel. A full description of this Fritchman model is provided in Chapter 3.

2.2 Traffic Models

Traffic models are important to this investigation since there is a direct correspondence between the compressibility of the source and the number and size of compressed packets generated by the compressor. This section identifies why existing traffic works are not entirely appropriate to the environment being investigated. Chapter 4 provides a new survey and naturally contrasts against previous works.

There are many examples of network traffic pattern analysis projects that concentrate on analyzing large dumps collected between high–speed stub or gateway networks [5, 18, 25, 35]. However, these papers are concerned with traffic over the "wide–area" more so than over an endpoint. While they give insight into the behavior of many protocols throughout the Internet, the traffic collections are not appropriate for low–bandwidth network study for the following reasons:

Single Endpoint – Traffic on a network backbone is a collection of many users' data. It is difficult to determine the behavior of a specific endpoint since there is no guarantee

[1]Because of the similarity of results, the authors only report on the four state models.

that all host traffic will traverse the backbone.

Reduced Server Traffic – Traces on a LAN will contain packets generated by applications that probably will not run on a low–bandwidth endpoint (NFS, routed, standard X windows protocol, etc.). While these applications have the ability to run over such links, they are either too bandwidth intensive (NFS) or not necessary (routed).

Maximum Transmission Unit (MTU) – Packet size distributions may be different since some protocols can select a variable MTU before operation. This can affect the amount of protocol overhead presented to a compressor.

User Mind Set – Users will behave differently when using a low–bandwidth network. For instance, the link speed significantly affects the decision of whether or not to download a one megabyte file or to view graphically intensive WWW pages.

Filtering – Since such large volumes of data can be collected on a high–speed backbone, many collection processes keep only a filtered subset of data. Many packets are never saved for analysis. In a high speed network, these filtered packets probably do not affect the link to a large degree. However, the high transmission latency produced by any packet being transmitted on a low–bandwidth link can significantly affect all other packets.

To identify traffic patterns on low–bandwidth endpoints, this work, as further described in Chapter 4, provides a traffic analysis in low–bandwidth conditions similar to those encountered in wide–area wireless networks. The investigation identifies gross overall traffic trends and identifies the top application level protocols in terms of packet and byte counts for users of low–bandwidth networks. It then compares these results to those collected from high speed networks.

The proposed data collection work is unique because it deals with the data collection of an individual user's session and concentrates on low–bandwidth networks. Since the collection is taken on a low–bandwidth link, current systems have more than enough CPU and disk capacity to collect and store all packets for later analysis (usually with no visible overhead to the user). This is in contrast to other systems that usually filter out many packets and only collect subsets of the available data. A full collection allows us to accurately determine the mix of traffic appearing on the link, and thus, as input to the packet compressors.

2.3 Compression Based Enhancements

The previous section outlined the mechanisms used to select traffic types that will become the input to packet level compressors. This section addresses the algorithms that will operate on that traffic to form compressed packets streams. A review is presented of various compression protocols. Since this investigation presents new header and payload compression protocols, the focus will center on those types of compressors.

Before delving into the protocols, one should understand that packet based compressors are typically created by modifying a general data compression algorithm. These modifications usually focus on packetization and synchronization issues. The general algorithms themselves have been thoroughly investigated by such works as [20, 29, 48]. These texts typically have a pedagogical flavor and deal with generalizations and theoretical limits of various algorithms. While such texts address the problem of variability in compression ratios, their quantifications are not at the granularity of application level protocols. They instead try to give overall averages for data compression. This investigation specifically targets packet based compressors. The following sections review working implementations.

2.3.1 Specific Application Stream Compression

Datagram compression can significantly improve the performance of low–bandwidth connections. In such systems, the sender applies a compression algorithm to selected portions of a datagram and the receiver reverses the process to obtain a duplicate of the original datagram. Algorithms must be selective about which portions of a datagram are changed since some protocol information must remain intact. They typically trade off CPU processing and memory for compression ratio.

Several specific systems have been developed that capitalize on compression or other enhancements at the protocol level. A well known example is the XRemote protocol, later to evolve into the Low–Bandwidth X (LBX) protocol, developed by Network Computing Devices (NCD). This protocol is specifically designed to enhance transport of the X–Windows data stream. It employs several mechanisms of compression including an application of LZW compression to the entire data stream. Another example is TechSmith Corporation's ProtocolAssist. This technology optimizes client server interactions over low–bandwidth dial up links. Both these examples enhance communications between a client and a specific server through combinations of intelligent caching, differencing, prefetching, and in some cases, compression. While such enhancements may employ packet level data compression, their goals are to optimize a specific protocol. In contrast, the goals presented in this work are aimed at enhancing general compression algorithms that will support all traffic types.

2.3.2 Header Compression

Datagram compression can be applied to the protocol header information found in packets. Such compression can significantly reduce the communications overhead of certain protocols. These routines require few CPU cycles and typically give you a fixed gain. Several

types of non–proprietary header compression have been discussed in the Internet community. Some of the earliest are the Thinnet protocols. Thinnet is proposed in Request for Comments (RFC) 914 [13] and consists of three schemes employing varying degrees of complexity. Thinnet is described for TCP, UDP, and TP4. However, it is a general scheme that can be applied to any new protocol that arises. Mathur and Lewis [31] present another algorithm for compression of Novell IPX headers. Most important for this discussion is VJ compression [23]. It is similar in nature to Thinnet II but is tailored to TCP. It provides significantly better compression than Thinnet II while relaxing the constraints imposed on the data link layer.

Of particular interest to this area is work performed by Degermark et al.[9, 10] on soft state header compression strategies for IP version six (IPv6). Their work has similar goals to the work presented in Chapter 6. They start out with similar assumptions about wireless networks. Namely, that these networks will be bandwidth limited, that packet header bandwidth is large when packets are small, that you can trade off processing and storage for saved bandwidth, and that links will have many consecutive packets from the same connection. However, Degermark's work is based on the notion of *soft state* and applies to non–TCP related IPv6 traffic versus the TCP based IPv4 traffic of our algorithm.

A key difference in the Degermark work relates to the reliability of the transport. Since their work is targeted towards UDP based traffic, there will be no retransmissions for lost packets. This means that VJ's TCP retransmission detection approach to state monitoring will not work. To maintain synchronization, the authors employ a *soft state* algorithm that uses periodic state updates. By associating generation counts within the state, lost synchronization can be detected and recovered. Degermark et al., make the assumption that the unmodified VJ algorithm will be used for TCP streams. Since the Degermark and

VJ algorithms target different protocols, they can operate concurrently.

A novel feature of the work is their use of a *slow start* algorithm. This algorithm attempts to balance the need for sending many header updates to allow for quick synchronization recovery versus sending few header updates to allow for high channel utilization and compression. With the algorithm, their compressor starts out sending updates over small intervals. The intervals exponentially increase until a nominal rate is achieved. A synchronization loss causes the interval to reset.

The soft state algorithm is less prone to synchronization loss when packets are dropped. It has some similarities to the state maintenance used in the LZW compression work described in Chapter 7. However, their work assumes simplex links while the LZW work makes use of a duplex link that has a reliable signalling channel. The Degermark algorithm does not suffer from the drawbacks of the original VJ algorithm and it addresses synchronization loss over non–reliable streams.

2.3.3 Payload Compression

Algorithms that are applied to the payload of a datagram can significantly reduce the datagram size. There are many examples of payload compressors in the Internet community. Several Request for Comments (RFC) documents [7, 15, 38, 49] propose control mechanisms for various payload compressing algorithms. Indeed, the LZW algorithm employed in Chapter 7 has been in use for many years. However, the study of datagram compression performance in the presence of noise seems to be under–represented in the literature. The author is unaware of any published studies of data–link payload compression in the presence of noise.

2.4 ARQ Mechanisms

As mentioned at the beginning of this chapter, an overview of ARQ mechanism is important for understanding the results presented in Chapter 7. A review of previous ARQ works is now presented with the understanding that such works could be used in conjunction with the new error tolerant protocols developed in this dissertation.

Packet based compression techniques typically require some mechanism for dealing with loss of synchronization. As suggested in the Compression Control Protocol RFC [37], compression algorithms should provide their own mechanism for detecting and dealing with packet loss. Most current algorithms work in conjunction with a reliable transmission layer. The reliability can either be provided below the compressor at the data link level, or above, at the transport layer. Since the performance of the retransmission system critically affects the overall performance, it is natural to review that work.

Considerable research has gone towards optimizing Automatic Repeat Request (ARQ) protocols for noisy channels. Any work with such a system must define both a model for the noise and a model for ARQ (Stop–and–wait, Go–Back–N, Selective–Repeat, etc). Noise models usually characterize errors as either dependent or independent and apply either to data bits or packets. Dependent channel errors are considered to be more appropriate for wireless links.

One approach for enhancing ARQ protocols is to modify the data link layer to support reliable datagram delivery. Through combinations of error detection, forward error correction, and retransmission it is possible to hide many link layer errors from the higher level transport protocols. Farber et al. [13] propose a reliable SLIP link. Choski [8] gives a comprehensive overview of much of the work on data link layer (DLL) retransmission schemes. A particularly interesting work is that by DeSimone, et al. [11]. This investigation not

only shows the benefits of DLL retransmissions, but quantifies the adverse effects generated by competing transport and data link layer retransmission schemes. In many cases, especially as error rates rise, the competition renders the benefits of DLL retransmission useless. Finally, it may be the case that a reliable link is simply not available. Such is the case with standard PPP. Non–reliable data link layers will be assumed within the context of this investigation.

A second approach to enhancing ARQ operation is proposed by Cáceres and Iftode [6]. This work proposes an end–to–end quick retransmission scheme for TCP. The rationale behind such a scheme relies on the fact that TCP interprets packet loss as being caused by network congestion. Upon such loss, congestion avoidance mechanisms are initiated [24]. These mechanisms lead to unacceptably high delays for retransmissions. The authors show that differentiating between network congestion and packet loss due to link errors can improve response of TCP connections. Their new retransmission scheme behaves better under error conditions that are not caused by congestion.

Finally, several investigators have proposed variations of what is known as split TCP. Such systems take a TCP connection between a fixed host and a mobile host and split it into two separate connections, one between the fixed host and the base station and one between the base station and the mobile host. The mobile support base station transfers datagrams between the two TCP connections. Bakre and Badrinath [4] suggest that each split TCP link consists of a standard TCP connection. Yavatkar and Bhagawat [50] propose a special wireless link between the mobile host and base station. A similar system to the split TCP concept is proposed by Amir et al. [1]. They suggest the addition of a retransmission snoop layer at the mobile host base station. Such a layer intercepts and handles retransmission requests. The snoop layer moves part of the retransmission engine closer to the source of

most errors. In all cases, the idea is to shield the sender from the error characteristics of the wireless link.

The work presented in this investigation is an enhancement independent from ARQ strategies, but reliant upon such strategies. It should work in conjunction with all ARQ strategies. Loss of synchronization in our algorithms will appear as packet loss to an ARQ strategy, so the more effectively an ARQ strategy deals with packet loss in a noisy environment, the more effectively our scheme will work. This is assumed to be especially true of ARQ strategies that work well for dependent packet error environments. Since loss of synchronization means that sets of contiguous packets will be dropped, strategies that handle burst packet loss will excel. It is important to note that this investigation relies on the standard TCP retransmission scheme. There is little reason to believe that error tolerant compression protocols would adversely affect any of these schemes.

Chapter 3

The Collection Environment

This chapter discusses the testbed used to gather all the statistics presented in later chapters. A working implementation was chosen over both analytically based and simulation based approaches due to the complexity of the system. While analytical solutions can be derived for some of the projected system parameters (e.g., average bit error rates for a given noise model), the analytical model will not allow for a detailed understanding of the interactions between different communications layers. An analytical model with the needed number of parameters would become intractable. Making the required simplifying assumptions would hide many of the interesting aspects that arise in real systems. Simulations have their own set of drawbacks. Simulations are only as good as their design assumptions. Verification and validation of a simulation model is a demanding process. If proposed simulated solutions cannot be implemented in a reasonable fashion then they do not hold much value. The work involved in a reasonable software based simulation approach is similar to that needed to implement a working system since, in both cases, the algorithms need to be coded. Having an implementation guarantees that practicality will be maintained. Further, a working system reduces the need to justify many design decisions since one is already be working with *real data*.

3.1 The Communications Testbed

A testbed was developed that has characteristics similar to wireless networks (low–bandwidth and possibly noisy links) while avoiding the drawbacks (high cost, limited access, and limited control over outside interference). This testbed is based on low–bandwidth connections that use the Point–to–Point (PPP) protocol [43]. It consists of a two station network where each station is directly connected through a serial link as well as connected through an Ethernet adaptor. The serial link is used to emulate a wireless media while the Ethernet link allows for monitoring the serial communications without significantly affecting the results. Both systems are running a modified Mach kernel under the NeXTSTEP operating system. Both systems contain a non-standard implementation of the BSD Packet Filter (BPF) [32] and a modified version of TCPDUMP [46].

3.1.1 The Berkeley Packet Filter

The Berkeley Packet Filter (BPF) is a kernel level module that provides a protocol independent interface to the data link layer of a system. It works by allowing higher level applications to define a *filter program* that runs on a *Filter Machine* that exists inside BPF. This *Filter Machine* provides an instruction set, accumulator, registers, scratch memory, and program counter. By fashioning an appropriate filter program, applications can select datagrams that they find interesting. The BPF will then hand to applications copies of those datagrams passing the filter.

3.1.2 The TCPDUMP Program

TCPDUMP is an application level program that works in conjunction with BPF. TCP-DUMP provides a high level interface to the BPF *Filter Machine* and has knowledge of

standard frame formats like PPP, SLIP, and Ethernet. Using TCPDUMP, one is able to construct boolean operations that are converted to Filter Machine programs and installed inside BPF. TCPDUMP can then analyze the headers of any packets that pass the filter and, optionally, store those packets for later recall.

3.1.3 BPF and TCPDUMP Modifications

The BPF and TCPDUMP packages are used to gather most packet based statistics for this investigation. However, because of the nature of data compression and the statistics needed, both BPF and TCPDUMP were modified to work in a non–standard manner. These modifications were not to the *Filter Machine*, but rather to the packet interface between BPF and the application level programs.

The BPF typically passes framed packets up to the requesting application. It is up to the requesting application to parse the framed packets and understand their formats. Unfortunately, part of this investigation required that packet information be gathered during multiple stages of framing and compression. Such information allows one to monitor the efficiency of the compressor. The drawback is that all information needed to parse a compressed data frame is not available in the frame. Instead, some information is maintained with the *state* of the compressor and decompressor algorithms.

To account for this, the interface between BPF and TCPDUMP was modified to allow more information to pass between the kernel and the requesting application. These additions allow the application to receive enough information so that it can gather appropriate statistics on the compression process. The downside is that the TCPDUMP application reads and writes files that are not compatible with standard TCPDUMP.

3.2 The PPP Implementation

The PPP driver is an RFC compliant client/server implementation. It consists of two pieces that, together, provide PPP connectivity. *PPPD* is a user level daemon that implements the PPP negotiation engine. This daemon works in conjunction with a loadable kernel server that provides PPP framing and data compression at the data link layer of the OS kernel. Compression is negotiated and controlled via the user level daemon through the Compression Control Protocol (CCP). This protocol allows optional negotiation of software compression mechanisms. The actual compression/decompression engines are implemented in the kernel.

The compression algorithms supported by the testbed include the popular Van Jacobson (VJ) TCP Header compression mechanism [23] and a payload compressor that is based on the LZW scheme used in the BSD *compress* program. The VJ algorithm will be discussed in detail in chapter 6. The LZW scheme was modified to operate in a packet based environment and will be further discussed in chapter 7.

Together, PPP, BPF, and TCPDUMP allow tracking of compression and timing information on individual packets as they flow through selected portions of the protocol stack on each machine. During any data collection session, the system stores a time–stamped copy of the first 68 bytes of each packet sent and received. The packet header allows determination of the higher level protocols encapsulated in the frame. In addition, the system stores timing and compression ratio information about each packet as it passes through various stages of the compression routines. The information collected describes all compression that is applied including: PPP specific address/control/and protocol compression, Van Jacobson TCP header compression, and LZW packet payload compression.

Having two directly connected and independent systems allows for full control over both

ends of the communications network. Having such control frees us to make modifications on the *server* as well as the *client* endpoints of a point–to–point link. The PPP environment itself was chosen for several reasons. The optional software based compression allows for selective modification and application of compression under various circumstances. Also, since it is not a wireless environment, we have tight control over outside interference. Extraneous noise and co–channel interference are not present over the standard link. However, for this investigation, a noise stream is required. The following subsection describes how noise is modeled on this system.

3.3 The Noise Models

Chapter 2 reviewed several methods that can be used to generate bitstreams that show dependent error characteristics similar to those found on wireless links. This section will detail the method selected to generate and insert noise over the PPP testbed link. A special type of Markov model, known as a Fritchman Partitioned Markov Chain, was chosen because it accurately represent noise found on wide–area links using existing wireless technology. It is reiterated that any reasonable error model will suffice. An exact noise environment is not necessary since no error correction is done at the data link level. A single bit will force a packet CRC checksum to fail just as effectively as a stream of dependent errors. Before describing the Markov model, a review of error characterization statistic is presented. This is necessary since these statistics are used to describe the output of the resultant error model.

3.4 Dependent Error Characteristics

Dependent errors (errors generated on channels having memory) are typically used to model wireless links. These errors can be described by a set of standard error statistics. Figure 3.1 describes the potential regions that may occur on a dependent error channel. The *gap* is defined as the region that exists between any two consecutive errors. It contains

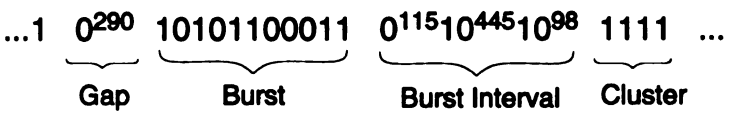

Figure 3.1: Error Statistics

no error bits. A *burst* is a region where the the local bit error rate exceeds a given threshold value. A burst region must start and end with an error bit. A subsequent burst cannot start with the ending error from the previous burst. The *burst interval* is the region between two consecutive bursts. Finally, a cluster is a set of contiguous error bits and also obviously qualifies as being a *burst*. By appropriate analysis of these regions, one may derive the *Error–Free Run* probability denoted by $P(0^m|1)$ (see Kanal et al. [26] for details). This is the probability that an error will be followed by at least m error–free bits.

3.5 Fritchman Partitioned Markov Chains

Figure 3.2 shows the general structure of a Fritchman Partitioned Markov model ([16]) with one error state. In such models, all states in partition A are reduced error states, meaning their transitions have a low probability of producing errors. The states in partition B (one state in this simplified model) are the high probability error states. Transitions while in partition B will most likely produce error bits. In general, partitioning is done between

the low probability error and high probability error states. In a Fritchman model, no

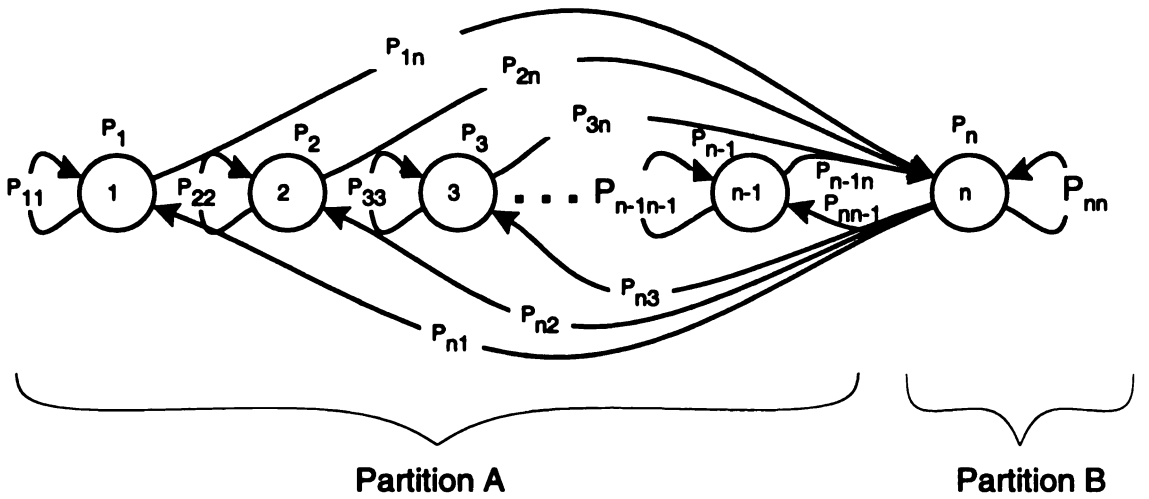

Figure 3.2: A One State Fritchman Partitioned Markov Model

transitions are allowed between states in partition A. Transitions to partition B signal that errors will probably occur. Transitions out of state B signal that reduced error states will begin. The probability of remaining in state B affects the cluster distributions while the probabilities of entering state B affect all other error statistics. This investigation used a *simplified* Fritchman model which limits the number of states in partition B to one and simplifies the probability of generating an error on a transition. In the simplified model, transitions in states of partition A never generate an error and transitions in the single state in partition B always generate errors.

