Flexible Application Driven Network Striping over Wireless Wide Area Networks

by

Asfandyar Qureshi

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

March 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
March 21, 2005

Certified by . . . .
John V. Guttag
Professor, Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Flexible Application Driven Network Striping over Wireless Wide Area Networks

by

Asfandyar Qureshi

Submitted to the Department of Electrical Engineering and Computer Science
on March 21, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Inverse multiplexing, or network striping, allows the construction of a high-bandwidth virtual channel from a collection of multiple low-bandwidth network channels. Striping systems usually employ a packet scheduling policy that allows applications to be oblivious of the way in which packets are routed to specific network channels. Though this is appropriate for many applications, many other applications can benefit from an approach that explicitly involves the application in the determination of the striping policy.

Horde is middleware that facilitates flexible striping over Wireless Wide Area Network (WWAN) channels. Horde is unusual in that it separates the striping policy from the striping mechanism. It allows applications to describe network Quality-of-Service (QoS) objectives that the striping mechanism attempts to satisfy. Horde can be used by a set of data streams, each with its own QoS policy, to stripe data over a set of WWAN channels. The WWAN QoS variations observed across different channels and in time, provide opportunities to modulate stream QoS through scheduling.

The key technical challenge in Horde is giving applications control over certain aspects of the data striping operation while at the same time shielding the application from low-level details. Horde exports a set of flexible abstractions replacing the application's network stack. Horde allows applications to express their policy goals as succinct network-QoS *objectives*. Each objective says something, relatively simple, about the sort of network QoS an application would like for some data stream(s).

We present the Horde architecture, describe an early implementation, and examine how different policies can be used to modulate the quality-of-service observed across different independent data streams. Through experiments conducted on real and simulated network channels, we confirm our belief that the kind of QoS modulation Horde aims to achieve is realistic for actual applications.

Thesis Supervisor: John V. Guttag
Title: Professor, Computer Science and Engineering

For my parents,

who have given so much to their sons.

Acknowledgments

This thesis owes a profound debt to John Guttag. Despite many obstacles, John made sure that I wrote, rewrote and, finally, that I stopped rewriting this thesis. Without his suggestions and the insights I gained during our regular discussions, this would have looked very different. John has been an invaluable advisor ever since I was an undergraduate. He has supervised my research with care, provided direction when I was adrift, and yet allowed me an enormous amount of freedom.

Hari Balakrishnan was the first to suggest the use of WWAN striping. Muriel Medard was a great help in understanding WWAN channel technology. Dina Katabi helped me try to make sense of my WWAN packet traces. Frans Kaashoek occasionally let me shirk my TA duties to focus on thesis-related work. Karen Sollins was a remarkable undergraduate advisor. Michel Goraczko and Dorothy Curtis were always there to help. Allen Miu, Godfrey Tan, Vladimir Bychkovsky, Dave Andersen and Eugene Shih made important intellectual contributions to this work, even if neither of us could tell you the specifics. Magdalena Balazinska made sure not to disturb me, when I collapsed on our office couch in between conference papers and thesis chapters. Ali Shoeb's prolonged reflection, and eventual enlightenment gave me hope. Research is an inevitably collaborative endeavour and, just now, I'm probably forgetting someone. Sorry.

Most of all, I feel indebted to my family. Without their love, lessons, insights, inspiration, flaws, strength, support, sacrifice, protection, patience, passion, genes, successes, and failures—without them, none of this would have been possible.

Finally, the middleware described in this thesis owes its moniker to the hopelessly unhealthy Warcraft-addiction my brother and I once shared.

Contents

1 Introduction **1**

1.1 Motivating Application 5

1.2 Goals 6

1.3 WWAN Striping: Challenges 8

1.4 Thesis Contributions 11

1.5 Thesis Organization 12

2 Background **15**

2.1 System Design 15

2.2 Wireless WAN's 15

2.3 Network Striping 18

2.4 Video Encoding and Streaming 19

3 Horde Architecture **23**

3.1 Architecture Overview 25

3.1.1 Application View 26

3.1.2 Horde Internals 26

3.2 Application Interface 27

3.2.1 Horde Data Streams 27

3.2.2 Application Data units 28

3.2.3 Bandwidth Allocation 29

3.2.4 Application Policy 32

3.2.5 Reporting Network Latency 32

3.3 Network Channel Management . . . 33
3.3.1 Channel Manager Modules . . . 33
3.3.2 Channel Pool Manager . . . 34
3.3.3 Congestion Control . . . 34
3.3.4 Transmission Tokens . . . 35
3.4 Summary . . . 35

4 Transmission Tokens **37**
4.1 The `txSlot` Abstraction . . . 38
4.2 Scheduler Look-Ahead: *Phantom* Tokens . . . 40
4.3 Summary . . . 42

5 Network Scheduling **43**
5.1 Application Perspective: Derived Utility . . . 44
5.2 Scheduler: Policy and Mechanism . . . 47
5.3 'Optimal' Scheduling . . . 50
5.4 Simple Scheduling Approaches . . . 51
5.4.1 Round-Robin Scheduling . . . 51
5.4.2 Randomized Scheduling . . . 52
5.4.3 Callback Scheduling . . . 52
5.4.4 Channel Pinned Scheduling . . . 53
5.5 Summary . . . 53

6 Objective Driven Scheduling **55**
6.1 Objectives . . . 55
6.2 Objective Specification Language . . . 57
6.3 Objective Driven Scheduling . . . 61
6.4 Scheduler Implementation . . . 63
6.5 Summary . . . 66

7 Evaluation **69**
7.1 QoS Modulation: Real Channels (`E1`) . . . 70

7.2 QoS Modulation: Simulated Channels 75
7.2.1 Simulation Setup 75
7.2.2 Randomized-Round-Robin (S0) 77
7.2.3 Policy: Low Latency (S1) 78
7.2.4 Policy : Low Latency and Low Loss (S2 and S3) 80
7.2.5 Policy: Low Correlated Loss (S4) 84
7.3 Impact of Varying Scheduler Look-Ahead 88
7.4 Network Channel Managers: CDMA2000 1xRTT 90
7.4.1 Congestion Control: verb 91
7.4.2 Expected Latencies for txSlot's 94
7.5 Summary 95

8 Conclusions **99**
8.1 Goals and Contributions 99
8.2 Directions for Future Work 102

A WWAN Channel Characteristics **103**
A.1 Summary 106
A.2 Experimental Setup 107
A.3 CDMA2000 1xRTT Link Characterization 109
A.3.1 Background 109
A.3.2 Network Hops 110
A.3.3 Link Capacity Estimates 112
A.3.4 Upload Bandwidth 113
A.3.5 Packet Round-Trip-Times 115
A.3.6 Packet Size 120
A.3.7 Loss Characteristics 121
A.3.8 Disconnections 122
A.4 GPRS Link Characterization 124
A.4.1 GSM/GPRS Background 124
A.4.2 Network Hops 125

A.4.3 Upload Bandwidth 125
A.4.4 Packet Round-Trip-Times 126
A.4.5 Packet Size 130
A.4.6 Loss Characteristics 131
A.4.7 Disconnections 132
A.5 Multi-Channel Effects 133
A.5.1 Aggregate Bandwidth 134
A.5.2 Bandwidth Correlation 137

B HOSE: the Horde Objective Specification Environment 139
B.1 Augmented BNF Notation Overview 140
B.2 The HOSE Language 142
B.2.1 Defining Objectives 143
B.2.2 Objective: Context 143
B.2.3 Objective: Goal 145
B.2.4 Objective: Utility 145
B.2.5 Numbers, Booleans and Strings 146
B.2.6 ADU and Stream Variables 149
B.2.7 Probability Distributions 150
B.3 Sample Objectives 152
B.4 Complete HOSE Grammar 152

List of Figures

2-1 Characteristics of WWAN channels. The throughputs are estimated averages in kilobits-per-second. 802.11 is only listed as a reference. . . 16

3-1 A modular breakdown of the *Horde* internals. Solid lines represent the flow of data; dashed lines represent the direction of control signals. The video and audio applications are both using multiple data streams. 25

3-2 Pseudo-code for the bandwidth allocation process. 31

4-1 The components of the `txSlot` abstraction. 38

6-1 The objective `Y1`, expressing the policy that `txSlot`'s carrying I-frames from stream 17 should have lower loss probabilities than slots for other frame types. 58

6-2 The objective `Y2`, expressing the policy that the average latency on stream 7 should be less than one second. 59

6-3 The objective `Y3`, expressing the policy that the utility derived from the delivery of ADU's falls off linearly as the latency rises above one second. 59

6-4 Pseudo-code for the `random-walk` scheduler. We used a `WALK_LENGTH` of 25 and the random schedule constructor given in figure 6-5. The method `evaluate_objectives` iterates over every active objective, evaluating that objective for the given schedule and returns the sum of the utilities. 64

6-5 Pseudo-code for the random schedule constructor. 65

7-1 Throughput provided by Horde in the stationary experiment `E1`, using three co-located interfaces. 71

7-2 Packet round-trip-time distributions for each of the channels in the `E1` experiment. The three interfaces were co-located and stationary. . . . 72

7-3 Round-trip-time distributions with two different schedulers. The graphs show the median and the upper and lower quartiles for the round-trip-time experienced by ADU's in each stream. 74

7-4 An objective expressing the fact that the utility derived by an application from an ADU from stream 2 or 4 falls off, or rises, linearly depending on how close the expected latency of that ADU is to one second. 74

7-5 The characteristics of the simulated channels. 75

7-6 The Markov process used to model bursty packet losses on the simulated channels. q is the packet loss parameter and $s = \Sigma_{i=1}^{i \leq \lceil \log_2 (max\ burst\ size) \rceil} \left(0.2\right)^i$. 76

7-7 Experiment `S0` results. The `RRR` scheduler (a) stripes every stream in the same way over the available channels; (b) giving similar latency distributions. 78

7-8 The low-latency objective `X1`. Limited to streams `0` and `3`, this objective assigns `0` utility to an ADU with an expected round-trip-time greater than $700ms$; and linearly increasing utilities for ADU's with lower round-trip-times. 78

7-9 Round-trip-time distributions in experiment `S1`. The graph shows the median and the upper and lower quartiles for the stream ADU RTT's. 79

7-10 Channel usage in experiment `S1`. Compared to `S0`, slots from the lowest-latency channel are shifted to the latency sensitive streams. This is shown by a negative `cdma2` usage change for streams `1` and `2`, and a positive change for `0` and `3`. 79

7-11 Round-trip-time distributions in experiments `S1` and `S2`. 80

7-12 The low-loss objective X2. This objective specifies that if an ADU from stream 3 has a probability of loss less than 10%, its utility rises linearly with the probability that it is delivered. 80
7-13 Channel usage for streams in experiment S2. The loss and latency sensitive stream 3 is assigned slots on the low-loss medium-latency channel cdma1 in exchange for slots on the lowest-latency channel cdma2. All other streams gain slots on cdma2 and lose slots on cdma1. 81
7-14 Loss rates in experiment S2. 81
7-15 Comparing loss rates in experiments S1 and S2. 82
7-16 The low-latency objective X3. Identical to X1, except for the fact that this objective is restricted to stream 3. 82
7-17 Channel usage and latency distributions for streams in experiment S3. 83
7-18 Loss rates in experiment S3. 83
7-19 The low-correlated-loss objective X4. This objective adds positive utility whenever two ADU's from stream 0 have a correlated loss probability below 25%. 84
7-20 Stream loss rates and ADU round-trip-time distributions for S4. . . . 85
7-21 The fraction of stream packets lost in correlated losses in S3 and S4. . 86
7-22 Comparing losses on the streams in S3 with losses in S4. 86
7-23 Channel usage in the experiment S4. Compared to S3, almost 10% of the stream 0 ADU's are moved from the cdma2 channel and spread over the GPRS channels, in order to reduce correlated losses. 87
7-24 The results of varying look-ahead in the random-walk scheduler with a solitary low-latency objective. The graphs show—for various scheduling cycle periods—the mean ADU round-trip-time for the stream with the objective. The upper-bound of $580ms$ is the mean RTT with a randomized-round-robin scheduler. 89
7-25 Queuing on a CDMA2000 channel using a constant-bit-rate sender. . . 91
7-26 Comparing verb goodput to other congestion control schemes. This graph shows the distribution of a windowed-average for each scheme. 92

7-27 Packet losses and goodput in a stationary experiment using `verb` on an actual `CDMA2000 1xRTT` channel. 93

7-28 Measured packet losses and goodput in a mobile experiment using `verb`. 94

7-29 Comparison of how well an exponentially-weighted moving-average (the `srtt`) tracks the actual packet `rtt` on a `CDMA2000` channel, when the RTT's are (a) normal; and (b) elevated. In both cases, the `srtt` is often inaccurate by over $100ms$. 96

7-30 Conditioned RTT probability distributions derived from the stationary `verb` experiment traces. These graphs show the distributions of the conditional probabilities $p_1(x \mid z) = P(rtt_{i+1} = x \mid rtt_i = z)$, where z in each graph is given by the $rtt(now)$ line. These graphs demonstrate that the current RTT can provide significant information regarding what the next RTT will look like. 97

A-1 Observed mean TCP throughputs for bulk data transfers using different WWAN interfaces. Each set shows the throughputs obtained for three independent bulk data transfers for: downloading data from an MIT host over the `CDMA2000` interface; uploading to MIT from the `CDMA2000` interface; downloading from MIT over the `GPRS` interface; and uploading data using `GPRS`. The `GPRS` interface is significantly asymmetric and the `CDMA2000` interface provides much more bandwidth than `GPRS`. 105

A-2 The Boston-West safe-ride route. 108

A-3 `traceroute` from the `CDMA2000` interface to `W20-576-26.MIT.EDU`. . . 111

A-4 `traceroute` from `W20-575-26.MIT.EDU` to the `CDMA2000` interface. . . 111

A-5 Median packet round-trip-times for the traceroute to the `CDMA2000` interface from `W20-575-26.MIT.EDU`. The last IP hop's latency dominates. 112

A-6 Extracts from the output produced by `pathrate` using the `CDMA2000` interface as the sender and an MIT host as the receiver. 113

A-7 Measured raw UDP upload bandwidth on a CDMA2000 interface when (a) stationary ($\mu = 129.59, \sigma = 4.57$); and (b) moving ($\mu = 119.45, \sigma = 21.49$). 114

A-8 Observed distributions of single packet round-trip-times for small packets on a CDMA2000 link when (a) the transmitter is stationary; (b) when the transmitter is moving; (c) when the transmitter is stationary, but the packets; are TCP-SYN packets instead of ICMP packets; and (d) The distribution of $\Delta_{rtt} = (rtt_{i+1} - rtt_i)$ for all ICMP round-trip-times from (a) and (b). 116

A-9 Dynamic behaviour of single packet round-trip-times for small ICMP packets on a CDMA2000 link when the transmitter is: (a) stationary and (b) moving. 117

A-10 Observed dynamic behaviour of round-trip-times when UDP packet trains are sent using a high rate packet generator, under varying network conditions: (a) normal network conditions; (b) abnormal network conditions can triple times. 118

A-11 Observed distribution of round-trip-times for packets generated at a high constant rates, with various packet sizes, over the CDMA2000 link: (a) distribution of round-trip-times; (b) distribution of inter-arrival times. 119

A-12 Observed effects of changing UDP packet sizes on the performance of the CDMA2000 interface: (a) other than for small packets, packet size does not significantly impact achievable throughput; but (b) as packet size is increased beyond a certain point, the round-trip-time increases with packet size. 121

A-13 Observed distribution of packet losses from a number of different experiments with CBR senders using a CDMA2000 interface. For each packet loss, (a) the histogram tracks how many consecutive packets surrounding that lost packet were also lost; and (b) shows the cumulative density of this loss distribution. 122

A-14 Packet loss observations from a single experiment using the CDMA2000 interface with 1400 byte packets and transmitting at a constant rate of around $120kbits/sec$: (a) shows the loss timeline, each thin vertical line representing a single packet loss; (b) shows the packet burst-loss length distribution. 123
A-15 tcptraceroute from the GPRS interface to WEB.MIT.EDU. 126
A-16 Observed behaviour of raw UDP upload throughput on a GPRS interface, when (a) stationary ($\mu = 24.95$, $\sigma = 0.68$); and (b) moving ($\mu = 18.92$, $\sigma = 5.31$). 127
A-17 Observed distributions of single packet round-trip-times for TCP-SYN packets on a GPRS link when the transmitter is (a) stationary; and (b) moving. 128
A-18 Observed behaviour of single packet round-trip-times for TCP-SYN packets on a GPRS link when the transmitter is (a) stationary; and (b) moving. 129
A-19 Observed effects of changing packet sizes on the GPRS interface: (a) medium and small packet transmissions are highly inefficient on the link, but for packets larger than 768 bytes, packet size has no impact on goodput (b) the median round-trip-time for large packets is 1.5 times as large as the median round-trip-time for smaller packets. . . 130
A-20 Observed distribution of packet losses from many different experiments with constant-rate senders using a GPRS interface. For each packet loss, (a) the histogram tracks how many consecutive packets surrounding that lost packet were also lost; and (b) shows the cumulative density of this distribution. 132
A-21 Packet loss timelines for constant-rate senders using a GPRS interface in three different experiments. 133
A-22 Observed dynamic behaviour of available raw UDP upload bandwidth when multiple stationary interfaces are being used simultaneously: (a) aggregate upload bandwidth (b) individual channel bandwidths. . . 135

A-23 Observed dynamic behaviour of available raw UDP upload bandwidth when multiple interfaces are being simultaneously used on a moving vehicle: (a) aggregate upload bandwidth (b) individual channel bandwidths. 136

A-24 Observed raw UDP bandwidths, for both moving and stationary multi-interface experiments. Averages are only for as long as the channel was active; they do not count periods of disconnection as zero. 136

A-25 Channel bandwidth correlation coefficient matrices. Redundant values are omitted for clarity; the matrices are symmetric. 138

B-1 An objective to limit reordering. The `ADU` variables are bound to consecutive ADU's from stream 0. 145

Chapter 1

Introduction

This thesis describes *Horde*, middleware that facilitates flexible network striping over Wide Area Wireless Network (WWAN) channels. Inverse multiplexing, or network striping, allows the construction of a high-bandwidth virtual channel from a collection of multiple low-bandwidth network channels. Unlike earlier network striping middleware, bandwidth-aggregation is not the only service provided by Horde. Horde separates the striping policy from the mechanism. It allows applications to describe Quality-of-Service (QoS) based policy objectives that the striping mechanism attempts to satisfy. Horde can be used by a set of application data streams, each with its own QoS policy, to flexibly stripe data over a heterogeneous set of dynamically varying network channels (e.g., a set of WWAN channels). With such channels, different striping policies can provide very different network QoS to different application data streams. The key technical challenge in Horde is giving applications control over certain aspects of the data striping operation (e.g., an application may want urgent data to be sent over low latency channels or critical data over high reliability channels) while at the same time shielding the application from low-level details. This thesis discusses the design of Horde and evaluates the performance of a preliminary implementation on both real and simulated channels.

Our work on Horde was motivated by our inability to find an existing solution to support the development of a system on which we were working. As part of a telemedicine project, we wanted to transmit real-time uni-directional video, bi-

directional audio and physiological data streams (EKG, blood pressure, etc) from a moving ambulance using public carrier networks.

Our research leverages the widespread cellular wireless data networks. In most urban areas, there are a large number of public carrier wireless channels providing mobile connectivity to the Internet, using standard cellular technologies such as `GSM` and `CDMA2000`. Notably, these providers have overlapping coverage areas, allowing us to connect to more than one provider at the same time. Other researchers have also investigated the use of higher bandwidth wireless technologies, such as 802.11 [3, 16], for public networks with varying degrees of success.

Despite the 3G hype, individual cellular Wireless Wide Area Network (WWAN) channels do not provide enough bandwidth for our telemedicine application. The upstream bandwidth[1] offered by these channels is rather limited and usually less than the downstream bandwidth. Presently, the best WWAN channels available to us in the greater-Boston area provide no more than a hundred and forty kilobits per second of *peak* upstream bandwidth. In our experience, achievable average bandwidth is normally lower and varies significantly over time.

Even in the near future we do not expect public carrier data networks to improve enough to meet the upstream bandwidth requirement of our telemedicine application. Our application's bandwidth demand is fairly extreme for WWAN's: our goal is to be able to deliver video whose quality is as close to television broadcast quality as possible. With effective compression, this video quality can be achieved with a data rate of around one megabit-per-second[2]. At this time, no WWAN provider has an economic incentive to provision their network to accommodate this demand.

Furthermore, as carriers optimize their communications infrastructure, WWAN channels increasingly favour downstream bandwidth at the expense of upstream bandwidth. Since network carrier decisions are driven by their economic needs, asymmetric

[1]Here the term upstream bandwidth refers to how much data we can *send* and the term downstream bandwidth refers to how much data we can *receive*. Our telemedicine system is primarily concerned with sending patient data to the hospital.

[2]Since the video will be rendered on a relatively large screen at the hospital, we cannot assume that the destination is a small-screen. This assumption is used to reduce the bandwidth demand of video sent over cell-phones.

demand leads to asymmetric provisioning[3]. In the US, typical individual demand for upstream bandwidth is far lower than the levels we require for our video streaming application. Even with the introduction of picture and video messaging, it is unlikely that average individual upstream demand will soon grow to be on the same order of magnitude as the upstream demand of our application.

WWAN channels also provide little in the way of quality-of-service guarantees for data packets. WWAN channels are handicapped by low bandwidth, high and variable round trip times, occasional outages, considerable burstiness, and much instability when moving (see appendix A and [36, 14]). In our experiments we have observed that WWAN's with overlapping coverage have widely varying channel characteristics. There is variation across channels and in time. Motion and location cause many additional complications, the details depending on the network technology and service provider. For example, average packet round-trip-times on different channels can easily differ by $500ms$[4]. Even on a single channel, at different times and locations, individual packet times can vary by this much.

These issues led us to consider using inverse multiplexing, or network striping, to aggregate several of these physical channels to provide virtual channels. Network striping takes data from the larger source virtual channel and sends it in some order over the smaller physical channels, possibly reassembling the data in the correct order at the other end before passing it to the application.

The available diversity in present-day WWAN environments makes network striping an especially appealing approach. By taking advantage of service provider diversity, overlapping coverage, and network technology diversity (e.g. GPRS and CDMA), we attempt to provide each application with the illusion that a reliable stable high-bandwidth channel is available. The existence of technological and provider diversity—a benevolent side-effect of the competitive nature of the cellular provider market—is likely to bolster the virtual channel, making it more reliable. The underly-

[3]The *Verizon* `CDMA2000 1xEV-DO` networks in the Washington DC and New York City areas seem to provide around three times as much downstream bandwidth as the older `1xRTT` network in Boston, but both networks provide the same amount of upstream bandwidth.

[4]Compare the `GSM/GPRS` and `CDMA2000 1xRTT` channels in appendix A

ing channels are more independent than if the same technology or the same provider were being used.

A great deal of work has been done on network striping [24, 19, 7, 39, 36, 33]. Most of this work is aimed at providing improved scheduling algorithms under the assumption that the underlying links are relatively stable and homogeneous. If this assumption holds, there is little reason to give applications control over how the striping is done, and allowing applications to be oblivious to the fact that striping is taking place is appropriate.

In our environment, however, the underlying links are neither stable nor homogeneous. Therefore, the manner in which the middleware decides to schedule the transmission of application packets can have a large influence on observed packet latencies, stream loss rates, and bandwidth. Furthermore, the application streams in our telemedicine system are heterogeneous with respect to which aspects of the network service they are sensitive: some applications care about average latency, some not; some care about loss more than others; and some care more about the variance of the latency than they do about the average latency.

In our WWAN environment the packet scheduler can modulate application observed QoS on a data stream and different applications can care about very different aspects of this QoS. This leads us to want to give applications some control over how striping is done.

The Horde middleware allows a collection of application data streams to provide abstract information about desired behavior to a striping subsystem. The subsystem uses this information to stripe data from these streams across a set of dynamically varying network channels. The key technical challenge in horde is giving applications control over certain aspects of the data striping operation (e.g., an application may want urgent data to be sent over low latency channels or critical data over high reliability channels) while at the same time shielding the application from low-level details. Horde does this by exporting a set of flexible abstractions to the application, in effect replacing the application's network stack.

1.1 Motivating Application

While this thesis is not about the application that motivated our work on Horde, understanding some aspects of the application is useful in understanding many of the design decisions underlying Horde. The application is part of a project to improve emergency health-care. We digress briefly to provide an overview.

For every 100,000 people in the United States, it is estimated that 794 have had a stroke [4]. Every year, 400,000 stroke survivors are discharged from hospitals. Since many of these people will be left with life-altering disabilities, the socio-economic impact of strokes on American society is one of the most devastating in medicine.

In many situations, the timely application of an appropriate therapy is of critical importance. Particularly, individuals suffering from acute ischemic stroke—a significant proportion of stroke victims—are potential candidates for thrombolytics therapies [4]. Effectiveness of treatment depends upon administering the pharmaceutical within a short time after onset of the stroke.

A diagnosis of ischemic stroke involves a fourteen step examination procedure called the National Institute of Health Stroke Scale (NIHSS) exam. NIHSS exams are normally conducted by a stroke specialist after the patient arrives at a medical center. All too often, patients arrive at the hospital too late: to be most effective, therapy must begin within around three hours of symptom onset. What should be preventable, becomes a life-long burden.

Ideally, one would like to perform the diagnosis and administer the treatment either at the scene of the stroke, or in an ambulance on the way to the hospital. Unfortunately, while EMTs—often the first medical personnel to treat stroke victims—can administer the therapy, they do not have the training needed to accurately administer a NIHSS exam. The exam is necessary before therapy can be started.

Experience with inter-medical-center and mobile remote medicine suggests that real-time video distributed over a high speed Internet connections can be a vital aid in diagnosing ischemic stroke (and other conditions) [42]. Notably, such a real-time video link enables the stroke specialist to remotely diagnose a patient. EMTs could

then begin therapy en-route to the hospital.

Our overall research project aims to provide advanced remote diagnostic capabilities for patients in an ambulance moving about in an urban area.

1.2 Goals

The primary goal of the work described in this document is the construction of a stable high-bandwidth virtual communication channel. We have chosen to use the Internet because of its ubiquity. As a necessary part of building a mobile telemedicine application, we need to establish a stable, high-quality video stream between the ambulance and the hospital. The video stream need not be duplex, but it must be possible for the ambulance occupants to interact with the doctor using real-time audio. Audio and other medical sensor streams—EKG, blood pressure and the like—are relatively easy to deliver, compared to the video, since they require at least an order of magnitude less bandwidth than the quality of video we wish to deliver.

The virtual channel must be as insensitive to motion as possible. While it is possible to require that the ambulance stop every time the video link needs to be used, a system that does not inhibit motion is closer to our aim of speeding up the treatment of patients. Unfortunately, even when striping, some degradation in link quality due to motion is likely to occur. Doppler effects, resulting from the motion, can be correlated across all channels. Similarly if provider base-stations are co-located, loss and bandwidth on the different channels may exhibit correlation.

In the context of inverse multiplexing, our system must deal with a *heterogeneous* set of physical channels. The different physical channels can vary in their bandwidths, the distributions of their packet latencies and their loss characteristics. QoS on each channel is also likely to vary significantly in time. The motion of the ambulance can cause this dynamic variation to increase. Our system must therefore assume that the underlying physical channels are neither similar nor necessarily stable.

In order for the telemedicine system we design to be useful, the system must be economically viable to build, deploy and operate in an urban environment. Many of

our decisions related to networking technologies and hardware are driven by this goal. Particularly, we focus on developing a system built out of conventional off-the-shelf components and the existing communications infrastructure so that the telemedicine application becomes easy to deploy.

A mobile telemedicine system based on WWAN striping would be fairly easy to deploy. No special communications infrastructure would need to be set up; all that would be required is a computer on the ambulance with the appropriate network interfaces, equipped with standard cellular data plans. Since our system is designed to deal with a diverse set of network interfaces, as the cellular providers improve their networks, the virtual channel's bandwidth would scale.