As described by Tsai [45], exponential curve fitting techniques may be applied to the *Error-Free Run* distribution of an existing error stream in order to derive appropriate values for the parameters shown in the figure. Using such parameters allows the model to generate an error stream that has a similar *Error-Free Run* distribution. Such curve fitting was performed on an actual wireless error stream and the results for a Simplified Fritchman Partitioned Markov Chain were derived.

Table 3.1: Fritchman Transition Values

Parameter	DPSK Freeway
p_{11}	0.371954
p_{22}	0.900030
p_{33}	0.999464
p_{44}	0.385774
p_{41}	0.416705
p_{42}	0.124011
p_{43}	0.073510
p_{14}	0.532679
p_{24}	0.099970
p_{34}	0.000536

3.6 The Fritchman Parameters

In 1994, Swarts and Ferreira [44] applied Tsai's error modeling technique to the results of experiments that were undertaken in both urban and freeway environments in South Africa. Using existing hardware and custom error recording equipment, they evaluated the performance of several existing wireless digital communications systems employing FSK, DPSK, QPSK, and 8-ary PSK modulations. By comparing the received data to the transmitted data, the authors were able to determine the error performance of each modulation scheme in a specific but real environment. From the recorded statistics, they were able to apply the curve fitting techniques and generate a Fritchman Partitioned Markov Chain. The resulting four state model is shown to accurately represent the observed error characteristics. The parameters for their models are given in [44] and are subsequently used to generate the error patterns in the PPP testbed. Table 3.1 lists the transition values.

3.7 Error Generation

The parameters shown in Table 3.1 model the output of a differential phase shift key (DPSK) modulator working in a city environment. These values were placed into a Mathematica model for a Simplified Fritchman Partitioned Markov Chain and an appropriate error stream was generated. This error stream consisted of a series of bits with a one representing an error. The stream was compressed and stored as a static array suitable for inclusion into a C program. This array was linked into the testbed's Mach kernel and referenced by the PPP kernel level routines. While in error mode, the PPP routines would determine the length of time it took to transmit packets of data. They would then check an equivalent number of bits in the bitstream to see if an error occurred. If so, PPP would purposely corrupt the CRC checksum so that the appropriate packets would fail during decoding.

A random error stream was generated along with the dependent errors created from the Markov model. This random stream has the same probability of bit error as the dependent stream and allows for investigation of short bursts of traffic without regard for the long gaps and burst intervals that may arise with dependent noise. The following figures show the error statistics for the two error streams. Figures 3.3 and 3.4 show the *Gap Distribution* and *Error-Free Run Distribution* for both the random and dependent channels. Figures 3.5 and 3.6 show the *Burst Distribution* and *Burst-Interval Distribution.* These last two are only for the dependent errors as the random stream does not display burst characteristics. They use a value of 0.60 as the threshold local ratio value for determining a burst.

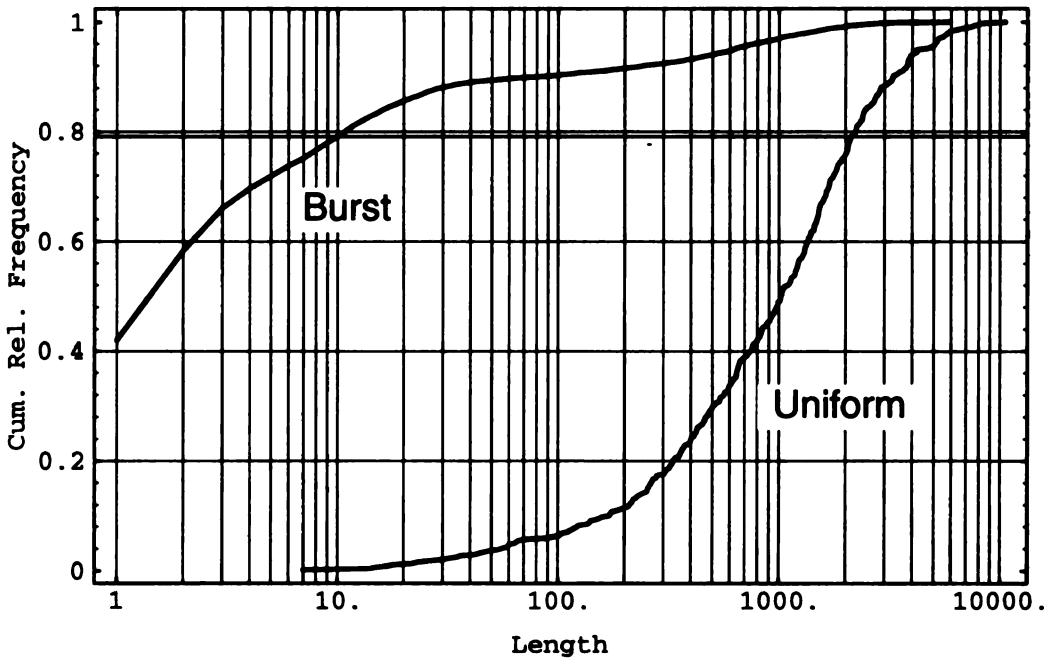

Figure 3.3: The Gap Distribution

3.8 Summary

This chapter described the data collection testbed that was used to collect the results presented throughout this work. The testbed is based on the industry standard Point–to–Point protocol (PPP). It models low–bandwidth systems by operating over restricted baud serial links. Using the standard PPP negotiation mechanisms, the links can be configured to support (or deny) various packet compressing algorithms. Specifically supported are the Van Jacobson (VJ) TCP header compression algorithm and a LZW based packet payload compressor.

The system also uses the Berkeley Packet Filter and TCPDUMP utilities. This testbed is capable of collecting detailed low–bandwidth traffic statistics in real time and with little noticeable overhead. Along with the ability to collect standard traffic dumps, the collection

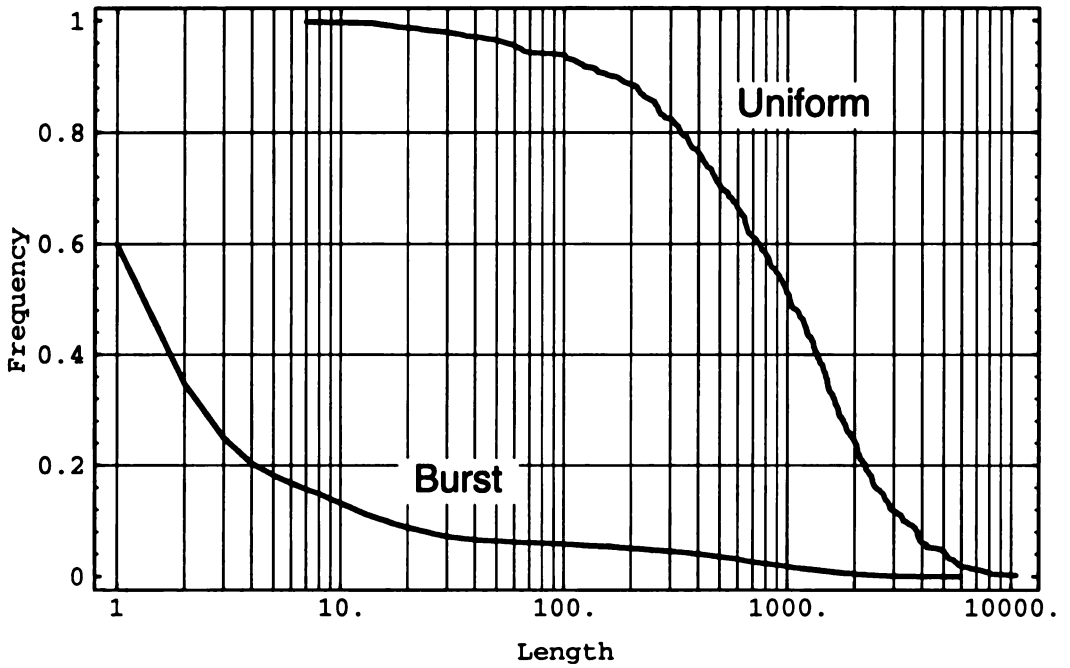

Figure 3.4: The Error–Free Run Distribution

facility is aware of the existence of the data compressors. Through BPF modifications, the testbed is able to track the compression statistics of packets as they flow through the system. Such capabilities will be necessary to understand where and how compression affects specific protocols.

Finally, the testbed has the ability to inject errors into the datastream. Such streams will be used to develop noise tolerant protocols that operate in wireless environments. The selected noise model was presented along with a justification of its suitability for this investigation.

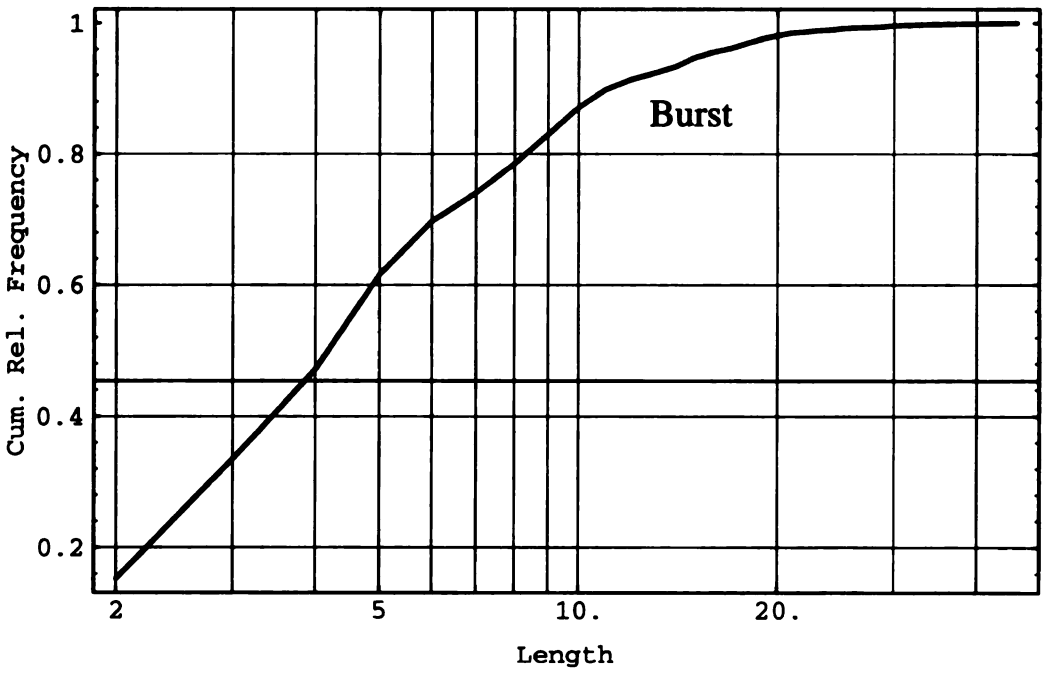

Figure 3.5: The Burst Distribution

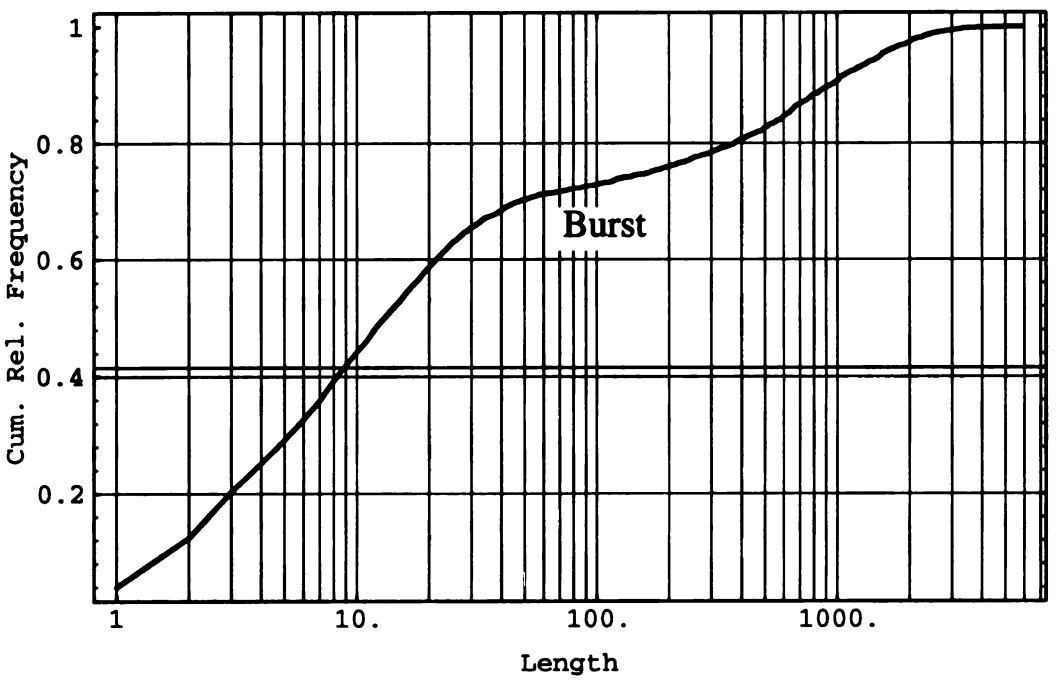

Figure 3.6: The Burst Interval Distribution

Chapter 4

Traffic Analysis

4.1 Introduction

A thorough understanding of low–bandwidth traffic characteristics is necessary in order to understand the input that is fed to packet level data compressors. Previous wide–area traffic studies gathered data from high speed LANs and network backbones. However, future wireless networks will make use of physical connections having orders of magnitude lower bandwidth than is available on wire based networks. Such low–bandwidth networks (LBNs) will impose restrictions that fundamentally change the way users access the network, and thus, the traffic patterns that flow over the network. To understand these new patterns, traffic studies must be performed in conditions near that of projected wireless systems.

This chapter describes an application level traffic analysis for LBNs. This analysis will quantify traffic generated by users of low–bandwidth systems. Data are collected over low–bandwidth endpoints that use point–to–point connections. Gross behavior patterns and detailed protocol analysis information are presented and compared to previous high bandwidth traffic analysis. The analysis provides a mechanism for selecting appropriate traffic models for use as input to packet level compression routines.

4.2 The Collection Environment

The intent of this analysis is to gain an understanding of the application level traffic that flows over low–bandwidth wide–area wireless networks. Existing wireless networks present several barriers for effective data collection and traffic analysis. First, access is not widely available. Many wireless systems exist either as pilot programs in a single city or are deployed only around major cities. Second, the high cost of access can significantly affect the traffic mixture. Most wireless systems charge either by connect time, packet count, or byte count. This rate system can cause users to change their normal access behavior. To address these problems, data were collected on systems that are not wireless but present a similar environment and promote similar usage.

Traffic dumps for this study were gathered using the PPP implementation described in chapter 3. For this portion of the investigation, noise generation was disabled. The dumps were generated with the help of a group of nine volunteers spread throughout three countries (Germany, United Kingdom, and the USA). A survey of the volunteers shows that there is a mix of both recreational and professional users. These volunteers modified their systems such that the collection process automatically starts when the IP layer of their link comes up. During their session, the system stores a timestamped copy of the first 68 bytes of each packet sent and received. This collection size is chosen because 68 bytes is enough to capture the IP and transport headers of the packets. When the link is brought down, the collection process is stopped and the dump file is augmented with overall system and session statistics. Finally, the dump is compressed, PGP encrypted (since there is potentially sensitive information in the dump), and eventually electronically mailed to the collection site.

4.3 Results of Analysis

A suite of programs was written to analyze the data. Since the information collected is based on many individual sessions (versus the large dumps taken in previous papers), there are two types of statistics that can be collected. The first set, called *Gross Statistics*, deals with the overall statistics on the dialup PPP sessions. It includes statistics measuring time of day during which the dump was taken, the duration of the session, the speed of the link, and the number of both packets and bytes sent and received. The second set of statistics are called *Protocol Statistics*. They are generated as a result of analyzing what went on during a particular session.

4.3.1 Gross Statistics

The following subsection gives a high level summary of the statistics gathered on sessions. Table 4.1 describes average counts and connect times for users. Since users are equipped with different speed modems, sessions are broken into groups based on user modem speed. The *Modem Speed* column describes the connect speed of the modem. While actual transmission speeds of modems may vary dynamically with signal and line quality, it was decided to categorize systems based on the maximum speed of modems. The direction indicators (*In* or *Out*) describe the direction of data when viewed from the endpoint collection host (a PPP client). Thus, *In* is coming from the LAN/Internet to the endpoint and *Out* is leaving the collection host and going to the LAN/Internet.

It is interesting to note that the ratio of incoming bytes to outgoing bytes is a little over five to one for 14.4K modems and closer to two and a half to one for 28.8K modems. As identified by Paxson in [35], data sets can vary significantly from site to site. The variance in ratios shows that the same holds true for our person to person collection (which is really site

Table 4.1: Gross Statistics

Stats	Speed 14000	Speed 28800
Num Sessions	149	261
Total packets	621174	1012836
Total Bytes In	167783683	138867597
Total Bytes Out	31800322	57567647
Avg. Bytes In	1126064	532060
Avg. Bytes Out	213424	220566
Avg. Dur. (Sec)	3865	2348

to site). Another interesting statistic is the difference between link speeds, packet counts, and total bytes. The packet totals are higher for 28.8K (as one might expect). However, the average packet size and incoming byte usage is higher for 14.4K links. Part of this is due to the choice of MTU size selected by individual users.

4.3.2 Protocol Statistics

Since the focus of this investigation is to determine the input to packet based compressors, this section will concentrate on protocol statistics that are derived from viewing the entire data collection. To properly interpret the following statistics, one must have a clear understanding of exactly where in the transmission stage this information was collected and how this affects the results. Unlike high speed links, packets that are sent over a low–bandwidth physical link typically undergo several stages of data compression (i.e., VJ TCP header compression [23] and hardware compression by the transmitting device). These compression schemes variably affect how many bytes of a packet actually get transmitted over the link. For the compression schemes implemented at the endpoint (e.g., VJ Header Compression), the endpoint can determine the effects. However, for intermediate devices such as compressing modems, endpoints cannot determine exact compression

statistics. Therefore, this chapter does not consider compression. Packet lengths are given either before any compression takes place (for transmitting) or after all decompression has been applied (receiving). This allows us to directly compare byte counts to previous studies.

Table 4.2 shows the top six highly used protocols (based on byte count) for the LBNs compared to the top six (in 1994) as described by Paxson in [36]. As shown, the mixture is quite similar in a few categories. However, there are the notable differences of X11 and POP3. As expected, the protocols are even less similar when compared to the LAN traffic described by Gusella in [18]. In Gusella's paper, the presence of NFS and Sun's ND traffic accounts for approximately 65% of his analyzed traffic but do not occur at all in this survey.

This protocol breakdown supports the assumption that users will attempt to use LBN systems in a similar manner as their high speed counterparts. Further it verifies that previous system surveys do not give a completely accurate profile of LBN systems. To further study this issue, Figures 4.1, 4.2, and 4.3 take Table 4.2 and graphically break down

Table 4.2: Top Six Protocols

Protocol Statistics		
Rank	Low Speed	High Speed
First	nntp	ftp
Second	smtp	nntp
Third	ftp	X11
Fourth	http	http
Fifth	pop3	shell
Sixth	exec	smtp

the information based on packet count for each protocol according to direction of transfer. Figure 4.1 gives an overall picture of the total data flow in the system. Figures 4.2 and 4.3 show the same graph when filtered to show just incoming or outgoing traffic respectively. In each figure, the X axis describes the packet size while the Y axis determines the number

of packets of that size.

The two protocols that probably need mentioning are *Combined* and *UnknTCP*. To make the display of the graph manageable, *Combined* is a collection of packet counts for protocols that did not have enough total packets to reach a threshold. For these graphs, the threshold was set at 20,000 packets. The *UnknTCP* category refers to packets that did not seem to be destined for any *well-known* TCP port. While the data have been combined for the graph display, the data reduction was done after the collection. Since all packets were captured in this study, information is available for each protocol in either of these groups.

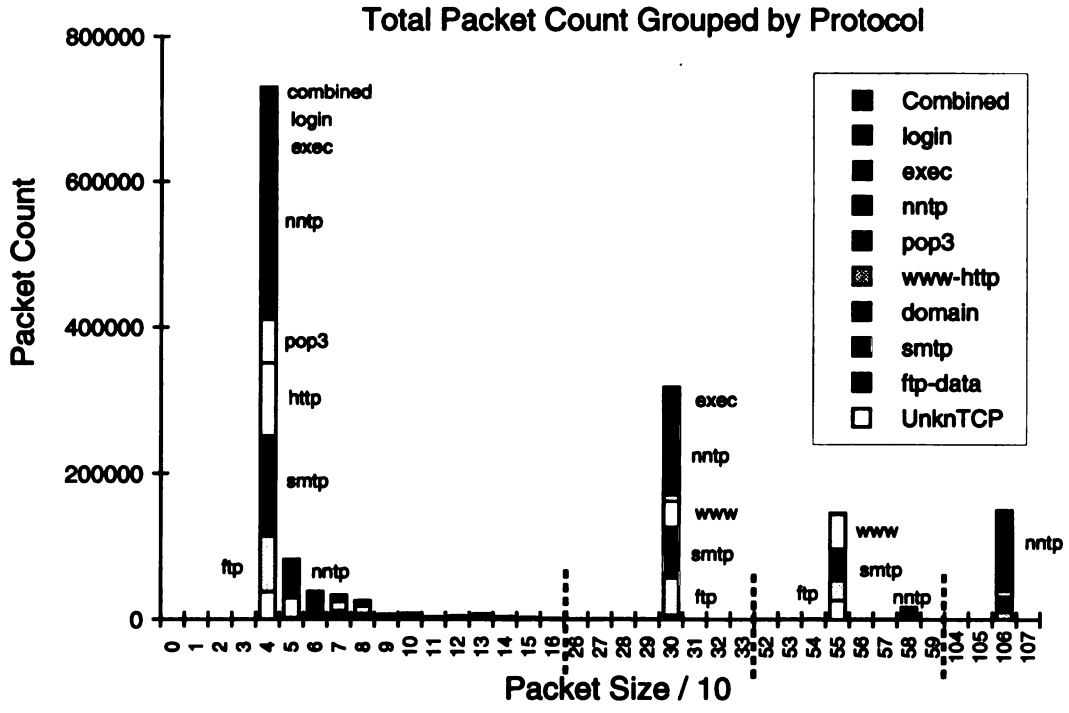

Figure 4.1: Total Incoming and Outgoing Traffic Grouped by Protocol

Table 4.3 shows the mixture of all the *well known* ports [40]. This table gives an appropriate overview of the typical usage patterns of current users. As in the graphs, many protocols have been checked against the threshold and combined. The counts in parenthesis

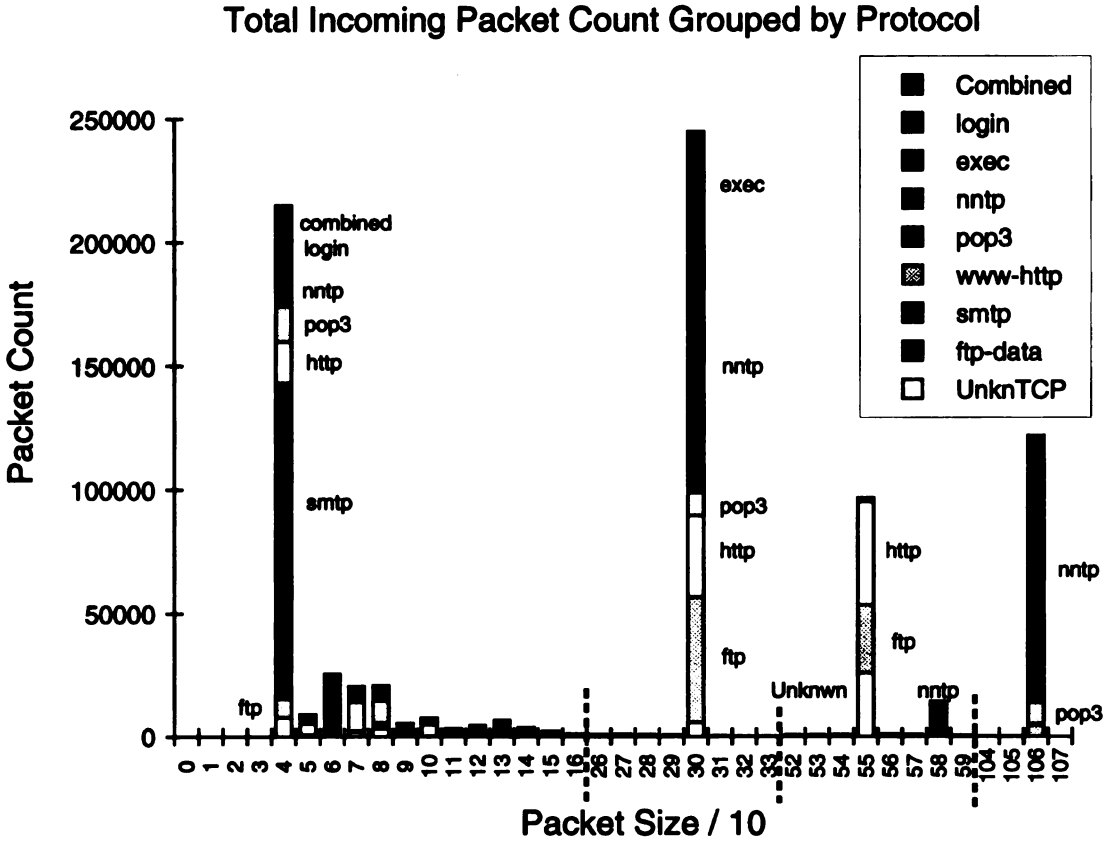

Figure 4.2: Total Incoming Traffic Grouped by Protocol

show the number of different protocols in the threshold group as well as the threshold size (1000).

In order not to be mislead as to the "amount" of data sent (versus the number of packets), Figure 4.4 describes the same statistics based on byte count. It was previously shown that incoming traffic accounts for approximately two and a half to five times the amount of outgoing traffic. But as one might expect, mail activity accounts for a large majority of outgoing bytes.

This SMTP traffic is non–interactive and much of the incoming is a mix of interactive and non–interactive. This is important because header compressors help to enhance interactive traffic, and thus must be very sensitive to the interaction between outgoing non–interactive

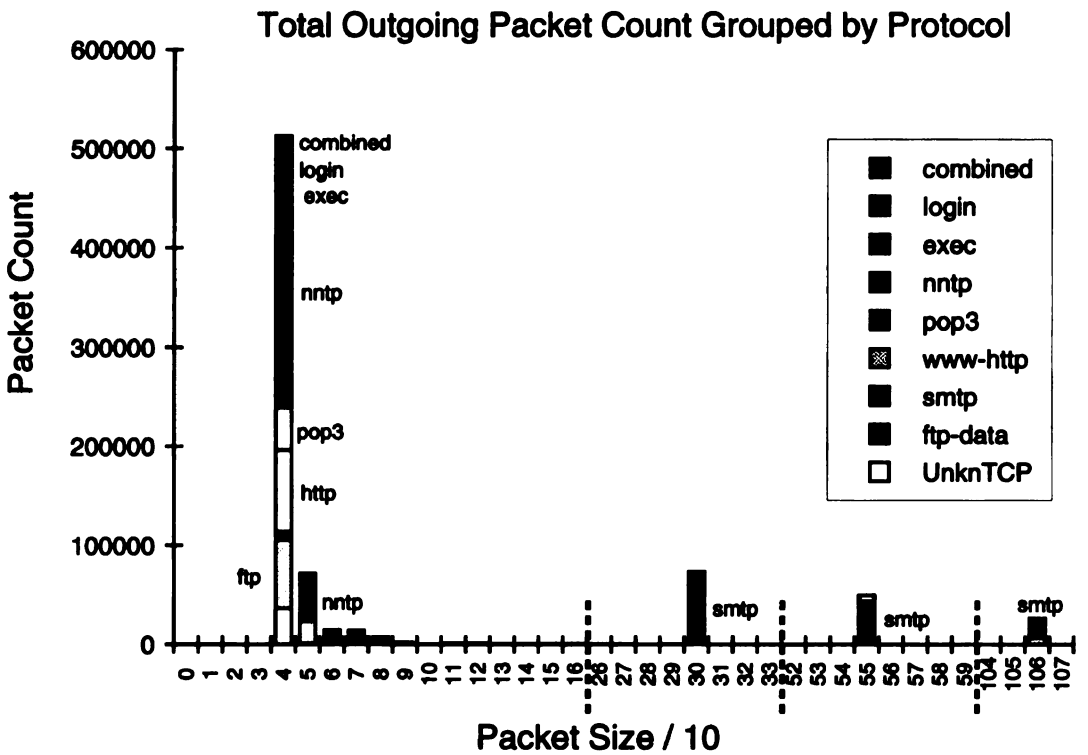

Figure 4.3: Total Outgoing Traffic Grouped by Protocol

traffic and outgoing interactive acknowledgments. It is important to provide good support for these types of interactive services.

4.4 Analysis of Approach

It is necessary to pause and validate the approach taken. Will the point–to–point traces be representative of future wireless wide–area access? To answer these questions, it is necessary to analyze the assumptions.

Traffic Similarity From the standpoint of the OSI network model, our data collection occurs at the data link layer. However, the results analyzed are for the encapsulated protocol datagrams at the network and transport layers. The separation of layers

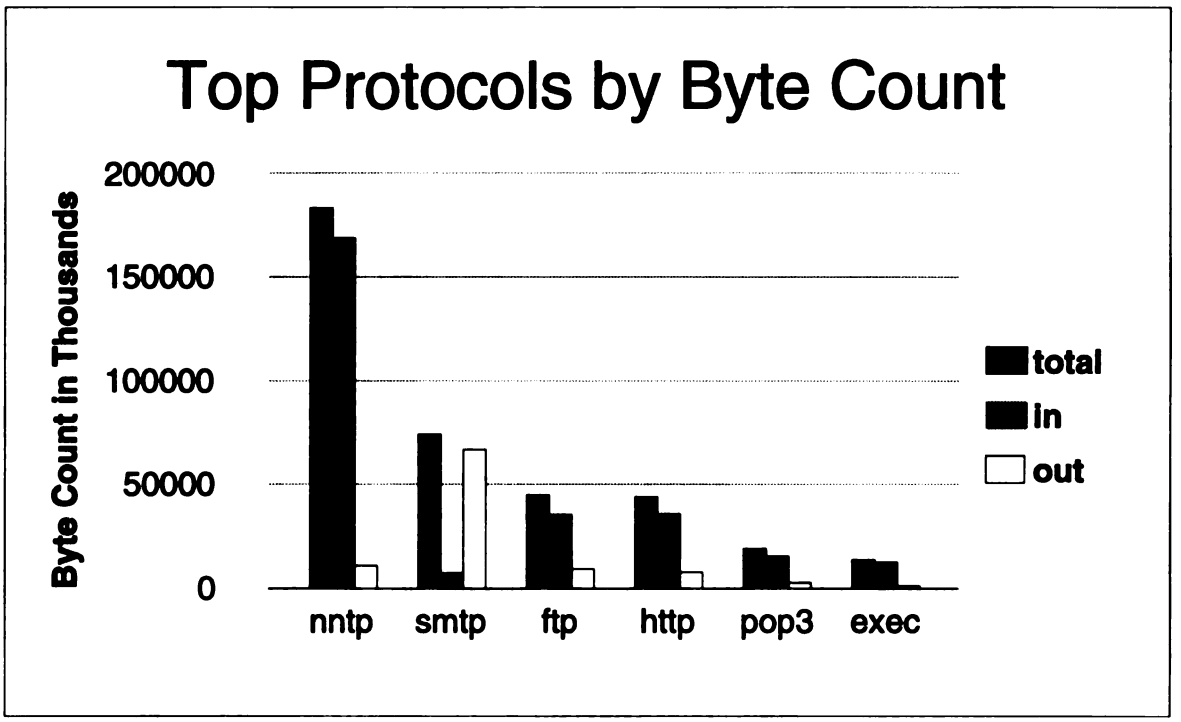

Figure 4.4: Total Incoming and Outgoing Byte Count Grouped by Protocol

ensures that the mechanics of physical layer transmission are hidden below the network layer. Link speed is one of the few physical layer properties that will be indirectly visible at higher layers. The network layer will not know if the physical medium is point–to–point or broadcast. Thus, the choice of IP over point–to–point dialup connections versus IP over actual wireless connections should not significantly affect the results. Further, the Cellular Digital Packet Data (CDPD) forum has recently adopted the TCP/IP suite of protocols as their standard. This lends credence that this suite of protocols will be important for future wide–area wireless networks.

LBNs will remain LBNs It is generally acknowledged that there is a lack of allocated spectrum for effective wireless traffic[33]. This is not necessarily a technological problem. While faster wireless interfaces can no doubt be designed, the fundamental

Table 4.3: All Protocols

Protocol Usage	
Protocol	Num Packets
nntp	563004
smtp	284235
http	194367
ftp+ftp–data	186602
pop3	135435
Unknown TCP	77380
exec	73433
login	50167
domain	32005
telnet	10004
time	8300
TCPThresh (28@1000)	3819
Unknown UDP	1748
sunrpc	1426
ntp	1342
UDPThresh (19@1000)	347

limitation of available spectrum will keep wide–area wireless devices from attaining high bandwidths.

4.5 Summary

This dissertation deals with data compression algorithms operating in noisy environments. Since the performance of these algorithms is tightly coupled to the characteristics the input data, one must justify the types of traffic that will be used in a compression investigation. Previous traffic studies dealt with traffic flowing over high bandwidth links. These studies do not accurately describe the traffic patterns found in low–bandwidth networks. This chapter described a mechanism that effectively captures session statistics for actual low–bandwidth user traffic. The collected dumps were analyzed to provide individual user level behavior statistics that are not found in previous work. The results are used to justify

the selected traffic patterns that will form the input to the data compressors in Chapters 6 and 7.

Chapter 5

Compression Analysis

5.1 Introduction

In the ubiquitous wireless architecture, the low–bandwidth link is a critical resource that must be tuned for performance. Data compression is a primary mechanism employed for this enhancement. Theory, experience, and common sense tell us that the compressibility of information varies according to the input content. The previous chapter describes how a low–bandwidth testbed was employed to derive usage patterns for a class of low–bandwidth network users. This chapter continues that work by presenting an analysis of the noise–free compression characteristics for the top protocols found in the traffic survey. An understanding of the compression characteristics in this *clean* environment will give a benchmark against which one can measure performance of the same algorithms in an error prone environment. The rest of this chapter is organized as follows: Section 5.2 describes compression statistics for the traffic gathered in Chapter 4. Section 5.3 analyzes the collected transition information and Section 5.4 summarizes the work and relates it to the overall direction of this dissertation.

5.2 Compression Analysis

This section describes the results of the compression analysis of application level protocols. It investigates compressibility factors, and in some cases transition probabilities, for the top six protocols identified in Chapter 4. These protocols, in order, are: Network News Transfer Protocol (NNTP), Simple Mail Transfer Protocol (SMTP), HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Post Office Protocol 3 (POP3), and the Login Protocol (LOGIN). While these protocols were identified in the last chapter, that work did not include the kernel modifications necessary to track the compression sizes. Consequently, new data had to be generated using the controlled conditions of the testbed. In each case, the characteristics of the protocol are maintained. Each subsection will describe the methods used to generate the data.

Graph Descriptions

During communication sessions, size information is collected for each packet as it traverses through the various compression and encapsulation routines. Specifically, size information is stored on the original received size, the size before/after Van Jacobson TCP header compression, and the size before/after LZW packet compression. The negligible effects of PPP address, control, and protocol compression can be inferred from these values and are subsequently ignored. For inbound packets, the original size included the PPP level framing bytes as well as the entire compressed payload. For outbound packets, the original size was that of the IP packet as delivered to the link layer from the protocol stack. During post processing, packets were quantized based on their original received size and direction of travel. Packet sizes referred to in graphs actually represent quantized packet sizes.

A stacked bar graph was generated for each of the desired protocols. The graph describes

the average compression of all packets based on their quantized size. Compression percentages include the quantization, averaging, and roundoff error inherent in such techniques. Directionality is maintained by providing a separate graph for the inbound and outbound packets. Individual stacked bars in a graph are labeled by a quantized packet size. The number at the top of each bar describes the percentage of total packets that fall into that quantization size. The *Counted* field at the bottom of each graph describes the total percentage of packets represented by the graph. This number is influenced by a *Threshold* value which is also supplied when used. Quantization levels having fewer than *Threshold* packets do not have a bar displayed. This leads to a less cluttered and more understandable graph. Finally, the *Overall [De]Comp* field describes the overall effect of compression. This value is calculated as the appropriate ratio between the total received bytes and the total bytes after all [de]compressions have been applied.

Graphs that describe inbound packets are designed to show the increase in packet size as the packet goes through various decompression routines. For inbound packets, LZW decompression is applied followed by VJ decompression. Decompression is shown as a percentage of original packet size. Thus, all packets start out as 100 percent of their original size and grow thereafter. For outbound packets, VJ compression is applied followed by LZW compression. Again, all sizes are shown as a percentage of the original size so outbound packets start out as 100 percent of their original size and shrink accordingly.

The following subsections describe each protocol in more detail. Attention is given to both the method of collection and analysis of results. In some cases, a more detailed analysis is performed on individual protocols. In these cases, protocol size transitions are included to help clarify a certain behavior. Since the FTP protocol exhibits predictable behavior, it was selected as the first graph to be presented even though it is not the top low–bandwidth

protocol in terms of usage. This gives the reader a familiar base against which to compare the other graphs.

The File Transfer Protocol

The File Transfer Protocol (FTP) is a well known and well understood protocol. However, the problem with using FTP for a compression study is that the compressibility figures depend heavily on the type of file being transmitted. To help address this problem, three different FTP graphs are represented. Figures 5.1 and 5.2 describe the compressibility of inbound and outbound packets for the transfer of a previously compressed file. Figures 5.3 and 5.4 represent the same information for uncompressed text files while Figures 5.5 and 5.6 are for executable files.

The collection process used the FTP *put* command to transfer the files. Statistics were also gathered on the *get* command with the expected result that the inbound and outbound packet graphs were reversed. The collected data includes the protocol exchange that sets up the FTP session and only includes Domain Name Service DNS and FTP packets. To help reduce the chance of anomalies, each session transferred multiple but distinct files of the same type (compressed, text, or executable).