Additionally, the use of a set of channels provides choice to the maintainers of the telemedicine system: they can decide how much bandwidth they want, and provision accordingly, without having to involve the carriers in this decision. If a particular ambulance decides to use a roughly 200 kilobit-per-second connection for the video, they can achieve this by acquiring two `CDMA2000` interfaces from different providers; if instead, they decide to implement a higher-quality link at one megabit-per-second, they only need to acquire additional interfaces and plug them into the ambulance's computer. No negotiations with the carriers are needed for this bandwidth flexibility.

The Horde middleware is designed to ease the development of applications that need to use WWAN striping. By separating striping policy from mechanism, Horde allows programmers to focus on describing desired QoS policy. Horde's approach aims to decreases the programming costs associated with building complex mobile systems that use network striping.

Our research intends to develop ideas that can be generalized beyond any particular problem. In fact, we focus on building a set of flexible tools that can be used by engineers to build high-performance mobile applications. This thesis does not focus on telemedicine, or even the problem of wireless video streaming; we tackle the problem of developing a general-purpose high-bandwidth wireless communication infrastructure, using widely available technology.

1.3 WWAN Striping: Challenges

A great deal of work has been done on network striping in the past. The simplest network striping approach is to use round-robin: send the first packet from the virtual channel down the first physical channel, the second packet down the second physical channel, and so on until we come back to the first channel. Much of the past work on network striping aims to provide improved scheduling algorithms assuming that the underlying links are relatively stable and similar and that applications are oblivious to the fact that striping is taking place.

Most contemporary striping systems assume that they are striping their data over a *homogeneous* set of network channels. For example, across n ISDN lines [7] or over m CDPD modems [39]. These systems compensate only for the relatively low-level heterogeneity that exists among channels, resulting from different levels of congestion on each channel.

This assumption of homogeneity is unrealistic for the class of applications we are addressing. In most areas the bandwidth/latency/loss characteristics of the available public carrier wireless channels vary by more than an order of magnitude (such as among a set of channels containing both 2.5G and 3G cellular data channels). We also expect there to be a high degree of dynamic variation in each wireless channel's *quality*—partly due to the motion of the ambulance and partly due to the fact that, since the WWAN is a shared resource, we are competing with other users for bandwidth. Moreover, since Horde will be deployed in moving vehicles, the set of available channels will change unpredictably. The experimental results documented in appendix A provide evidence of high variability in bandwidth and packet-latency on real WWAN channels. Spatial variation in the QoS provided by different carriers is well-known and largely depends on the carrier's placement of cell-towers relative to the terminal. One of the reasons for temporal variation is that bandwidth demand in a WWAN cell varies over time, while the total available bandwidth in that cell remains roughly constant. Motion also causes a great deal of temporal variation.

Horde exposes certain aspects of the striping operation to the application to allow

a modulation of the network QoS for data streams. Since there are multiple heterogeneous channels available to Horde, the order in which the middleware decides to schedule the transmission of application packets is crucial. Horde attempts to schedule packets at the sender so as to maximize the expected *utility* derived by the applications from the resulting sequence of packet receptions. The utility is defined by the application's expressed policy. The scheduler takes into account such things as expected latencies, observed loss-rates, and expected loss correlations.

In contrast, most striping systems normally aim for invisibility, hoping to hide from applications the fact that data is being striped over multiple channels. In order to achieve this goal, the design of these striping systems must be aware, in a way similar to NAT's, of the application layer protocols that run on top of them. For instance, many such systems try to minimize packet reordering within TCP-flows. While the goal of invisibility is sensible when applications cannot, or should not, be modified, we believe that the resulting behaviour is not always appropriate. An invisible striping system has to make two possible choices: either optimize for the common case (e.g. reliable in-order delivery [27, 39, 7]); or optimize for each *known* protocol separately ([36, 32]). Ultimately, neither approach is flexible.

While invisible striping can ignore application QoS sensitivities, an alternative approach that does not do so is application-specific code to perform the striping. Many multi-path video streaming applications [37, 12, 9] exemplify this approach. In these systems, network-striping code is intertwined with the application logic. In contrast, Horde allows one to build a multi-channel video streaming application that cleanly separates the network striping and network channel management operations from the application logic.

Allowing applications to actively influence the striping operation can be beneficial. By allowing each application to express its desired goals, Horde can arrive at efficient transmission schedules that can lead to high aggregate application utilities. Consider, for example, our telemedicine system. An encoded video stream may contain both keyframes (I-frames) and unidirectionally encoded frames (P-frames). In order for P-frames to be decoded properly at the receiver, the I-frames they depend on must have

been successfully transmitted across the network. Clearly, on one level, an application derives more utility from an I-frame than it does from a P-frame. This suggests that I-frames should be sent over more reliable channels. If two I-frames from different video layers contain similar information, it may make sense to ensure that they are sent over channels with uncorrelated losses. Additionally, during playback the audio must be kept synchronized with the video. In this case, an application may value receiving an audio packet at roughly the same time as the related video packets.

Horde is ambitious: our goal is to show that striping *policy* can be sufficiently decoupled from *mechanism*, the latter being determined by the applications in a fairly abstract manner, thereby easing the development of diverse bandwidth-limited[5] mobile-aware applications. We have designed Horde to be expressive enough to provide the application with an appropriately abstract level of control over the striping process. In this thesis, we present results from a preliminary implementation of the middleware to show that this decoupling of policy and mechanism is possible.

The primary challenge in the design and implementation of the Horde architecture was deciding how applications should express their goals and how these goals should be translated into data transmission schedules. The Horde middleware allows a group of application data streams to provide abstract information about desired behavior to a striping subsystem. The subsystem uses this information to stripe data from these streams across a set of dynamically varying network channels between the application and a host recipient. Horde gives the application control over certain aspects of the data striping operation (e.g., latency and loss) while at the same time shielding the application from low-level details.

Horde is most useful when dealing with:

Bandwidth-Limited Applications Situations in which applications would like to send more data than individual physical channels can support, justify both the network striping operation and the additional processing cost associated with Horde's QoS modulation framework.

[5]By *bandwidth-limited* we imply that the bottleneck in the application is the network. Particularly, we assume that end-point processing is not constrained.

Sensitivity to Network QoS If no application data streams are sensitive to network QoS, Horde's QoS modulation framework becomes redundant.

Heterogeneous and Dynamically Varying Network Channels With such channels, the scheduler has an opportunity to significantly modulate observed QoS. Additionally, Horde's approach is most useful when the available network channels have characteristics that are predictable in the short term. As we report in appendix A, public carrier WWAN channels are of this nature.

Heterogeneous Data Streams When different streams gain value from different aspects of network performance, trade-offs can be made when allocating network resources among those streams. An example is an application in which some streams value low latency delivery and others value low loss rates.

Horde is not meant to be general networking middleware: as long as most application data can be sent down a single, stable link, using Horde is overkill. More generally, in situations where one is dealing with a fixed set of relatively homogeneous and stable channels, other techniques [39, 7, 27] may be more appropriate.

1.4 Thesis Contributions

This thesis makes the following major contributions:

Inverse Multiplexing Middleware

This thesis describes the design and implementation of the Horde middleware, which provides applications with an inverse multiplexing subsystem. Horde is designed to be flexible and modular, allowing application designers to optimize for many different types of networks (e.g. wired, wireless, CDMA, 802.11) and also allowing for different types of packet schedulers to be written for the middleware. We use a preliminary implementation of Horde to explore notions about dealing with channel heterogeneity and application policy.

Separating Striping Policy from Mechanism

We present techniques to separate striping policy from the actual mechanism used and evaluate how well our striping mechanism is able to interpret the policy expressed by applications. Horde provides an *objective* specification language that allows QoS policy to be expressed for data streams. We have implemented a basic packet scheduler whose striping policy can be modulated using such objectives. Through real-world experiments and simulation we show that a system in which applications modulate striping policy is feasible. Furthermore, our experiments demonstrate that the ability to modulate striping policy provides tangible benefits to applications.

Heterogeneous and Unstable Channels

Horde has been designed from the ground-up with the assumption that the set of channels being striped over are not well-behaved and can have significantly different channel characteristics that may change unexpectedly (e.g. due to motion relative to a base-station). Horde uses channel-specific managers to provide congestion control and channel prediction logic for each available channel. We argue that congestion control belongs below the striping layer and should be performed in a channel-specific manner because of the idiosyncrasies of WWAN channels.

WWAN Channel Characteristics

Through a number of real-world experiments and discussions of two different WWAN technology standards, we explore the characteristics of real WWAN channels. Not part of the core thesis, these experiments are documented in appendix A. We show that `CDMA2000 1xRTT` and `GSM/GPRS` channels are handicapped by low bandwidth, high and variable round trip times, occasional outages, considerable burstiness, and much instability when moving.

1.5 Thesis Organization

This thesis continues by providing some background on related problems, such as network striping and WWAN channels in chapter 2. A more detailed characterization

of WWAN channels is left for appendix A. We provide an overview of the Horde architecture in chapter 3, and follow in chapter 4 with a discussion of the abstractions we use to decouple the Horde packet scheduling logic from details about individual network channels and channel behaviour models. Chapter 5 begins our discussion of packet scheduling in the domain of network striping. Chapter 6 introduces the policy expression framework and the scheduler we have developed for Horde (a more detailed discussion is left for appendix B). Chapter 7 evaluates an early implementation of the Horde middleware using both real and simulated experiments. Finally, chapter 8 provides conclusions and directions for future work.

Chapter 2

Background

This chapter covers some background material.

2.1 System Design

Our design uses the application level framing principle (ALF), as enunciated by Clark et al [17]. Our use of *application data units* (ADU's) and extensive event handler callbacks is reminiscent of the Congestion Manager (CM) [8]. However, instead of following the CM's pure-callback based approach to scheduling, Horde mixes callbacks with an abstract interface that can be used to specify complex application-specific policies. Horde began as an incarnation of the congestion manager for multiple channels. Given the complexity of the QoS modulation problem, the design of the library evolved to the point where it is quite different from the CM. Additionally, the mechanism we use to divide the total bandwidth among data streams is similar to the mechanism used by XCP [31]. Finally, the use of transmission tokens as network transmission *capabilities* borrows from capability based systems, like the MIT exokernel OS [20].

2.2 Wireless WAN's

Figure 2-1 enumerates the various wireless wide-area data network technologies that might be available to a terminal running Horde. 802.11 is not a WWAN technology,

Channel	Latency	Loss	Throughput/Down	Throughput/Up
GSM/GPRS	Highest	Low	40	20
CDMA2000 1xRTT	High	Low	120	120
CDMA2000 1xEV-DO	High	Low	300	120
802.11*	Low	Medium	>1000	>1000

Figure 2-1: Characteristics of WWAN channels. The throughputs are estimated averages in kilobits-per-second. 802.11 is only listed as a reference.

but is included as a reference point.

In most urban areas, there are a large number of public carrier cellular wireless channels providing mobile connectivity to the Internet, using standard cellular technologies such as GSM/GPRS and CDMA2000. Notably, there are multiple providers and these providers have overlapping coverage areas, allowing us to connect to more than one provider at the same time. More details about these channels can be found in appendix A; we provide a brief overview here.

In connections over these WWAN's, the wireless link dominates in determining the quality of the connection. Because of the restricted wireless channel throughputs (see figure 2-1), the bottleneck link is likely the last wireless hop. Doppler effects due to motion and the positioning of the terminal relative to the provider's base-stations can significantly reduce this already low throughput.

In our experiments, the last IP hop's latency accounted for over 99% of the overall latency of the entire route taken by packets sent from a WWAN terminal. Providers using GPRS use IP tunnels to transfer a terminal's data packets through the provider network. The last IP hop in such a situation is the entire provider network, which might explain why it dominates.

WWAN channels are handicapped by low bandwidth, high and variable round trip times, occasional outages, considerable burstiness, and much instability when moving. The quality of a WWAN link is highly dependent not only on the technology used, but perhaps even more so on the way the provider has decided to set up the network (e.g. the distribution of a provider's base-stations).

CDMA2000 channels are better than GPRS channels. Figure 2-1 notes that a CDMA2000 link has six times the upload bandwidth as a GPRS link. For small TCP-SYN pack-

ets and stationary terminals, we have measured average packet round-trip-times of around $315ms$ on the `CDMA2000` link and average times of $550ms$ on the `GPRS` link. For larger 768-byte UDP packets and stationary terminals, on both types of channels we have measured an average packet latency of around $800ms$ with a standard deviation of around $100ms$.

Motion causes the throughput of these links to degrade. Average throughputs aren't affected much by motion, but on both channels the standard deviation of the throughput increases by a factor of five. Since the `CDMA2000` channel's throughput variance is a much smaller fraction of its total throughput, this increase in variance is less significant for `CDMA2000` than it is for `GPRS`.

When moving, the `CDMA2000` latencies are not affected much, but for the `GPRS` link the average latency is multiplied by 1.5 and the standard deviation is multiplied by 4.5. On `GPRS` channels latency spikes are, generally speaking, correlated with motion; when nodes are stationary the round-trip-times are relatively stable. In contrast, on the `CDMA2000` channels, we observed no significant motion artifacts and a predictable pattern in packet round-trip-times.

Disconnections on WWAN channels are uncommon. Occasionally we experienced disconnections on the `GPRS` links during our experiments. We could reconnect immediately without changing location in almost all of these situations. The `CDMA2000` link, never disconnected, though we did notice occasional service disruptions while moving about Boston.

In this thesis we do not assume a high-bandwidth connection, such as 802.11, will be available to us. If a stable 802.11 connection were available, it would make sense to send as much data down the 802.11 channel as possible. Other researchers have investigated the use of wireless technologies like 802.11 for public networks [3, 16, 2] with varying goals and varying degrees of success.

802.11 is not a WWAN technology. It was not designed to be used over large distances, in multi-hop wireless networks, and with vehicular mobile terminals. Therefore its use in this way poses a number of technological challenges. For instance, technologies like `CDMA2000` have been designed to seamlessly switch base-stations

with *soft* hand-offs whenever appropriate, while 802.11 has not. 802.11 also has a higher loss rate than the WWAN channels because the WWAN channels use heavier coding and retransmissions on the wireless link [25].

We expect 802.11 connectivity to be rare, but possible (through the techniques used by the Astra project [2], for example). Further, since the ambulance is moving, this connectivity is likely to be short lived (less than thirty seconds). With the appropriate channel manager, whenever an 802.11 connection is available the Horde scheduler can shift to using that connection. The remainder of this thesis assumes that 802.11 connections are never available.

2.3 Network Striping

Schemes that provide link layer striping have been around for a while [24, 19, 7, 39]. Most such schemes were pioneered on hosts connected to static sets of ISDN links. These schemes mostly assume stable physical channels. Instabilities and packet losses can cause some protocols to behave badly [7]. Usually IP packets are fragmented and distributed over the active channels, if one fragment is lost, the entire IP packet becomes useless. This results in a magnified loss rate for the virtual channel. Horde avoids this magnification by exposing ADU fragment losses to the application, consistent with the ALF principle.

Almost all link layer striping systems optimize, quite reasonably, for TCP. Mechanisms are chosen that minimize reordering, and packets may be buffered at the receiver to prevent the higher TCP layer from generating DUPACKS [7]. When loss information must be passed to higher layers, such as in the case of fragmented IP packets, partial packet losses can be exposed to the higher layer by rewriting TCP headers before the striping layer passes the packets up to the TCP stack [39].

Implementing flow control under the striping scheduler is not a new idea. The LQB scheme [39] extends the deficit-round-robin approach to reduce the load on channels with higher losses. Adiseshu et al [7] note that they used a simple credit based flow control scheme below their striping layer, to reduce losses on individual

channels. The use of TCP-v, a transport layer protocol, to provide network striping with the goal of achieving reliable in-order delivery [27] also results in per-channel flow control. However, whereas Horde implements *channel-specific* flow control, past work has used the same flow-control mechanism on each type of channel. With our use of flow control and channel models, we take the idea of flow control below the striping layer further than any previous system of which we are aware.

Some recent proposals for network striping over wireless links have proposed mechanisms to adjust striping policy. Both MAR [36] and MMTP [32] allow the core packet scheduler to be replaced with a different scheduler implementing a different policy. This is the hard-coded policy case discussed later.

Recently, the R-MTP proposal by Magalhaes et al [33] considered the problem of striping over heterogeneous channels in a mobile environment. R-MTP uses a novel congestion control scheme that is optimized for wireless channels. R-MTP is primarily concerned with bandwidth aggregation and, although implemented as a transport level mechanism, keeps the striping operation hidden from the application. In contrast, Horde considers the problem of bandwidth aggregation to be less important than the QoS modulation techniques this thesis presents, and striping is exposed to applications.

2.4 Video Encoding and Streaming

We were motivated by the need to develop a multi-path video streaming application. This section provides some background in the area of video encoding.

Most video compression techniques, such as the ubiquitous MPEG standards [34], not only exploit the spatial redundancy within each individual frame of video (neighbouring pixels are likely to be similar), but are also able to exploit the temporal redundancy that exists between video frames (in scenes from the real world, adjacent frames are similar). Usually, the higher the sampling frame rate of the source video, the more temporal redundancy exists between adjacent frames.

A simple encoder/decoder (codec) such as one for MPEG4 video transforms the

input from a video source, a sequence of images, into a series of coded frames. Frames are coded as either keyframes (also called intra-coded frames, $I\text{-}frames$), predictively coded frames ($P\text{-}frames$), or bi-directionally predicted frames ($B\text{-}frames$). I-frames are coded independently of all other frames (these may be coded in the same way as JPEG images, using quantization and discrete cosine transforms to reduce the amount of data that must be stored). P-frames are coded based on the previously coded frame. Finally, B-frames are coded based on both previous and future coded frames. I-frames carry the most information, followed by P-frames which carry more information than B-frames. Various forms of motion prediction can be used to produce the P-frame and B-frame encodings.

If a constant video quality is desired, the bit-rate of the compressed video may vary over time. The ability of an encoder to produce the smaller P-frame and B-frame encodings depends on the amount of redundancy, which can vary depending on how the video scene progresses over time. If a constant bit-rate is required, the quality may vary. Modern codecs usually provide both options [6].

When streaming video over networks, the codec must provision for the possibility that the delivery process will introduce errors. Many decoding errors can be concealed from the human eye through prediction: spatial, simple temporal, and motion-compensated temporal interpolation techniques can all be used. The loss of I-frames is the most damaging: since these contain the most information, the P-frames and B-frames that depend on them cannot be decoded properly. Forward error correction (FEC) techniques can be used, but erode the gains made by the compression mechanism. Retransmissions can also be used for lost data, but this may not be an option for real-time video, such as ours.

Additionally, with network video streams, there are likely to be hard deadlines for the arrival of video ADU's. Generally, video data is useless if it arrives too late into the decoding process. Modern players use a *playout buffer* to compensate for delay. Such buffers can help mitigate for network delay, variance in this delay (or *jitter*), and even to provide enough time to retransmit lost packets [21].

Scalable video coding techniques can be used to encode the video as a base layer

and one or more enhancement layers. Multi-description (MD) video coding encodes the video into a number of descriptions, each of roughly the same value to the decoder. With MD coding, losing all the packets for an encoding still leads to good video quality, without the error propagation that a single-description MPEG-like codec would suffer from. Receiving data for multiple descriptions only enhances video quality at the decoder.

Setton et al [37] provide an example of such an application that uses multiple-descriptions of the source video, spreading them out over the available paths, based on a binary-metric quality of the paths. Begen et al [12] describe a similar scheme. Horde allows more flexible scheduling of video packets over the available paths.

More comprehensive overviews of video encoding can be found elsewhere [5, 23, 6], but are largely irrelevant for the purposes of this thesis. The following two facts are important: different types of video ADU's provide different amounts of utility to the video decoder; and video ADU's have dependencies between them: if one fails to be delivered then the other provides less, or possibly no, utility to the decoder.

Chapter 3

Horde Architecture

The Horde middleware can be used by applications to stripe data streams over a dynamically varying set of heterogeneous network channels. Horde provides the capability to define independent and dynamic striping policies for each data stream. Applications communicate with Horde by instructing it to deliver application data units, *ADU*'s, which are then scheduled on the multiple available network channels. The Horde middleware provides to applications the ability to abstractly define striping policy; per-network-channel congestion control; and explicit flow control feedback for each stream. We have implemented Horde in user-space.

Our design of Horde grew out of our work with hosts equipped with many heterogeneous wireless WAN interfaces[1]. In practice, we observed that these WWAN interfaces experienced largely independent intermittent service outages and provided a network quality-of-service that varied significantly with time (see appendix A). The service provided by such a set of WWAN interfaces represents a local resource that must be multiplexed among independent data streams. The primary purpose of the Horde middleware is to manage this resource. Mainly because of the large WWAN QoS variability, the management of this resource presents a number of challenges that have shaped the architecture presented here:

[1] We use the terms *WWAN channel* and *WWAN interface* interchangeably.

Modulating QoS We observed high time-dependent variability on the WWAN interfaces, additional quirks introduced by motion, and large differences in QoS across different WWAN technologies. This presents an opportunity to significantly modulate observed QoS across different data streams through the use of different packet scheduling strategies. The Horde scheduler is designed to allow applications to explicitly affect how the middleware modulates QoS. A flexible QoS policy expression interface allows applications to drive the packet scheduling strategy within Horde.

Network Congestion Control Horde needs independent congestion control for each channel, to determine available bandwidth and to be fair to other users of the shared WWAN. Furthermore, channel-specific congestion control mechanisms are needed for WWAN's, due to the dominating effect of the wireless link. Congestion control is not a focus of our research, but chapter 7 evaluates a WWAN-channel-specific congestion control mechanism to provide justification for our arguments.

Stream Flow Control Horde provides explicit flow control for each data stream, informing each stream about how much of the overall throughput has been allocated to that stream. Horde enforces these allocations. A mediator within the middleware divides the network bandwidth among data streams, even as total available bandwidth varies widely. Since the demands of streams can outstrip the availability of the striped-network resource, enforced bandwidth allocations and explicit flow-control feedback become particularly important in providing graceful degradation. Although an important component of the overall system, flow control and bandwidth allocation mechanisms are not a focus of our research.

Bandwidth allocation vs QoS modulation Horde separates the issue of dividing available bandwidth among active data streams from the issue of modulating the network QoS for each of those streams. Given a bandwidth allocation, different scheduling strategies can modulate QoS in different ways. The QoS modulation mechanism has a lower priority than the bandwidth allocation mechanism: ignoring short-term deviations, all transmission schedules obey the bandwidth allocations.

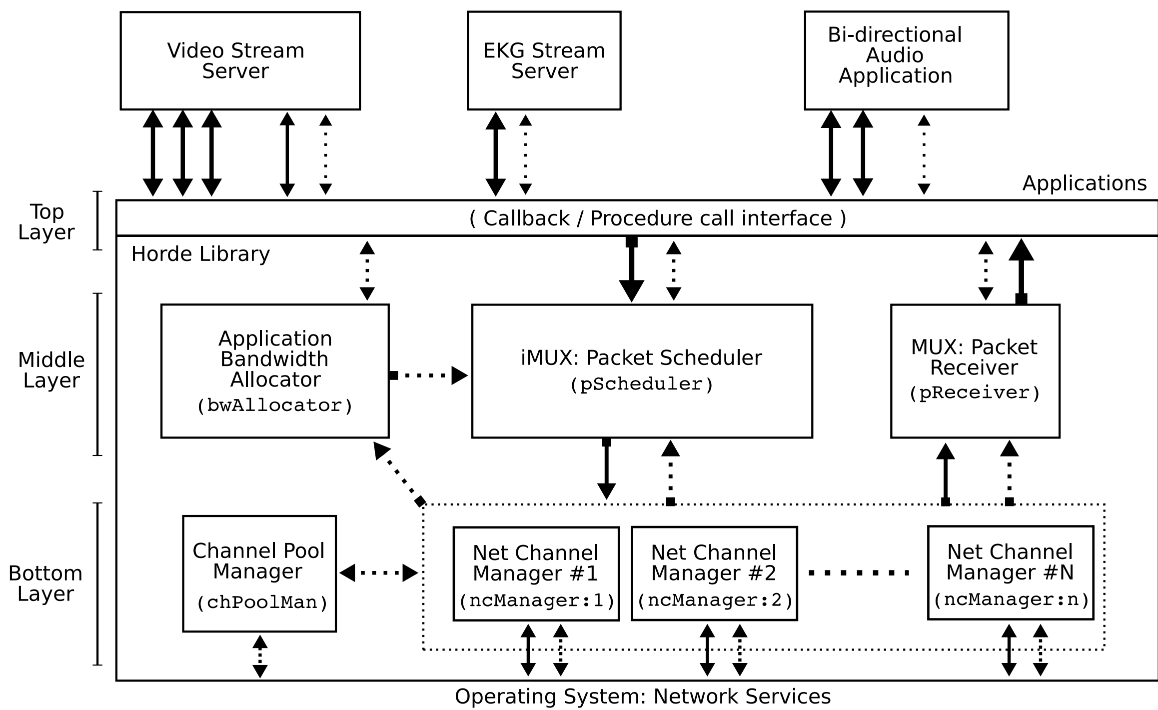


Figure 3-1: A modular breakdown of the *Horde* internals. Solid lines represent the flow of data; dashed lines represent the direction of control signals. The video and audio applications are both using multiple data streams.

This chapter describes the overall structure of the middleware and discusses various aspects of the architecture in more detail. In particular, we describe the basic interface exported by Horde to applications and Horde's network channel management layer. Chapter 4 covers the abstractions used to decouple the packet scheduler from the network layer. We defer a detailed discussion of the packet scheduler and Horde's policy expression interface till chapters 5 and 6.

3.1 Architecture Overview

This section presents a high level overview of Horde. We first summarize the architecture from an application's perspective, outlining the capabilities the middleware provides and the interfaces it exposes to applications. We then consider the internal structure of our implementation of the middleware. Figure 3-1 shows a sketch of how the middleware is structured.

3.1.1 Application View

An application can open one or more Horde data streams (§3.2.1) and then use these streams to send *application data units*, ADU's (§3.2.2), over the striping subsystem. If the application is sensitive to some QoS aspects on any stream, it can inject *objectives* (chapter 5) into the middleware in order to modulate the observed network QoS for those streams. For every ADU an application sends, it will eventually receive an `ack`, if the ADU was delivered, or a `nack` if the middleware detects a loss.

Streams receive bandwidth allocations from Horde (§3.2.3). For each stream, the application must specify some parameters related to the desired bandwidth for that stream. Furthermore, for each stream, the middleware sends *throttle* events to the application to notify it about changes in allocated bandwidth. Applications can use this information to prevent over-running sender side buffers and for more complex behaviour, such as, changing compression or coding strategies.

3.1.2 Horde Internals

Internally, Horde is divided into three layers (see figure 3-1). The lowest layer presents an abstract view of the channel to the higher layers of the middleware. It deals directly with the network channels, handling packet transmissions, flow control, and probes. The middle layer, is composed of the inverse multiplexer and the bandwidth allocator. The highest layer interfaces with application code.

In our implementation, the highest layer provides a simple IPC interface. Applications use this interface to inject and remove ADU's and policies. Horde also delivers ADU's and invokes event handler callbacks using this interface.

In the middle layer, the outgoing packet scheduler, `pScheduler`, decides how to schedule ADU's from the unsent ADU pool over the available channels (chapter 5). Incoming packets are delivered to the relevant data streams by the `pReceiver` module. The bandwidth allocator, `bwAllocator` (§3.2.3), decides how to divide up the bandwidth, provided by the channel managers, among the data streams.

In the lowest layer, the various network channel managers (§3.3) deal directly

with the network interfaces. Most importantly, they perform congestion control and inform the rest of the middleware how much data can be sent along each interface. The channel pool manager monitors network connectivity on all a host's interfaces, making sure there is a channel manager for each active interface.

3.2 Application Interface

This section describes the interface exported by Horde to applications. The highest layer of the middleware communicates with applications. The present implementation of the Horde API provides a set of C++ classes and methods allowing applications to operate on streams and ADU's. Our implementation encapsulates the middleware as a daemon process. The API makes calls into the middleware (using standard Unix IPC methods) to open Horde data streams, execute operations on open streams, transfer ADU's, or modulate scheduler policy. Horde invokes event handler callbacks into the API code, using the same IPC pathways, to send feedback[2]. The remainder of this section describes the capabilities provided by the Horde interface.