Careful inspection of graphs 5.1 and 5.2 will show expected results. Discussion starts with the transfer of previously compressed files (using GNU's gzip utility). In such a condition, the packet payload contains data that is not compressible. Van Jacobson TCP header compression, which is unaffected by packet payload, accounts for all the compression applied to packets. As expected, VJ compression significantly impacts the small return TCP acknowledgment packets. It has a reduced effect on the larger outbound traffic, as its effect must be amortized over the larger size of those packets.

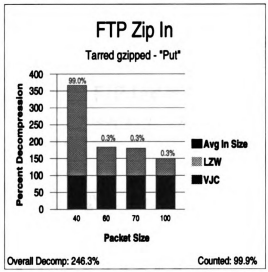

Figure 5.1: Inbound FTP (Gzip)

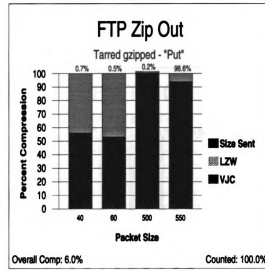

Figure 5.2: Outbound FTP (Gzip)

In Figures 5.3 and 5.4, the transfer of text based material shows the significant effects of LZW packet compression. In this case, packet payload is highly compressible. It is interesting to note that the smaller inbound packets do not tend to compress well with the LZW algorithm. Of further interest, and as shown in the final sets of FTP figures, is the fact that executable files tend to be highly compressible. It seems that compression of such files would be beneficial for low–bandwidth network based file systems.

The Network News Transfer Protocol

The Network News Transfer Protocol (NNTP) traffic represents traffic generated by an NNTP based news reader. It is assumed that the majority of NNTP traffic will be inbound as users typically query news server for articles. The traffic collection does not include traffic generated by posting articles to the server. As the majority of NNTP transfers consist of textual data, compression ratios are expected to be high. Figures 5.7 and 5.8 validate this assumption.

It is interesting to note that using a small threshold (5 packets) resulted in 13% of

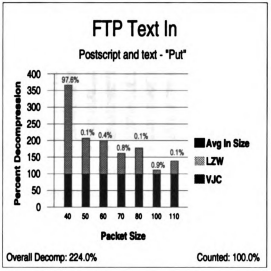

Figure 5.3: Inbound FTP (Postscript)

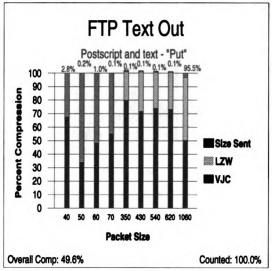

Figure 5.4: Outbound FTP (Postscript)

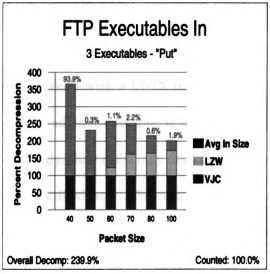

Figure 5.5: Inbound FTP (Executables)

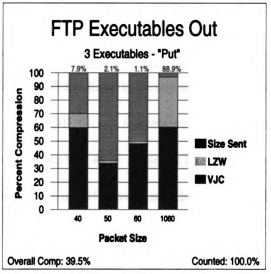

Figure 5.6: Outbound FTP (Executables)

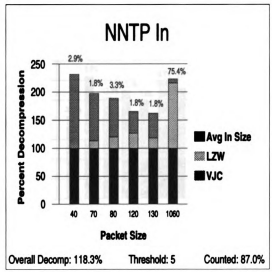

Figure 5.7: Inbound NNTP Traffic

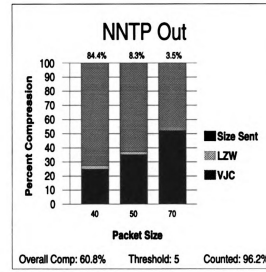

Figure 5.8: Outbound NNTP Traffic

the packets failing to pass the filter. This shows a wide variability in packet size for this protocol. A closer examination of size transitions showed that packets of size x were followed by packets also of size x only 72% of the time. Contrast this to a 95.8% average (over all three file types) for the FTP protocols previously discussed.

The Simple Mail Transfer Protocol

Simple Mail Transfer Protocol (SMTP) traffic was collected by monitoring sendmail transactions with a UNIX mail server. Since low–bandwidth links typically operate in disconnected environments, it is assumed that they run POP clients for retrieving mail. Accordingly, this traffic only describes outbound SMTP transfers while inbound POP transfers are described later. The collection of messages used in the investigation includes both ASCII and MIME mail. The mail messages selected for transmission are duplicates of those sent by third parties to one of the authors during one standard business day. These messages were collected and resent through the testbed mechanism.

While MIME mail may contain many types of data, this investigation concentrates

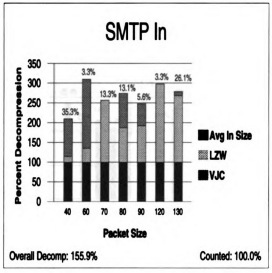

Figure 5.9: Inbound SMTP Traffic

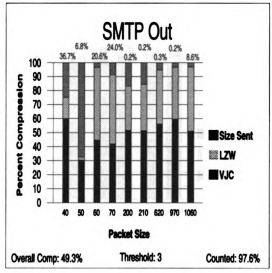

Figure 5.10: Outbound SMTP Traffic

mainly on text based messages. No graphics, video, or sound were included in the messages. It is asserted that including such items in the messages leads to the same variabilities as seen in the FTP protocol. For now, it is apparent that most mail messages are mainly composed of text. Future work may address compressibility of MIME messages with varying content.

When viewing Figures 5.9 and 5.10, it is again interesting to note the variability of packet sizes. These variabilities are in fact due, in part, to the HELO protocol initiated by sendmail. The multiple stages of this protocol keep the distribution from looking like the expected bimodal distribution as seen in the POP protocol (described later). Further, one might notice that the majority of packets are small. This is due to sendmail opening a new connection for each outbound message. There are proposals for SMTP pipelining [14] that address just this issue. Overall, the text based nature of mail and HELO lend themselves well to packet compression.

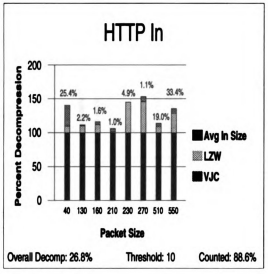

Figure 5.11: Inbound HTTP Traffic

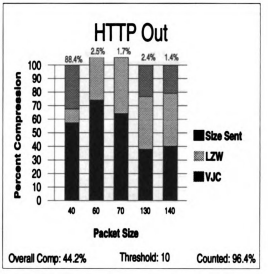

Figure 5.12: Outbound HTTP Traffic

The HyperText Transfer Protocol

The HyperText Transfer Protocol (HTTP) data were gathered using LightHouse Design's OmniWeb browser. This is a multi–threaded browser that provides support for most Netscape extensions. The browser supports data and image caching. For collection, the cache was initially cleared and graphics were set to display on each page. Data were collected by starting at the Yahoo World Wide Web site (http://www.yahoo.com/) and then following related links through several areas of interest. Only the HTTP protocol was used. The mail, news, gopher, and file transfer options of the browser were not used.

As with NNTP, it was surprising to see the variability of packet sizes seen during the sessions. A ten packet threshold filtered approximately 12% of the packets. Since browsers are mainly consumers of information, a more defined bimodal packet distribution might have been expected. Only 57% of inbound packets were followed by packets of the same size. However, 81% of outbound packets were followed by packets of the same size. The overall inbound compression was not as high as might be desirable. This is likely due to the high graphic content present on most WWW pages. Since World Wide Web traffic is

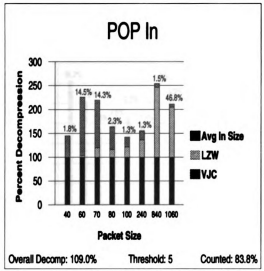

Figure 5.13: Inbound POP3 Traffic

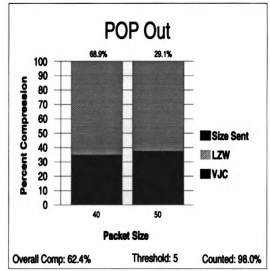

Figure 5.14: Outbound POP3 Traffic

starting to account for a major portion of traffic flowing over networks, this protocol may merit special attention for compression and optimization.

The Post Office Protocol

The Post Office Protocol (POP) traffic was collected by monitoring the receipt of the same messages that were previously sent in order to gather the SMTP traffic. It is interesting to note that without the HELO protocol, there is a much larger percentage of large packets. Further, the acknowledgment traffic is far more consistent. As expected, the compressibility is fairly good and the overall compression figures for outbound SMTP and inbound POP are similar.

The Login Protocol

The LOGIN protocol (produced using the *rlogin* command on UNIX systems) is very similar to the TELNET protocol. It is a mechanism to open a text based terminal to another host computer. Typically, such sessions are characterized by many outbound packets

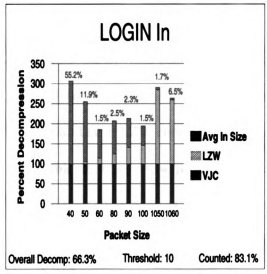

Figure 5.15: Inbound LOGIN Traffic

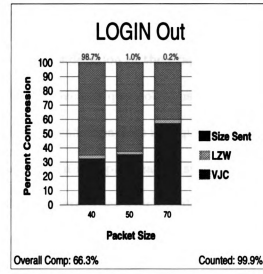

Figure 5.16: Outbound LOGIN Traffic

produced by individual keystrokes and many inbound packets corresponding to individual acknowledgments (used to echo typed characters). Occasionally, large bursts of data are received as local screens get updated with new information.

Traffic collection for this protocol was performed by opening a terminal onto a peer computer. During the session, files were edited, mail was read, standard UNIX commands were issued, and WWW sites were viewed using the *Lynx* text based WWW browser. It is asserted that regardless of what applications are used over the link, the text based nature of the traffic patterns will remain fairly consistent. Thus, the behavior of the user probably will not have much effect on the compression ratios observed. As can be seen, the majority of packets are small and do not benefit from the LZW compression. Different compression schemes that work well on small packets may merit consideration. Such schemes may be generally applicable over all TCP sessions regardless of protocols.

Discussion

Besides the obvious compression figures, careful consideration of the graphs yield some less obvious results. The first was a bit surprising but became quite obvious in retrospect. The variability in LZW compression leads to a wider spread of packets into quantized levels. While this does not cause problems, it may be of concern for optimization systems. It was assumed that small packets would have regular size characteristics which could be exploited. LZW seems to introduce irregularities into the system.

Also of interest are the discrepancies for LZW compression of small acknowledgment packets. Return packets for some protocols (namely NNTP and some FTP) were not as susceptible to LZW compression as other protocols. Since VJ compression had already been applied, it was assumed that the bodies of these packets would all contain fairly similar data and that they would all compress in a similar manner. Because acknowledgments make up such a large percentage of packets, it may be worthwhile to investigate a hybrid environment that includes special compressors for smaller packets.

Of further surprise was the poor compression characteristics of HTTP. It appears that the graphic content, which is generally not very compressible with standard algorithms, dominates many pages. Even a large text page with a small graphic may have the transmission dominated by the transfer of the graphic. Images in JPEG format should be considered as being fairly incompressible. However, other types of binary data may be more susceptible to data compression. This protocol probably merits special attention.

Finally, one of the most surprising result was that many protocols did not appear to have a strong bimodal distribution as originally thought. For instance, outbound SMTP traffic was originally thought to consist of many large packets that carried the mail messages to the server (most mail messages were large enough to fully fill several packets). However, less

than 10% of the packets were considered large (the size of a packet's maximum transmission unit). The majority of packets were under 100 bytes long. The compression schemes employed work better when there are larger runs of data. As mentioned above, a hybrid compression scheme that uses packet size as a selector may perform well.

5.3 Size and Protocol Transitions

Traffic collected during the original analysis described in Chapter 4 does not contain enough information to determine compressibility statistics. However, protocol and size transition information can be derived. For the sake of completeness, this section describes the transition information for the traffic dumps previously collected. These transition figures are much more indicative of what a low–bandwidth communications system may actually encounter.

Table 5.1: Inbound transitions (794590 packets counted)

First Order Inbound Transitions		
Previous Size	Next Size	Percentage
40	40	20.149511
300	300	22.523565
550	550	10.901849
1060	1060	13.617589
Previous Size = Next Size		72.699150†
Previous Size ≠ Next Size		27.300848†
Encompasses 67.2% of packets † – not affected by 5% Threshold		

Second Order Inbound Transitions		
Prev Two Sizes	Next Size	Percentage
40,40	40	17.380684
300,300	300	21.144390
550,550	550	10.057527
1060,1060	1060	12.721923
Prev 2 Sizes = Next Size		64.677879†
Prev 2 Sizes ≠ Next Size		35.322121†
Encompasses 61.3% of packets † – not affected by 5% Threshold		

Tables 5.1 and 5.2 show the first and second order quantized size transitions for inbound and outbound packets respectively. Percentage thresholds are applied to limit the number of transitions displayed. The *Encompasses* percentage describes the number of packets

passing the filter and included in the transition information. Percentages are also shown for transitions that include a change in quantized size. These size change percentages are not affected by the threshold and apply to all packets traveling in the required direction.

Table 5.3 views the same transitions from the perspective of application level protocols. In this case, only change transitions were gathered and included. Thresholds were not applied to this collection so these numbers represent percentages gathered for all packets.

Table 5.2: Outbound transitions (737650 packets counted)

First Order Outbound Transitions		
Previous Size	Next Size	Percentage
40	40	56.540966
50	50	5.539611
300	300	9.450810
550	550	5.466677
Previous Size = Next Size		81.704994†
Previous Size ≠ Next Size		18.295006†
Encompasses 77.0% of packets † – not affected by 5% Threshold		

Second Order Outbound Transitions		
Prev Two Sizes	Next Size	Percentage
40,40	40	52.393955
300,300	300	9.164780
Prev 2 Sizes = Next Size		73.374336†
Prev 2 Sizes ≠ Next Size		26.625664†
Encompasses 61.6% of packets † – not affected by 5% Threshold		

Table 5.3: Protocol Transitions

Protocol Transitions on Inbound packets	
Prev Protocol = Next Protocol	91.226387
Prev Protocol ≠ Next Protocol	8.773617
No Threshold in effect Encompasses 794590 packets	

Protocol Transitions on Outbound packets	
Prev Protocol = Next Protocol	89.989731
Prev Protocol ≠ Next Protocol	10.010271
No Threshold in effect Encompasses 737650 packets	

Discussion

It is clear from the tables that packets come in runs of the same size and protocol. Since many low–bandwidth systems act mainly as accessors of information, one might expect the inbound traffic stream to have quite a variability in protocol mix. While this is the case, the transitions between different protocols does not occur often. Long runs of the same protocol mean that compression algorithms can rely on the ability to train their dictionaries. Further, from the previous chapter it can be readily seen that the packet size distribution is skewed in the well known bi–modal distribution. This validates previously held beliefs about protocol behavior and bolsters optimization methods that rely on such assumptions.

As mentioned previously, packet compression variability tends to scatter packets into many quantization levels. Enabling LZW compression will probably increase the percentage of packets that are followed by packets of a different size. Such a scheme would reduce the length of runs for packets of the same size. Schedulers that rely on this information would not be as effective.

Finally, it would be interesting to compare these transitions to the data collected in the previous section. However, this would be inappropriate. The LZW compression graphs are limited to specific protocols and occur under controlled conditions. Protocol transitions would obviously be skewed since only one protocol would be in use. Size transitions would also be inaccurate since packet mixture from concurrently active protocols would not exist.

5.4 Summary

Packet payload compression is an effective mechanism for better utilizing the limited bandwidth of wireless networks. The previous chapter identified the application level protocols commonly used in low–bandwidth networks. This chapter further analyzed those

protocols to generate compressibility statistics. Since the data compressors are working in a noise free environment, the results presented here give us a base for comparison against the same algorithms operating in the presence of noise. Such operation is described in the following chapters.

Chapter 6

Header Compression

6.1 Introduction

As shown in Chapter 5, data link level packet compression can significantly improve the performance of low–bandwidth connections. One widely used transport protocol header compression algorithm is known as Van Jacobson TCP header compression, named after is creator [23]. An implementation of this algorithm now exists for almost every Point–to–Point (PPP) and Serial Line IP (SLIP) package available today. Van Jacobson TCP header compression, commonly designated as VJ compression, is mainly used to increase the performance of interactive communications sessions. The compressor can reduce the header size of common TCP traffic from approximately forty bytes to five or less. On low speed links, this reduction allows interactive connections to exhibit better response.

Implicit in the VJ compression process is the notion that both endpoints maintain state information. Each end must remain tightly synchronized with its peer. Loss of synchronization leads to incorrect results from the decompressor. Incorrect results in turn mean dropped packets, TCP retransmissions, and timeout latencies. On relatively clean

links, such as those provided by an error correcting telephone modem, loss of synchronization is infrequent and performance effects are negligible. However, in the case of wireless links, errors (especially lost packets) are more frequent. Loss of synchronization can significantly degrade the performance of sessions employing VJ compression.

This chapter will focus on the problems caused by corrupted state and loss of synchronization during VJ compression. The effects of corrupted state are quantified and a new approach for resynchronization is proposed. This new noise tolerant approach removes much of the dependency information implicit on the link. It is shown that the penalty for this new method is small for noiseless links while the gain is significant when errors are present. The rest of this chapter is organized as follows: section 6.2 gives brief overview of VJ compression along with a performance evaluation, section 6.4 describes the removal of dependency information and its related performance gain, and section 6.6 describes the overhead of the new technique.

6.2 The VJ Algorithm

Van Jacobson TCP header compression is fully described, along with a reference implementation, in RFC 1144 [23]. While it is not our intention to explain all the intricate details of the algorithm, this section will review some of the basic concepts to give a base for discussion of the modifications.

VJ compression is a transport header compression that is specifically tailored for TCP. It uses intimate knowledge of TCP characteristics to achieve considerable compression of TCP headers. VJ compression treats the physical link as consisting of two simplex links, one each direction going from compressor to decompressor. This is important to note since it implies that there is no direct backwards flow of information from the decompressor to

the compressor.

The basic premise of the VJ algorithm is that roughly half of the bytes of a TCP/IP header will remain constant over the lifetime of the TCP connection. Of the half that change, most change in a predictable fashion. With a copy of the header of the last packet received on a connection, the VJ algorithm is able to discern these changes through a small set of differential values that come in the VJ compressed header of the current packet. Applying these changes to the old copy and updating that old copy allows the algorithm to reconstruct the TCP/IP header at the decompressor and be ready for the next incoming packet. Through some special casing, the forty byte TCP/IP header can be reduced to five or even three bytes.

VJ Compression gives you a near fixed size reduction (plus or minus a few bytes). It is unaffected by the size or type of data held in a TCP packet payload. As the maximum transfer unit (MTU) of a datagram shrinks (i.e., with wireless systems), the overhead of the TCP/IP protocol header increases. Smaller packets (those with less payload) exhibit better compression ratios since the TCP/IP protocol header makes up a larger portion of the total packet size. As the figures in the previous chapter show, VJ compression is useful for overhead reduction even with non–interactive traffic. Under ideal conditions, VJ compression should never hurt performance as the header will never get larger.

One main characteristic of the algorithm is that the differential is applied between successive packets. If a packet is dropped by the link layer due to noise, then that packet will never reach the VJ decompression routine. Because the differential never gets applied and the local copy of the TCP/IP header never gets updated, then future packets will become corrupted as they get updated with incorrect information. This is known as losing synchronization. Synchronization will be lost anytime that a packet gets dropped by noise.

The solution to lost synchronization is for the compressor to send an uncompressed TCP/IP packet through to the decompressor. The decompressor can then save this entire correct header and begin applying the future differentials again.

6.2.1 Packet Loss

It is important to understand where packets get dropped using this differential scheme. There are potentially two places where errors can occur. The first is in the TCP stack. The decompressor does not have a direct means to determine if it has lost synchronization. If a packet gets dropped by the data link layer, then the next packet to arrive at the VJ decompressor will be updated as normal, albeit incorrectly because of the lost differential update of the previously dropped packet. No errors will occur in the VJ algorithm and the packet will be passed on as usual. It is only later, in the TCP stack, that the packet will fail the TCP checksum and be dropped. The lost packet causes the decompressor to lose synchronization. Each successive packet that arrives gets dropped by TCP until an uncompressed packet arrives and resynchronizes the system.

The second place that packets can get dropped is right at the start of the VJ algorithm. One of the options available to VJ is to compress the connection identifier. Using this option, the system will use the same connection as in the previous packet when no connection ID is present. Unfortunately, if a packet gets dropped, then you may have missed a connection identifier switch and you may be applying updates to packets on the wrong connection. In some circumstances, there is the possibility that this type of error will go undetected even by TCP. The VJ algorithm can be notified of a packet drop by the lower layer. If this notification gets sent, then the VJ decompressor will toss every packet until it receives one with an explicit connection ID.

6.2.2 Synchronization

The issue of synchronization is discussed in the VJ document. Since there is no backwards flow of information from the decompressor to the compressor, the compressor must find some way to determine when the decompressor has lost synchronization. The mechanism used is to detect TCP retransmissions. Since loss of synchronization is one reason that a packet may not have arrived intact at the destination TCP stack, the VJ algorithm conservatively sends all retransmissions as uncompressed. Since uncompressed datagrams will resynchronize the peer then *if* a loss occurred, it will be fixed during the retransmission. Of course, if there was a loss of synchronization, all packets that are sent before resynchronization will eventually be dropped and retransmitted uncompressed.

6.3 Performance Analysis of the VJ Algorithm

VJ compression is mainly designed for interactive traffic but can also provide small gains for bulk data transfer applications. Because of it low overhead and general effectiveness, it is desirable to always use VJ compression. This is especially true as the datagram MTU shrinks. Smaller MTUs means a larger percentage of the packet contains transport protocol overhead. VJ compression can effectively remove much of that overhead. On a noiseless channel, VJ compression should never decrease the throughput of a connection. Unfortunately, this is not true when the channel becomes noisy.

The method chosen to measure the effectiveness of VJ compression is throughput of bulk data transfers. Bulk transfers are used because they give a more accurate long term measure of compression effectiveness. For this analysis, throughput values were collected for three different files of sizes 344K, 328K, and 550K. Each file was sent five times, and the

average value used for graphing. Since VJ compression is not affected by packet payload content, the content of the files, although shown, is unimportant.

To simulate a noisy wireless environment, a noise signal was combined with the datastream at the data link layer. The noise, as described in Chapter 3, consists of a stream of bits, zero representing error free transmission and one representing noise, that were added to the incoming and outgoing data bits. For completeness, two separate noise streams were tested. The first is a random noise stream having bit probability of error of approximately 7.3×10^{-3}. This model is useful in determining the effectiveness of the new algorithm independent from the variances introduced by large error–free gaps that arise in burst models. However, since wireless channels are better represented by a bursty model, the second error stream was generated using the four state Fritchman–partitioned Markov Chain. While both data streams have approximately the same probability of bit error, the random model has a higher probability of packet error. This is because no error correction is done at the data link layer. A single bit error will drop a packet as easily as a burst of bit errors. Since the random model has relatively uniform error–free gaps, it consequently has a higher probability of packet loss.

Figures 6.1 and 6.2 show the performance of VJ compression both in a noiseless and a random noise environment and with differing MTU values. It should be noted that in the error free case, VJ compression always provides better throughput than no VJ compression. Further, the effects of VJ are more pronounced for the smaller MTU packet.

As noted in section 6.2.1, it is possible for the VJ algorithm to lose synchronization. When this happens, the decompressor starts to toss packets. When the VJ compressor sees TCP retransmits, it will send the packets uncompressed thus resynchronizing the two endpoints. You can see the effects if synchronization loss by looking at the performance in

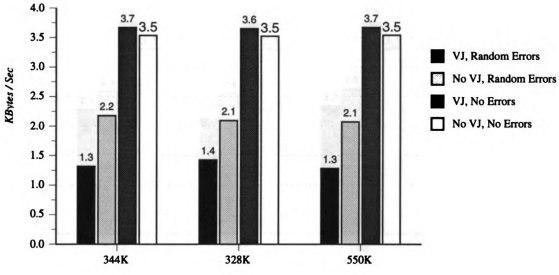

Figure 6.1: Standard VJ, Random Noise, 1500 Byte MTU

the presence of noise. It is far better to avoid the use of VJ compression than to use it in the presence of noise. This is especially true for large MTU values.

6.4 The New VJ Algorithm

VJ compression is beneficial for use on noiseless links, especially those using small MTU values. Wireless links now present a challenge. Since they typically have small MTU values, it would be useful to employ VJ compression. However, since the links are error prone, it is advantageous to avoid the use of VJ compression. The question that arises is what can be done to improve VJ performance in the presence of noise? If one could raise the performance of bulk transfers to the level achieved when not using VJ compression, then one would break even on bulk transfers and still gain the advantages of VJ for interactive use. Outperforming the no VJ case would indicate a measurable reduction in protocol overhead. The following subsections describe a new algorithm that can attain this.

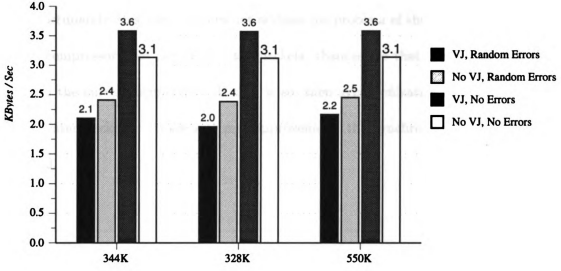

Figure 6.2: Standard VJ, Random Noise, 296 Byte MTU

6.4.1 No VJ Slot Compression

From the previous discussions on packet loss, one area to investigate would be the tossing of packets with no connection identifiers. Table 6.1 gives some idea of how many packets actually get discarded due to the compression of connection identifiers in noisy links. The *packet error* field indicates the number of errors detected by the data link layer. The *toss* field gives the number of packets that were discarded by the VJ decompressor *in addition to* the lost packet at the data link layer. These numbers correspond to the transfer of the file *combined.ps* in the random noise graph.

Table 6.1: Effects of Slot Compression in a Noisy Link

MTU	Number of Packet Errors	Number of Additional Packets Discarded
296	169	707
1500	184	291

The obvious solution would be to always send the connection ID. In such cases, one always adds an extra byte to the compressed TCP header with the benefit that the

decompressor does not get confused about connection identifiers and does not discard packets. Unfortunately, this solution does not address the problem of the missing update. Even if the decompressor does not discard the packets, chances are that the errored packet belonged to the current connection. If that is so, then synchronization loss causes TCP to drop all later packets. To see any real improvement, the synchronization issue must be addressed.

6.4.2 Selective Application of VJ

A second approach to dependency removal would be to selectively apply VJ compression to a subset of packet headers. Since large packets are more prone to be affected by an error, one could choose to only apply VJ compression to small packets. In doing so, one gains the benefits of VJ compression for the interactive traffic but does not not suffer many of the drawbacks of operation over a noisy link. The bimodal traffic pattern described in chapter 4 provides a good value for a cutoff point based on packet size. As shown in figure 4.1, a cutoff point between 110 and 290 bytes would capture a significant amount of the packets passing a link. The exact value really depends on the MTU and resides just below that MTU value. Figures 6.3 and 6.4 show the performance of compressing packets smaller than 160 bytes.

One can note a slight gain, in some cases, when using large MTU transfers. However, for bulk transfers, the gain only provides throughput similar to that of not using VJ. As the MTU size decreases, you lose the benefits of VJ compression in conjunction with small MTU packet overhead. For this reason, a better approach is sought.

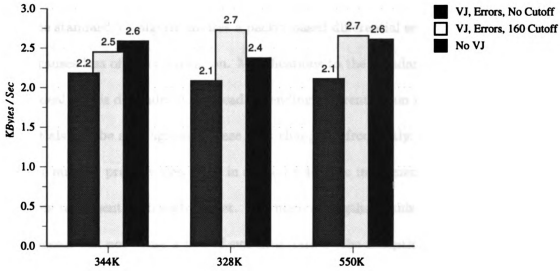

Figure 6.3: Standard VJ, 1500 Byte MTU, Cutoff Size of 160

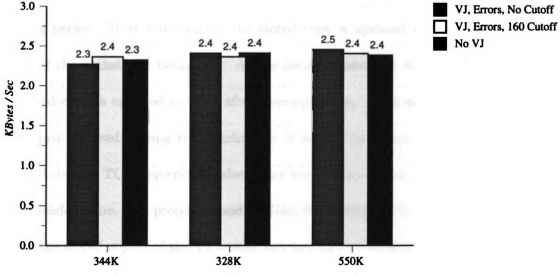

Figure 6.4: Standard VJ, 296 Byte MTU, Cutoff Size of 160

6.4.3 Interpacket Dependency Removal

The previous subsections have outlined the reasoning behind loss of synchronization. Since the standard VJ algorithm uses a packet based differential encoding scheme, loss of a packet causes loss of synchronization. Modifications to the standard algorithm can remove a great deal of this dependency. Instead of sending differentials on a packet by packet basis, differentials can be sent against a base that changes infrequently. Further, to combat the sequence number problem described in section 6.4.1, the implementation forces connection identifiers to be sent with each packet. The main strengths of this approach are that most packet losses will not cause a loss of synchronization (the exceptions are described below) and that the core operation of the proven VJ algorithm remains unchanged. The drawbacks include a minor increase in memory usage and a minor increase in the size of the compressed header.

In the VJ algorithm, packets are encoded and decoded against a stored copy of the previous packets header. The differentials are calculated and replace the header in the outgoing packet. More importantly, the stored copy is updated to be a duplicate of the header of the packet just being sent. At the decompressor, the reverse happens. Namely, the stored copy is updated so that, after decompression, it is a copy of the header of the packet just received. Using this mechanism it is easy to detect retransmissions because they will contain TCP sequence numbers that are the same as or smaller than the current stored header value. The proposed modification, called New VJ in this discussion, is to not update the stored copied of the VJ header but to continue using the VJ algorithm in other respects. Since all differential encoding is done using a fixed base at both ends, packet loss will not cause endpoints to lose synchronization.

Such a scheme is not quite as simple as removing the header update code. Care must be

taken to maintain enough information such that the retransmission and special case policies of VJ still function properly. This entails expanding the amount of information kept about each active TCP connection. Along with the TCP/IP header that is kept, one must also now keep the TCP window, acknowledgment, sequence number, and size of the last packet flowing over each connection. This information is always updated with the last packet while the stored TCP/IP header remains unchanged.

The new VJ algorithm is still able to use the same conservative mechanism that the original VJ algorithm uses for detecting retransmissions and resynchronizing the link. The question now arises as to why resynchronization is necessary? Since packet encoding is performed on base copies, there is no interpacket dependency. Unfortunately, it is entirely possible that the compressor may find it necessary to transmit an uncompressed packet. In addition to retransmissions, such packets are occasionally sent to combat overflow problems with the differential encoding. As they flow to the decompressor, they become the new base packets against which differentials are calculated. If an uncompressed packet gets dropped due to noise, then the algorithm still loses synchronization. In such cases, operation degenerates to the same procedure as the standard VJ algorithm until an uncompressed packet makes it through to the decompressor.

6.5 Performance Evaluation of the New VJ Algorithm

The following figures illustrate the performance characteristics of the new algorithm. Figures 6.5 and 6.6 show the performance degradation of the new algorithm in a noiseless environment. This degradation is due to several factors and will be discussed in section 6.6. In all cases, this degradation is small and the new algorithm still provides increased performance as the MTU size decreases. The rest of the figures describe performance for

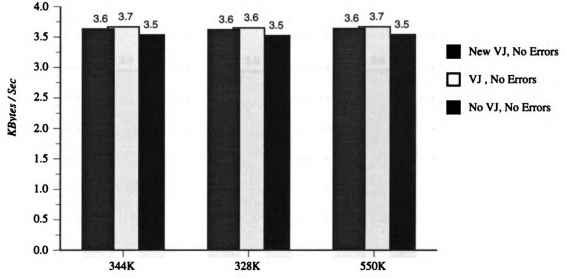

Figure 6.5: New VJ Performance With No Noise, 1500 Byte MTU

random and burst error models. One can note significant gains, in some cases up to 77%, over the old VJ algorithm. The new algorithm also shows gains over the case of no VJ compression.

The variances due to using differing MTU values in the same noise environment reflect the increased probability of packet loss for large MTU transfers and the associated cost of retransmissions when a packet is lost. Also, as previously discussed in section 6.3, the random error environment has lower throughput because of its higher probability of packet loss. However, it is important to note that the gains of the new algorithm increase in such an environment. The performance of the old algorithm is intimately associated with the number of dropped packets. The new algorithm has removed much of this dependency. The approach of dependency removal has attained the goals of allowing transport header compression to exist in a noisy environment while attaining better performance than if no compression were applied.

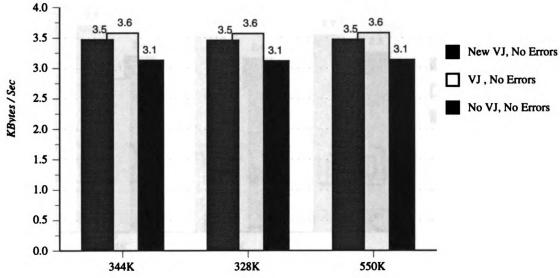

Figure 6.6: New VJ Performance With No Noise, 296 Byte MTU

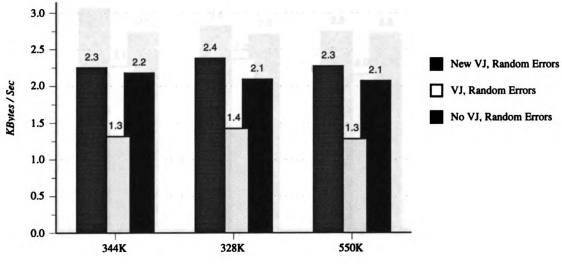

Figure 6.7: New VJ Performance With Random Noise, 1500 Byte MTU

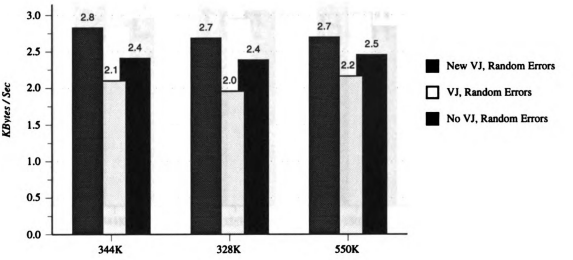

Figure 6.8: New VJ Performance With Random Noise, 296 Byte MTU

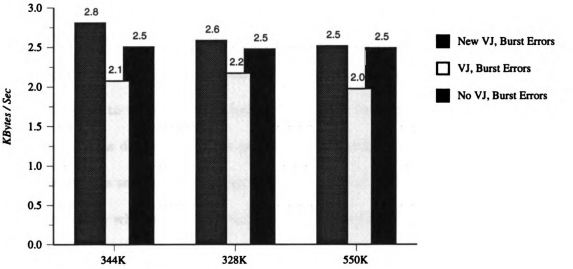

Figure 6.9: New VJ Performance With Burst Noise, 1500 Byte MTU

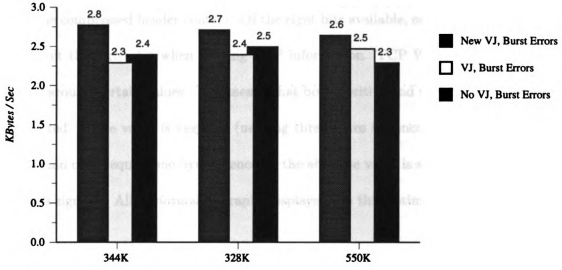

Figure 6.10: NewVJ Performance With Burst Noise, 296 Byte MTU

6.6 Overhead and Optimizations

It is important to understand the overhead involved in the new approach. Dependency removal adds some overhead to packet size and thus reduces its effectiveness over noiseless channels. One fixed overhead addition is the added byte needed for a connection identifier. This is required so that lost packets will not cause the decompressor to become confused about the session to which a packet belongs. A second overhead is the reduction in the coding efficiency as differential values grow larger that 255. Both the standard and new algorithms use the same variable length encoding scheme. Positive values smaller than 256 take one byte while others values take three. Since the original VJ algorithm uses interpacket dependencies, changes are usually small. However, when one starts to generate changes based on a non–changing header, some values may grow larger than usual. To help reduce these overheads, a few areas were exploited to possibly provide a marginal increase in performance.

The compressed VJ header contains, as its first byte, a change mask that describes the rest of the compressed header content. Of the eight bits available, only seven are used. One can exploit this extra bit when sending TCP information. TCP Window values typically oscillate around certain values. This means that both positive and negative values must be transmitted. If the value is negative (needing three bytes for encoding), and its absolute value would only require one byte to encode, the absolute value is sent and the extra bit is used as a sign bit. All performance graphs displayed use this optimization.

A second approach would be to use the bit for backwards communication with the TCP compressor. With such an approach, the bit could be used to signal that the decompressor has become unsynchronized. This would obviate the need for the compressor to send retransmissions uncompressed. In order to exploit this approach, the TCP decompressor must perform a TCP checksum over the reconstructed packet. Failure of the checksum indicated loss of synchronization. This approach was investigated and then discontinued. Besides the obvious overhead of calculating the TCP checksum twice (our TCP stack and VJ Decompressor do not communicate), the gains would not justify the other overheads. Backwards communication adds a significant amount of protocol state and complexity. The only benefit would be to reduce the number of uncompressed packets that flow across the link. Since a new VJ header requires approximately 6 bytes and an uncompressed packet requires 40, one only gains a savings of approximately 34 bytes per uncompressed packet. In large transfers, the extra overhead of uncompressed packets will only amount to a few thousand bytes total. This was verified by adding a mechanism to allow sending the first copy of any retransmission compressed while allowing any further retransmissions to go uncompressed. No significant performance gain was attained under the error conditions present.

6.7 Summary

This chapter was devoted to showing how state maintenance in the VJ TCP header compression algorithm could be modified to better tolerate errors. The original algorithm was shown to behave poorly on noisy channels. The first proposed enhancement required that connection identifiers always be sent with the compressed packet. This keeps the decompressor from tossing packets while it waits for a new connection identifier. Unfortunately, it did not address the loss of synchronization. This technique alone was not sufficient to increase performance.

The results of Chapter 4 lead to the second enhancement. By selecting an appropriate cutoff based on packet size distributions, an enhanced algorithm was proposed that selectively applies VJ compression. A cutoff size, somewhat smaller than the MTU, allows one to capture a significant amount of traffic while avoiding compression of the large error prone packets. This approach increased the performance to around that of not using VJ compression. The benefit of the approach is that VJ compression is still applied to the small packets typically found on interactive sessions.

In an effort to outperform a connection that avoids the use of VJ, a new noise tolerant protocol was developed that removed most of the interpacket dependencies. These modifications do not alter the foundations of the VJ algorithm and they still maintain the assumptions of simplex operation over an unreliable channel. Resynchronization, when needed, is performed in the same manner as the original algorithm (i.e., by detecting TCP retransmissions). The new protocol is tolerant of errors and allows superior throughput on noisy links while maintaining good performance on noiseless links.

Chapter 7

Payload Compression

7.1 Introduction

Protocol header compression algorithms, as described in the last chapter, are important because they can significantly reduce the overhead of transport protocols for interactive sessions. Further, they can provide noticeable benefits for systems that require a small maximum transmission unit (MTU). Unfortunately, the algorithms are usually tied closely to the structure of a protocol header and cannot be extended to the datagram payload. The compression investigation in Chapter 5 shows that payload compression algorithms can also significantly reduce the datagram size. Measurements on bulk data transfers show byte reductions that far exceed the effects of header compression. This is of utmost importance for wireless and other low–bandwidth interfaces and is the motivating force behind investigating compression algorithms in the presence of noise.