3.2.1 Horde Data Streams

Horde provides a connection based model to an application. The applications on the source and destination hosts set up a connection, or *stream*, before they start sending data to each other. A Horde stream represents a sequence of application data units. Applications can register callbacks to manage events on each stream. Important events for streams are: ADU `ack`'s, ADU losses, and flow control related `throttle` events.

Establishing a Connection To accept connections from an *a priori* unknown hosts, the middleware allows an application to register callbacks so that the appli-

[2]This implementation is appropriate when traffic along these IPC pathways is relatively low. For our current application the largest virtual channels we expect are on the order of a megabit, so the cost of crossing process boundaries is not prohibitive. Another, single-application implementation is designed to run inside the application's process, if higher bandwidth virtual channels are needed. Only the callback/procedure-call code stubs need to be changed.

cation can react to specific connection requests. Each connection request carries a service identifier that allows the middleware to deliver that request to the appropriate callback. This is analogous to a non-blocking `accept` on a POSIX TCP socket.

If an application wants to initiate a connection to a known host, it must make a call into Horde—providing a service identifier, in addition to naming that host—and then wait for the local instance of Horde to successfully negotiate a stream connection with a remote instance of the middleware.

As in TCP, once the connection has been established, no distinction is made between the two end-hosts.

Naming Network Hosts Since Horde is striping data over multiple interfaces, at least one of the end-hosts is equipped with multiple interfaces. Horde uses these multiple addresses—at least one address must exist for each active interface—as equivalent aliases for naming that host. To connect, the host initiating the connection only needs to know one alias; the handshake protocol that establishes a stream connection exchanges all active aliases for the end-hosts of that connection. In our implementation, addresses are always IP addresses.

Stream Flow Control As discussed later, an application requests, and is granted, some—possibly time-varying—amount of bandwidth for each stream. The middleware maintains an independent queue for each stream and if the application sends data at a higher rate than can be accommodated, given the bandwidth allocation for that stream, ADU's are dropped from the stream's queue in an unspecified order.

3.2.2 Application Data units

When sending and receiving data on a stream, applications communicate with Horde at the granularity of *application data units* (ADU's). Horde allows ADU's to be injected and removed dynamically by applications.

Horde ADU's consist of some amount of data—transmitted across the network—and arbitrary header fields—not transmitted across the network. Header field values

can be modified after the ADU has been injected into Horde. The header fields can be used as part of the policy expression framework on the sender host. For example, our video streaming application could define the `frame_type` field, for use within policy definitions for video streams, set to either `'I', 'P' or 'B'` in each video ADU. Some fields are predefined and managed by Horde (e.g., `stream_id`, and `latency`).

Partial-ADU losses and network ADU reordering are exposed to applications. This design decision draws from previous arguments for application level framing [17, 8]. Although ADU's from a stream are always delivered to the scheduler in the order they appear in the stream, they can arrive out of order and, since ADU's may need to be fragmented for packet based delivery[3], partial-ADU losses can occur.

If stream manager callbacks are registered, applications are notified about all ADU acknowledgment and ADU loss events. Horde network peers send acknowledgments for each ADU fragment they receive. This is the consequence of needing feedback for congestion control in the channel managers (§3.3). These acknowledgments can be prioritized to be always sent along the lowest latency available network path, resulting in the fastest possible feedback over the network.

3.2.3 Bandwidth Allocation

Every active Horde stream is allocated some fraction of the available bandwidth. For each stream, an application either specifies the maximum rate it wants to send data along that stream, or marks that stream as being driven by an `elastic` traffic source. Horde in return informs the application the rate at which it can send on that stream. If an application sends more than its allocated rate, ADU's are dropped in an unspecified order inside the middleware, because of sender-side buffer overruns.

As available bandwidth changes, `throttle` events—reflecting the new bandwidth allocations—are generated by Horde and delivered to each stream manager callback. In practice, the total available bandwidth can vary significantly with time. Simple

[3]For simplicity, this thesis largely ignores the fragmentation issue. The Horde implementation discussed in this thesis never fragments ADU's. ADU fragmentation has been studied in other real systems [8], and is less interesting to us than other aspects of Horde, such as QoS modulation.

applications can ignore these events, always sending at their maximum rates, letting the middleware decide which ADU's to drop. Conversely, adaptive applications can use the information conveyed by throttle events to guide changes to their behaviour. Some applications may respond to `throttle` events by simply ramping their sending rates up or down. Others may change their behaviour in more complex ways, for example: down-sampling the stream; changing stream encodings; omitting the transmission of less important data, etc.

Enforced bandwidth allocations and explicit flow-control feedback are important in providing graceful degradation when independent data streams exist. The set of active network managers can provide only a finite—and time varying—amount of bandwidth to be distributed among the active data streams. This amount will often be outstripped by total demand. As alluded to in figure 3-1, multiple independent applications may simultaneously be using Horde data streams to try to send data over the network. Further, for a single application, callbacks managing different streams can be independent of each other. A mediator is therefore needed inside the middleware to divide the bandwidth. This mediator transforms changes in available bandwidth into some set of changes in the bandwidth of each stream. In Horde, stream managers only need to respond to `throttle` events; the applications do not need to consolidate bandwidth management code.

The present implementation of Horde uses a simple adaptive min-max fairness policy to allocate available bandwidth among streams. The pseudo-code in figure 3-2 illustrates the bandwidth allocation process. This process is managed by Horde's bandwidth allocator (`bwAllocator`) module. The set of active streams is repeatedly iterated over until all available bandwidth has been allocated. In each iteration an additional unit of bandwidth is allocated to every stream that has less bandwidth than it requested. An `elastic` stream is assumed to request an infinite amount of bandwidth.

```
unallocated := total available bandwidth
extra := max((unallocated - ∑∀x requested(x)), 0)
while (unallocated ≥ extra)
   for ∀x ∈ {streams}
      if (x.allocated < x.requested)
         then  unallocated := unallocated - 1
               x.allocated := x.allocated + 1
```

Figure 3-2: Pseudo-code for the bandwidth allocation process.

This approach to allocation has some notable properties: no stream is ever allocated more bandwidth than it requests; any extra bandwidth is wasted; and if

$$min_{\forall x}(\texttt{x.requested}) \geq \frac{\texttt{total}}{\texttt{|\{streams\}|}}$$

then bandwidth is divided evenly among all streams.

When multiple destinations exist, a different bandwidth allocation process might be needed. When Horde is striping multiple streams to multiple end-hosts, the achievable throughput to one host may be larger than the achievable throughput to another host. Dividing the overall total bandwidth using the earlier process can waste bandwidth. To avoid wastage streams can be divided into different classes—based on the destination hosts—before bandwidth allocations are derived. However, if the sending interface is the bottleneck, this approach can result in the same number of transmission slots used for every end-host. This is not appropriate unless all destinations are equally important for applications. Since our telemedicine application stripes to a single destination host, our work on Horde all but ignores the multi-destination case.

Bandwidth allocation is separated from QoS modulation in Horde. This is because different scheduling strategies with a bandwidth allocation can modulate QoS in different ways. For example, n bytes of data from a stream can be sent on a fast channel instead of spreading those n bytes over fast and slow channels. The same amount of bandwidth is used, but the stream is provided different QoS.

In Horde, the QoS modulation mechanism has a lower priority than the bandwidth allocation mechanism. When producing transmission schedules, the amount of data

sent for each stream is restricted based on the stream allocations[4].

3.2.4 Application Policy

Using Horde, an application is able to define a dynamically varying policy, which in turn impacts the packet scheduling strategy within the middleware. Chapter 6 contains a more detailed discussion of the various ways in which applications can express policies within the framework of our middleware and how these policies affect the scheduler.

In short, Horde applications can define independent local *objectives* for each data stream (e.g., degree of reordering in the stream), between streams (e.g., that losses should be uncorrelated), or at the granularity of ADU's. An objective specifies a particular QoS property that can be achieved using some packet transmission schedule for that stream. The application also defines how much *utility* it derives from the fulfillment of each objective. The Horde scheduler's goal then is to pick packet transmission schedules that provide high utility to applications.

3.2.5 Reporting Network Latency

Applications sometimes need estimates for average network round-trip-time and the round-trip-time variance. These estimates might be used, for example, to set timers for timeouts, or to size video play-out buffers. One option is for Horde to provide a single number that professes to estimate the average network latency—possibly a weighted average across all channels. Horde does not attempt to provide such a number. Applications using Horde are not oblivious to the fact that multiple and heterogeneous channels are being used, so there is no reason to collapse latency information into a single number.

An application can calculate its own estimates for a stream, using that stream's `ack` event sequence. `ack` events for ADU fragments are exposed to stream manager

[4] Over small time-scales this may not be true. The scheduler can produce imbalanced transmission schedules if it determines ADU's from some stream(s) should be delayed and sent on a channel that is not ready to accept data.

callbacks. Every `ack` event specifies the round-trip-time for the related ADU fragment transmission.

Instead of explicitly exposing the number of channels to the application, the Horde interface instead reveals the channel equivalence classes. Horde allows an application to acquire a set of latency average-variance pairs. This set provides the major modes of the latency distribution seen by the Horde scheduler. Each element of this set either represents a single channel[5], or multiple channels that Horde has determined are equivalent (e.g., two channels with the same average latency and correlated variations). With two GPRS channels and one CDMA2000 channel (§7.1), the smallest number of major modes seen by the Horde scheduler is two—one for GPRS and one for CDMA2000—and the largest number of modes is three—one for each channel. Both sets of modes are valid interpretations of the Horde interface specifications.

3.3 Network Channel Management

This section describes the part of Horde that deals directly with the underlying network channels. Other parts of the middleware are built on the layer outlined here.

3.3.1 Channel Manager Modules

Horde uses a set of Network Channel Manager modules (the `ncManager`s) to manage the available network channels. Each `ncManager` manages a single network interface. `ncManager`s maintain an internal model of their channels and perform dynamic measurements to determine the present state of their channels. This may involve active probing. The `ncManager`s are responsible for sending and receiving all packets from these channels and also perform channel-specific congestion control.

The purpose of an `ncManager` is to provide higher layers with an abstract interface to each channel, an interface that is independent of the underlying channel technology. There may be different implementations of the `ncManager` interface for different networks (e.g., 802.11, CDMA2000 1xRTT, GSM/GPRS, EDGE).

[5]Our implementation always reports a unique mode for each channel.

The primary utility of using different channel manager implementations lies in allowing different channel models and congestion control schemes to coexist without complicating the packet scheduler. Other researchers have presented congestion control schemes optimized for different WWAN channels (e.g., [38]). Furthermore, channel technology specific knowledge can be easily incorporated into a probabilistic model for that channel (§7.4.2). Such models can provide better predictive capabilities than generic models. The use of different `ncManager` implementations allows us to abstractly use channel optimized flow control and predictive logic for each channel.

3.3.2 Channel Pool Manager

A separate module instantiates the `ncManager`s, based on the set of available network channels. The Channel Pool Manager (`chPoolMan`) periodically monitors the host computer's network interfaces and activates or deactivates the `ncManager`'s appropriately, in response to changes in network connectivity.

The use of the `chPoolMan` facilitates software hand-offs. Normally, hand-offs for streams occur in response to interactions between feedback from the `ncManager`s, application policy, and the scheduling algorithms. A stream may be shifted off a channel by the scheduler if the QoS on that channel changes. The `chPoolMan` ensures that, when channel connectivity varies, the scheduler has enough information to shift streams on to or off the affected channels.

3.3.3 Congestion Control

Congestion control in a striping system should be implemented below the striping layer, independently for each channel. When data is being striped, since there are multiple, independent channels, there are multiple, independent congestion domains. Thus each channel manager should run an independent congestion control session[6]. An application, above the striping layer, does not have enough information to implement efficient congestion control, unless the application can perfectly simulate

[6]When there are multiple destination hosts, each `ncManager` may need to maintain a unique congestion control session for each of these hosts.

the scheduling algorithm. We do not believe that having a single congestion control session straddling channels—as previous striping systems have done—is the right approach. For instance, approaches based on ADU loss as an indicator for network congestion may either be inefficient (e.g., if some of the underlying channels provide explicit congestion notification ECN feedback) or just plain wrong (e.g., a loss on one channel may only indicate that the application should send less data down that particular channel; not that it should decrease it's overall sending rate since other channels may have spare bandwidth). Even older systems [7, 39] realized the need for some form of minimal per-channel congestion control below the striping layer.

Congestion control for WWAN channels should be implemented in a channel-technology specific manner. The behaviour of the last hop wireless link can often dominate in determining how well a congestion control scheme works on a given channel technology [10, 28]. WWAN links exhibit many idiosyncrasies (appendix A). Researchers have investigated congestion control schemes for wireless channels [10, 38] and we demonstrate a channel-specific congestion control scheme (§7.4.1) that performs better than generic congestion control on the type of channel it was designed for.

3.3.4 Transmission Tokens

Using a congestion-control mechanism, a channel model and recent measurements as its guides, each `ncManager` generates tokens in the form of *transmission slot*, `txSlot`, objects. These objects provide an abstract interface for the scheduler to access channel information, providing transmission capabilities and expectations about channel behaviour. We will discuss this abstraction in more detail in chapter 4.

3.4 Summary

The Horde API provides many services to applications. The API defines calls into the middleware to open Horde data streams, execute operations on open streams, transfer ADU's, and modulate scheduler policy. Horde invokes callbacks in applications, to

send stream-related feedback (e.g., ADU loss and acknowledgment events). Every active Horde stream is allocated some fraction of the available bandwidth. This allocation is based on available bandwidth, and negotiations between Horde and all active data streams. As available bandwidth changes, `throttle` events are generated for streams. Horde also allows applications to modulate network QoS for streams using a flexible policy expression framework, discussed more fully in chapter 6.

Horde uses channel-specific managers to provide congestion control and channel prediction logic for each available channel. The primary utility of using different channel manager implementations lies in allowing different channel models and congestion control schemes to coexist without complicating the packet scheduler. We have also argued that congestion control should be implemented below the striping layer.

The remainder of this thesis largely deals with issues related to the Horde packet scheduler. Chapter 4 examines the interface between the network layer and the scheduler; chapters 5 and 6 develop the notion of objective driven network striping; and chapter 7 mainly evaluates our scheduler implementation using real and simulated packet traces.

Chapter 4

Transmission Tokens

Horde uses the notion of transmission tokens, in the form of `txSlot` objects, to provide a well-defined and abstract interface between the channel manager modules and the packet scheduler. The scheduler makes its decisions based solely on the information contained in the `txSlot` objects. A `txSlot` is an abstract representation for the capability to transmit some amount of data on some network channel at a given time, along with the expected QoS that will be derived from that transmission.

Each channel manager can also be asked to *look ahead* into the near future (e.g. the next second) and estimate how many more slots are likely to become available on that channel. *Phantom* transmission tokens provide an abstract interface to this look-ahead logic. Phantom `txSlot`'s cannot be used to transmit data, but encapsulate expectations about future channel behaviour.

This chapter covers the transmission token abstractions in Horde. We first describe the basic `txSlot` abstraction. The remainder of the chapter discusses the phantom `txSlot` abstraction.

Field	Description	Range
`channelID`	Parent network channel for this slot.	
`sequence`	The sequence number for this slot on its parent channel.	
`lossProbability`	The estimated loss probability for a packet transmitted in this slot.	$loss \in [0, 1]$
`expectedRTT`	Expected time between the transmission of data in this slot and the reception of an acknowledgment.	$milliseconds \geq 0$
`lossCorrelation(other)`	Compares two transmission slots to see if packet losses in the two slots are expected to correlate. This is an estimate for $P(both\ lost \mid either\ one\ lost)$.	$corell \in [0, 1]$
`cost(x)`	Cost of transmitting `x` bytes in this slot.	$cost(x) \geq 0$
`maximumSize`	The maximum number of bytes that can be transmitted in this slot.	$size > 0$

Figure 4-1: The components of the `txSlot` abstraction.

4.1 The `txSlot` Abstraction

`txSlot` objects represent transmission slots on specific network channels. The main fields of the `txSlot` abstraction are outlined in figure 4-1.

Transmission Capabilities Each `txSlot` object represents a *capability* that the `ncManager` grants to the scheduler, allowing the transmission of some data along that channel. `txSlot`'s are produced by a `ncManager` whenever its congestion control algorithms determine it is safe to transmit data on the underlying channel. The packet scheduler acquires these `txSlot`'s from all active channel managers, maps ADU fragments to `txSlot`'s, and passes that data-to-`txSlot` assignment back down to the channel managers, resulting in data being transmitted by the channel managers that generated those `txSlot`'s.

Latency and Loss Expectations Each `txSlot` encapsulates information about the expected characteristics of the transmission slot, derived from a dynamically updated probabilistic model of the channel's behaviour.

When it creates a `txSlot` the `ncManager` uses a channel model to derive the

expectations for that slot. A simple `ncManager` implementation could use a weighted moving average of RTT values to determine the expected latency and a simple average ($\frac{total\ delivered}{total\ lost}$) for the loss probability. While moving-averages seemed to work well in our experiments, we found better estimators could be constructed using WWAN channel-technology specific knowledge (§7.4.2). More elaborate estimators plug in easily into our framework: a `ncManager` that uses signal strength readings to estimate a loss probability, or one that uses an accelerometer input to predict when a motion sensitive channel will start experiencing more losses, could be added without any complications.

Correlated Loss When two slots suffer losses, we say that the losses in those slots are correlated. There are two types of correlated losses that are important to us: burst losses on a single channel and correlated losses on different channels. In our experiments (appendix A) we observed that burst losses were common on wireless WAN channels. Correlated losses on different wireless WAN channels can occur because of signal fading or external factors that cause cross-channel loss correlation (e.g., because two channels have antennas on a tower that is occluded).

The `txSlot` interface provides a loss correlation metric to help the scheduler make judgments about correlated losses. Reasoning about loss correlations can be important in some application domains. Applications such as those streaming multiple description video [9] can be sensitive to correlated losses. As figure 4-1 indicates, two `txSlot` objects can be compared to see if a loss in one slot is expected to correlate with a loss in the other slot. The result of this comparison is a conditional probability for the event that both slots will experience losses if either of them experiences a loss. Suppose it is known that on a single channel 90% of the losses occur in bursts of length 2 or greater, but only 3% of the packets are actually lost. Comparing two back-to-back slots on this channel would yield a correlated loss probability of 0.9.

Striping presents multiple opportunities to reduce correlated losses. ADU's can be sent down different channels, provided these channels exhibit uncorrelated losses. ADU's can also be spaced out temporally on the same channel, interleaving multiple

data streams to maximize throughput and reduce correlated losses on any one stream. When a scheduler is comparing two `txSlot` objects, it does not need to know which of these factors (temporal or spatial) is causing losses for those slots to be uncorrelated.

Transmission Costs There are often monetary costs associated with the transmission of each bit on each channel. For some applications, it will be appropriate to let cost influence decisions, e.g., to prefer a lower-cost higher-loss `txSlot` over a higher-cost lower-loss one.

Each `ncManager` maintains its own cost model configured with provider-specific information. For instance, for a WWAN account with a data transmission quota, the cost for each `txSlot` above the quota is higher than for earlier slots. Since the cost model for each parent channel is accessible from the appropriate `txSlot`'s, cost models can be used in policy decisions by the scheduler.

Notice that the only difference between low and higher bandwidth channels is that higher bandwidth channels will result in more `txSlot`'s being generated. This simplifies the scheduler. Similarly, channels exhibiting a great deal of dynamic variation in their latency and loss characteristics—such as our WWAN channels—do not represent special cases. The scheduler treats two `txSlot`'s from the same channel, but with widely different loss rates or RTT's, as if they were `txSlot`'s from different channels.

The abstraction of `txSlot`'s makes it easier to reason about scheduling algorithms and application policy, especially when we are dealing with unconventional or ill-behaved channels with high QoS variances. By encapsulating channel-specific information inside one `ncManager` implementation tailored for GPRS channels and another `ncManager` tailored for the CDMA2000 channel (§7.4), we can build relatively simple schedulers that make use of this channel specific information (§6.4).

4.2 Scheduler Look-Ahead: *Phantom* Tokens

Phantom transmission tokens provide an elegant way for the scheduler to make decisions based upon expected future channel behavior. Each channel manager can be

asked to look ahead into the near future (e.g. the next second) and estimate how many more transmission slots are likely to become available on that channel. Using their channel models, `ncManager`'s can provide phantom `txSlot` objects tagged with a confidence level, a measure of how sure the channel manager is of its prediction. Phantom slots are only useful for their expectations; unlike other `txSlot`'s, they do not represent capabilities to transmit data, so the scheduler cannot actually use them to send data on any channel.

A good scheduler needs this sort of look-ahead logic. Depending on network configurations and policies, in some scheduling cycles the available `txSlot`'s are too few—or may lack critical information that can be easily predicted by a channel model—to make good decisions about schedules. Imagine, for example, that at some point in time only high latency `txSlot`'s are available. If a low latency slot will be available shortly, it may be better to defer scheduling an urgent packet. Experiments (§7.3) show that the use of phantowm slots boosts the accuracy of our policy driven scheduler. The alternative to predictive logic is to use infrequent scheduling cycles, increasing average queuing delays for ADU's.

The look-ahead logic needs to be in the channel managers. This logic is part of the Horde network layer, and must lie below the scheduler. Furthermore, this logic is derived from the channel-specific model and congestion control algorithms. We have used a simple look-ahead algorithm that relies on average transmission and loss rates from the past to predict the future. This predictor works reasonably well but, as with the logic for deriving expected latencies for `txSlot`'s, this predictor can be improved by incorporating additional knowledge about the underlying channel technology.

Phantom transmission tokens provide a good way to capture expected future channel behavior. Using the basic `txSlot` interface to facilitate the use of look-ahead logic allows us to keep the interface between the scheduler and the channel managers simple. Phantom `txSlot`'s look the same as normal `txSlot`'s—except for being marked as phantoms and tagged with confidence levels below 100%.

Phantom slots allow us to simplify the implementation of our schedulers. Predicted slots can be examined and compared the same way as actually available slots.

With phantom slots, factoring channel predictions into scheduling decisions does not require special-cases inside the scheduler[1].

Our `random-walk` scheduler implementation (§6.4) uses phantom slots. In any scheduling cycle, our scheduler uses application policies in conjunction with both normal and phantom `txSlot`'s to find a good schedule. The scheduler only transmits as much data as it can on the normal `txSlot`'s. In the next cycle, the process of finding a good schedule starts afresh; the scheduler does not try to remember assignments made to phantom slots. Our present scheduler does not use the confidence levels inside phantom slots, but a more sophisticated scheduler could easily use this additional information to make more informed decisions.

4.3 Summary

The interface between the Horde network layer and the Horde scheduler is composed, almost entirely, of a single abstraction: the transmission slot. `txSlot`'s serve two purposes: they represent transmission capabilities and they encapsulate expectations about channel behaviour. Channel latency and loss expectations are part of `txSlot`'s that can be used to transmit data. Phantom `txSlot`'s, on the other hand, cannot be used for transmissions and represent expectations about future slot availability. Phantoms encapsulate latency and loss predictions for these expected slots.

The use of `txSlot`'s decouples the scheduler from the channel-specific models and congestion control algorithms inside the channel managers. With phantom `txSlot`'s, channel-specific look-ahead logic can be incorporated elegantly into the scheduler.

[1]Depending on the scheduler design, the constraints imposed by the `bwAllocator` can introduce wrinkles in how phantom slots are used. The scheduler must obey the constraints imposed by the `bwAllocator`. These constraints require that a stream get a certain number of actual `txSlot`'s over a period of time, sometimes preventing the scheduler from delaying that stream's data using phantom slots. In our implementation, we found that these wrinkles were more than compensated for by the other advantages of using phantoms.

Chapter 5

Network Scheduling

At the core of any striping system lies a packet scheduler that decides how to transmit packets on the multiple outgoing links. In Horde, the packet scheduler does not determine the data-rate for a stream, but it can modulate the Quality-of-Service (QoS) for that stream. The bandwidth allocator's policy, in conjunction with the congestion control algorithms running within the channel drivers, ultimately limit the rate at which ADU's can be sent on each data stream. After the application data streams have been appropriately rate-limited, the scheduler must decide which transmission slot carries which ADU. These decisions have the potential to modulate the latency and loss QoS observed by applications on each data stream. For example, by consistently picking slots with low expected latencies for ADU's from data stream x, the scheduler can decrease the average latency observed by the parent application on stream x.

When dealing with a set of homogeneous and stable channels, simple round-robin schedulers work reasonably well. If all the network channels are similar in terms of average latency and loss, and per packet latencies and loss probabilities are close to these averages, transmitting a given packet on one channel instead of the other is not likely to affect the expected loss or latency of that packet. Therefore, it is not surprising that research in striping over wired and/or homogeneous channels has primarily been concerned with scheduling packets so as to prevent reordering [24, 19, 7, 39].

Conversely, with a heterogeneous and/or dynamically unstable set of available channels, the scheduler's task becomes more complicated. The available transmission slots, from the different channels at different points in time, can vary significantly in terms of expected loss and latency characteristics. Consequently, the manner in which the scheduler decides to transmit application packets can be crucial in determining the network QoS seen by an application on a given data stream.

We expect horde to be used in situations where available channels differ by nearly an order of magnitude, in terms of bandwidth and latency. Furthermore, even on a single WWAN channel we have observed packet latencies to vary in time by up to an order of magnitude (see appendix A). Therefore, packet scheduling techniques and their effects on application observed QoS are an important part of our research.

In this chapter we introduce the scheduling problem for network striping. We cover application utility functions and their relation to scheduling policy; we argue that it is beneficial to separate policy and mechanism in a striping scheduler for WWAN channels; we define what we consider to be optimal scheduling; and finally we review simple approaches to striping schedulers. Chapter 6 discusses the novel *objective* driven scheduler and policy expression interface we have developed for Horde.

5.1 Application Perspective: Derived Utility

Before we delve into different approaches to scheduling, it helps to consider what aspects of the network service are important from an application's perspective.

During some period of network service in which an application actively sends ADU's, it obtains some utility from the consumption of its ADU's at another host. We refer to this abstract utility as the application's *utility function* and represent it as a numerical value function—much in the same way microeconomics textbooks choose to model the utility consumers assign to goods that they consume [35]. A utility function defines a partial order on possible transmission schedules, which can be used by the scheduler to rank its scheduling choices.

The application's utility function represents *net* utility. *Gross* utility refers to

the total value derived by the application. Net utility is the difference between the gross utility and the *cost* of the network service. Gross utility is never negative, but net utility may well be, if the cost of using the network exceeds the utility derived from that service. *Expected* net utility can be obtained by using latency and loss expectations as inputs to the utility function.

In our formulation, a sender application's utility function is based solely on the ADU header fields, the sequence of packet transmissions and the sequence of `ack` receptions. Non-zero transmission costs may cause utility to be affected by channel usage and transmission times. Utility can also be directly related to the number, identity or delivery time of the delivered ADU fragments. However, since the sender doesn't know the delivery times, we work with network round-trip-times, derived from the ADU fragment transmission sequence and the `ack` arrival sequence. When ADU fragments have been sent but `ack`'s have not been received, the expected round-trip-times and, therefore, the expected utility can be used. Finally, factors such as local queuing time, parent stream identity and ADU type (e.g. I-frame) can also impact utility. The ADU header fields provide this information.

The idealized form of the utility function can be written as:

$$utility_{app}(\tau) \sim f_{app}\big(history_{\{tx\}}(\tau),\ history_{\{rx\}}(\tau)\big)$$

Where, over a period τ: packet transmission history ($history_{tx}$) is a set of triples of the form $(adu, fragment, txslot)$; `ack` reception history ($history_{rx}$) is a set of $(adu, fragment, time)$ triples; and f_{app} is an arbitrary application defined function.

The remainder of this section considers in more detail the factors influencing utility in our formulation.

ADU Transmission Cost Non-zero transmission costs may cause net utility to be affected by channel usage and transmission times. Fragments can be transmitted on multiple channels, each channel having a different time-varying cost model associated with it. Using a more costly transmission slot to deliver an ADU fragment

may be valued less than if that fragment could be delivered using a less costly slot. Cost models can be accessed through the `txSlot` interface. Cost can be in terms of the electrical power required for the transmission or in terms of the monetary cost associated with transmitting on a leased channel. For example, when striping over WWAN channels: one channel may have an unlimited data plan, another may charge a fixed cost for the first n bytes of data and then x cents for every additional kilobyte, and another may charge y cents for every kilobyte sent.

Delivered ADU's Utility is directly related to the set of delivered ADU fragments. Fewer delivered fragments are likely to result in lower utility since an undelivered ADU fragment likely represents lost potential, the inability to do some work. However, it is not always the case that delivering more fragments is equivalent to delivering more value. Which ADU's get delivered can also matter: some ADU's can be more important than others. For example, I-frame ADU's in a video stream provide significantly more utility than the B-frame ADU's in that stream. Furthermore, with redundant coding techniques an application might derive the same utility from the delivery of n ADU's as it would derive from the delivery of $(n + m)$ ADU's,. A simple example is when two ADU's and their bitwise-`XOR` are transmitted. Reception of any two of these allows the recipient to reconstruct the original two ADU's.

ACK Arrival Times Application utility may be affected by ADU network round-trip-times. If an application is sensitive to the passage of time, then the sequence of deliveries becomes important in determining the application's utility. Our primary use for a utility function is to rank possible transmission schedules. This ranking is only useful at the sender, in the outgoing packet scheduler. However, delivery times are in terms of events at the receiver. Without fine-grained clock synchronization between the two end-hosts, delivery times and transmission times are not truly comparable. We assume that the hosts are not synchronized. Therefore, utility functions are expressed exclusively in terms of what is known at the sender: the sequence of receptions of acknowledgments from the receiver. Because of network queuing and

reordering of the acknowledgments, the `ack` arrival times may not match the sequence of ADU fragment deliveries.

Expected Utility The packet scheduler uses *expected* utility instead of actual utility. A real-world scheduler will not have perfect information about the future. If our aim is to rank possible transmission schedules, in order to pick which ones to use, then we must account for this imperfect information. Within a period τ, a sender has perfect information about which packets it has transmitted and about the `ack`'s it has received. The scheduler lacks information about the packets for which `ack`'s have not yet been received, about available transmission slots that are being considered but have not yet been used, and about the slots that will be available in the future. Channel models can provide expectations for this missing information. We can pass these expectations as inputs to the utility function, to determine expected utility, and then this expected utility can be used in our scheduling decisions. The `txSlot` interface allows the scheduler to access these expectations.

ADU Fields Information in ADU header fields can be used by an application's utility function. Some header fields are predefined and managed by Horde. An example is the `latency` field which represents the time between an ADU's insertion into Horde and the reception of the final `ack` for that ADU. In addition to the network round-trip-time, this field factors in the time the ADU spends queued inside Horde. Furthermore, ADU's can be annotated (e.g., the application-specific video `frame-type` field and the Horde `stream-id` field). As we demonstrate later (chapters 6 and 7) annotations can be useful in utility function definitions.

5.2 Scheduler: Policy and Mechanism

The packet scheduler has a *policy.* The scheduler transforms its policy into transmission schedules based on the offered load and using what it knows—through models and measurements—about the present and near-future behaviour of those channels. For

example, a packet scheduler following a policy of maximizing bandwidth while minimizing receiver-side ADU reordering within a single data stream will try to transmit packets on the available channels so as to minimize that expected ADU reordering. Such a scheduler may try to minimize reordering as it stripes the data, but—with imperfect predictions about the future and imprecise past measurements—it may not be able to do so.

Most contemporary striping systems use a *static* scheduler policy [27, 39, 7] that cannot be modulated by applications. Often, the policy in these systems is inseparable from the scheduling mechanism itself[1]. These systems can work well, when the assumptions they make regarding their channels and the utility functions of the streams hold.

Using a single static policy is not always desirable. For a single data stream, using a scheduler whose policy does not conform to that stream's utility function can result in significantly sub-optimal results. In the worst case, the scheduler's policy can be directly opposed to the stream's utility function, yielding the worst possible transmission schedules.

As an example of sub-optimal scheduling, consider a scheduler optimized for TCP, like those used in [39, 7]. Such a scheduler may optimize its transmission schedules assuming that all data units provide the same amount of value to the application. Further, it may assume that *any* reordering in a stream of data units—as they pass through the striping layer and over the network—is to be avoided as much as possible. Suppose this scheduler were to be used with an application streaming video, encoded as a stream of I, P and B frames. First, the data units in this stream do not all provide the same utility to the application. I frames are the most important, followed by P frames. B frames do not tend to provide much utility. Second, small scale ADU reordering is not particularly important—a playback buffer will probably need to be used to compensate for the delay jitter anyway. Reordering is therefore tolerated by the application. If there were two channels being striped over, one having a far

[1]Generally speaking, this appears to be the result of choosing simplicity in mechanism as one of the primary design goals.

lower loss rate but a slightly larger latency than the other, the best schedule for the video application may be to route the I-frame ADU's consistently along the lower-loss channel while sending the P-frames consistently along the higher-loss channel. This approach would knowingly induce reordering within the data stream—something a scheduler with a TCP-optimized policy is not likely to do.

Furthermore, instead of a single stream, multiple streams—each with a different utility function—may be using the striping subsystem at the same time. In such a case, it may be possible to adapt the scheduler's policy to accommodate the objectives of multiple streams.

Different streams can be sensitive to different aspects of the QoS they receive from the striping subsystem. The streams in our telemedicine system are heterogeneous with respect to which aspects of the network service they are sensitive to. For example, we simultaneously transmit video and EKG data. Video ADU's are likely to be highly sensitive to latency. EKG ADU's are generated at a low frequency and are not as sensitive to latency. Additionally, the low sampling frequency gives the EKG application time to retransmit lost ADU's, so the EKG stream is not particularly sensitive to ADU losses.

Appropriate transmission schedules can be constructed that accommodate both the video and EKG streams. In order to do so, however, a single static scheduler policy cannot be used, unless we know enough about the policies of the streams *a priori* to construct an appropriate scheduler.

In Horde, the striping policy is separated from the striping mechanism itself; the policy being dynamically driven by application-specific logic. Horde allows applications to define the striping policy at run-time, providing a generalized mechanism within the scheduler to facilitate many different policies. Furthermore, the philosophy underlying our system's design implies that an application may decide to change its policy on-the-fly. We use these ideas to guide our scheduling algorithms and the design of our policy expression interface, presented in chapter 6.

5.3 'Optimal' Scheduling

An *optimal* scheduler would provide a set of streams, over some time period, with the best possible transmission schedules, given the policies expressed for those streams and the state of the network during that period.

Since no real-world scheduler has perfect knowledge of the future, in the context of Horde, we choose to define an *optimal* scheduler as one that would pick the short-term schedules most valued by the application given expectations about the future.

For a period τ, such a scheduler has complete information about some things (which ADU's have been sent; which `ack`'s have been received) and expectations about others (latency and loss expectations for the ADU's that have been sent but not yet `ack`'d; expectations about the currently unused `txSlot`'s; and expectations about the phantom `txSlot`'s).

Over τ, such a scheduler would find a transmission schedule that maximizes the sum of the utility functions for all of the applications[2] using the expectations about future channel behaviour provided to the scheduler[3]:

$$maximize\left[\sum_{\forall app}\left(utility_{app}(\tau)\right)\right]$$

As posed above, the optimal scheduling problem is a computationally infeasible online optimization problem. Infeasible since scheduling decisions may need to be made many times a second in an actual system. In our scheduler we only try to find *high utility* transmission schedules instead of optimal schedules.

[2]If fairness is of concern, this definition can be modified to take that into account. In this thesis we restrict our consideration of fairness to bandwidth allocations. Further, since fair bandwidth allocations for each stream are provided to the packet scheduler by another component in Horde, fairness does not enter our definition of scheduler optimality.

[3]These expectations are encapsulated in concrete and phantom `txSlot`'s.

5.4 Simple Scheduling Approaches

Before we move on to describe our *objective* driven scheduler, let us first consider some possible ways to implement simple packet schedulers within a striping subsystem.

5.4.1 Round-Robin Scheduling

The simplest network striping approach is to use round-robin: send the first ADU down the first channel, the second ADU down the second channel, and so on until we come back to the first channel. A great deal of past work on network striping has used some variant of round-robin scheduling [24, 19, 7, 39], primarily because of its simplicity and the fact that more complex schedulers are not likely to provide much additional utility when the underlying network channels are stable and the applications are unaware that striping is taking place.

A powerful variant often used is deficit-round-robin: the scheduler cycles through the different network channels, sending as many packets, as the deficit/weight for each channel will allow. In practice, there are many different ways to derive the weights [7, 39]. Weights can be set statically or dynamically updated, based on estimated bandwidths, loss rates, etc.

Older systems (e.g. [7]) using round-robin schedulers assume that all packets entering the striping subsystem are part of the same stream, the single virtual network channel. When there are multiple input streams to the scheduler, as in Horde, we must first pick an input (an application data stream) and then pick the output (the `txSlot`). These choices could both be made in a round-robin fashion; application ADU's could be serviced in a FIFO manner (as in the older single virtual channel approaches) and channels in a round-robin manner; or some other selection techniques could be employed.

5.4.2 Randomized Scheduling

Deterministic round-robin scheduling with multiple inputs can result in situations where one or more inputs receive an unfair allocation from the striping subsystem. Consider the case when ADU's from a given input x are always mapped to the output channel y. If all network channels are similar, always sending data from input x down channel y is probably reasonable. However with heterogeneous channels, like those in our set of WWAN channels, such a consistent mapping could result in some input stream experiencing much higher average loss or latency than the other streams.

Randomization is one technique that can help avoid this behaviour within the round-robin framework. With multiple inputs and multiple outputs, one can pick the output using some round-robin variant and pick the input using a random variable, or vice-versa. The randomization can evenly distribute good and bad `txSlot`'s.

5.4.3 Callback Scheduling

Another scheduling approach is to expose every `txSlot` to the application, simply asking it to assign data to each transmission slot. This is reminiscent of the congestion manager framework [8]. Since the application is itself forming the schedules, it can provide specialized code to process each `txSlot`, to attempt to maximize the utility it derives from the resulting schedules. The application's policy is hard-wired in the callback code. Some striping systems have followed this approach [36, 32].

In principle, a callback scheduler specifically implemented for a given application policy has the potential to be more computationally efficient than the flexible scheduler we propose. The drawback is that such callback schedulers can be both difficult to implement and difficult to reason about for even moderately complex applications. This is especially true when there are multiple applications and multiple streams each having a non-trivial local policy. The monolithic callback scheduler implementation must take into account the state of all active data streams when processing each `txSlot`. Often, it is easier to think in terms of local stream policy. Our *objective* driven scheduler uses localized policy expression.

5.4.4 Channel Pinned Scheduling

An approach popular with schedulers optimized for multi-path video streams is to allow applications to pin each data stream to a selected network channel. Such schedulers only send packets from stream x along some specifically defined network channel y, discarding all packets the channel cannot hold, even if other channels have spare capacity. Adaptive schemes based on this idea redistribute the streams as channel conditions change. Redistribution is not frequent; short-term channel condition variations are ignored.

5.5 Summary

The application's utility function defines the value that the application derives from the network service provided by Horde. In our formulation, a sender application's utility function is based solely on the ADU header fields, the sequence of packet transmissions and the sequence of `ack` receptions. A utility function defines a partial order on possible transmission schedules, which can be used by the scheduler to rank its scheduling choices. An 'optimal' scheduler would use transmission schedules that maximize application utility.

Most contemporary striping systems use a scheduler policy that cannot be modulated by applications. This can result in transmission schedules that yield low application utilities. Techniques such as round-robin variants, specialized schedulers and channel-pinning have inflexible scheduler policy. The Horde scheduler, in contrast, separates striping policy from the striping mechanism. The next chapter describes the Horde *objective* driven scheduler.

Chapter 6

Objective Driven Scheduling

By providing a specialized scheduler and a sufficiently abstract policy expression interface, Horde allows applications to drive the striping operation. This chapter is an overview of the Horde scheduler and the Horde Objective Specification Environment (`hose`). Appendix B provides a detailed description of `hose`; this chapter provides an informal introduction to the most important aspects of `hose`.

6.1 Objectives

An important issue in the design of Horde was deciding how applications express their policy goals and how the scheduler translates these goals into data transmission schedules. `hose` allows applications to express their policy goals as succinct *objectives*. Objectives represent the modular decomposition of an application's utility function. The set of expressed objectives drives the packet scheduler towards transmission schedules that provide high utility to applications.

The set of objectives describes the application policy regarding network quality-of-service. Each objective says something, relatively simple, about the sort of network QoS an application would like for some stream(s). The set of objectives describes how much an application values the assignment of a given type of `txSlot` to some ADU. An application injects one or more objectives into the library. Objectives can be injected and removed dynamically, allowing for the specification of time-varying

utility functions. When an application does not specify an objective for some QoS aspect along some stream, it is implicitly assumed that the application does not care about that aspect of QoS on that stream. In the special case when no application injected objectives exist for a given stream, the scheduler implicitly assumes that no application is sensitive to any aspect of QoS on that stream. Such a stream is likely to be assigned those transmission slots left over after existing objectives have claimed the good slots for their own streams. However, bandwidth allocations are enforced before transmission schedules are constructed, so every stream always receives its fair share of transmission slots. Nonetheless, since a stream may receive slots more likely to result in losses, that stream may not receive its fair share of throughput. Fair shares do not imply that all streams are allocated equal bandwidth (refer to §3.2.3 for information on the notion of fairness used by the bandwidth allocator).

Objectives can be *positive* or *negative*. Positive objectives define the utility that would be gained if the objective were met by a schedule. Negative objectives define the amount of utility that would be lost if the objective's associated property were part of some schedule.

Objectives will usually be concerned with the policy of a single data stream but it is possible—and sometimes useful—to define objectives that straddle multiple streams. For instance, an objective from a multi-description video streaming application could express the application goal of minimizing correlated frame losses on the different video data streams.

It is likely that applications will set up competing policy objectives. For example, one objective could value latency while another could value order preservation on the same stream. Alternatively, two streams could both have associated objectives valuing low latency, when not all transmission slots have low latency. The tension between competing objectives must be resolved by the scheduler.

The modular reasoning provided by the use of objectives is intended to simplify application development. For those applications at which we have looked, the objectives tend to be relatively simple, e.g., favour lower loss `txSlot`'s for certain types of ADU's in a stream (e.g., video I-frames), favour lower latency `txSlot`'s for some

specific streams, and avoid correlated losses for certain ADU's. Conversely, the monolithic utility functions of these same applications tend to become rather complex.

6.2 Objective Specification Language

The purpose of the `hose` language is to provide a flexible mechanism in which application objectives can be expressed and evaluated. Using literals, ADU's, streams, header fields, and latency and loss probability distributions as the basic units, `hose` allows the construction of complex constraints.

`hose` is a type-safe language. Every expression in `hose` has a well-defined type and there are specific rules governing how expressions of the same or different types can be composed and what the type of the composition will be. The `numeric` type in the language definition only supports integers. Probabilities are represented as percentage values ranging from 0 to 100.

Objective All valid fragments of `hose` code must start with an `objective` definition. An `objective` consists of three sections: a `context` (variable bindings); a `goal` (a boolean predicate); and a `utility` section. Figures 6-1, 6-2 and 6-3 show samples of `hose` code. For a schedule in which the `goal` predicate is `true`, the `hose` interpreter uses the `utility` section of the `objective` to determine how that `goals` fulfillment has affected the utility derived from the associated transmission schedule. Active `objectives` are evaluated by the scheduler in an unspecified order.

Context An `objectives` `context` section specifies a mapping between the `goal` and a set of ADU's that can be used to achieve that `goal`. Each `context` defines a set of filters on all possible ADU's. For every `ADU` or `stream` variable used in the `goal` or `utility` sections, the `objectives` `context` contains a filter predicate, expressed in terms of ADU header fields or stream identifiers. Fields like `stream_id` and `stream_seq` are predefined for ADU's. Applications can arbitrarily define other header fields to selectively activate `objectives` for marked ADU's.

```
objective {
   context {
      adu:foo { (stream_id == 17) && (frame_type == "I") }
      adu:bar { (stream_id == 17) && (frame_type != "I") }
   }
   goal { prob(foo::lost?) < prob(bar::lost?) }
   utility { foo { 100 } }
}
```

Figure 6-1: The objective `Y1`, expressing the policy that `txSlot`'s carrying I-frames from stream 17 should have lower loss probabilities than slots for other frame types.

The example objectives `Y1`, `Y2` and `Y3` show different types of `context`s. The `context` for `Y3` (figure 6-3) specifies that the objective `Y3` applies to all ADU's from stream `937`. The `context` for `Y2` (figure 6-2) specifies that the objective `Y2` applies to stream `7`. The `context` for `Y1` (figure 6-1) is the most complex example here. The objective `Y3` is defined in terms of two `ADU` variables `foo` and `bar`: `foo` is always bound to an `I`-frame ADU from stream `17`; and `bar` refers to any non-`I`-frame ADU from stream `17`. The `frame_type` field is an example of an application-defined field. It is used here to integrate application-specific ADU annotations into `hose`.

Goal The `goal` section specifies a predicate that can be evaluated on some schedule (i.e., on a mapping of ADU's to `txSlot`'s). `goal` predicates can be reasonably involved: simple probability, boolean and numerical expressions can be progressively composed to produce a complex boolean predicate.

Each of the objectives `Y1`, `Y2` and `Y3` has a different type of `goal`. `Y2`'s `goal` is `true` whenever the average latency on stream `7` falls below one second. `Y3`'s `goal` is met whenever an ADU from stream `937` can be sent in a slot having a loss probability below 20%. `Y1`'s `goal` is `true` whenever an `I`-frame ADU can be sent in a slot that has a lower loss probability than a slot used to send a non-`I`-frame ADU.

Utility The `utility` section specifies how application utility is affected when a `goal` has been met. The `utility` section contains a numeric valued expression for each `ADU` or `stream` variable whose utility is affected. Negative utilities bias the

```
objective {
   context { stream:foo { stream_id == 7 } }
   goal { foo::latency_ave < 1000 }
   utility { foo { 100 } }
}
```

Figure 6-2: The objective `Y2`, expressing the policy that the average latency on stream 7 should be less than one second.

```
objective {
   context { adu:foo { stream_id == 937 } }
   goal { prob(foo::lost?) < 20 }
   utility { foo { 5 * min((1000 - expected(foo::latency)), 100) } }
}
```

Figure 6-3: The objective `Y3`, expressing the policy that the utility derived from the delivery of ADU's falls off linearly as the latency rises above one second.

scheduler against schedules with the property specified in the related `goal`; positive utilities promote such schedules. The base utility of transmitting an ADU is zero. Therefore, within the bandwidth allocator's constraints, a scheduler implementation can decide not to transmit an ADU whenever the set of `objectives` specifies that all possible transmission slots give negative utilities for that ADU.

Complex numeric expressions are possible. Objectives `Y1` and `Y2` are simple, adding a constant positive utility when their goal is met, but objective `Y3` (figure 6-3) uses a fairly complex `utility` section. `Y3`'s goal is met when an ADU from stream `937` can be transmitted in a slot with less than a 20% chance of experiencing a loss. When the `goal` is met, `Y3` specifies that the utility gained from transmitting in that slot falls off linearly as the expected latency for that slot rises above one second. If the expected latency is higher than $1100ms$, `Y3` becomes a negative objective, driving the scheduler away from assigning such slots to stream `937`'s ADU's.

Probabilities The notion that the scheduler is working with imperfect information is included in the language. `hose` defines special `latency` and `lost?` fields for each `ADU` variable. `latency` represents the elapsed time between when that ADU was

inserted into Horde by the application and when the `ack` event for that ADU was delivered to the application. `lost?` is `true` in the event that the ADU was dropped. These fields are treated in the language as typed random variables (equivalently: as probability distributions over some type). The need for random variables exists because manipulating these fields as normal numeric or boolean variables would ignore the fact that they represent indefinite quantities. For example, until an `ack` has been received, the actual value of `latency` is not known. Its value can, however, be estimated, using channel models.

`hose` provides operators to covert these random variables to numeric values. The `expected` operator can be used to get the expected value of a numeric random variable. The `prob` operator can be used to get the probability (as a percentage, between 0 and 100) of a boolean random variable being `true`[1].

`hose` also defines a `correlated_loss` operator. `correlated_loss` operates on pairs of ADU's and provides an estimate for the following probability:

$$P(both\ ADU's\ were\ lost \mid either\ ADU\ was\ lost)$$

Stream Variables `stream` variables are included to allow the expression of certain types of constraints. `stream` variables have the fields: `stream_id`, `latency_ave` (mean stream latency), `latency_sdev` (stream latency standard deviation), and `loss_rate` (stream loss rate). `stream` variables are needed, for example, in objectives based on the ADU latency average or jitter. Constraints on stream latency variance cannot be expressed in `hose` using only `ADU` variables. Furthermore, the average latency on a stream is different from the *expected* latency of a transmission slot. The later is what the `ADU`'s `latency` field provides. For example, the expected latency for a single

[1]An earlier incarnation of the `hose` language allowed random variables to be composed in the same way that boolean and numeric expressions can be. The `hose` version described in appendix B is a lot more restrictive. Allowing the arbitrary manipulation of random variables has the potential to prohibitively increase scheduler complexity. Furthermore, another reason for not providing a generalized random-variable (or PDF) type and generic PDF manipulation operators is that the underlying network models used by Horde are not mature enough to generate probabilities for arbitrarily conditioned network events. Correlated ADU loss, single ADU loss, and ADU `ack` arrival time are well-defined network events, which Horde's network models can handle.

transmission slot may be high (e.g., if the channel manager noted a temporarily low signal strength and predicted that multiple link-layer retransmits might be needed), but the average latency for the stream may remain low in spite of instantaneous latency spikes.

6.3 Objective Driven Scheduling

Given the set of injected objectives, the set of ADU's and the set of available `txSlot`'s, an objective driven scheduler uses those objectives to find a good high-utility transmission schedule. A schedule is a mapping of `txSlot`'s to ADU's. This section describes the basic semantics governing how schedules are derived from objectives.

The optimal scheduler's goal (from §5.3) in terms of objectives is:

$$maximize\left[\sum_{\forall objective}\left(utility_{objective}(\tau)\right)\right]$$

The Horde scheduler runs in cycles. It builds a short-term transmission schedule during each consecutive scheduling cycle, using a relaxed form of the above goal to construct the schedules. The period τ above is the time since the last scheduling cycle. A scheduling cycle runs at least every T milliseconds, but may run sooner if more than m slots are available. At the start of each cycle, the scheduler collects ADU's from the heads of the data stream queues, and `txSlot`'s from the channel managers. Then, using the injected objectives, the scheduler finds a high-value schedule.

The scheduler only considers *valid* transmission schedules when determining which schedule to use. All schedules that are composed of currently available `txSlot`'s and obey the allocations assigned by the `bwAllocator` (§3.2.3) are considered to be valid. However, since our look-ahead mechanism (§4.2) may reveal future transmission slots better suited to ADU's being considered in the current cycle, the scheduler is allowed short-term deviations from the `bwAllocator` allocations. In the long run, however, the Horde scheduler must ensure that these allocations are maintained.

Given an `objective` and a schedule, there may be multiple valid `context` bindings

for that `objective`. The set of valid `contexts` for an `objective` can be determined by applying the `ADU` filter defined in the `context` to the union of the set of ADU's in the schedule and the ADU transmission history. The `context` provides bindings for the `ADU` variables used in the `objective` and the `txSlot`'s (from the schedule and the history) provide the latency and loss probability distributions that might be referenced in the `goal` and `utility` sections.

Evaluating an `objective` adds (or subtracts) some amount of utility to the parent schedule. For each valid schedule, an `objective` can be evaluated once for every valid `context` in that schedule. When the `objectives` define a negative utility for some {`ADU`, `txSlot`} pair, the scheduler may decide to assign another ADU to that slot.

Horde schedulers don't guarantee optimal scheduling. The comprehensive evaluation of objectives over all possible transmission schedules in every cycle would result in optimal scheduling. Unfortunately, the number of valid transmission schedules grows as the factorial of the number of transmission slots in the schedule. Therefore, in practice, the Horde scheduler must use some other method of evaluating objectives. Our implementation efficiently finds a high utility schedule, not necessarily the highest utility schedule, by restricting the set of possible schedules it considers.

Scheduling with Look-Ahead

A good scheduler needs look-ahead logic. In some scheduling cycles the available `txSlot`'s can be too few—or may lack critical information, that can be easily predicted by a channel model—to make good decisions about schedules. Imagine, for example, that at some point in time only high latency `txSlot`'s are available. If a low latency slot will be available shortly, it may be better to defer scheduling an urgent packet. Experiments (§7.3) demonstrate that look-ahead boosts the accuracy of our policy driven scheduler. The alternative to predictive logic is to use infrequent scheduling cycles, increasing average queuing delays for ADU's.

Phantom transmission tokens (§4.2) provide a way to factor expected future channel behavior into scheduling decisions. Each channel manager can be asked to indicate, by creating phantom `txSlot`'s, how many slots it expects to have available in the

near future (e.g., the next second). Predicted slots can be examined and compared in the same ways as actually available slots. With phantom slots, incorporating channel predictions into scheduling decisions does not require special-cases in the scheduler[2]. In any cycle, Horde schedulers use both normal and phantom slots to find good schedules, only transmitting as much data as can be sent using the normal `txSlot`'s. In the next cycle, the process of finding a good schedule starts afresh; the scheduler does not try to remember assignments made to phantom slots.

6.4 Scheduler Implementation

This section describes our objective driven scheduler implementation: the `random-walk` scheduler. In constructing our scheduler, we have consciously attempted to make it as simple as we could. Our goal was to show that even a very simple algorithm can provide enough benefits to justify the Horde QoS modulation framework. Chapter 7 demonstrates that this scheduler works reasonably well.

In each scheduling cycle, our scheduler uses a random bounded-search of the transmission schedule space to find what looks like the best schedule, given the set of active objectives. In the absence of any objectives, the `random-walk` scheduler works like the randomized-round-robin scheduler: any valid mapping of slots to ADU's is equally likely. The scheduler creates k random transmission schedules[3], evaluates all objectives over each of these k schedules and picks the schedule with the highest aggregate utility. A scheduling cycle runs every T milliseconds. During each cycle, the scheduler acquires from every channel manager the currently available `txSlot`'s and the phantom `txSlot`'s for N milliseconds into the future. Based on the number of slots, ADU's are collected from the data streams. The constraints imposed by the bandwidth-allocator govern how many ADU's can be collected from each stream. Streams are treated as a FIFO queues. Our implementation does not drop ADU's

[2]Depending on the scheduler design, the constraints imposed by the `bwAllocator` can introduce wrinkles in how the scheduler uses phantom slots. In summary: the scheduler must ensure that, in the long run, the allocations are met even though phantoms may cause short-term deviations.

[3]In the experiments documented here, we used 25 for the value of k.

```
tx_schedule random_walk_scheduler() {
  // collect all tx_slot's with some amount of look-ahead
  slots = collect_slots( LOOKAHEAD_MSECS );
  // collect as many adus as there are slots
  adus = collect_adus ( slots.size() );

  // do a random walk
  int max_util = MIN_INTEGER;
  tx_schedule best_schedule = NULL;
  for (int i = 0; i < WALK_LENGTH; i++) {
    // get a random schedule
    tx_schedule random = create_random_schedule( slots, adus );
    // evaluate all objectives over this schedule and get util
    int util = evaluate_objectives ( random );
    // see which schedule has been the best so far
    if (util > max_util) {
      max_util = util;
      best_schedule = random;
    }
  }

  return best_schedule;
}
```

Figure 6-4: Pseudo-code for the `random-walk` scheduler. We used a `WALK_LENGTH` of 25 and the random schedule constructor given in figure 6-5. The method `evaluate_objectives` iterates over every active objective, evaluating that objective for the given schedule and returns the sum of the utilities.

locally, even if including those ADU's in the chosen schedule results in a negative utility. T, N and k are constants.