One widely used payload compression algorithm is based on Welch's modifications to the popular Ziv–Lempel technique presented in 1978 [51]. Welch's algorithm, known as LZW (Lempel–Ziv–Welch) [47] is the same algorithm that is used in the common BSD UNIX

compress program. A free implementation of this algorithm now exists for a popular public domain Point–to–Point Protocol (PPP) package [30]. LZW compression is mainly used to increase the performance of bulk transfers over links employing low–bandwidth connections. Spending extra CPU cycles at the endpoints can significantly reduce the amount of data that must pass through the bottleneck low–bandwidth link of the network connection.

As with VJ Compression, loss of synchronization can significantly degrade the performance of sessions employing LZW compression. This chapter will focus on the problems caused by corrupted state and loss of synchronization during LZW payload compression. The effects of corrupted state are quantified and a new approach to LZW state maintenance is proposed. This new method removes much of the dependency information implicit on the link. It is shown that the penalty for this new method is small for noiseless links while the gain is significant when errors are present.

7.2 LZW Packet Payload Compression

Chapter 6 dealt with header compression over simplex links (i.e., no backwards communication from the decompressor to the compressor). This section will introduce the second algorithm, a packet payload compressor. Unlike VJ header compression, this algorithm provides a logical control channel at the link layer. This allows full communication between the decompressor and the compressor and allows more flexibility in the design of the algorithm. The payload compression algorithm is based on an LZW implementation. LZW compression is mainly used to increase the performance of bulk transfers over links employing low–bandwidth connections. Spending extra CPU cycles at the endpoints can significantly reduce the amount of data that must pass through the bottleneck low–bandwidth link.

In the following discussion, it will be assumed that the input to the LZW compressing

algorithm will be an infinite stream of input symbols. The compressed output will transmitted over an unreliable link to a peer process that will decompress the stream to its original state. Unlike file based compression where the communication between compressor, decompressor, and information source (a file on disk) is virtually noiseless, the unreliable nature of the communication link requires the use of extra mechanisms to ensure synchronization between the compressor and decompressor. This section will provide an overview of both the algorithmic process of LZW compression and the important details of the implementation. A good understanding of both is required for understanding the proposed modifications. Readers wanting a more thorough treatment of the LZW algorithm are referred to the excellent presentation by Williams [48].

7.2.1 The LZ78 Compression Algorithm

Before describing the LZW technique, a review of its predecessor, LZ78, will be presented. To provide compression, the LZ78 algorithm uses the notion of a *dictionary of phrases*. Starting with an empty dictionary, the algorithm scans the input datastream and finds the longest sequence of characters that matches a phrase already in the dictionary (this may be the empty string). It then sends a pair of values to the output. The first is an index to the matching phrase found in the dictionary while the second is the first input character following the longest matching phrase. Once the two codes are emitted, the algorithm adds to its dictionary a new phrase that consists of the concatenation of the matched phrase and the new character.

The dictionary can be conceptualized as a digital search tree. Similar to Figure 7.1, each node is numbered and represents the phrase found by following the path from the root to the node. To match the datastream input, one simply follows the tree using the input data as a guide. Once a leaf node is encountered, the algorithm issues that node number as

output. Further, it emits the next data input character and then adds that character as a new leaf node. Thus, the dictionary grows by one character for each pair of codes emitted.

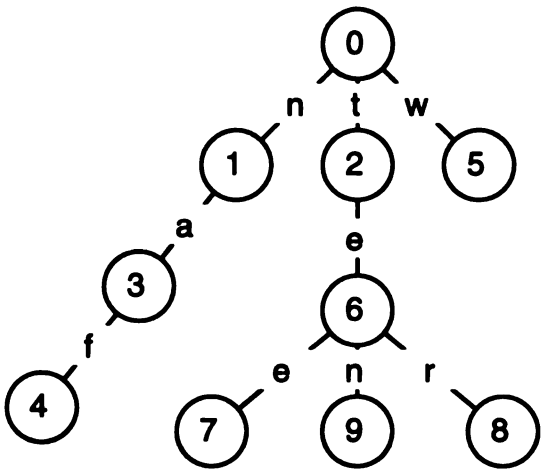

Figure 7.1: LZ Dictionary

The receiver performs decompression in a similar manner. The decompression dictionary starts out empty and, as the phrase index and next character pair arrive, the decompressor reconstructs the dictionary and rebuilds the original input. It is important to note that no explicit dictionary is ever transmitted to the receiver. Instead, the decompression dictionary is dynamically recreated based on the input. This recreation requires that the input to the decompressor precisely match the output of the compressor. Any discontinuity in the datastream will result in an incorrect rebuilding of the decompressor's dictionary and subsequent incorrect generation of the uncompressed data. Maintaining this synchronization is a key issue that will be discussed later.

7.2.2 The Lempel–Ziv–Welch Algorithm

The LZW algorithm is a variant of the LZ78 algorithm. Instead of explicitly transmitting the first symbol found after a phrase match in the dictionary, the algorithm emits the

dictionary index only. It then uses the next character as the first in the subsequent phrase lookup. In order for this algorithm to work, the dictionaries must initially be seeded with each of the possible input symbols. Further, given an input phrase, the decompressor is unable to update its copy of the dictionary until it decodes the first character of the next phrase. These differences introduce a few special cases that must be addressed with the LZW algorithm. However, removal of the explicit transmission of the last character is the distinguishing factor of the algorithm.

Like LZ78, LZW compression gives highly variable data reduction and is critically dependent on the characteristics of the input data. Under normal circumstances, it is relatively unaffected by the size the datagram since the algorithm conceptually operates on an infinite input stream that is created by the concatenation of all outgoing datagrams. As will be seen later, practical considerations force the implementation to make less than ideal concessions.

7.2.3 The Algorithmic Implementation

Implementation details must address a variety of concerns that are conveniently glossed over in an algorithmic description. Data constructs, error control, and synchronization maintenance are all critical details that must be addressed. As will be shown, certain implementation choices can critically affect the performance of the LZW algorithm. This subsection will address those choices and analyze their effect on performance.

Packets arrive at the compressor in the form of TCP/IP datagrams. If a VJ header compressor is available, TCP headers are compressed by that algorithm first. The (possibly VJ compressed) datagrams are then run through the LZW compressor and framed for sending. Since the compressor and decompressor must remain tightly synchronized, part of the framing information contains a unique sequence number that is increased for every

compressed packet. Also, since certain framing and protocol information must remain intact, the algorithm preserves certain portions of the datagram by creating a logical data stream to pass to the compressor.

7.2.4 Packet Compression and Emitted Size

The Markovian nature of the LZW scheme leads to poor performance when the datastream contains pre–compressed data. Link layer interleaving of different sources can further reduce the performance of an LZW compressor. To account for this, the compressor will compare the size of the original TCP/IP datagram with that of the compressed datagram. It will always send the smaller of the two datagrams. Using such a mechanism ensures that performance of the compressor will not drop below that of the uncompressed case. However, such a choice does lead to two types of output packets. Compressed datagrams that contain a unique link layer sequence number and normal TCP/IP datagrams that do not contain a unique link layer sequence number.

An interesting byproduct of this mechanism is that the compressor's dictionary gets modified regardless of whether a compressed or uncompressed packet gets sent. This is an important process since an empty dictionary must be seeded with initial information. An empty dictionary typically yields a few uncompressed packets as it acquires a critical mass of phrases and begins to really work. However, this also means that the uncompressed packets that do not contain unique link layer sequence numbers can affect the dictionary. This will be important for discussion of decompression in the presence of noise.

7.3 Performance Analysis of the LZW Algorithm

The method chosen to measure the effectiveness of LZW compression is throughput of bulk data transfers. Bulk transfers are used because they give a more accurate long–term measure of compression effectiveness. For this analysis, throughput values were collected for two different files of sizes 344K and 550K. Each file was sent three times and the average value was used for graphing. Since LZW compression is critically affected by packet payload content, the content of the files must be justified.

The compression investigation in Chapter 5 shows that LZW compression is not effective at compressing already compressed data. However, the packet size detection mechanism discussed in Section 7.2.4 allows operation to proceed on such packets with no noticeable decrease in performance. Fortunately, the results of Chapter 4 show that the top protocols are mostly text based. So, for this investigation, files were chosen that contain highly compressible data. For completeness, Figures 7.2 and 7.3 include the transfer of a previously compressed file. Later graphs do not include this file. It is important to remember that the actual content and exact throughput numbers are not as important as is the fact that the data are compressible and causes a mix of compressed and uncompressed packets to be sent. This gives the basis for examining modifications to the compression algorithm when it is used in the presence of noise.

Figures 7.2 and 7.3 show throughput rates for two different maximum transmission unit (MTU) sizes. The large 1500 byte MTU values work well for bulk data throughput but tend not to work well in noisy environments. The smaller 296 byte MTU will be used throughout the rest of the paper. Smaller MTU values are typical of wireless links and their use tends to lead to better performance in the presence of noise. Figures 7.2 and 7.3 are for a noiseless environment and represent throughput for varying choices of the size of the dictionary table

(as described in the next section). In these, and the following experiments, VJ header compression is disabled. This is to remove the dependencies that VJ compression imparts on the link. It is apparent that on a clear channel, LZW compression provides significant throughput improvements independent of VJ compression and MTU values.

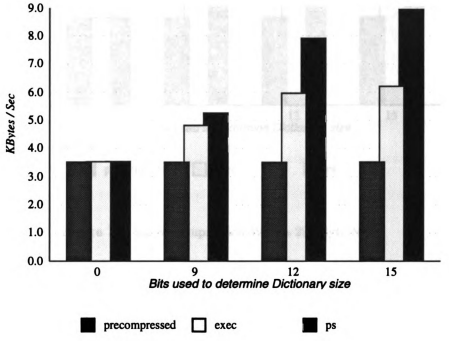

Figure 7.2: LZW Compression With a 1500 Byte MTU

7.4 Dictionary Size and Clearing

If the LZW algorithm is operated as previously described, the compressor's dictionary size will increase forever. Since this is not practical, mechanisms must be in place to manage the dictionary size and to control if and when the dictionary is cleared. The dictionary's maximum size is controlled by the output length of a dictionary index. If the compressor is able to output an index that requires nine bits for representation, then the compressor

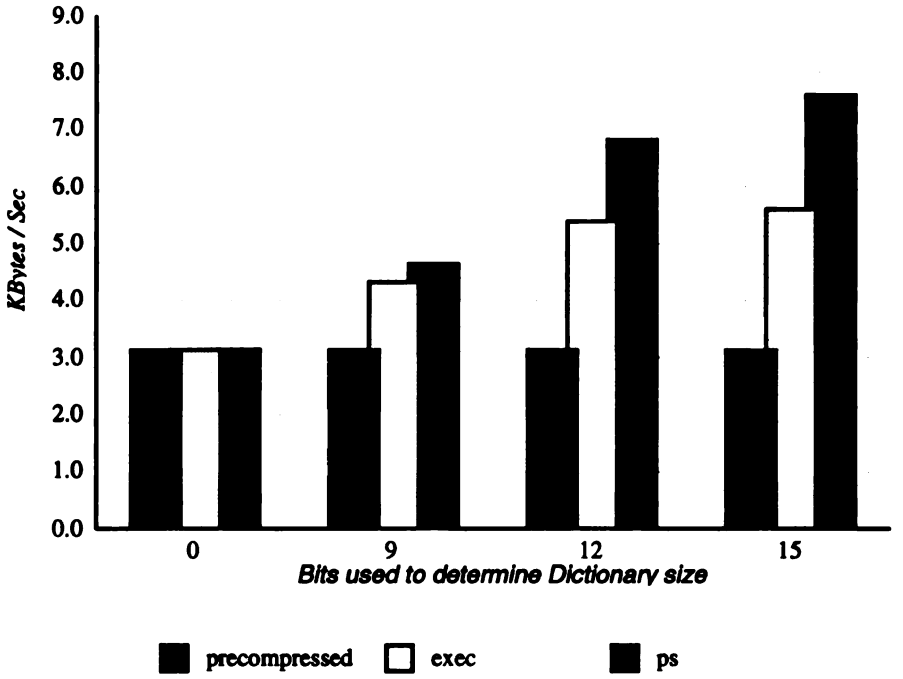

Figure 7.3: LZW Compression With a 296 Byte MTU

must maintain a structure that has $2^9 = 512$ entries. For this implementation, code sizes are allowed to range from $n = 9$ to $n = 15$ bits[1].

The previous conceptualization of the LZW algorithm used a digital search tree for representation. This search tree is actually implemented using a dual mode array. In one mode the array is indexed from 0 to $2^n - 1$. These indices represent the node numbers in Figure 7.1. Each index specifies a phrase in the dictionary and is output to the peer. Since the LZW scheme requires the dictionary to be seeded with all possible input characters, the first 256 slots (0–255) are always reserved. The largest code currently existing in the dictionary will be deemed *max_ent* and can have any value between 256 and $2^n - 1$.

The second use for the array is as a hash table. As phrases are built during compression,

[1]The actual index representation uses a variable encoding scheme that uses just enough bits to encode the number. So, the index size can range from 9 to n bits depending on the magnitude of the index value.

hash keys are generated for indexing into the array for comparison. Upon a hash failure, a matching phrase has been found and a new entry is added. New entries use the index value of *max_ent + 1*. When the table becomes full ($max\_ent = 2^n - 1$), new values are no longer added. In this case, compression still works, but only for the static codes that are now contained in the table.

Once the dictionary becomes static, it is unable to adjust to variances in the input stream. In such circumstances, a mechanism must be in place to clear the dictionary so that it can restart growth using the characteristics of the input data. The current implementation maintains a measure of the local compression ratio. Once the dictionary becomes full, the algorithm will periodically compare the local ratio to a previously calculated ratio. Only when the ratio appears to drop does the system issue a dictionary clear (i.e., setting *max_ent* to 256).

This clearing mechanism is very important to the performance of the algorithm. Its specific operation is based on a parameter called the *check_gap*. This value determines the minimum number of input characters that pass between checks of the local compression ratio. The word minimum is used because checks can only occur at the end of a datagram since clears are only allowed between datagrams. Further, checks only occur when the dictionary is full. If the dictionary is not full, the check will be bypassed. A large *check_gap* reduces the number of checks allowing small transient variations in compression to go unnoticed. Smaller values of *check_gap* allow more frequent checks of the ratio and possibly more frequent clears.

Under error–free conditions, the *check_gap* size can significantly effect the operation of the compressor. Figure 7.4 shows the average throughput of the two compressible files using differing values of *check_gap* and an MTU of 296. The *exec* file is a concatenation of 3

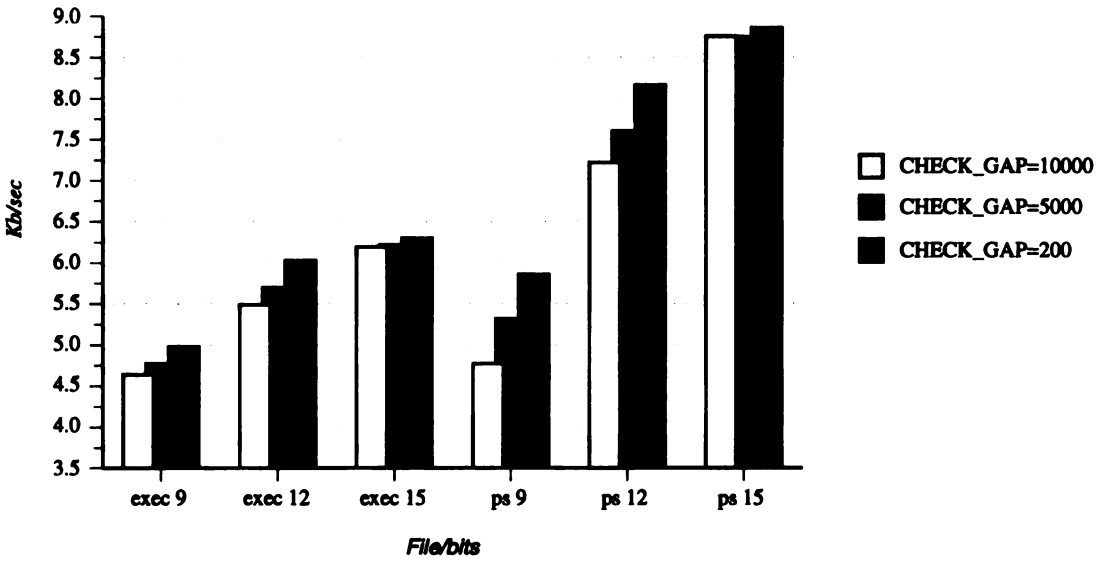

Figure 7.4: Compression With Varying Values of *check_gap*

common UNIX executable files while the *ps* file is a concatenation of 3 postscript files that represent technical documents. Each file was transmitted using different values of n, the number of bits used to represent an index. A larger n leads to a larger dictionary and better compression. For these specific files, a smaller *check_gap* value results in better compression. Having too large a value of *check_gap* means that the dictionary cannot clear often enough to efficiently track the changing nature of the input data. This concept will be revisited later.

7.5 Packet Loss and Synchronization

State maintenance is a critical operation for a compression implementation. Proper decoding requires that the compressor and decompressor maintain identical copies of the dictionary. Since the decompressor updates its dictionary based on received input, any data loss will destroy the continuity between the two endpoints. While data loss is typically not

a problem in file–based systems, the unreliable medium of a communication network forces the implementation to add mechanisms to ensure both sides remain synchronized.

This implementation uses a combination of stop–and–wait ARQ combined with sequence numbers. The PPP framing mechanism includes a checksum that will detect bit errors. Upon error detection, packets, which in this case may be compressed framed IP datagrams, will be dropped. The data compressor will not know that an error has occurred until it receives the next numbered packet. Care must be taken here to note that not all packets are numbered. As explained in section 7.2.4, uncompressed packets may be sent using an unnumbered mode (no sequence number is included in the packet framing). So, it is quite possible that several packets will arrive before the decompression engine realizes that synchronization has been lost.

Upon the detection of synchronization loss, the decompression engine will set a flag that causes all incoming packets to be dropped. It then issues a reset request (RESETREQ) to the peer. Such a request, when received by the compressor, causes the compressor to clear its dictionary to an empty state, flush any queued but already compressed datagrams, and issue a reset acknowledgment (RESETACK). When the RESETACK is received back by the decompressor, the decompressor resets the drop packet flag, clears its dictionary and begins operation again. Operation of the RESETREQ/RESETACK control channel is relegated to a user level program that provides reliable delivery through a retransmission mechanism. The important things to note about this round trip reset are that all packets get dropped between the detection of a lost packet and reception of the final RESETACK. This may include some unnumbered packets that slip by before an error is detected on the next numbered packet. These unnumbered packets get dropped by TCP when their checksums fail. Further, the result of the RESETREQ/RESETACK is that both sides completely

clear their dictionary and start over with an empty copy. As mentioned in section 7.2.3, this probably means that the next few packets will be sent uncompressed.

7.5.1 Performance Analysis in the Presence of Noise

One common trait of the wireless medium is noise. Data corruption and the subsequent packet losses can significantly affect throughput numbers. Techniques such as forward error correction (FEC) and TCP retransmission modifications [1, 4, 6, 50] have been proposed in an attempt to reduce the effects of data errors. FEC is useful for removing errors while TCP modifications help cope with the required retransmissions when unrecoverable errors do occur. Unfortunately, data compression adds a new variable to the picture. Namely, the compression dictionary synchronization issue. Loss of synchronization will lead to the discarding of a large number of packets that arrive over the medium with no bit errors. While these packets arrive intact, they cannot be decoded properly because of dependencies on the compression dictionaries. This subsection investigates the extent of this dependency.