Figures 6-4 and 6-5 describe the scheduler in pseudo-C++ code. In practice, the scheduler is a little more complex, since it must deal with the constraints imposed by the bandwidth allocator. Our implementation never fragments ADU's for delivery.

The `random-walk` scheduler is not tied to the `hose` language. The language imposes restrictions on what types of objectives can be expressed. The scheduler itself assumes nothing about the properties of individual objectives. The `random-walk` scheduler only requires a set of functions that map schedules to numeric values. The scheduler can even be used with a single monolithic value function replacing

```
tx_schedule create_random_schedule(slots, adus) {
  tx_schedule random;

  // randomly reorder
  random_shuffle( slots );
  random_shuffle( adus );

  // produce schedule:
  //   slot[i] contains adus[i]
  for (int i = 0; i < adus.size(); i++)
   random[ slots[i] ] = adus[i];

  return random;
}
```

Figure 6-5: Pseudo-code for the random schedule constructor.

`evaluate_objectives` in figure 6-4. To differentiate good schedules from bad ones, `evaluate_objectives` should define a partial order on the set of possible schedules. A better scheduler could use the information in a `hose` objective (e.g., an objective favours low latency for stream x) to direct the search toward better schedules. A directed search, instead of the random one used here, is likely to eventually find better schedules.

The simulations documented in chapter 7 used pre-compiled objectives. We have implemented a `hose`-subset interpreter using the `lex`/`yacc` utilities but have not integrated it into the `random-walk` scheduler. Interpretation would be functionally equivalent, so the simulation results would not be affected.

One of the goals of our research was to prove that an objective driven scheduler was feasible in a real system, in-spite of the fact that the optimal scheduling problem itself is a prohibitive optimization problem. Our experiments demonstrate that a very simple scheduler can provide good results. A more intelligent scheduler implementation may be able to provide QoS to streams closer to their optimal allocations. In practice, there are many other ways to find approximate solutions to the optimal scheduling problem posed earlier. For instance, the scheduling problem can be mapped to a SAT-optimization problem, or to a graph-flow-optimization problem.

Our present scheduler does not use the confidence levels inside phantom slots, but a more sophisticated scheduler could easily use this additional information to make more informed decisions.

Scheduler Processing Cost

The Horde scheduler has a higher processing cost than conventional striping schedulers. It requires more computation and the maintenance of per-flow state. In practice, the cost of running the Horde scheduler is not a problem. The scheduler requires much less processing time than other application operations (e.g. MPEG video compression). Furthermore, with bandwidth limited applications, the network is the bottleneck and processing is not particularly constrained. Finally, in our WWAN environment, it does not help to run scheduling cycles more frequently than roughly once every $50ms$, since individual `CDMA2000` channels cannot generate more than one `txSlot` every $50ms$. Our initial implementations of the Horde scheduler indicate that the extra computation and state will not be a problem in practice. In our experiments with unoptimized code containing many debugging log statements, the scheduler implementation runs fast enough for our purposes.

6.5 Summary

Horde provides an abstract application policy expression interface and a flexible striping scheduler that can interpret application policy. Horde applications express their policy as a set of modular objectives, using the `hose` specification language. By interpreting these objectives, an objective driven scheduler attempts to provide the network quality-of-service requested by the application for each data stream. Given the set of injected objectives, the set of ADU's and the set of available transmission slots, an objective driven scheduler finds high-utility transmission schedules.

The `random-walk` scheduler is our implementation of an objective driven scheduler. This implementation demonstrates that even a very simple algorithm can provide enough benefits to justify the Horde QoS modulation framework. In each

scheduling cycle, the `random-walk` scheduler uses a random bounded-search of the transmission schedule space to find what looks like the best schedule, given the set of active objectives. Since a good scheduler needs channel look-ahead logic, the scheduler includes phantom tokens in the search.

Chapter 7

Evaluation

The preceding chapters have covered the overall design of the Horde library and the objective specification environment it provides. This chapter evaluates the performance of a preliminary implementation of the library, using both real and simulated network packet traces.

Horde provides three main services: bandwidth aggregation, network QoS modulation, and channel specific congestion control. Bandwidth aggregation is a service almost all previous striping systems focus on providing. Therefore, the other two services form the novel contributions of our research. This chapter focuses on these services. We show that QoS modulation is both feasible to implement and useful to applications. We also provide a case-study to support our earlier assertion that having channel specific congestion control and prediction logic has advantages over using generic methods for every type of channel.

We have implemented a very preliminary version of the Horde library, and are working on building a mobile video streaming system using this implementation. Thus far we have focused on implementing what we consider to be the two most interesting parts of the Horde framework: the channel managers and the packet scheduler. The implementation evaluated in this chapter uses stubs to simulate the simpler modules (such as the bandwidth allocator).

The bulk of this chapter deals with the Horde scheduler: how well our implementation allows QoS to be modulated for individual streams on real WWAN channels

(§7.1) and in simulation (§7.2); and the advantage gained by using a scheduler look-ahead mechanism within our system (§7.3). This chapter also introduces a network channel manager specially designed for `CDMA2000 1xRTT` channels (§7.4). Using experimental results from an actual WWAN channel, we argue that our specialized manager is superior to a more generic manager that cannot use *a priori* knowledge about `CDMA2000` channel characteristics.

7.1 QoS Modulation: Real Channels (`E1`)

In this section we report on an experiment (`E1`) conducted over actual WWAN channels. We show that a `random-walk` scheduler can be used to provide low-latency for specific data streams.

Implementation Note The `E1` experiment was conducted with an older version of the Horde implementation than the one used in the simulated-channel experiments (§7.2). Prediction logic for `txSlot`'s was sparse and there was no look-ahead logic. The scheduler implementation was a variant on the `random-walk` scheduler. We used the same generic AIMD congestion control scheme for all the channels.

Experimental Setup The `E1` experiment used a laptop computer connected to a single `CDMA2000 1xRTT` interface and multiple `GSM/GPRS` interfaces. All `GPRS` interfaces were connected to the same service provider; the `CDMA2000` interface was connected to a different provider. The `GPRS` interfaces used standard cell-phones connected to the laptop over a bluetooth link. The `CDMA2000` interface was a PCMCIA-based modem. The devices were in close proximity to one another and did not use specialized antennas.

The experiment reported here used three stationary WWAN interfaces. It consisted of sending as many 768 byte packets[1] as Horde's congestion control layer al-

[1]Our UDP experiments on WWAN channels in appendix A show that packet size affects both achievable throughput and packet latency. Generally: smaller packets experience lower latency; larger packets yield higher raw UDP throughput. 768 byte packets seem to be a good compromise.

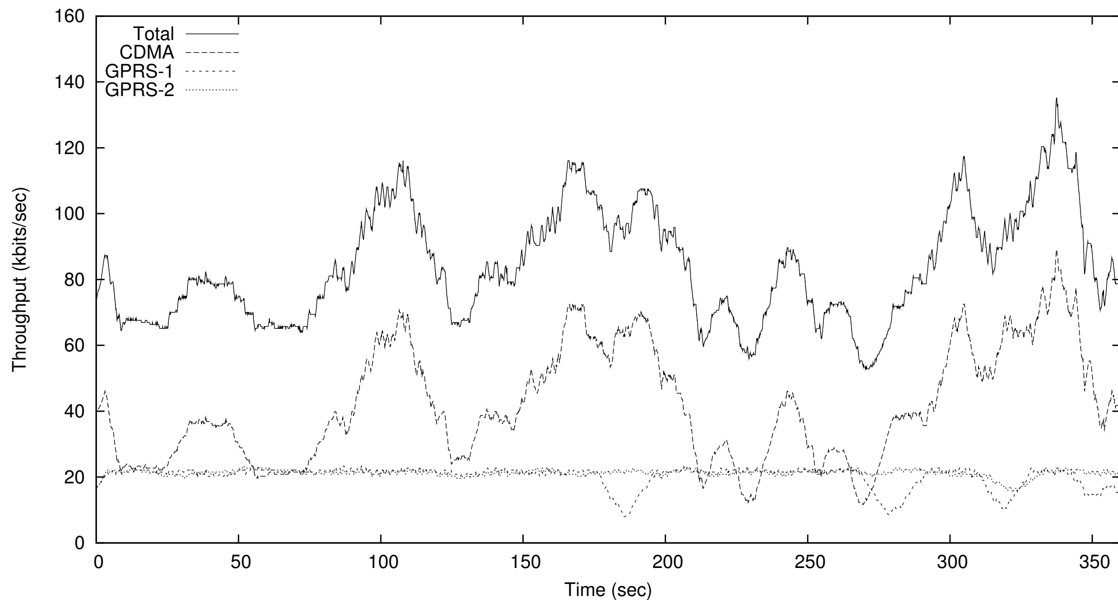


Figure 7-1: Throughput provided by Horde in the stationary experiment `E1`, using three co-located interfaces.

lowed, from the laptop to a host on the MIT ethernet. Packet traces were collected and used to simulate different schedulers.

Measured Throughput Figure 7-1 depicts the throughput provided. The throughputs were calculated using a 10 second sliding window average. The GPRS bandwidths are relatively stable in this experiment. This is consistent with previous experiments measuring raw UDP throughput (appendix A), which did not use Horde. Those experiments also indicate that vehicular motion leads to significant variability in throughput. We conjecture that the variability in CDMA throughput was either caused by contention with other users or due to the fact that our AIMD congestion control scheme is ill-suited to the channel, causing the throughput to oscillate. Other experiments—including those that investigated `verb` congestion-control later in this chapter—have shown the CDMA interface providing upload bandwidth of up to 120 kbits/sec on its own. Generally speaking, bandwidths on all of these WWAN interfaces varied greatly, over short time scales (as seen in the figure), over longer time scales, and geographically.

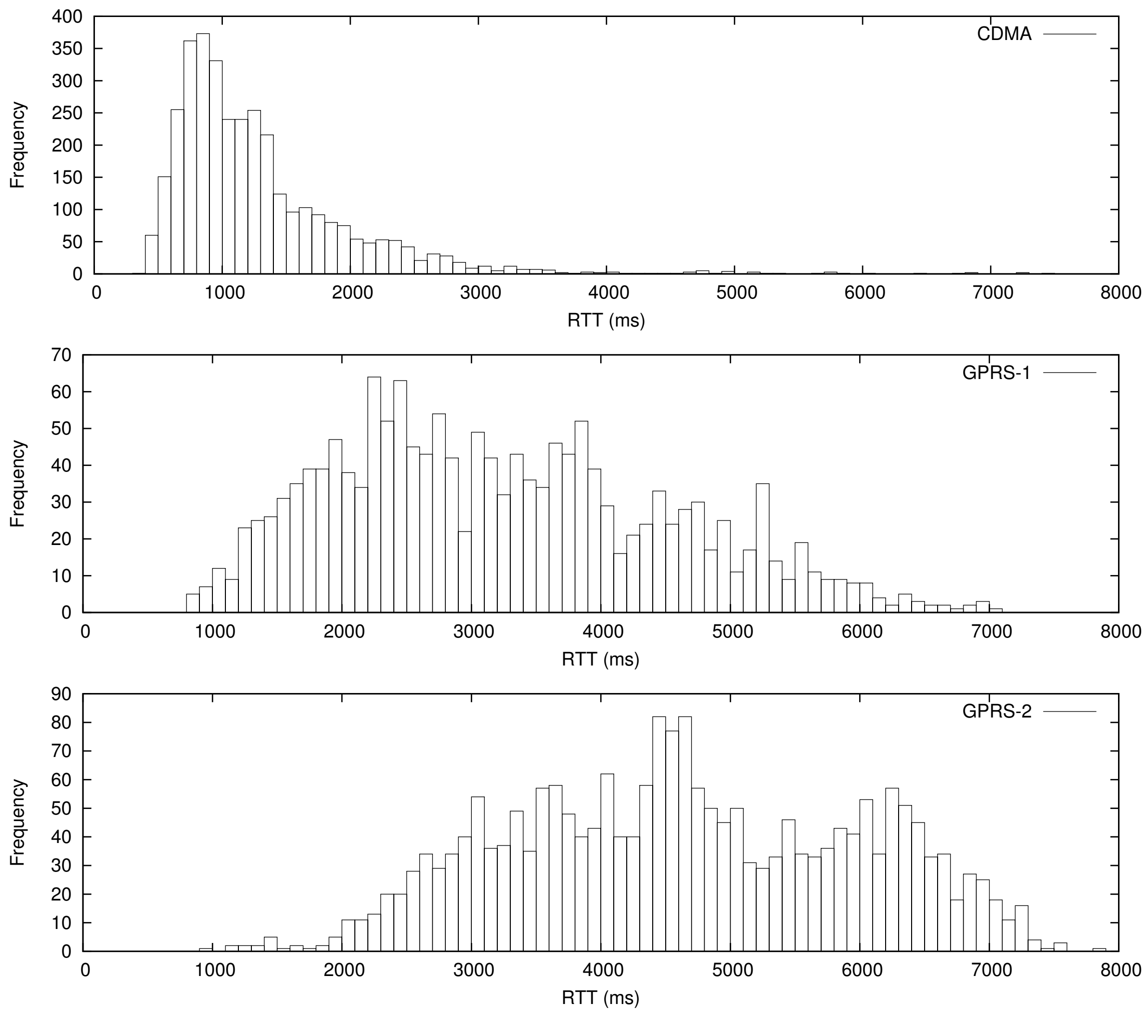


Figure 7-2: Packet round-trip-time distributions for each of the channels in the `E1` experiment. The three interfaces were co-located and stationary.

Measured Packet Latencies Figure 7-2 shows the packet latency distributions for each channel. The Horde congestion control layer was configured to always send acknowledgments back over the lowest latency channel (the CDMA channel in this case). Consequently, the GPRS distributions do not represent the packet round-trip-time distributions that would be seen for those channels in isolation. Typically, those are longer and more variable.

All of our observations, in this and in other experiments, show that the CDMA interface has significantly lower average packet latencies than the GPRS interfaces. However, the GPRS latencies reported here are higher than reported elsewhere. In

experiments conducted in other settings, we found that the median was usually within 2 seconds for 768 byte packets. We conjecture that the higher RTT's in this experiment may have been caused by a low quality wireless link to the GPRS base station. GPRS uses link layer retransmits, often elevating the RTT in order to mask channel losses [25, 14].

Channel Heterogeneity Figure 7-2a and 7-2c exemplify the very large differences that exist between two co-located WWAN channels using different technologies and providers. Figure 7-2b and 7-2c show that even two WWAN channels from the same provider and using the same technology can provide significantly different QoS. The existence of such marked differences highlights the potential impact of scheduling decisions on the network QoS observed by each application data stream.

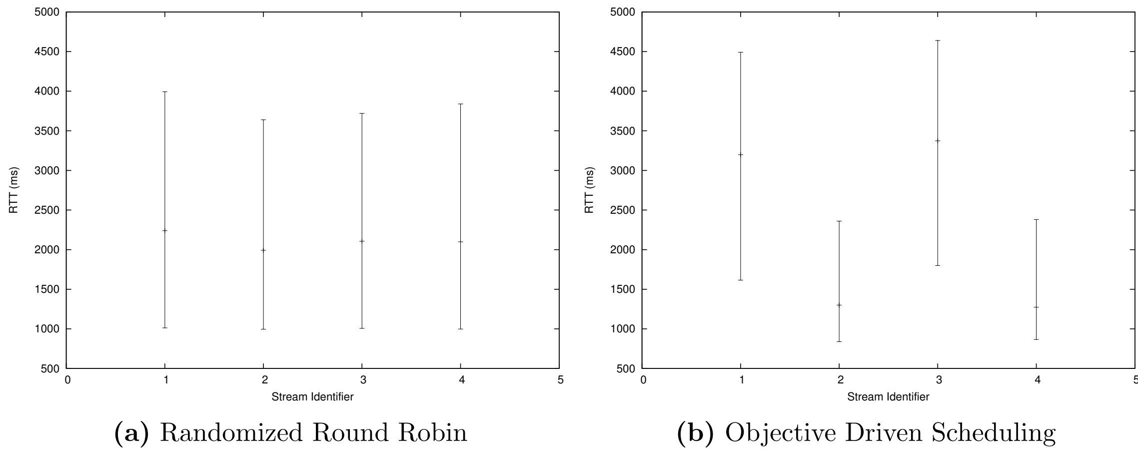
Scheduling The results of figure 7-3 were obtained by simulating the Horde scheduler on the packet traces from the above experiment, using simple channel models[2] and no look-ahead. Using the packet traces in this way, we could compare the effects of using different schedulers and policies under identical network conditions. Figure 7-3 does not include the queuing delay packets would experience within the library due to the cycle delay of the Horde scheduler; actual ADU latencies would therefore be slightly larger.

Figure 7-3 compares a randomized round robin scheduler to the schedule generated by Horde with a single objective that implied streams 2 and 4 were latency sensitive. We set up four data streams, each of which demanded and received the same amount of bandwidth from the `bwAllocator`.

As expected, when using randomized round robin, there were only small variations in the latency distributions across the four streams. Figure 7-3a shows the distribution of the latencies observed by each stream when using a round robin scheduler.

The objective driven scheduler was able to provide a much lower mean latency to the two latency sensitive streams at the cost of a slight increase in the mean latencies

[2] In the `txSlot` expectations, expected packet latency was calculated using a weighted moving average of known packet latencies. Loss probabilities were, similarly, averages.



(a) Randomized Round Robin

(b) Objective Driven Scheduling

Figure 7-3: Round-trip-time distributions with two different schedulers. The graphs show the median and the upper and lower quartiles for the round-trip-time experienced by ADU's in each stream.

```
objective {
   context { adu:foo { (stream_id == 2) || (stream_id == 4) } }
   goal { true }
   utility { foo { 5 * (1000 - expected(foo::latency)) } }
}
```

Figure 7-4: An objective expressing the fact that the utility derived by an application from an ADU from stream 2 or 4 falls off, or rises, linearly depending on how close the expected latency of that ADU is to one second.

of the other two streams. Figure 7-3b shows the distribution of observed latencies when the objective driven scheduler was used. No objective was specified for streams 1 and 3, indicating that the application wasn't sensitive to any aspects of QoS on those streams. Streams 2 and 4 were given the objective in figure 7-4. This objective states that added value is derived in proportion to how far the expected round-trip-time (RTT) deviates from one second; expected RTT's above one second yield negative values[3]. Based on this single objective, the scheduler was able to infer that it should preferentially assign transmission slots with an expected low latency to ADU's from streams 2 and 4. Figure 7-3b clearly shows that the scheduler recognized that QoS unfairness can be beneficial to the application.

[3]The choice of the constant 5 was arbitrary during the related experiment, and is ultimately unimportant, given that there are no other objectives.

Channel	Bandwidth	Latency μ	Latency σ	Loss	Loss Burst Size
cdma1	120 (kbits/sec)	600 ms	50 ms	0-1%	1, 2, 4, 8, and 16
cdma2	···	300 ms	15 ms	≈20%	···
gprs1	22 (kbits/sec)	900 ms	100 ms	1-5%	1, 2 and 4
gprs2	···	···	···	···	···
gprs3	···	1000 ms	100 ms	···	···
gprs4	···	···	···	···	···

Figure 7-5: The characteristics of the simulated channels.

7.2 QoS Modulation: Simulated Channels

In this section we report on experiments using simulated WWAN channels. These experiments investigate how well our scheduler can handle a set of objectives derived from our telemedicine application.

7.2.1 Simulation Setup

We felt it was necessary to use simulation to fully evaluate the performance of the Horde scheduler. Our simulated channels are based on measurements from actual WWAN channels (appendix A). Our use of simulated channels was motivated by the fact that the set of real channels available to us (a single `CDMA2000` interface connected to one provider and multiple `GPRS` interfaces connected to another provider) did not provide enough QoS modulation options to the scheduler. For instance, any sort of loss or latency QoS sensitivity would drive a stream to the `CDMA2000` channel. Using a more diverse set of simulated channels allowed us to better investigate the performance of the scheduler.

We simulated the set of channels shown in figure 7-5. `cdma1` was based on a model of a standard `CDMA2000 1xRTT` interface; `cdma2` was based on a model of a `CDMA2000 1xRTT` interface from a different provider with link-layer retransmissions disabled[4].

[4]For `cdma2` we have conjectured that disabling link level retransmissions on a `CDMA2000` channel causes losses to rise by an order of magnitude, but reduces the packet latency. Furthermore, we have assumed that the provider for `cdma2` has optimized their network for latency, unlike the `cdma1` provider. While this model may not be entirely accurate for any real-world `CDMA2000` channel, it provides us a high-loss low-latency channel in our simulations. An `802.11` interface would also provide a high-loss low-latency channel, but with an order of magnitude more bandwidth and an order of magnitude lower latency than `CDMA2000`.

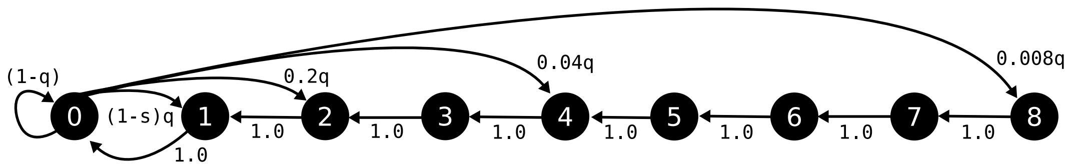


Figure 7-6: The Markov process used to model bursty packet losses on the simulated channels. q is the packet loss parameter and $s = \Sigma_{i=1}^{i \leq \lceil \log_2(max\ burst\ size) \rceil} \left(0.2\right)^i$.

The `gprs` channels were based on a model of `GSM/GPRS` channels. `gprs1` and `gprs2` were assumed to be from one provider, and `gprs3` and `gprs4` were assumed to be from another provider. The first `GPRS` provider was assumed to provide a slightly better average packet latency.

Each simulated channel was modeled using two random variables: T_i (round-trip-time) and L_i (loss). T_i was a Gaussian random variable[5] for the latency experienced by packet i on the channel. Statistics collected from actual WWAN channels (appendix A) show that the latency distributions are approximately Gaussian[6]. L_i was a boolean random variable denoting whether packet i was lost. Since our experiments with WWAN channels show a bursty loss nature, L_i was derived using the multi-state Markov process shown in figure 7-6. Whenever a simulated slot was generated, if the Markov process's state was a non-zero state, that slot had $L_i = true$. Furthermore, the process had a state transition after each generated slot. The given Markov process is equivalent to using a boolean random variable to determine if a burst-loss should start (with probability q of being $true$) and then an exponentially decaying random variable to determine how long that burst should last[7]. In the interest of simplicity, instead of arbitrary burst sizes we only used burst sizes that were powers of two. The possible burst sizes used for each simulated channel are given in figure 7-5.

Each experiment used a different packet trace. We ran the scheduler using channel managers that produced `txSlot`'s based on the simulated channel models described above. The packet traces for all the experiments had the same latency and loss distributions.

[5]Using the means and standard deviations given in figure 7-5 for each channel.
[6]With means and variances similar to those noted in figure 7-5.
[7]The exponential decay is based on our measurements in appendix A.

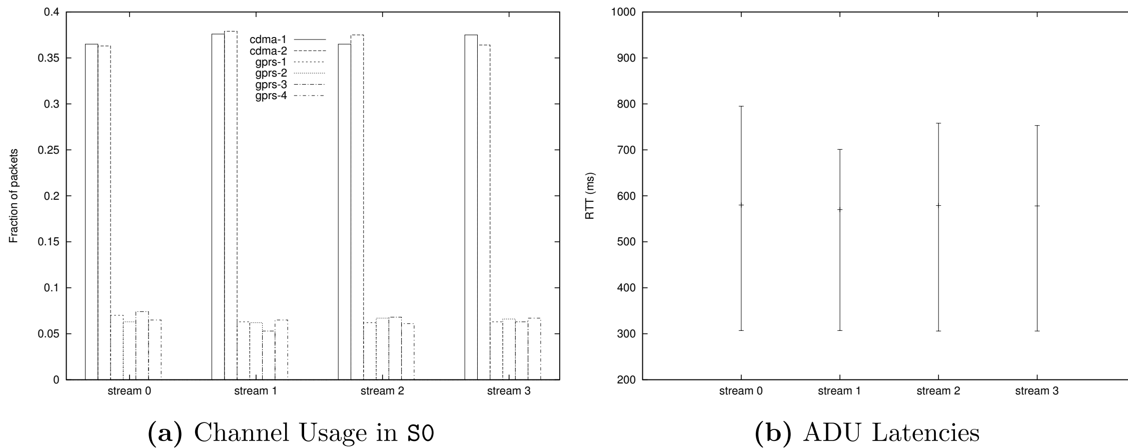
The channel managers for the simulated channels produced `txSlot`'s with a constant-bit-rate (τ). In `E1` `txSlot`'s were produced based on the `ack` arrival times. When asked to provide phantom slots for λ milliseconds into the future, the simulated channel managers used their generation rate to figure out how many phantom slots should be returned ($\lfloor \frac{\lambda}{\tau} \rfloor$).

Both phantom and normal `txSlot` expectations were initialized using simple models. An exponentially-weighted-moving-average of past round-trip-times was used for latency, and loss probabilities were derived from a simple average ($\frac{losses}{total}$). The channel managers used a exponentially decaying estimator, based on the Markov model in figure 7-6, to determine the correlated loss probability for two transmission slots. Slots from different channels were predicted to be perfectly uncorrelated (probability of zero): adjacent slots from the same channel were predicted to be perfectly correlated (a 100% correlated loss probability); and slots more than eight transmissions apart on the same channel were also predicted to be perfectly uncorrelated.

The simulations documented here used pre-compiled objectives. We implemented a `hose`-subset interpreter using `lex/yacc` but have not integrated it into the `random-walk` scheduler. Interpretation would be functionally equivalent, so the simulation results would not be affected.

7.2.2 Randomized-Round-Robin (`S0`)

The first simulated experiment, `S0`, reported here used a randomized-round-robin scheduler on the simulated channels from figure 7-5. The results of `S0` establish a stream QoS baseline reference for the subsequent simulations. Four streams were created, each requesting the same throughput. Figure 7-7a shows how the data from these streams was striped over the six available channels. Figure 7-7b shows the streams' ADU latency distributions. All streams have similar latency distributions because the scheduler stripes each stream in roughly the same way. The `cdma1` channel carries about six times the amount of data as the `gprs1` channel for each stream because of the bandwidth ratio of the two channels.



(a) Channel Usage in `S0` **(b)** ADU Latencies

Figure 7-7: Experiment `S0` results. The `RRR` scheduler (a) stripes every stream in the same way over the available channels; (b) giving similar latency distributions.

```
objective {
  context { adu:foo { (stream_id == 0) || (stream_id == 3) } }
  goal { expected(foo::latency) <= 700 }
  utility { foo { 700 - expected(foo::latency) } }
}
```

Figure 7-8: The low-latency objective `X1`. Limited to streams `0` and `3`, this objective assigns `0` utility to an ADU with an expected round-trip-time greater than $700ms$; and linearly increasing utilities for ADU's with lower round-trip-times.

7.2.3 Policy: Low Latency (`S1`)

The simulated experiment `S1` used the low-latency objective `X1` (The `hose` form is given in figure 7-8), to drive down the median round-trip-times for streams `0` and `3`. `S1` is similar to the earlier experiment `E1`. Figures 7-9, 7-10, and 7-15a show the relevant results from the experiment. No other objectives were used.

`X1` causes the scheduler to reduce latencies for streams `0` and `3`. Comparing figure 7-11a to the randomized-round-robin case (figure 7-7b) shows that median latencies for streams `0` and `3` are reduced in `S1`. The latencies of the other two streams rise.

Figure 7-10b shows that the objective `X1` causes slots from the lowest latency channel `cdma2` to be shifted from streams `1` and `2` to the latency-sensitive streams `0` and `3`. Because of the bandwidth allocator's fairness constraints, the latency sensitive streams must give up some of their slots on the medium latency `cdma1` channel.

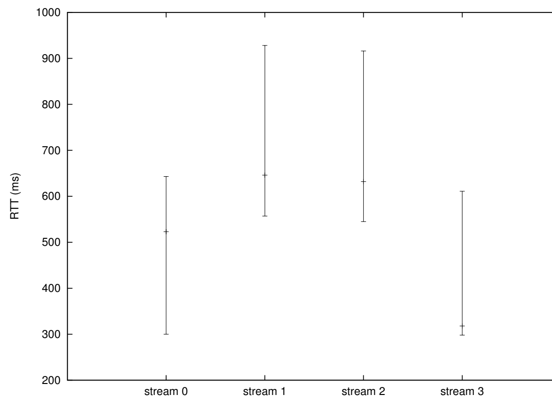
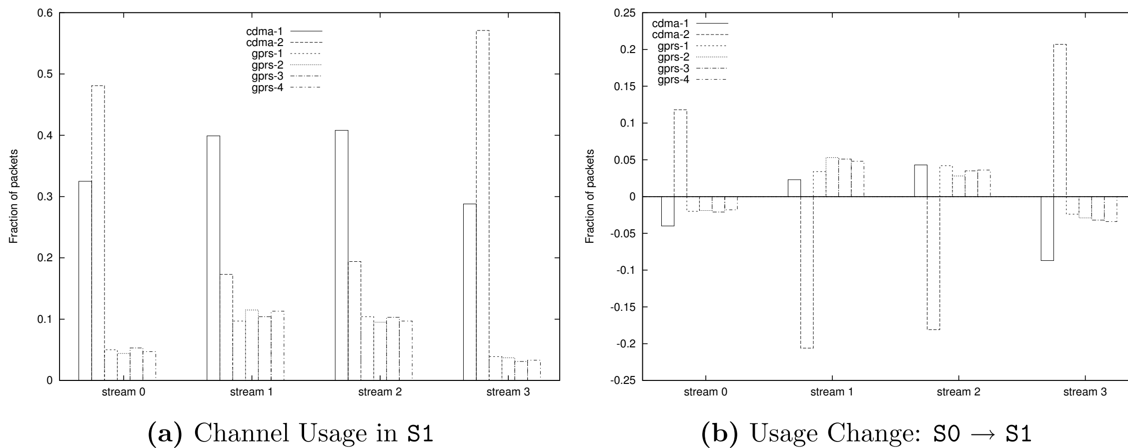


Figure 7-9: Round-trip-time distributions in experiment `S1`. The graph shows the median and the upper and lower quartiles for the stream ADU RTT's.

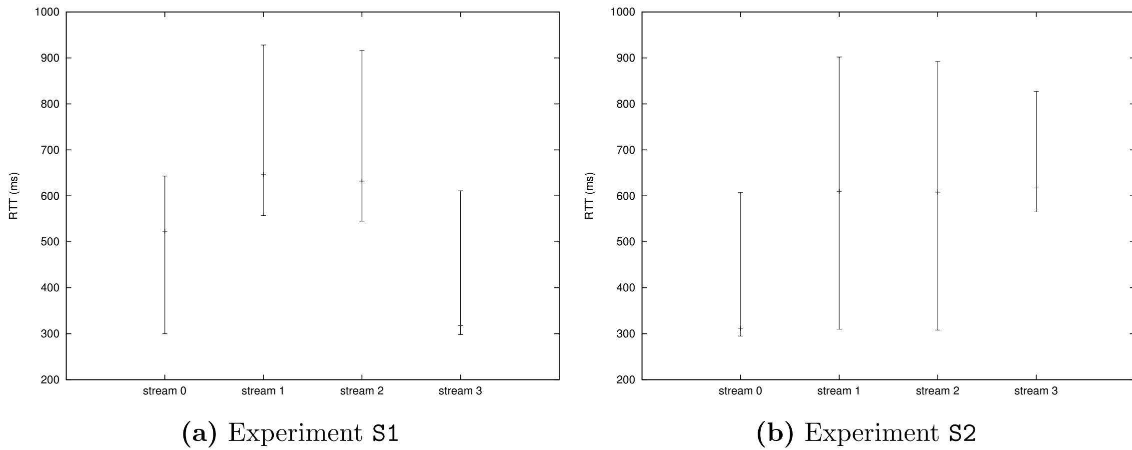


(a) Channel Usage in `S1` **(b)** Usage Change: `S0` $\rightarrow$ `S1`

Figure 7-10: Channel usage in experiment `S1`. Compared to `S0`, slots from the lowest-latency channel are shifted to the latency sensitive streams. This is shown by a negative `cdma2` usage change for streams `1` and `2`, and a positive change for `0` and `3`.

Our scheduler is not perfect. Even though streams `0` and `3` have approximately the same inter-quartile range and both upper quartiles are below the $700ms$ specified in the objective, nevertheless the fact that stream `3` is given more slots from the `cdma2` channel than stream `0` exposes the suboptimality of the `random-walk` scheduler. This suboptimality is not significant because, although median latencies are about $250ms$ apart, the mean latencies are close, within $50ms$ ($\mu_0 = 524ms$ and $\mu_3 = 476ms$)[8].

[8] The large difference in median stream RTT's is likely due to the large $300ms$ difference in mean latency between `cdma1` and `cdma2`.



(a) Experiment `S1`

(b) Experiment `S2`

Figure 7-11: Round-trip-time distributions in experiments `S1` and `S2`.

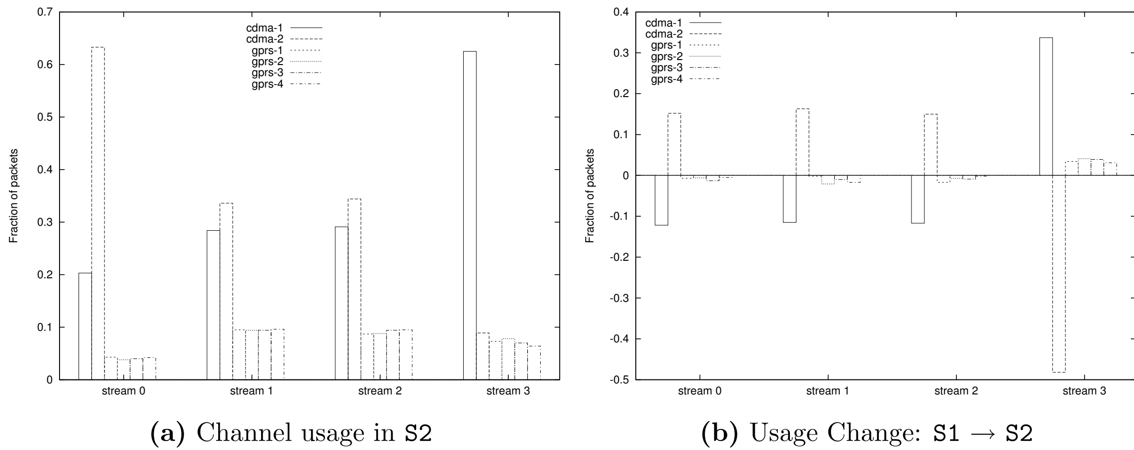
```
objective {
  context { adu:foo { stream_id == 3 } }
  goal { prob(foo::lost?) < 10 }
  utility { foo { 100 * (prob(foo::lost?) - 10) } }
}
```

Figure 7-12: The low-loss objective `X2`. This objective specifies that if an ADU from stream `3` has a probability of loss less than 10%, its utility rises linearly with the probability that it is delivered.

Furthermore, an *optimal* scheduler would send all ADU's from the latency sensitive streams down the `CDMA` channels. The `random-walk` scheduler sends a small proportion of latency sensitive ADU's down the high-latency `GPRS` channels. Part of this problem might be due to the way in which our implementation handles the bandwidth allocator's constraints: if a stream x hasn't been serviced for a while (e.g., because of phantom slot related delays), x will get priority in the next scheduling cycle. This might result in streams being assigned `CDMA` slots even though they shouldn't be.

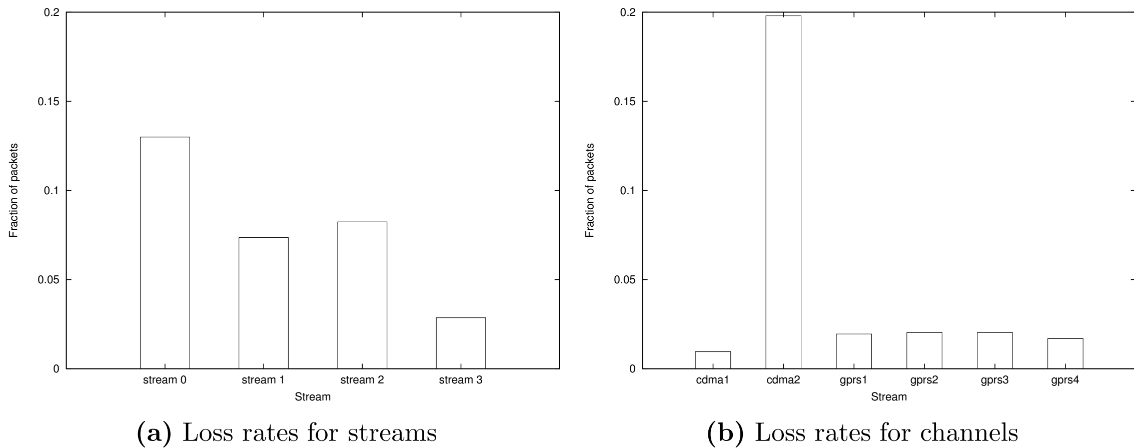
7.2.4 Policy : Low Latency and Low Loss (`S2` and `S3`)

The simulated experiment `S2` used the objective `X1`, as before, and additionally used the objective `X2` (figure 7-12) to specify that stream `3` was sensitive to losses. Figures 7-11b, 7-13, 7-14 and 7-15b show the relevant results from the experiment.



(a) Channel usage in `S2` **(b)** Usage Change: `S1` → `S2`

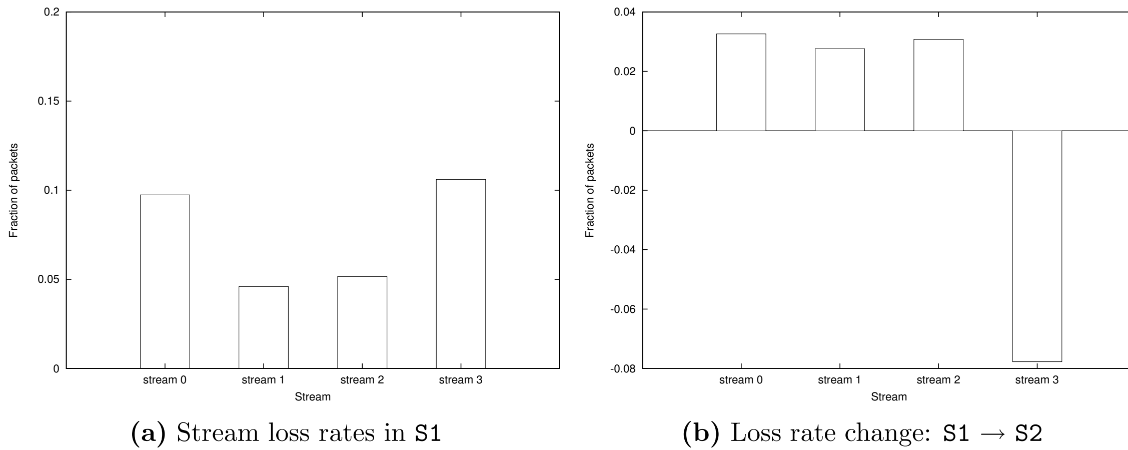
Figure 7-13: Channel usage for streams in experiment `S2`. The loss and latency sensitive stream `3` is assigned slots on the low-loss medium-latency channel `cdma1` in exchange for slots on the lowest-latency channel `cdma2`. All other streams gain slots on `cdma2` and lose slots on `cdma1`.



(a) Loss rates for streams **(b)** Loss rates for channels

Figure 7-14: Loss rates in experiment `S2`.

The existence of `X2` for stream `3` clearly reduces the losses on that stream. Figure 7-14a shows that stream `3` has the lowest loss-rate of all the streams in `S2`. Compared to `S1`, the scheduler assigns far fewer stream `3` ADU's to the `cdma2` channel in `S2` (see figure 7-13b). The `cdma2` channel has an order of magnitude higher packet loss-rate than the other channels (see figure 7-14b), so moving stream `3` ADU's away from `cdma2` reduces that stream's loss-rate and raises the overall utility. Figure 7-15b shows that compared to `S1`, loss rates for all the other streams increase, as low-loss slots are shifted over to stream `3`.



(a) Stream loss rates in S1

(b) Loss rate change: S1 → S2

Figure 7-15: Comparing loss rates in experiments S1 and S2.

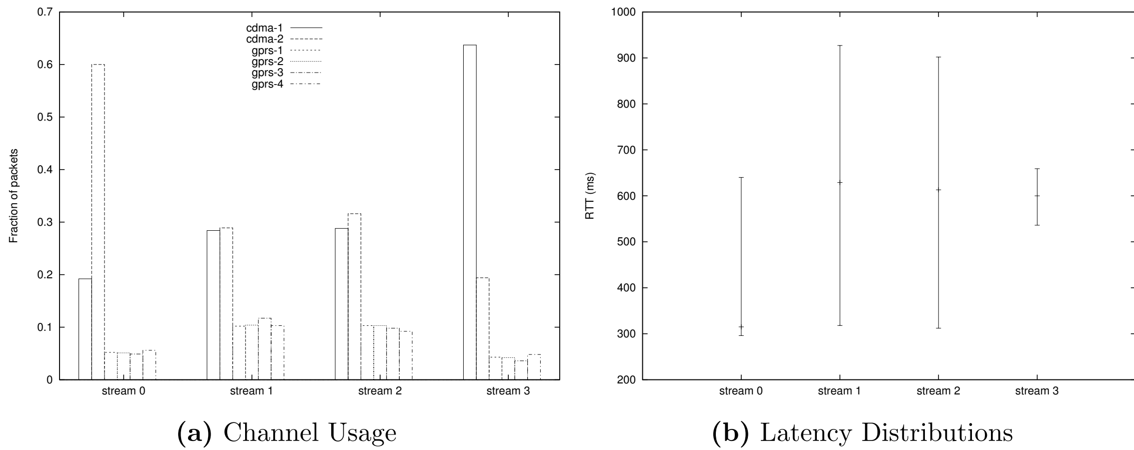
```
objective {
  context { adu:foo { stream_id == 3 } }
  goal { expected(foo::latency) <= 700 }
  utility { foo { 700 - expected(foo::latency) } }
}
```

Figure 7-16: The low-latency objective X3. Identical to X1, except for the fact that this objective is restricted to stream 3.

While the average latency on stream 3 is still below $700ms$[9] because of the objective X1, figure 7-11b shows that a significant number of ADU's experience latencies higher than $700ms$. Even though X1 specifies that low-latency slots provide higher utility on stream 3, the existence of X2 causes the scheduler to sometimes use the low-loss high-latency GPRS channels for stream 3 ADU's. This scheduling choice either represents the suboptimality of the random-walk scheduler or is a consequence of X2 dominating in the determination of utility for the stream.

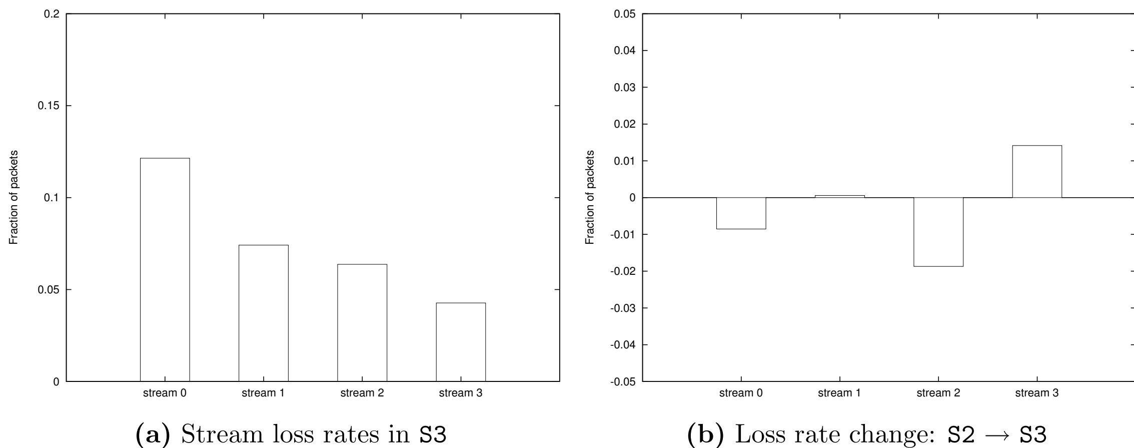
To investigate this, the simulated experiment S3 used the X3 objective (figure 7-16) in addition to the other two. The X3 objective further increases the utility gained on stream 3 from low-latency transmissions. Figure 7-17a shows that in S3 the scheduler assigned most ADU's from the latency-sensitive stream 0 to the high-loss low-latency cdma2 channel, and most ADU's from the loss-and-latency-sensitive stream 3 to the

[9]The mean is $673ms$ and the median is $617ms$.



(a) Channel Usage **(b)** Latency Distributions

Figure 7-17: Channel usage and latency distributions for streams in experiment S3.



(a) Stream loss rates in S3 **(b)** Loss rate change: S2 → S3

Figure 7-18: Loss rates in experiment S3.

low-loss medium-latency cdma1 channel. Figure 7-17b shows that the use of X1 and X3 kept most stream 3 ADU's latencies below $700ms$. Figure 7-18 shows that in S3 stream 3 still has the lowest loss rate, but this loss rate is slightly higher than it was in S2 (shown clearly in figure 7-18b, which compares figure 7-18a to figure 7-14a).

An optimal scheduler would always send ADU's from stream 3 on cdma1 and ADU's from stream 0 on cdma2. It would never use the GPRS channels to send latency-sensitive ADU's. As before, in S3 the random-walk scheduler sends a small fraction of latency sensitive ADU's in high-latency slots. The scheduler also occasionally uses high-loss low-latency slots for loss-and-latency sensitive ADU's.

```
objective {
  context { adu:foo { stream_id == 0 }
            adu:bar { stream_id == 0 }
  }
  goal { correlated_loss(foo, bar) <= 25 }
  utility { foo { 125 } }
}
```

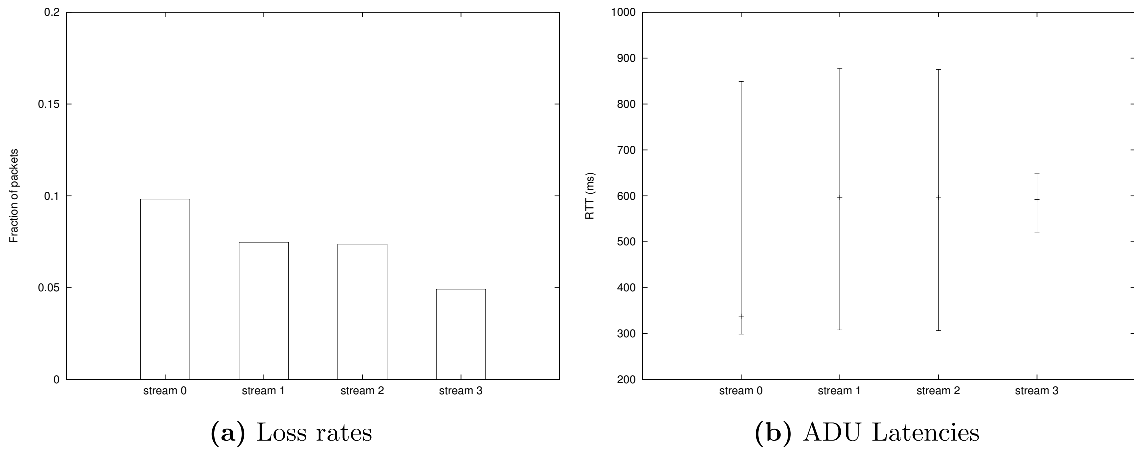
Figure 7-19: The low-correlated-loss objective `X4`. This objective adds positive utility whenever two ADU's from stream `0` have a correlated loss probability below 25%.

7.2.5 Policy: Low Correlated Loss (`S4`)

Correlated loss[10] objectives are more complicated than objectives that prioritize either loss or latency, such as `X2` and `X3`. For instance, the objective `X4` (see figure 7-19) specifies a desired relationship between two ADU's in a transmission schedule. Earlier objectives like `X3` specify a property that can be derived directly from a single transmission slot, without any knowledge about which other ADU's occupy slots in the candidate transmission schedule. The existence of combinatorial objectives such as `X4` complicates the use of a directed search inside the scheduler. The need to handle multi-ADU objectives such as `X4` is one of the reasons behind the zero-knowledge approach used by the `random-walk` scheduler. If only single-ADU objectives like `X3` were allowed, relatively simple, deterministic schedulers could be constructed that would be more accurate than the `random-walk` scheduler in situations like `S1`.

The primary mechanism the `random-walk` scheduler can use to reduce correlated losses for a stream is to spread ADU's from that stream over multiple channels. The scheduler can always determine that correlated losses can be reduced by spreading ADU's from a stream over multiple channels (comparing `txSlot`'s from different channels for loss correlation always returns 0; see simulation notes below). The use of phantom slots ensures that the scheduler considers transmission slots from all channels in every cycle. Since the scheduler works with a relatively small number of slots[11] for each channel in a scheduling cycle, the scheduler is not likely to reduce correlated

[10] Earlier, we defined a *correlated loss* for ADU's `X` and `Y` as the event that both ADU's were lost.
[11] In our simulations, the scheduler's window was smaller than the size of the maximum burst-loss.



(a) Loss rates

(b) ADU Latencies

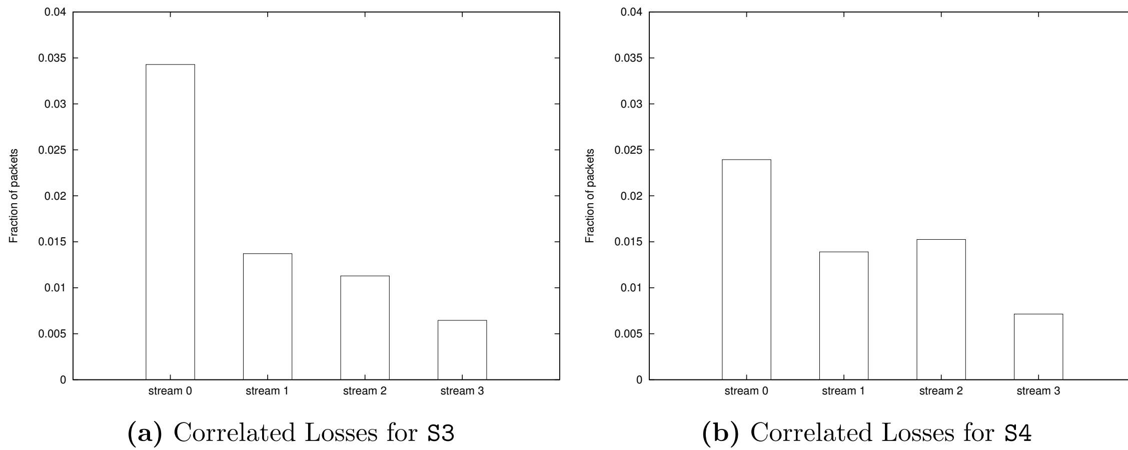
Figure 7-20: Stream loss rates and ADU round-trip-time distributions for `S4`.

losses by spacing out ADU's temporally on the same channel. In the absence of any objectives, the randomization inherent in the scheduler spreads streams over channels by default.

Simulation Notes Our simulated channel models assume that the channels are not correlated with each other. Consequently, in our simulations, correlated losses mainly occurred as a result of the bursty-loss nature of the simulated channels. In reality, WWAN channels do seem to exhibit cross-channel correlation (see appendix A) but—lacking a reasonable analytical model for this cross-channel correlation—we chose not to model it in our simulations. For simulation, we assumed that all channels were from different providers, connected to different base-stations, and therefore exhibited temporally uncorrelated losses.

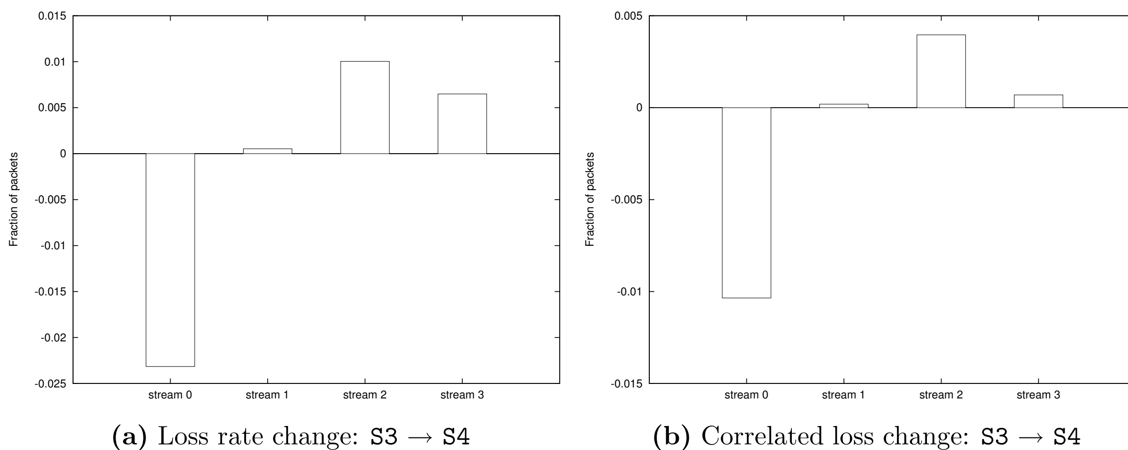
Since our correlated loss model may not be entirely realistic, the results in this section should be viewed as demonstrating the `random-walk` scheduler's ability to handle combinatorial objectives in general rather than as this scheduler's ability to handle correlated loss objectives in particular.

The simulated experiment `S4` added the low-correlated-loss objective `X4` (figure 7-19), in addition to `X1`, `X2` and `X3` from the previous experiment `S3`. Figures 7-20, 7-21, 7-22 and 7-23 show results from the experiment.



(a) Correlated Losses for S3 **(b)** Correlated Losses for S4

Figure 7-21: The fraction of stream packets lost in correlated losses in S3 and S4.

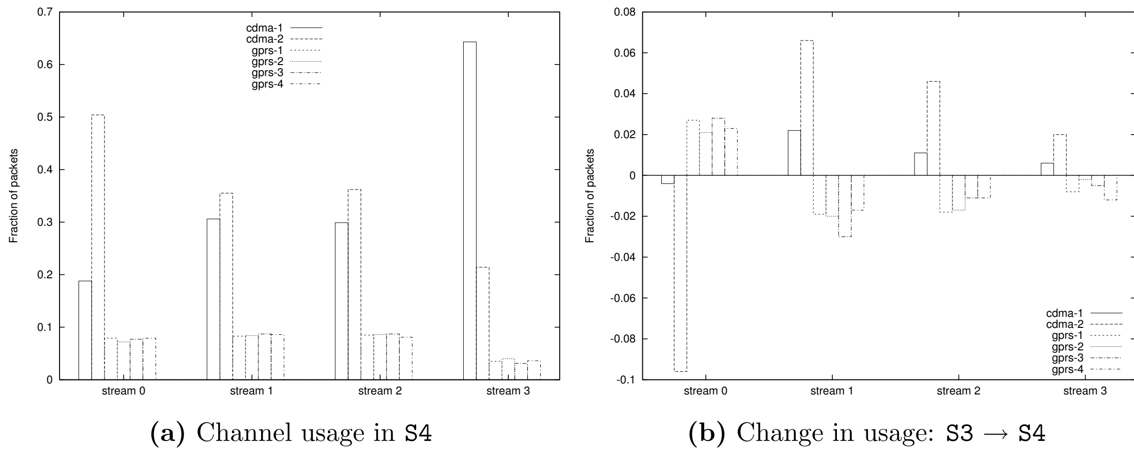


(a) Loss rate change: S3 → S4 **(b)** Correlated loss change: S3 → S4

Figure 7-22: Comparing losses on the streams in S3 with losses in S4.

The introduction of X4 drove down both the correlated losses and overall losses for stream 0. Figures 7-21 and 7-22b show that the fraction of correlated losses in S4 was roughly two-thirds the fraction of such losses in S3. Additionally, as figure 7-22a shows, reducing the correlated losses on stream 0 caused the overall loss rate to fall. The change in the loss rate was about twice the change in the correlated loss rate.