7.5.2 Throughput Figures

Figures 7.5 and 7.6 show the performance of LZW compression in a noisy environment using an MTU of 296 bytes, no VJ compression, and the standard LZW algorithm. For small dictionary sizes, it is apparent that performance is worse than when compression is not enabled (0 bits). The only gain comes when the dictionary is large enough to provide for significant compression. For these gains, the benefit of compression outweighs the penalty of synchronization loss and discarded packets. The compression benefit is composed of two parts. The first is due to the reduced number of bytes that must cross the low–bandwidth link. The second is due to the reduced probability that a small packet will be hit by noise

as compared to its large uncompressed counterpart. None of the runs exhibit the large increases displayed in the noiseless environment (Figure 7.3).

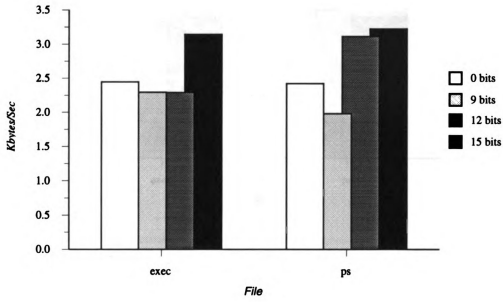

Figure 7.5: LZW Compression With Random Noise

It is well known that standard TCP is not suited for a noisy environment[1]. In situations of packet loss, TCP will assume congestion as the cause and will apply throttle–back and slow–start mechanisms. While these definitely affect the throughput numbers displayed, a more telling figure of the dependencies related specifically to LZW compression are the number of packets discarded and the number of retransmissions required to complete the transfers. Table 7.1 shows the average values for the graphs displayed. Each value represents the average counts for transferring one copy the *exec* file plus one copy of the *ps* file. *Pkt* represents the number of IP packets successfully handed to the higher layers of the stack, *Err* the number of errors encountered, *Toss* the number of packets discarded in addition to the error packet because endpoints are out of synchronization, *Retrans* the number of TCP

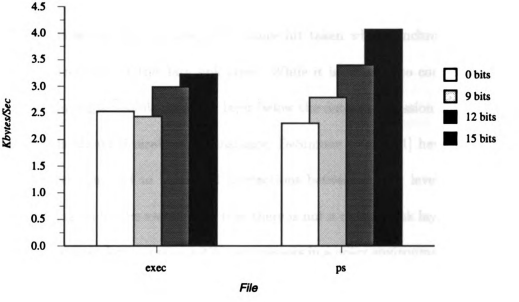

Figure 7.6: LZW Compression With Burst Noise

retransmissions, and *Ratio*, the ratio of received compressed bytes to uncompressed bytes. As can be seen, the dependencies induced by the use of LZW compression cause a dramatic increase in the number of retransmissions.

Table 7.1: Error Statistics for Standard LZW Scheme

Error Type–Bits	Pkt	Err	Toss	Retrans	Ratio
Random–0	3408	121	0	492	1.0
Random–9	3951	111	304	1680	0.82
Random–12	3906	83	278	1440	0.60
Random–15	3891	75	278	1407	0.56
Burst–0	3474	78	0	197	1.0
Burst–9	3811	71	205	421	0.82
Burst–12	3760	64	176	354	0.60
Burst–15	3739	52	168	309	0.53

7.6 The New LZW Algorithm

The previous section showed the performance hit taken when synchronization dependencies were introduced at the data link layer. While it is possible to completely remove errors by introducing a reliable data link layer below the data compression engine, it is not clear that this is always desirable. For instance, DeSimone et al. [11] have shown that a reliable link layer can lead to unwanted interactions between higher level retransmission schemes. Working under the assumption that there is not a reliable link layer, this work investigates mechanisms for operating LZW compressors in a noisy environment. This section will introduce techniques that reduce the cost of packet errors.

Close analysis of the compression behavior reveals undesirable characteristics of the current mechanisms. The first is that a RESETREQ/RESETACK results in both sides completely clearing their dictionaries. Since compressors do not perform well until they have lengthy strings against which to compress, the compression efficiency will temporarily degrade until dictionaries become partially full. The second is that once an error occurs, all compressed packets must be discarded until the RESETREQ/RESETACK has successfully restored the synchronization between the endpoints. All of these discarded packets must eventually be retransmitted by the higher transport layer.

To address these problems, a new model of transmission is adopted. This model, from here on referred to as a *generational model*, uses a series of dictionary checkpoints and the already present reliable RESETREQ/RESETACK round trip communication channel to remove many of the synchronization dependencies that exist in the current implementation. Figure 7.7 describes the structure of this model. Individual compressed packets are identified by a tuple consisting of the generation number and count within the generation. The generation number refers to a dictionary checkpoint (described in the next section) that

is guaranteed valid on both ends. The packet count is used to detect lost packets and to initiate generation changes at the appropriate points.

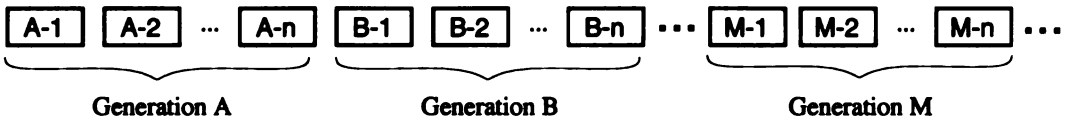

Figure 7.7: The Generational Model

7.6.1 Dictionary Checkpoints

Section 7.4 described the implementation of the LZW dictionary as a dual use array. A key component of this array is the value of *max_ent*. This variable holds the index into the dictionary where the next code will be inserted and is the growth point of the dictionary. As each code is inserted, *max_ent* is incremented. Clears have the effect of resetting *max_ent* to its initial value.

The generational model introduces the notion of a dictionary checkpoint. A checkpoint is a value of *max_ent* that is guaranteed to correctly exist on both sides. If both sides agree on *max_ent* then it is also the case that they agree on all values in the dictionary that have indices less than *max_ent* (i.e., they are synchronized for codes that have indices less than or equal to *max_ent*).

Using the notion of checkpoints, it is possible to remove some of the interpacket dependencies in the compressor algorithm. The key is to modify the compression algorithm so that it assigns a unique checkpoint to each generation. As packets from generation A are run through the dictionary, the compressor first grows the dictionary as it normally would when packets get compressed. However, the portion of the algorithm that emits the dictionary indices into the "compressed" packet is limited to emitting values that are less than

or equal to the value of the checkpoint for generation A. In essence, the maximum size of the dictionary is artificially limited to this checkpoint, guaranteeing that every code in the compressed packet has an index less than or equal to the checkpoint. This means that upon decompression, every code is guaranteed to exist in the decompression dictionary. Further, since all packets in generation A use the same checkpoint, a packet loss inside generation A does not render the decompressor unable to continue decompression within that generation.

Generational changes are a critical point during operation. Packets within a generation are independent from the decompressor point of view. Once n packets have been sent, the compressor will bump the generation count. This process entails two important steps. The first is that the checkpoint *max_ent* is changed to the real value of *max_ent*. This new *max_ent* checkpoint will either incorporates all of the new strings that were added from the last generation, or it will be reset to its initial position (a clear) if the local compression ratio is deemed poor. The new setting will be used for the next generation and should allow better compression since the dictionary has hopefully changed in a positive manner. The second thing that happens is that the sequence number is modified to reflect that it belongs to the next generation. This is an important step since this change signals the decompressor that it should bump its checkpoint to agree with the compressor.

7.6.2 Packet Errors and RESETREQ/RESETACK Behavior

Remember that even though packets are output with artificially limited indices, the compressor is still growing the real value of *max_ent*. As stated previously, a generational change means that both sides start to use a larger value of *max_ent* and thus gain the added benefit of longer strings in the dictionary. However, there is still the required maintenance of guaranteeing that both sides are consistent up to the next checkpoint. This maintenance

is performed through the RESETREQ/RESETACK mechanism.

If the decompressor determines that a packet loss has occurred, then the state of the decompression dictionary, somewhere past the current generational checkpoint, will be out of synchronization with the compression dictionary. Since the compressor is not susceptible to packet loss, then the decompressor can assume that its dictionary is in sync with the compressor up to the value of *max_ent* that was generated by the decompression of the last packet that successfully arrived in order. Using this knowledge, the decompressor can use the reliable RESETREQ/RESETACK channel to notify the compressor of its state prior to the detected loss. It does this by appending a value of the last known good *max_ent* and packet sequence number to the RESETREQ. The compressor, upon receiving these values, has reliable information about the decompressor's state and can make a decision about how to proceed. If it needs to perform a clear (or has detected it performed a clear in a generation gap that is already in the pipeline) it will reset its dictionary. If not, it will set its checkpoint to the value sent by the decompressor[2]. Either way, the compressor sends back a value of *max_ent* and a new sequence number that represents the start of a new generation. The decompressor will use these new values unconditionally. This allows the compressor to make the final decision on the new state.

7.7 Performance of the New LZW Algorithm

This section looks at the performance of the new algorithm both in a noiseless environment (Figure 7.8) and in noisy environments (Figures 7.9 and 7.10). Only LZW compression using a fifteen bit dictionary index length is addressed. This is the most memory intensive

[2]This has the added benefit of jumping the checkpoint further into the dictionary before a generational gap has arrived.

case and provides the best noiseless compression.

The error free case is shown for various generation lengths. One may recall that the compressor and decompressor are only allowed to clear their dictionaries between generations. This means that longer generations allow fewer clears. Part of the shown performance differences are due to the lack of clears and are analogous to the *check_gap* graphs as previously described in Section 7.4. Longer generation counts have the same effect as larger *check_gap* values. Longer generation counts can also have a negative effect on compression ratios. Since the dictionary only grows at generational gaps, more packets are compressed with an old (smaller) dictionary. Each of these packets does not benefit from the dictionary growth generated by the previous packets in the current generation.

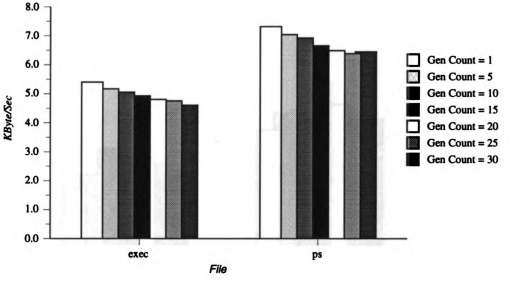

Figure 7.8: Various Generation Gaps Under Error Free Transmission

Noise adds a new factor in the determination of performance. Since packets within a generation are independent, then larger generations provide for fewer discarded packets. Recall that packets are only discarded when a RESETREQ/RESETACK is pending and

packets start to arrive from the next generation. Larger generation counts mean a higher probability that the RESETREQ/RESETACK will complete within the same generation. This must be balanced with the negative effects that large generation counts have on compression. Figures 7.9 and 7.10 exhibit a curve that shows performance increasing as the generation gap increases enough to balance the discarded packets. As the generation count grows too high, the poor compression outweighs the benefits of fewer discarded packets and performance drops off again. Table 7.2 shows this tradeoff. The ratio column clearly shows the compression dropoff while the retransmission column shows the reduction in retransmissions. Compared with Table 7.1, one can see significantly fewer discarded packets over the standard LZW scheme.

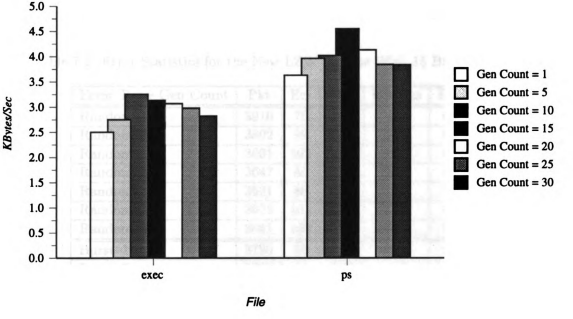

Figure 7.9: New Algorithm With Random Error

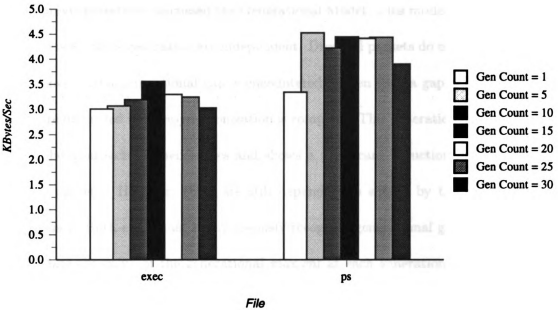

Figure 7.10: New Algorithm With Burst Error

Table 7.2: Error Statistics for the New LZW Scheme With 15 Bit Dictionary

Error Type–Gen Count	Pkt	Err	Toss	Retrans	Ratio
Random–1	3910	76	280	449	0.45
Random–5	3802	99	220	388	0.43
Random–10	3691	105	142	297	0.42
Random–15	3647	86	95	232	0.42
Random–20	3621	86	72	220	0.40
Random–25	3623	101	76	223	0.37
Random–30	3641	100	66	209	0.36
Burst–1	3790	58	211	351	0.46
Burst–5	3719	66	155	311	0.44
Burst–10	3691	73	132	277	0.43
Burst–15	3626	66	71	222	0.42
Burst–20	3635	69	68	223	0.41
Burst–25	3623	65	62	212	0.40
Burst–30	3634	76	73	256	0.35

7.8 Receiver Controlled Synchronization

The previous sections discussed the Generational Model. This model has the advantage that packets within a generation are independent. Dropped packets do not cause a synchronization loss until a generational gap is encountered. When such a gap is encountered, all packets are discarded until resynchronization is complete. This generational model removes some of the interpacket dependencies and shows a significant reduction in the number of discarded packets. However, there are still dependencies shared by the compressor and decompressor. Both ends must simultaneously recognize generational gaps and simultaneously update the value of the generational *max_ent* at each generational gap. Since only the receiver knows about packet losses, the system must endure large round latencies for a RESETREQ/RESETACK. The round trip latencies cause packets to cross generational gaps during error conditions and thus force the receiver into a packet discard mode. To remedy this situation a second modification was performed. This modification makes heavy use of the round trip control channel and gives the receiver complete control over when to perform a generational update. With this new method, the compressor will never perform a generational update unless requested by the receiver.

In the error free conditions, this second LZW algorithm is similar to the previous generational model. Operation of the compressor is essentially unchanged. It still compresses all packets with the initial generational *max_ent* and keeps a separate copy of the real *max_ent*. However, the compressor will ignore the packet count and never assume that a generational gap has been encountered. The decompressor also behaves in essentially the same manner, but unlike the compressor, it keeps track of the generational gaps as it normally would. After a fixed number of packets, the decompressor will signal the compressor through the control channel. The signal, called a GENREQ/GENACK, is similar

to a RESETREQ/RESETACK but does not indicate that synchronization has been lost. Such a message has the effect of telling the compressor that it may bump its generational *max_ent*. However, unlike the RESETREQ/RESETACK, the value of the real *max_ent* is left unchanged. So in essence, the GENREQ/GENACK notifies the compressor to perform a generational update.

The signalling channel is used in a stop–and–wait mode. At any given time, only one GENREQ/GENACK or RESETREQ/RESETACK can be outstanding. Since this control channel is reliable, there is no need to worry about losing such a request due to noise. If a request is outstanding and another request cómes through, it is held until the outstanding request is satisfied. Since both the RESETREQ/RESETACK and GENREQ/GENACK must share the channel, a priority scheme is used for arbitration. Outstanding RESETREQ/RESETACK packets will always override GENREQ/GENACK since the operations performed by a RESETREQ/RESETACK include those performed by a GENREQ/GENACK.

Use of the modified algorithm almost guarantees that the endpoints will never become unsynchronized and simplifies synchronization. The only case in which the endpoints may lose synchronization is if a GENACK (a very small packet) gets corrupted by noise and dropped. In this case, the compressor has already updated its value of the generational *max_ent* but the decompressor will not receive the GENACK and so not make the update. Any following packets on the link will then get dropped because they crossed a generational boundary without a decompressor update. The decompressor can detect when this situation arises since it will receive packets in a new generation. On occurrence, it will active the standard RESETREQ/RESETACK to resynchronize the dictionaries.

Table 7.3 shows the results of the same transfers using the new algorithm. As the

Table 7.3: Error Statistics for the Receiver Directed LZW Scheme With 15 Bit Dictionary

Error Type–Gen Count	Pkt	Err	Toss	Retrans	Ratio
Random–1	4761	77	20	173	0.41
Random–5	4438	71	14	132	0.41
Random–10	4128	74	16	170	0.41
Random–15	3927	77	8	175	0.41
Random–20	3842	78	9	167	0.40
Random–25	3785	78	7	176	0.39
Random–30	3770	73	4	189	0.40
Random–35	3724	75	3	165	0.40
Random–40	3710	70	0	166	0.40

generation size grows, the chance of losing a GENACK decreases. One can note that in the final stages there are almost no discarded packets and at forty, our results show zero extra discarded packets.

Just as in the first modification, the receiver directed approach does have some drawbacks. Since the system uses a stop–and–wait approach to generational updates, the round trip time of the GENREQ/GENACK limits the speed at which these updates may occur. This can lead to poor tracking of dictionary clears (analogous to having too large of a value for the *CHECK_GAP*). This directly affects the compression ratios given by the algorithms.

7.9 TCP Failure

With the severe reductions in discarded packets, one may ask why there are not better throughput gains. This is an excellent question that prompted further investigation. It turns out that even with the reduced number of dropped packets and reduced retransmissions, the compression engine is still waiting on the TCP implementation. The current TCP stack is unable to keep the data pipe full with packets. Figures 7.11 and 7.12 show the increase in TCP sequence numbers versus time for the best case performance of the first modification

of the LZW algorithm (using fifteen bits of compression and a generation count of fifteen). Two arbitrary sections show significant time where the sequence numbers increment slowly or not at all. Because it is know that there are relatively few retransmissions, it is asserted that these slow increments are not due entirely to retransmissions. Instead, they are due to the congestion avoidance mechanisms of TCP. Indeed, these graphs are reminiscent of those produced by Cáceres et al. [6] during their investigations in improving TCP performance in mobile computing environments. Since TCP is above the level at which this compression is implemented, any of the previously mentioned TCP optimizations (see Section 2) can co–exist with this work. It is asserted that those mechanisms will be able to adjust the TCP parameters in such a way that TCP will then be able to keep the pipe full. Under such circumstances, the reduction in the number of retransmissions should increase the performance of the link.