X4 caused ADU's from stream 0 to be spread out over the GPRS channels to reduce correlated losses (see figure 7-23). In S3, stream 0 ADU's were concentrated on the medium-latency cdma2 channel because of X1. When a stream's ADU's are transmitted on the same channel, multiple ADU's from that stream can be lost together in



(a) Channel usage in `S4` **(b)** Change in usage: `S3` → `S4`

Figure 7-23: Channel usage in the experiment `S4`. Compared to `S3`, almost 10% of the stream `0` ADU's are moved from the `cdma2` channel and spread over the `GPRS` channels, in order to reduce correlated losses.

a channel burst-loss. Because of `X2` and `X3`, displacing stream `3` ADU's from `cdma1` would have reduced the overall utility. Stream `0` ADU's were therefore transmitted on the `GPRS` channels. The objective `X1` was not met for any `GPRS` transmission slots due to their high latencies, but the utility gained from uncorrelated losses (`X4`) offset the utility lost by raising the latency of some ADU's.

Figure 7-20b shows the latency distributions of the four streams in `S4`. In `S4`, the ADU latency distribution of stream `0` was still centered at a low latency, but compared to `S3`, it had a longer tail. The use of the `GPRS` channels to reduce correlated loss caused the upper quartile latency of stream `0` to rise above the $700ms$ goal from `X1`.

We found that correlated losses could be reduced further by tweaking `X4`. For instance, using 150 instead of 125 halved correlated losses for stream `0`, but shifted the median latency for that stream up by $200ms$. Using a value of 300 caused `X4` to dominate the policy for stream `0`, causing all streams other than stream `3` to have the same latency distribution. Using 300 in `X4` implied that stream `0` received as much utility by avoiding a likely correlated loss as it did by using a slot with an expected latency of $400ms$. `X1`, `X2` and `X3` together prevented `X4` from displacing stream `3` from the low-latency slots.

An optimal scheduler would probably not be able to do much better. Ignoring

the use of temporal spacing, spreading out stream `0` ADU's onto the high-latency low-loss `GPRS` channels is probably optimal. `X3` and `X2` effectively reserve the low-loss low-latency `cdma1` for stream `3`. Therefore stream `0` correlated losses can only be reduced by interleaving that stream's ADU's on `cdma2` and the `GPRS` channels. .

7.3 Impact of Varying Scheduler Look-Ahead

To quantify the effects of using phantom `txSlot`'s, we ran a large number of simulated experiments, in which we varied the scheduling cycle period and the look-ahead time. We used a single low-latency objective similar to `X3`[12], four streams and the earlier channels. We set the loss rates and latency variances to zero on all the channels to keep the experiments simple. The mean latency achieved by `X3` was used as the metric of scheduler accuracy.

Without the use of phantom `txSlot`'s the `random-walk` scheduler can degenerate to perform as badly as the randomized-round-robin scheduler. For example, if the cycle period is $50ms$, most scheduling cycles might never consider the `GPRS` slots as candidates for less important ADU's. This is because the `CDMA` channels each generate one slot every $48ms$[13] and the `GPRS` channels each generate one slot every $279ms$[14].

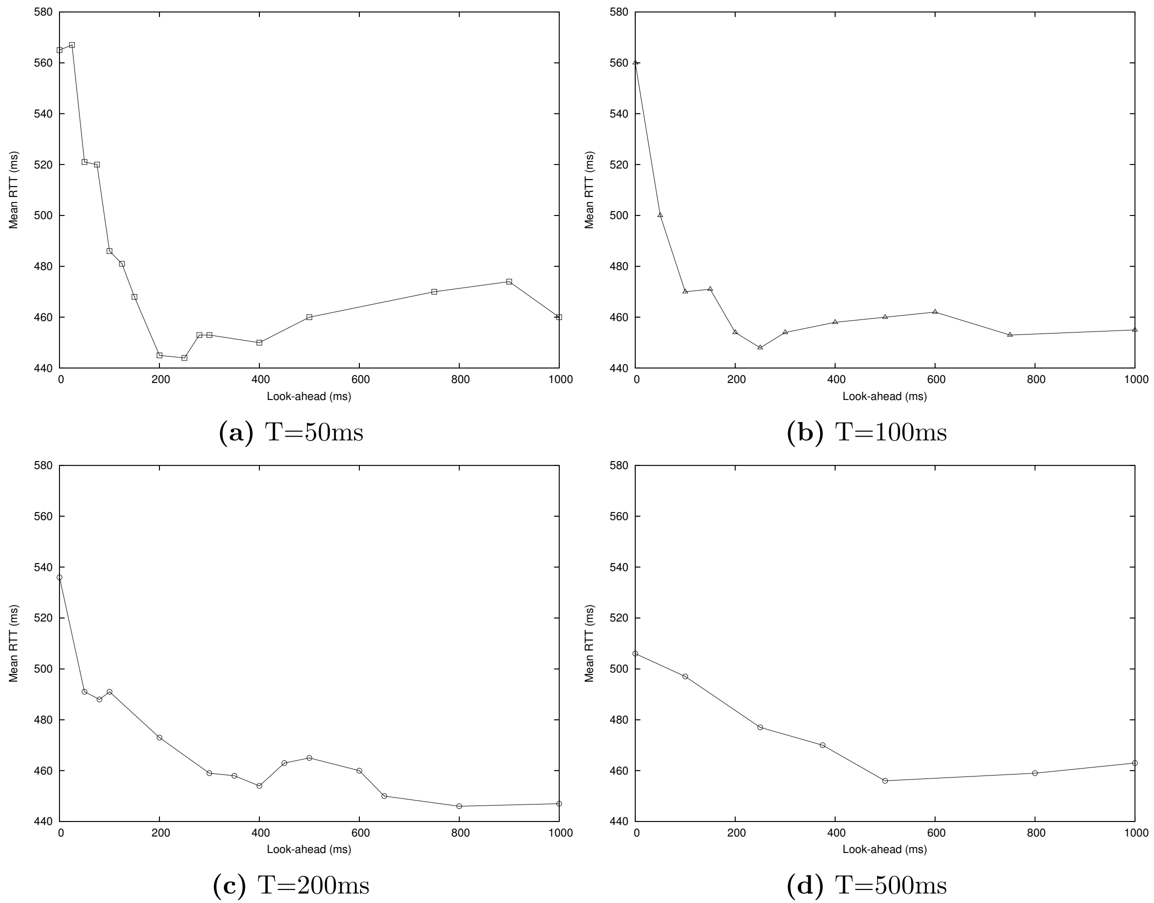
We were interested in how low an average latency the scheduler would provide to the only latency sensitive stream. Figure 7-24 shows how this average ADU latency varied with the two scheduler parameters (cycle period and look-ahead duration).

Our experiments demonstrate that some look-ahead is advantageous. With no look-ahead and a scheduling period less than $200ms$, the `random-walk` scheduler performed almost as badly as the randomized-round-robin scheduler. Adding look-ahead made the scheduler perform significantly better. As we increased look-ahead, for all the cycle periods in figure 7-24, the accuracy of the scheduler initially improved. This improvement was dramatic for cycle periods less than $500ms$. Look-ahead periods

[12] In these experiments we replaced 700 with 500 in the objective, to make it so that the objective would not be met on `cdma1`.

[13] 120 kbits/sec with 768 byte packets.

[14] 20 kbits/sec with 768 byte packets.



(a) T=50ms **(b)** T=100ms

(c) T=200ms **(d)** T=500ms

Figure 7-24: The results of varying look-ahead in the `random-walk` scheduler with a solitary low-latency objective. The graphs show—for various scheduling cycle periods—the mean ADU round-trip-time for the stream with the objective. The upper-bound of $580ms$ is the mean RTT with a randomized-round-robin scheduler.

greater than $280ms$[15], didn't seem to increase the accuracy of the scheduler as much.

In figure 7-24, all the graphs show that additional look-ahead stops being useful at some point, and beyond that point the mean RTT begins to rise. We believe this has to do mainly with the details of our implementation: the stream bandwidth constraints cause the scheduler to make sub-optimal decisions if it determines that some ADU's have been delayed in phantom slots for too long. Furthermore, because of the nature of the bounded random walk used by our scheduler, we expect scheduler accuracy to deteriorate when the number of phantom slots becomes too large. With more slots, the random search covers a smaller fraction of the possible schedules, so it

[15]One phantom slot per `GPRS` channel.

is less likely to find the better schedules. Nonetheless, figure 7-24 clearly shows that even a large amount of look-ahead is much better than having no look-ahead at all.

These graphs plot the mean *network* latencies for stream ADU's, not the actual latencies experienced by the application. Actual latencies include the queuing delays within the Horde library. Longer scheduling cycle periods can reduce the rate at which streams are serviced, increasing the queuing delay.

With respect to overall ADU latency, a shorter scheduling cycle period is preferable to a longer period. For instance, while using a $500ms$ cycle period without look-ahead can lower the network latency of the stream reasonably well, the queuing delay is likely to significantly delay transmissions. A transmission slot may be delayed up to $500ms$ before having an ADU assigned to it.

Processing cost concerns may dictate that a very small scheduling cycle period cannot to be used. In our earlier experiments, `S0` through `S4`, we had used a $100ms$ cycle period with look-ahead in the neighbourhood of $300ms$.

7.4 Network Channel Managers: CDMA2000 1xRTT

This section sketches out the techniques that can be used to implement a Horde network channel manager optimized for `CDMA2000` channels. This implementation incorporates a congestion control mechanism that looses fewer packets than TCP-style AIMD in the steady-state and seems to do as well when the channel is not providing a stable bandwidth. Furthermore, we outline effective channel prediction mechanisms that work well for `CDMA2000` channels.

This `CDMA2000` manager would not necessarily work well for other types of channels. Both our congestion control and latency prediction mechanisms make use of the fact that distinct queuing artifacts are normally visible in a sequence of round-trip-times for back-to-back packets on a `CDMA2000` channel (see figure 7-25 and appendix A). These kinds of artifacts are normally not present on other types of channels.

Other researchers have proposed congestion control schemes optimized for `GPRS` channels [38, 15]. Such schemes can be incorporated into the `GPRS` channel manager.

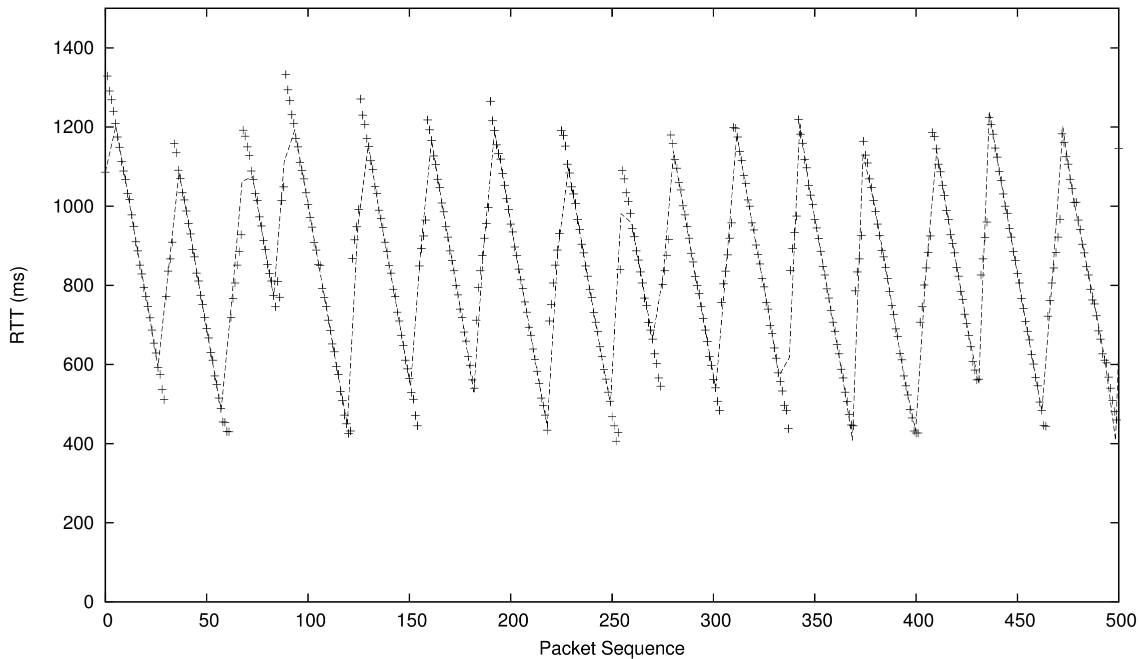


Figure 7-25: Queuing on a CDMA2000 channel using a constant-bit-rate sender.

7.4.1 Congestion Control: `verb`

We outline a congestion control scheme for CDMA2000 channels that performs better than generic AIMD congestion control when the channel is stable (e.g., when stationary) and about as AIMD the same when the channel is unstable. Our main goal is to minimize losses and avoid throughput oscillation on a stable channel.

The `verb` implementation is meant to be a proof-of-concept (designed and implemented over three days). There is significant room for improvement. We discuss it here to demonstrate that a channel specific congestion control scheme can outperform generic congestion control by using *a priori* knowledge about channel characteristics.

`verb` provides a high average goodput[16], without oscillating unnecessarily when the channel is stable[17]. Figure 7-26 shows the measured goodput achieved by different

[16]Goodput here is the number of bytes whose reception was successfully acknowledged by the network peer.

[17]`verb` parameters allow trade-offs to be made between steady state packet losses and the aggressiveness of the bandwidth probing. The parameters can be changed dynamically while the channel manager is running, from the horde daemon terminal window. `verb` parameters can be optimized for whether the application is mobile or stationary.

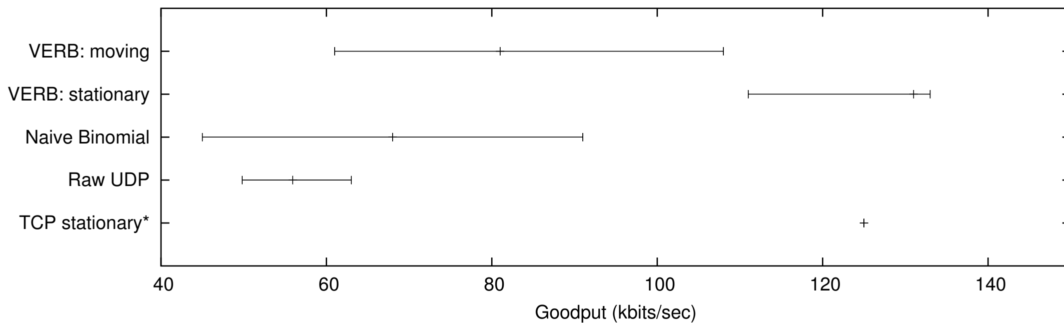


Figure 7-26: Comparing `verb` goodput to other congestion control schemes. This graph shows the distribution of a windowed-average for each scheme.

congestion control schemes on a real `CDMA2000` channel. The median goodput of `verb` when stationary is comparable to the average goodput during a bulk TCP transfer. Figure 7-27 shows that the goodput does not vary as much as an AIMD scheme would (e.g., see figure 7-1), primarily because `verb` does not back off from the 120 kbits/sec sending rate unless the network can no longer support such a high rate. When moving, `verb` varies about as much as a naive binomial congestion control implementation [11] does when stationary, but `verb` has a higher median goodput. Raw UDP gives us the lowest median goodput; most packets are lost. The achieved UDP rate is the same whether we send at a constant-bit-rate of 60 or 120 kbits/sec. It is possible that senders who refuse to back-off are penalized by the base-station we were next to during these experiments. Another possibility is that a high sending rate pushed the `CDMA2000` channel into an abnormal network state (appendix A), and without back-off the channel did not recover.

`verb` uses a coarse timer to periodically reset it's sending rate to the measured goodput from the previous period. The notion is to use feedback to stabilize to a sending rate to a rate not greater than the available bandwidth. In this way `verb` uses increasing packet round-trip-times as a signal that queues in the network are building up, backing off preemptively instead of waiting for a loss to occur. This avoids the unnecessary losses resulting from the constant probing used by generic AIMD TCP-style network congestion control. Using latency as a congestion signal is not a new idea [13, 33], but `verb` has been tailored to our `CDMA2000` channel, by

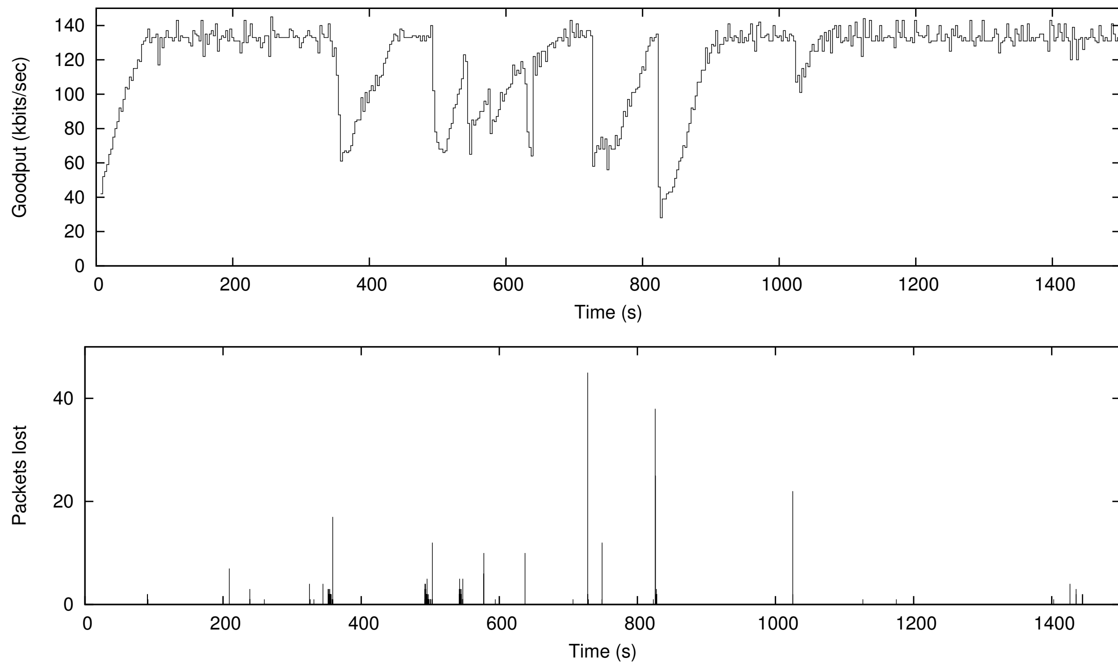


Figure 7-27: Packet losses and goodput in a stationary experiment using `verb` on an actual `CDMA2000 1xRTT` channel.

incorporating constant factors derived from our knowledge about such channels.

When the `verb` sending rate stabilizes—when `verb` is not sending more data than the network can support—`verb` tries to probe for additional bandwidth. There are two ways `verb` probes for bandwidth: non-aggressive AIMD; and by trying to move to the next throughput *plateau*.

`verb` can be parameterized with a set of plateaus[18]. The algorithm tries to stabilize the sending rate around a plateau. It always tries to edge upwards, using AIMD within the plateau boundaries, restrained by feedback containing increasing packet latencies. Once it has stabilized, the algorithm tries to ramp up, to see if the next plateau is viable. Once it has reached the top plateau, bandwidth probing is essentially turned off, avoiding unnecessary packet losses. Probing for more bandwidth than the `CDMA2000` specification supports is pointless.

Figures 7-27 and 7-28 show the dynamic behaviour of a network channel manager using `verb` congestion control for a `CDMA2000` channel in both stationary and mobile

[18] For `CDMA2000`, plateaus placed roughly 5 kbits/sec apart, ranging from 5 to 133 kbits/sec.

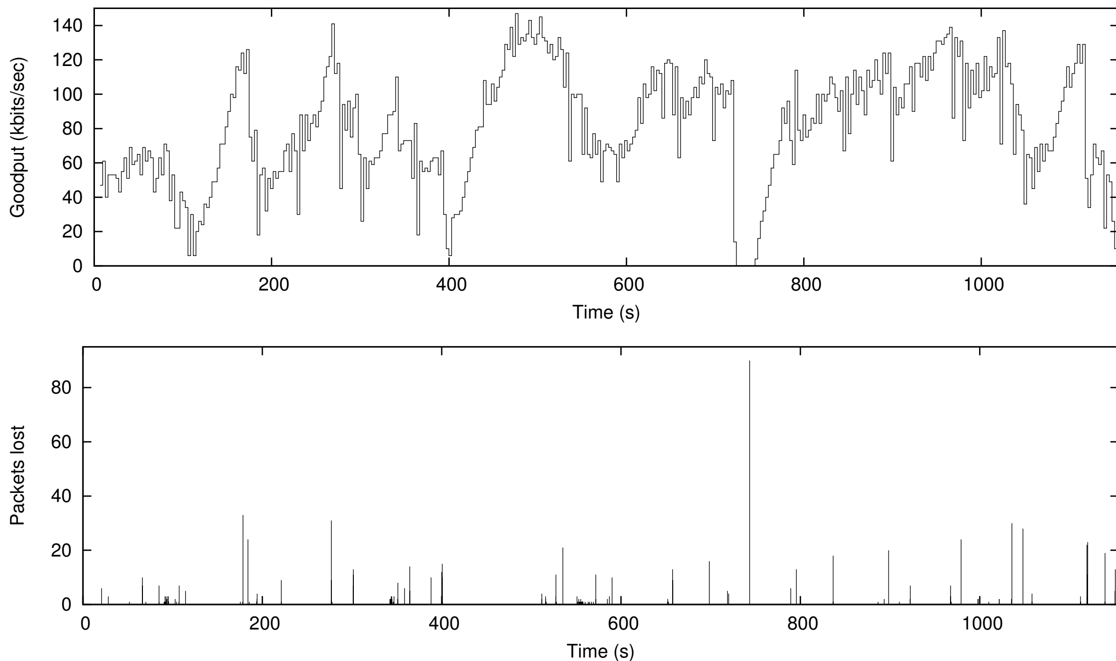


Figure 7-28: Measured packet losses and goodput in a mobile experiment using `verb`.

settings[19]. A `CDMA2000` channel's raw bandwidth varies significantly due to motion (see appendix A), so there is not much `verb` can do to stabilize the throughput when the terminal is moving.

7.4.2 Expected Latencies for `txSlot`'s

An objective driven scheduler may rely heavily on the expected latencies contained within `txSlot`'s to build transmission schedules that accurately reflect application policy. Inaccurate predicted latencies are likely to be a serious handicap.

Both in normal and abnormal network conditions, a generic TCP-style exponentially-weighted-moving average (the `srtt`) is a reasonable way to predict future latencies on a `CDMA2000` channel. Figure 7-29 shows how well such an estimator tracks measured packet round-trip-times in the `verb` experiments. The earlier experiments (`E1` and `S1` through `S4`) used the `srtt` to fill in the `txSlot` latency expectations. The

[19] Some annotations for figure 7-28: [80s-120s] crossing the Harvard bridge; [200s] goodput dips at underpass near BU; [500s] goodput maxima when passing the starbucks near the Prudential; [750s] blackout on corner of Dartmouth and Comm. Ave.; [1050s] dip when crossing the Harvard bridge.

experimental results show that, while often inaccurate by over $100ms$, this estimator does not cause any serious problems for the scheduler.

Nevertheless, the queuing effects on the channel (figure 7-25) and the presence of `verb` congestion control, allow us to build a better latency predictor than the `srtt`. On a `CDMA2000` channel, if the last packet-round-trip time was low, it is highly likely that the next one will be higher; and if the last round-trip-time was high, it is likely that `verb` will back-off and the next round-trip-time will fall as the queues drain.

The round-trip-time for packet i contains information about the round-trip-time for $i+k$, for small k. To see this, we can condition the probability distribution of the latency of the $(i+1)$'th packet on the latency of the i'th packet:

$$p_k(x \mid z) = P(rtt_{i+k} = x \mid rtt_i = z)$$

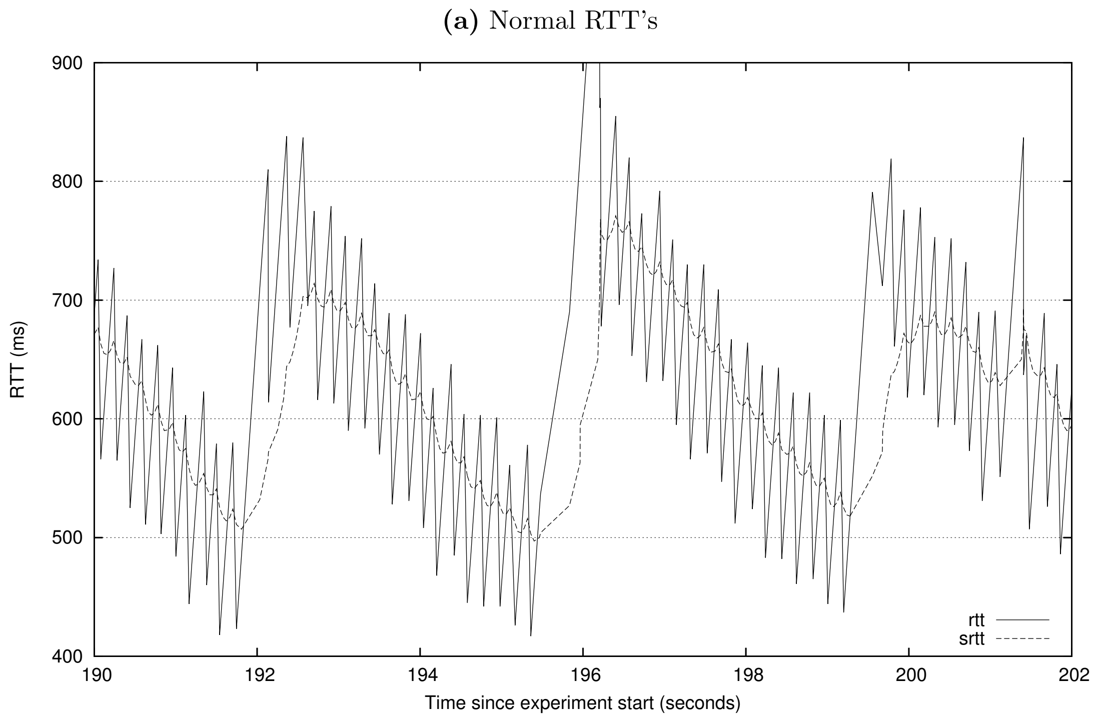
Figure 7-30 shows the above distribution, derived from the stationary `verb` experiment trace, for some values of z and $k = 1$.

A priori knowledge of the $p_k(x \mid z)$ distribution can be used to augment the `srtt` predictions. The `srtt` is often inaccurate by hundreds of milliseconds in figure 7-29. Using $p_k(x \mid z)$ has the potential to narrow this gap.

7.5 Summary

Through real and simulated experiments this chapter demonstrated that the `random-walk` objective driven scheduler implementation can be used to modulate the network quality-of-service provided to application data streams. This chapter also demonstrated that network look-ahead is important for an accurate scheduler with low queuing delays. Additionally, this chapter introduced `verb` congestion control, optimized for `CDMA2000` channels, demonstrating that channel-specific congestion control can be superior to generic congestion control. Finally, we discussed how a priori knowledge about the `CDMA2000` channel's latency distributions can be used to build an improved latency estimator for that channel.