7.10 Summary

This chapter was devoted to showing how state maintenance in a LZW compression implementation can be modified to better tolerate errors. Unlike the VJ protocol, the LZW protocol maintains a reliable duplex signalling channel that can be used to convey control information. This channel gives more latitude in making control decisions. The original LZW system was shown to behave poorly on noisy channels. Upon detection of a corrupted packet, it would drop excessive numbers of packets until the endpoints regained synchronization. A new protocol was developed that allowed the system to better tolerate errors. The new protocol can trade off compression efficiency for probability of synchronization loss. It dramatically reduces the number of discarded packets. Because of standard TCP mechanisms beyond the control of this testbed, throughput numbers did

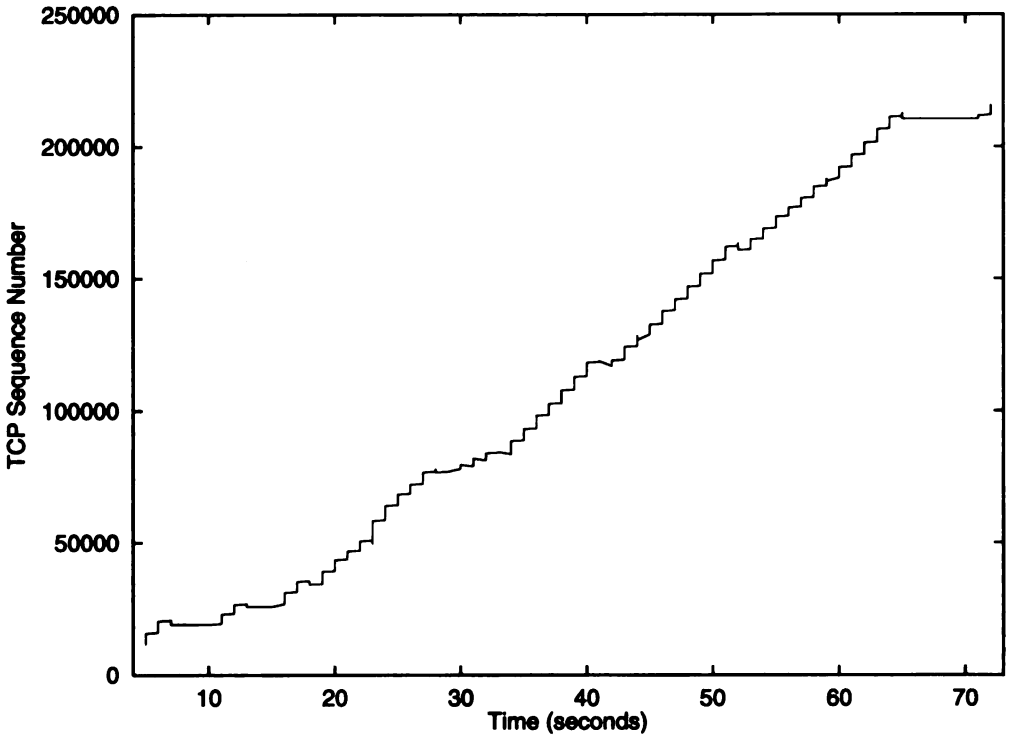

Figure 7.11: TCP Sequence Numbers Over Time

not reflect the gains that should be present from the reduced number of discarded packets. However, it was shown that the TCP implementation was at fault since it was unable to keep the link active. Other investigators have addressed the TCP problem and their solutions should solve that aspect of the operation.

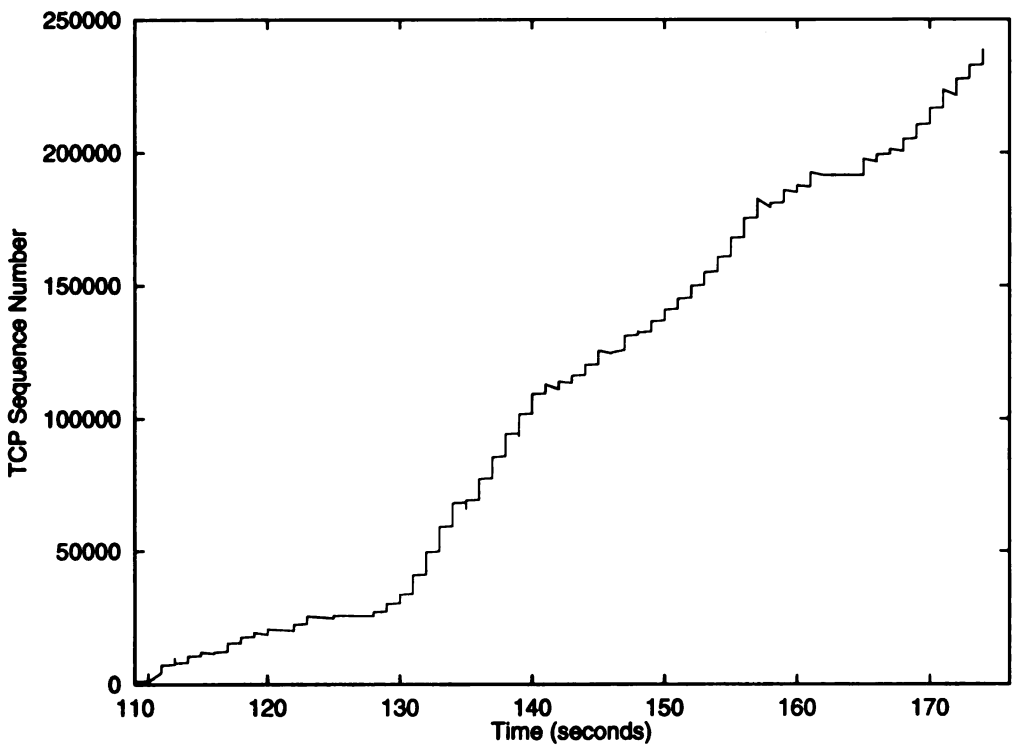

Figure 7.12: TCP Sequence Numbers Over Time

Chapter 8

Conclusion

8.1 Document Review

This dissertation investigates two noise tolerant compression protocols. The work starts with the basic assumptions that wide–area wireless networks are both bandwidth constrained and noise limited. From these assumptions it was surmised that the behavior of low–bandwidth networks users would produce traffic patterns different from those presented in previous works. Chapter 4 described a mechanism that effectively captured session statistics in a distributed fashion. The collected dumps were first compared to traffic patterns described in high speed traffic surveys. They were then further analyzed to provide individual user level statistics that were not found in previous works. It was observed that bandwidth constraints impose restrictions that cause traffic to vary from that provided in previous works. However, it was also noted that most of the application level traffic consists of the *expected* mixes of mail, news, login, and web.

The rationale for the traffic study was to justify the traffic types that would be used as input to packet based compressors. It was found that the most common low–bandwidth

application level protocols have the potential to compress very well. To understand just how these protocols compress, Chapter 5 analyzed the header and payload compression characteristics of the top protocols found in the traffic analysis. Besides giving a comparative base on the effectiveness of these types of compression, it showed that variations in packet payload compressibility lead to a flattening of the standard bi–modal packet size distribution. Any shift away from a small number of large sized packets has two effects that were of interest to this work. First, increasing the number of smaller packets increases the protocol overhead that exists on a channel. Additional protocol overhead cuts into the usable capacity of an already limited link. Second, more packets mean that dependent noise has the potential to disrupt a potentially large number of independent multiplexed streams. This causes synchronization problems with packet level compressors.

In Chapter 6, it was shown that Van Jacobson (VJ) TCP header compression can be especially good at removing TCP protocol overhead in a noiseless environment. Unfortunately, it was also shown that when moved to the noisy wireless environment, the performance of this compressor dropped to below that of not using header compression. The contribution of Chapter 6 is a description of modifications to the VJ communications protocol that allow it to better tolerate errors. Drawing on both the size distributions presented in Chapter 4 and the compression performance characteristics described in Chapter 5, several enhancements were made to the standard VJ algorithm. These enhancements allowed the use of VJ compression on interactive traffic while maintaining the same performance as links that avoid VJ compression. Further enhancements removed many of the dependencies that were implicitly transmitted on the link. This resulted in a new noise tolerant protocol that is nearly as effective as the standard protocol on noiseless links and can significantly outperform the standard on random or burst error channels.

With the success of the noise tolerant VJ protocol, Chapter 7 attempted to apply the same concepts to a LZW payload compressor. Unfortunately, the architecture of LZW communications channel was significantly different from that of the VJ channel. This meant that the VJ mechanisms could not be used directly.

The LZW bi–directional communications protocol allowed greater leeway in operation and provided a more sophisticated mechanism for detecting synchronization loss. Noise tolerance was approached by adding a generational update structure on top of the protocol. This structure helped to remove the interpacket dependencies implicit on the link. By modifying the size of the generation, one was able to tradeoff compression efficiency against the probability of synchronization loss. Such a scheme results in dramatic reductions in packet loss on the link. To improve the scheme further, the protocol was again modified to place all generational control in the decompressor. This removed almost all implicit information stored on the link and reduced the shared state that was present in the original algorithm. This modification resulted in even further reductions in packet loss. In some configurations, no additional packets were discarded when a packet was dropped due to noise.

The unfortunate reality of the LZW modifications is that throughput gains were not readily apparent. Even though the noise tolerant aspects of the protocol were successful (as shown by the dramatic reduction in discarded packets), throughput figures did not significantly increase. After specific investigations as to the cause, it was shown that the TCP implementation was unable to properly handle the noise present on the link. Even with almost no added packet discards, TCP was unable to keep the link filled with data packets. Frequent pauses, some lasting several seconds, reduced the efficiency of the transfer. The good news is that this is not an unknown phenomenon. Several independent works have

identified this problem and have suggested techniques for modifying TCP so that it will perform better on error prone links. It is asserted that these TCP modifications, which are totally independent from this work, can correct the TCP deficiency and would allow the noise tolerant LZW scheme to provide better throughput than its original counterpart.

8.2 Future Work

There are several directions that may be pursued with this work. This section outlines areas that merit further investigation.

8.2.1 Further LZW Analysis

The most obvious extension would be to port the noise tolerant algorithms over to a system that allowed modification of TCP. Then, one could combine the enhanced TCP algorithms with the noise tolerant LZW scheme to determine the exact throughput increases that could be generated.

A second direction for these algorithms would be to apply the *slow-start* algorithm suggested by Degermark et al. ([9] and as described in Chapter 2 on page 26) to the generational gap size in the receiver controlled LZW scheme. This mechanism would start out by sending generational updates frequently. This would allow both sides to quickly build up a their generational dictionaries at the expense of operating under a higher probability of synchronization loss. Over time, the algorithm then slowly increases generational gap to reduce the probability of synchronization loss. One would hope that this would help balance the tradeoff between generation size and compression efficiency.

8.2.2 Multiple Compressor Comparison

Chapter 5 investigated the compressibilty of those protocols that are highly used on low–bandwidth links. It would be interesting to implement some combination of the LZS-DCP, Predictor, Gandalf FZA, Stac LZS, and Microsoft per–packet algorithms and compare them using the same packet based graphing technique. Such a comparison could lead to hybrid compressors that outperform any one algorithm. A dynamic selection of algorithm combinations could also be investigated. Such a selector could tailor the algorithm choice based on parameters such as the signal–to–noise ratio of the channel, the past history of data flow (i.e. the transition information found in Chapter 5), and on packet size estimates.

8.2.3 CPU Tradeoffs and Alternate Implementations

VJ and LZW compression tradeoff memory and CPU processing to reduce transmission requirements. At some point there is a data rate at which one breaks even on the tradeoff. This break even point is a moving target based on the ever changing ratio of CPU cycles to link speed. With the success of programmable DSP based communications solutions, it would be interesting to investigate high–speed DSP based implementations of these algorithms to determine if they could be successfully applied on systems such as cable modems and xDSL. Such systems usually have highly asymmetric bandwidth constraints that could lead to further algorithmic enhancements. For instance, IEEE 802.14 based cable modems will contend for a two megabit per second upstream channel and have access to a forty megabit per second downstream. Upstream communication is subject to very high latencies while downstream communication is totally controlled by the headend. Providing the headend had enough computational power to provide compression, it is possible that an implementation of the compression protocols could take advantage of the asymmetric

bandwidth to reduce the latency of the upstream. An asymmetric programmable compression solution that outperforms V.42bis would be quite valuable.

8.2.4 IPv6

IPv6 opens a new realm for exploration. As partially addressed by Degermark et al. [9, 10], protocol overhead will account for a larger percentage of the available bandwidth than does IPv4. One question that needs answering regards the synchronization requirements that link level encryption adds to the system. Such encryption could interfere with the operation of data compressors. As an example, it appears that providing header compression before encryption would hide important protocol information from the encryption scheme. Applying header compression after encryption would fail since most header information would be unintelligible. Further, since encryption probably requires its own synchronization, the issue of their interoperation must be addressed. Algorithms need to be developed that would allow each of the independent synchronization schemes to co–exist on a noisy channel.

Bibliography

Bibliography

[1] Elan Amir, Hari Balakrishnan, Srinivasan Seshan, and Randy H. Katz, "Efficient TCP over networks with wireless links," in *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995, pp. 35–40.

[2] Jørgen Bach Andersen, Theodore S. Rappaport, and Susumu Yoshida, "Propagation measurements and models for wireless communications channels," *IEEE Communications*, vol. 33, pp. 42–49, Jan 1995.

[3] B.R. Badrinath and T. Imielinski, "Replication and mobility," Technical report, Department of Computer Science, Rutgers University, 1992.

[4] Ajay Bakre and B.R. Badrinath, "I-TCP: Indirect TCP for mobile hosts," in *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 30 – June 2 1995, pp. 136–143.

[5] Ramón Cáceres, Peter B. Danzig, Sugih Jamin, and Danny J. Mitzel, "Characteristics of wide–area TCP/IP conversations," *Proceedings of SIGCOMM '91*, vol. 21, pp. 101–112, 1991.

[6] Ramón Cáceres and Liviu Iftode, "Improving the performance of reliable transport protocols in mobile computing environments," *IEEE Journal on Selected Areas in Communications*, vol. 13, pp. 850–857, June 1995.

[7] D. Carr, "PPP gandalf FZA compression protocol," RFC 1993, Network Working Group, Aug. 1996.

[8] Ojas T. Choksi, "Performance of data link layer in cellular radio environment," Master's thesis, Rutgers University, May 1994.

[9] Mikael Degermark, Mathias Engan, Björn Nordgren, and Stephen Pink, "Low–loss TCP/IP header compression for wireless networks," Technical report, CDT/Department of Computer Science, Lulea University, S–971 87 Lulea, Sweden, 1996.

[10] Mikael Degermark and Stephen Pink, "Soft state header compression for wireless networks," Technical report, CDT/Department of Computer Science, Lulea University, S–971 87 Lulea, Sweden, 1996.

[11] Antonio DeSimone, Mooi Choo Chuah, and On-Ching Yue, "Throughtput performance of transport–layer protocols over wireless LANs," in *Proceedings of IEEE Globecom*, volume 1, November 29 – December 2 1993, pp. 542–549.

[12] E.O. Elliott, "Estimates of error rates for codes on burst–noise channels," *The Bell System Technical Journal*, vol. XLII, pp. 1977–1997, sep 1963.

[13] David J. Farber, Gary S. Delp, and Thomas M. Conte, "A thinwire protocol for connecting personal computers to the internet," RFC 914, Network Working Group, Sept. 1984.

[14] N. Freed and A Cargille., "Smtp service extension for command pipelining," RFC 1854, Network Working Group, Oct. 1995.

[15] R. Friend and W. Simpson, "PPP Stac LZS compression protocol," RFC 1974, Network Working Group, Aug. 1996.

[16] Bruce D. Fritchman, "A binary channel characterization using partitioned markov chains," *IEEE Transactions on Information Theory*, vol. IT–13, pp. 221–227, Apr. 1967.

[17] E.N. Gilbert, "Capacity of a burst–noise channel," *The Bell System Technical Journal*, vol. XXXIX, pp. 1253–1265, Sep 1960.

[18] Riccardo Gusella, "A measurement study of diskless workstation traffic on an Ethernet," *IEEE Transactions on Communications*, vol. 38, pp. 1557–1567, Sept. 1990.

[19] M. Hata, "Empirical formulat for propagation loss in land mobile radio services," *IEEE Transactions on Vehicular Technology*, vol. VT–29, pp. 317–325, 1980.

[20] Gilbert Held, *Data Compression, Second Edition.*, chapter 2, pp. 20–121 John Wiley & Sons, 1987.

[21] T. Imielinski and B.R. Badrinath, "Mobile wireless computing: Solutions and challenges in data management," DataMan Project, WINLAB, Rutgers University, 1992.

[22] John Ioannidis, Dan Duchamp, and Gerald Maguire, Jr., "IP–based protocols for mobile internetworking," in *SIGCOMM 91*, Sept. 1991, pp. 235–245.

[23] V. Jacobson, "Compressing TCP/IP headers for Low-Speed Serial Links," RFC 1144, Network Working Group, Feb. 1990.

[24] Van Jacobson, "Congestion avoidance and control," in *Proceedings of ACM SIGCOMM '88*, Aug. 1988.

[25] Raj Jain and Shawn A. Routhier, "Packet trains – measurements and a new model for computer network traffic," *IEEE Journal on Selected Areas in Communications*, vol. SAC–4, pp. 986–995, 1986.

[26] Laveen N. Kanal and A.R.K. Sastry, "Models for channels with memory and their applications to error control," *Proceedings of the IEEE*, vol. 66, pp. 724–744, July 1978.

[27] Raymond J. Leopold and Ann Miller, "The IRIDIUM communications system," *IEEE Potentials*, vol. 12, pp. 6–9, Apr. 1993.

[28] Allen H. Levesque and Kaveh Pahlavan, "Wireless data," in Jerry D. Gibson, editor, *The Mobile Communications Handbook*, chapter 35, pp. 553–567. CRC Press in Cooperation with the IEEE Press, 1996.

[29] Robert W. Lucky, *Silicon Dreams: Information, man, and machine.*, chapter 3,4, pp. 37–145 St. Martin's Press, New York, 1989.

[30] Paul Mackerras, "PPP–2.3," Available via anonymous FTP from cs.anu.edu.au, May 1996.

[31] S. Mathur and M. Lewis, "Compressing IPX headers over WAN media (CIPX)," RFC 1553, Network Working Group, Dec. 1993.

[32] Steven McCanne and Van Jacobson, "The BSD packet filter: A new architecture for user–level packet capture," Proceedings of the 1993 Winter USENIX Conference, San Diego, CA., and available via anonymous FTP from ftp.ee.lbl.gov:/papers/bpf-usenix93.ps.Z, 1993.

[33] Janice Obuchowski, "Wireless communications and spectrum conservation: Sending a signal to conserve," *IEEE Communications Magazine*, vol. , pp. 26–29, Feb. 1991.

[34] Y. Okumura, E. Ohmori, T. Kawano, and K. Fakuda, "Field strength and its variability in VHF and UHF land mobile radio service," *Rev. elec. Communications Lab*, vol. 16, pp. 825–873, 1968.

[35] Vern Paxson, "Empirically–derived analytic models of wide–area TCP connections," *IEEE Transactions on Networking*, vol. 2, pp. 316–336, Aug. 1994.

[36] Vern Paxson, "Growth trends in wide–area TCP connections," *IEEE Network*, vol. 8, pp. 8–17, July/August 1994.

[37] D. Rand, "The PPP compression control protocol (CCP)," RFC 1962, Network Working Group, June 1996.

[38] D. Rand, "PPP Predictor compression protocol," RFC 1978, Network Working Group, Aug. 1996.

[39] Theodore S. Rappaport, Rias Muhamed, and Varun Kapoor, "Propagation models," in Jerry D. Gibson, editor, *The Mobile Communications Handbook*, chapter 22, pp. 355–369. CRC Press in Cooperation wit the IEEE Press, 1996.

[40] J. Reynolds and J. Postel, "ASSIGNED NUMBERS," RFC 1700, Network Working Group, Oct. 1994.

[41] Martin S. Roden, *Analog and Digital Communication Systems.* Englewood Cliffs, New Jersey, 07623: Prentice Hall, 1991.

[42] Claude E. Shannon, "A mathematical theory of communication," *Bell Systems Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.

[43] W. Simpson, "The point–to–point protocol (PPP)," RFC 1661, Network Working Group, Dec. 1993.

[44] Francis Swarts and Hendrik C. Ferreira, "Markov characterization of digital fading mobile VHF channels," *IEEE Transactions on Vehicular Technology*, vol. 43, pp. 977–985, Nov. 1994.

[45] S. Tsai, "Markov characterization of the H.F. channel"," *IEEE Transactions on Communication Technology*, vol. COM-17, pp. 24–32, 1969.

[46] Van Jacobson, C. Leres, and S. McCanne, "tcpdump," Available via anonymous FTP from ftp.ee.lbl.gov, June 1989.

[47] T. A. Welch, "A technique for high–performance data compression," *IEEE Computer*, vol. 17, pp. 8–19, 1984.

[48] Ross N. Williams, *Adaptive Data Compression*. 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 1991.

[49] J. Woods, "PPP Deflate protocol," RFC 1979, Network Working Group, Aug. 1996.

[50] Raj Yavatkar and Namrata Bhagawat, "Improving end–to–end performance of TCP over mobile internetworks," in *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Dec. 1995, pp. 146–152.

[51] J. Ziv and A. Lempel, "Compression of individual sequences via variable–rate coding," *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, 1978.