(a) Normal RTT's



(b) Period of Elevated RTT's

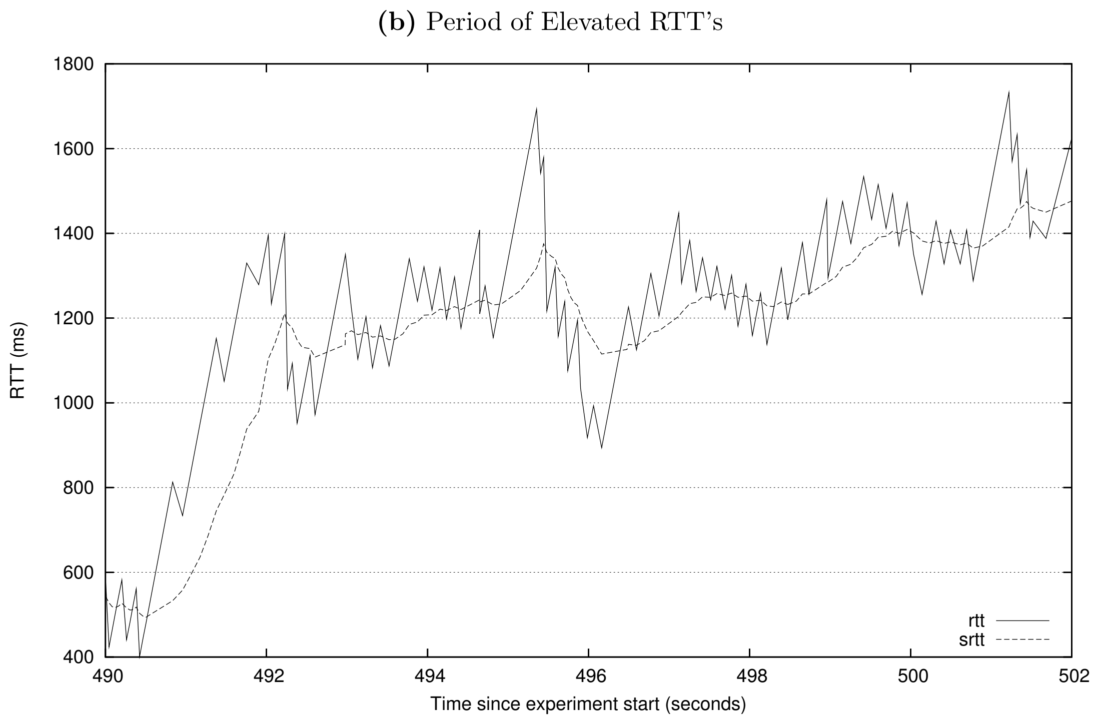


Figure 7-29: Comparison of how well an exponentially-weighted moving-average (the `srtt`) tracks the actual packet `rtt` on a `CDMA2000` channel, when the RTT's are (a) normal; and (b) elevated. In both cases, the `srtt` is often inaccurate by over $100ms$.

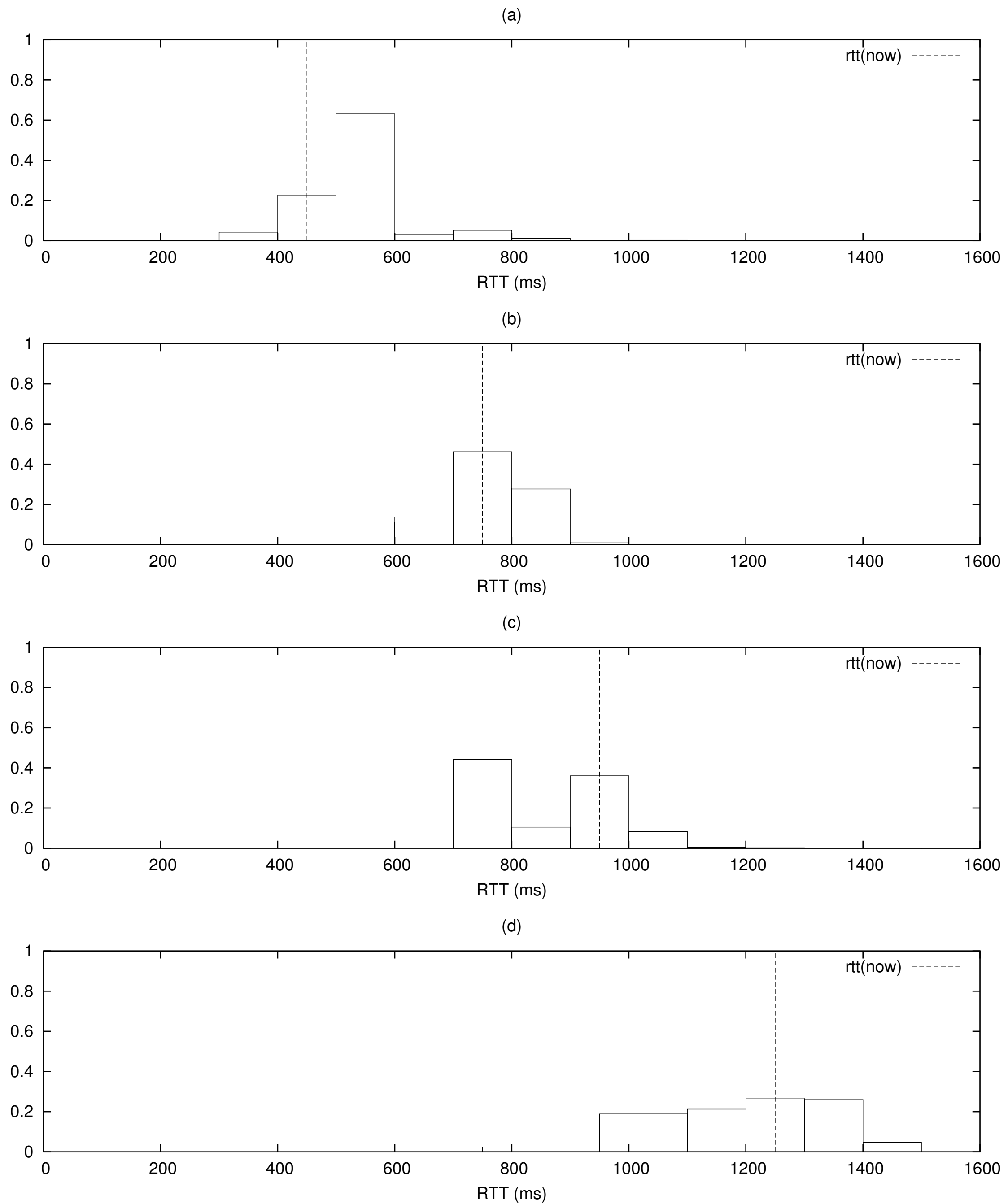


Figure 7-30: Conditioned RTT probability distributions derived from the stationary `verb` experiment traces. These graphs show the distributions of the conditional probabilities $p_1(x \mid z) = P(rtt_{i+1} = x \mid rtt_i = z)$, where z in each graph is given by the $rtt(now)$ line. These graphs demonstrate that the current RTT can provide significant information regarding what the next RTT will look like.

Chapter 8

Conclusions

In this chapter we conclude the thesis with a summary of its goals and contributions followed by proposed improvements and directions for future work.

8.1 Goals and Contributions

Our research leverages the cellular wireless data networks and the technique of inverse multiplexing to develop flexible user-space networking middleware. This thesis developed the *Horde* middleware, discussing its design and evaluating the performance of a preliminary implementation on both real and simulated channels.

Horde facilitates flexible network striping in WWAN environments for a diverse range of applications. Horde is unusual in that it separates the striping policy from the striping mechanism. It allows applications to describe objectives for each stream that the striping mechanism attempts to satisfy. Horde can be used by a set of application data streams, each with its own QoS policy, to flexibly stripe data over a highly heterogeneous set of dynamically varying network channels.

Horde is most useful when dealing with:

Bandwidth-Limited Applications Situations in which applications would like to send more data than individual physical channels can support, justify both the network striping operation and the additional processing cost associated with Horde's QoS modulation framework.

Sensitivity to Network QoS If no application data streams are sensitive to network QoS, Horde's QoS modulation framework becomes redundant.

Heterogeneous and Dynamically Varying Network Channels With such channels, the scheduler has an opportunity to significantly modulate observed QoS. Additionally, Horde's approach is most useful when the available network channels have characteristics that are predictable in the short term. As we report in appendix A, public carrier WWAN channels are of this nature.

Heterogeneous Data Streams When different streams gain value from different aspects of network performance, trade-offs can be made when allocating network resources among those streams. An example is an application in which some streams value low latency delivery and others value low loss rates.

The primary challenge in the design and implementation of the Horde architecture was deciding how applications should express their goals and how these goals should be translated into data transmission schedules. We wanted to give the application control over certain aspects of the data striping operation (e.g., applications may want more urgent data to be sent over lower latency channels or more critical data over higher reliability channels) while at the same time shielding the application from low-level details.

In this thesis, we described the basic abstractions upon which Horde is built. The two key abstractions are transmission tokens (`txSlot`'s) and policy *objective*'s. The `txSlot`'s are generated by Horde to abstractly capture the expected short-term performance of the network channels. The objectives are written by application programmers to abstractly specify network-QoS related objectives for individual data streams. The Horde scheduler uses `txSlot`'s and objectives to derive transmission schedules that provide better value to applications than do conventional scheduling algorithms.

Horde is ambitious: the problem of deriving good transmission schedules from application objectives is a complex one. We have implemented a simple scheduler that can be modulated by objectives. Experiments with this initial implementation

confirm our belief that the kind of QoS modulation Horde aims to achieve is both realistic and advantageous to actual applications.

In summary, our research makes the following major contributions:

Inverse Multiplexing Middleware

This thesis describes the design and implementation of the Horde middleware, which provides applications with an inverse multiplexing subsystem. Horde is designed to be flexible and modular, allowing application designers to optimize for many different types of networks (e.g. wired, wireless, CDMA, 802.11) and also allowing for different types of packet schedulers to be written for the middleware. We use a preliminary implementation of Horde to explore notions about dealing with channel heterogeneity and application policy.

Separating Striping Policy from Mechanism

We present techniques to separate striping policy from the actual mechanism used and evaluate how well our striping mechanism is able to interpret the policy expressed by applications. Horde provides an *objective* specification language that allows QoS policy to be expressed for data streams. We have implemented a basic packet scheduler whose striping policy can be modulated using such objectives. Through real-world experiments and simulation we show that a system in which applications modulate striping policy is feasible. Furthermore, our experiments demonstrate that the ability to modulate striping policy provides tangible benefits to applications.

Heterogeneous and Unstable Channels

Horde has been designed from the ground-up with the assumption that the set of channels being striped over are not well-behaved and can have significantly different channel characteristics that may change unexpectedly (e.g. due to motion relative to a base-station). Horde uses channel-specific managers to provide congestion control and channel prediction logic for each available channel. We argue that congestion control belongs below the striping layer and should be performed in a channel-specific manner because of the idiosyncrasies of WWAN channels.

WWAN Channel Characteristics

Through a number of real-world experiments and discussions of two different WWAN technology standards, we explore the characteristics of real WWAN channels. Not part of the core thesis, these experiments are documented in appendix A. We show that `CDMA2000 1xRTT` and `GSM/GPRS` channels are handicapped by low bandwidth, high and variable round trip times, occasional outages, considerable burstiness, and much instability when moving.

8.2 Directions for Future Work

Much additional work is possible within the context of this thesis. The Horde implementation at this point is more of a *proof-of-concept* than a deployable, real-world system. We are continuing our work on it.

Two possibilities for future work are obvious:

Scheduling Algorithms

The `random-walk` scheduler presented in chapter 7 was purposefully meant to be as simple as possible. Our goal was to show that even a very simple algorithm can provide enough benefits to justify the Horde QoS modulation framework. Simplicity, however, led to sacrifices being made in terms of the scheduler's accuracy.

We have experimented with more efficient schedulers. Some use heuristics, based on the objective language's semantics, to find good schedules. We are optimistic that a better scheduler implementation can both reduce the processing requirements of the striping subsystem as well as yielding transmission schedules closer to optimal.

Telemedicine Application

The application that initially motivated our work remains, as yet, incomplete. This thesis tackles many computer science problems not directly related to building the telemedicine system we outlined in chapter 1. Nonetheless, having the Horde middleware provides a good basis on which to build upon in the future.

Appendix A

WWAN Channel Characteristics

This chapter characterizes, at a high level, the characteristics of the service provided by WWAN's, in the greater-Boston area. We document a number of our end-to-end WWAN experiments. We also refer to previously published results. We restrict our attention to `CDMA2000 1xRTT` and `GSM/GPRS` channels. Where possible, we offer explanations for interesting experimentally observed behaviour. However, we are not attempting to build a definitive model for these WWAN links, nor to explain their observed idiosyncrasies. Such a detailed digression into the nature of the WWAN channels is beyond the scope of this thesis.

Given the ubiquity of WWAN's and the small number of core WWAN IP packet-data technologies being used (mostly `CDMA2000` and `GSM/GPRS`) throughout the world, one would expect there would be a reasonable amount of publicly available information about the performance characteristics of these networks. Details about the underlying networking technologies are well documented [25], but performance data is notoriously sparse.

Part of the problem is that the quality of a WWAN link is highly dependent not only on the technology used, but perhaps even more so on the way the provider has decided to set up the network. For example, the distribution of a provider's base-stations can significantly impact the quality of service experienced by a WWAN interface using that provider. Indeed, as the average cell-phone user must be acutely aware, the quality of a cellular network can vary dramatically from place to place,

even over short distances. With vehicular motion, base-station hand-offs, Doppler and other electromagnetic effects such as multi-path fading can cause further disruptions in the channel.

Presumably, WWAN providers frequently measure and optimize the performance of their own data networks, but it is not in their interest—given the nature of the market—to disclose detailed performance estimates. In fact, estimates published by providers are often hyped—theoretical optimals and word-play—in order to entice consumers into adopting the more costly data service plans. Some publicly available data provides information on `GSM/GPRS` channels [14, 36] and postings in online forums can help determine what average behaviour can be be expected from different providers in different cities. However, there is not enough information in these sources to adequately characterize WWAN links. The discussions in this thesis require a more detailed characterization.

It is widely accepted that the packet-data services provided by today's `CDMA2000` and `GSM/GPRS` networks are characterized by low and variable bandwidths and high and variable round-trip-times. This is confirmed by our experiments. The advertised bandwidth of `GPRS` is almost a third the advertised bandwidth of `CDMA2000` networks; and the `CDMA2000` standard is expected to provide significantly lower round-trip-times than the `GSM/GPRS` standard. Furthermore, given our expectation that Doppler and multi-path effects will impact CDMA based communication less than TDMA based communication [25], we expect the performance of `GPRS` to be affected more than that of `CDMA2000` by motion.

Before delving into the detailed WWAN statistics, figure A-1 shows the mean throughput obtained by TCP bulk data transfers on the two interfaces. The figure illustrates both the asymmetry between the upstream and downstream paths in the `GPRS` network, and the fact that the `CDMA2000` interface provides over six times as much upload bandwidth as the `GPRS` interface. Somewhat surprisingly, in these experiments the `CDMA2000` interface provides greater upload bandwidth than download bandwidth. A possible explanation is that the provider equally provisioned both downstream and upstream paths, but contention for the upstream path is lower.

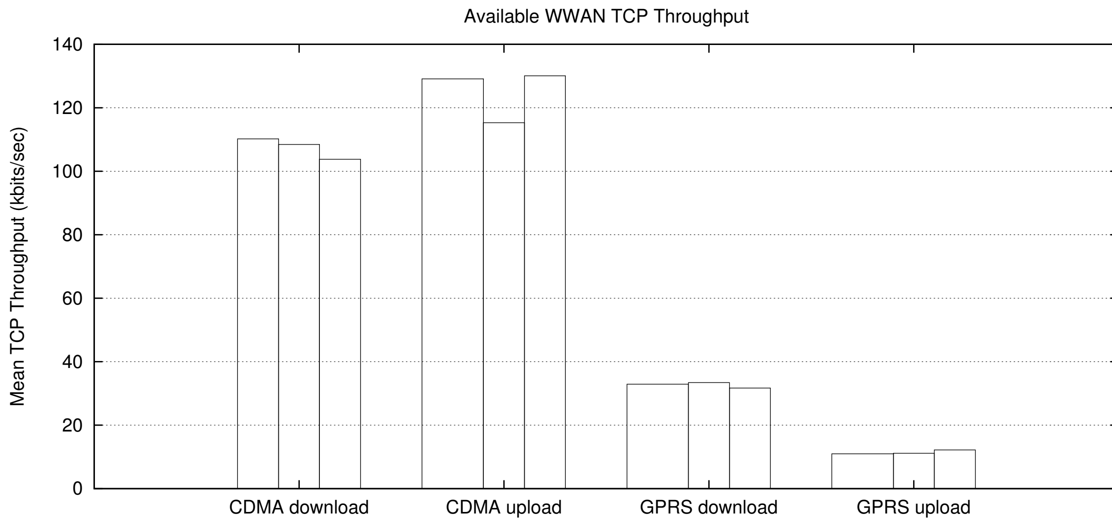


Figure A-1: Observed mean TCP throughputs for bulk data transfers using different WWAN interfaces. Each set shows the throughputs obtained for three independent bulk data transfers for: downloading data from an MIT host over the `CDMA2000` interface; uploading to MIT from the `CDMA2000` interface; downloading from MIT over the `GPRS` interface; and uploading data using `GPRS`. The `GPRS` interface is significantly asymmetric and the `CDMA2000` interface provides much more bandwidth than `GPRS`.

WWAN links are primarily used for web browsing, and service providers have optimized their networks keeping this in mind. Some providers and cell-phones provision resources asymmetrically, allocating more resources to handle downloads and fewer resources to handle data uploads. This results in observed instances, such as in figure A-1 where downstream bandwidth can be as much as three times as large as upstream bandwidth. Additionally, the preeminence of web browsing seems to have led to optimizations for TCP-like bursty traffic (as argued in §A.3.5).

Our experiments almost exclusively focus on the characteristics of the upload path. Our telemedicine application rarely uses the download path.

This chapter is structured as follows: §A.2 sketches out the basic experimental setup; §A.3 investigates the characteristics of a CDMA2000 data link; §A.4 examines the characteristics of a GSM/GPRS data link; §A.5 discusses experiments with multiple interfaces operating in close proximity; and §A.1 summarizes the chapter, providing a high-level view.

A.1 Summary

Before continuing, we present summarized characterizations of the WWAN channels.

CDMA2000 1xRTT When stationary, the `CDMA2000` interface provides around $130kbits/sec$ of upload bandwidth with little variance ($\sigma = 5$). Vehicular motion causes the quality of the link to deteriorate: while moving it provides around $120kbits/sec$ with a high variance ($\sigma = 22$). Packet round-trip-times are high, do not seem to be affected by motion, and are dependent on packet size. For small TCP-SYN packets, we measured a mean round-trip-time of $316ms$; and for large packets, we measured mean round-trip-times close to $800ms$. `traceroute`'s show that the bulk of this transmission delay (over 90%) occurs inside the WWAN service provider's network. When sending a sequence of back-to-back packets, we found consistent queuing artifacts in the resulting packet times: latencies were periodic, gradually decreasing from a maxima and then jumping back up. Losses on the `CDMA2000` channel were bursty. We experienced no disconnections, but temporary communication blackouts occurred, which seemed to depend entirely on the interface's spatial location. We also observed many periods of elevated round-trip-times during which packet latencies jumped well above two seconds.

GSM/GPRS When stationary, the `GSM/GPRS` interface can provide around $25kbps$ of upload bandwidth with little variance ($\sigma = 0.7$). Vehicular motion causes the quality of the link to deteriorate rapidly: when moving it provides around $19kbits/sec$ with a very high variance ($\sigma = 5.3$). Packet round-trip-times are high, are affected significantly by motion, and are dependent on packet size. The minimum average packet round-trip-times measured on stationary `GPRS` interfaces were around $550ms$ for small TCP-SYN packets (with vehicular motion these packets took $200ms$ longer). For large packets (768-bytes and up) and stationary transmitters, average round-trip-times ranged from $800ms$ to $1500ms$ depending on packet size. `tcptraceroute`'s show that the bulk of this transmission delay occurs inside the WWAN service provider's network, which makes sense since the entire `GPRS` providers network shows up as a

single IP hop. Losses on the channel were bursty. We experienced frequent disconnections, but most were temporary, requiring nothing more than than a renegotiation of the `pppd` session. Sometimes these disconnections were more serious, and we had to move to a new location before reconnecting.

Multiple Channels Multiple `CDMA` and `GSM/GPRS` channels can be aggregated using co-located interfaces connected to a laptop. Since these channels do not seem to exhibit strong correlations in their throughput fluctuations, the aggregate channel can be more robust than individual channels.

'Optimal' Packet Size 768-byte payload lengths appear to be optimal for both types of WWAN channels. Smaller packets are inefficient, and result in low throughputs. Larger packets tend to have larger average round-trip-times.

A.2 Experimental Setup

All our experiments were carried out using a Linux laptop and hosts on the MIT ethernet. The laptop was running an updated 2.6.x kernel from the Fedora Core 2 distribution. The Linux point-to-point protocol daemon (`pppd`) was used to establish connections over each WWAN interface.

CDMA2000 1xRTT For the `CDMA2000` experiments, we used an Airprime-5220 PCMCIA card, enabled with a Verizon wireless data service plan which used the `CDMA2000 1xRTT` option on the card (the card can also connect to `1xEV-DO` networks). The Verizon plan gave us a routable IP address and the ability to send and receive ICMP, UDP, and TCP packets.

GSM/GPRS For the `GPRS` experiments we used multiple Nokia 3650 phones enabled with T-mobile data service plans. We had to connect to the phones using bluetooth, introducing an additional wireless hop. We used a bluetooth module plugged into the USB port of the laptop to simultaneously connect to a maximum of four

Figure A-2: The Boston-West safe-ride route.

Nokia phones. `pppd` was configured to work over the `rfcomm` devices created by the Linux bluetooth subsystem. The effects of the bluetooth link are negligible in these experiments; the `GPRS` effects dominate. The `GPRS` link has round-trip-times and variances some orders of magnitude greater than the bluetooth link, and has an order of magnitude lower available bandwidth and variance.

Unfortunately the T-mobile `GPRS` service passed through a NAT, so we did not have routable IP addresses. Although we could send UDP packets from these interfaces, we could not receive UDP packets using them.

Motion Experiments Experiments to determine the effects of motion were carried out by placing the laptop inside the MIT 'Boston-West' safe-ride (see figure A-2). The safe-ride is a shuttle service that follows a fairly constant path, making frequent short stops, moving from the MIT campus into Boston over the Harvard bridge, east onto the Boston University campus, and then back to MIT over the BU bridge. Since all motion experiments were carried out late in the evening on a weekday or very early in the morning, no adverse traffic conditions hampered the motion of the vehicle.

A.3 CDMA2000 1xRTT Link Characterization

A.3.1 Background

This section provides an overview of the CDMA2000 standard. More detailed descriptions of the standard can be found elsewhere (e.g., [25, 1]).

Based on the earlier IS-95 standard, the CDMA2000 standard uses a wide-band spread-spectrum Radio Transmission Technology (RTT), and code division multiple access (CDMA). CDMA2000 satisfies the bandwidth requirements for the International Mobile Telephony 2000 (IMT-2000) standard, often perceived as the third-generation (3G) cell-phone system standard. A number of variants of CDMA2000 have been developed: 1xRTT and 3xRTT use different radio technologies; 1xEV-DO and 1xEV-DV are both high data rate evolutions of the basic 1xRTT architecture. In the US, 1xRTT is widely deployed, with two of the major service providers (Sprint and Verizon) using CDMA2000 1xRTT for their wireless networks. Deployment of 1xEV-DO has begun, but is limited to a small number of cities, primarily because of a lack of demand. 1xRTT provides average throughputs of around $120 kbits/sec$ in real-world networks, and 1xEV-DO and 3xRTT have the potential to provide in excess of $300 kbits/sec$.

The use of CDMA technology should provide better quality of service than narrow-band systems, such as GSM, by providing robust operation in fading environments and transparent (soft) handoffs. CDMA uses multi-path fading effects to enhance the quality of the received signal; conversely, in narrow-band systems fading effects can cause substantial degradation in the quality of the received signal.

The present CDMA2000 1xRTT technology supports a peak theoretical throughput of 153.6 kits/sec, but this peak can vary based on how providers decide to place and configure their base-stations. The underlying radio interface provides a number of fundamental physical channels connecting the mobile device to the base station, each having a bandwidth ranging from $750 bits/sec$ to $14.4 kbits/sec$. A supplemental channel (SCH) is also provided, having up to 16x the bandwidth of the fundamental channels, a maximum of 153.6 kbits/sec (16 x $9.6 kbits/sec$). The SCH has the additional benefit of improved modulation, coding and power control schemes.

As in IS-95, the CDMA2000 physical channels normally use fixed-length transmission frames of 20ms but CDMA2000 also allows 5ms, 40ms and 80ms frame lengths. Frames on a CDMA2000 channel can either contain packet data fragments, frames from the voice encoder, or signaling information. According to the CDMA2000 standard, it is possible for a data packet to be split among different transmission frames, with other data (e.g. voice) interleaved between that packet's fragments.

CDMA2000 assumes that the data being delivered over the SCH's is bursty and delay-tolerant traffic; particularly, consisting of small durations of traffic separated by large durations of no traffic. In such a case, it would be very inefficient to dedicate a permanent channel to a traffic call, since this would adversely affect voice traffic. Consequently, SCH's are designed so that they can be assigned and deassigned at any time by the base-station. Furthermore, for bursty and delay tolerant traffic, assigning a few scheduled fat pipes is preferable to dedicating many thin or slow pipes. The fat-pipe approach exploits variations in the channel conditions of different users to maximize sector throughput. The more sensitive the traffic becomes to delay, such as voice, the more appropriate the dedicated traffic channel approach becomes [1]. Unfortunately, there seems to be no way for an application to inform the CDMA2000 interface that the data being sent is delay sensitive, since a standard application communicates with the CDMA2000 interface through a point-to-point protocol stack.

Finally, the CDMA2000 1xRTT system extensively uses link-layer retransmissions and forward-error-correction (FEC) for data packets, increasing packet delays and delay variances, but keeping the bit-error-rate for the channel at a low level.

A.3.2 Network Hops

Figures A-3 and A-4 show the results of running `traceroute` over the CDMA2000 interface to a host on the MIT network. The asymmetric routing is unsurprising. The IP-hops inside the Verizon network (`*.myvzw.com`) show up only in the trace from MIT. It may be that the Verizon network is set up to ignore ICMP packets originating from CDMA2000 interfaces.

While there are many hops in the path between MIT and the CDMA2000 interface,

```
1 * * *
2 * * *
3 * * *
4 * * *
5 * * *
6 * * *
7 0.so-4-0-0.CL1.EWR6.ALTER.NET (152.63.16.78) 280.440 ms 278.414 ms 339.524 ms
8 0.so-7-0-0.XL1.NYC4.ALTER.NET (152.63.10.21) 285.572 ms 282.523 ms 274.600 ms
9 0.so-6-0-0.BR1.NYC4.ALTER.NET (152.63.21.77) 280.611 ms 322.486 ms 278.545 ms
10 so-0-0-0.edge1.NewYork1.Level3.net (209.244.160.181) 279.618 ms 279.540 ms
278.621 ms
11 ge-2-1-0.bbr2.NewYork1.Level3.net (64.159.4.149) 279.568 ms 419.340 ms 279.603
ms
12 as-2-0.mp2.Boston1.Level3.net (64.159.4.181) 279.584 ms 278.545 ms 380.478 ms
13 ge-11-1.hsa1.Boston1.Level3.net (64.159.4.162) 278.582 ms 273.541 ms 280.604 ms
14 p0-0.mit3.bbnplanet.net (4.24.88.50) 300.576 ms 300.644 ms 299.434 ms
15 NW12-RTR-2-BACKBONE.MIT.EDU (18.168.0.21) 480.354 ms 301.549 ms 299.954 ms
16 W20-575-26.MIT.EDU (18.187.0.45) 299.137 ms 300.514 ms 299.677 ms
```

Figure A-3: `traceroute` from the `CDMA2000` interface to `W20-576-26.MIT.EDU`.

```
1 NW12-RTR-2-W20.MIT.EDU (18.187.0.1) 4.812 ms 0.411 ms 0.339 ms
2 EXTERNAL-RTR-2-BACKBONE.MIT.EDU (18.168.0.27) 0.443 ms 0.529 ms 0.423 ms
3 g3.ba21.b002250-1.bos01.atlas.cogentco.com (38.112.2.213) 0.889 ms 0.873 ms
1.094 ms
4 g0-2.core01.bos01.atlas.cogentco.com (66.250.14.205) 13.014 ms 1.254 ms 13.161
ms
5 p5-0.core01.jfk02.atlas.cogentco.com (66.28.4.118) 6.198 ms 5.972 ms 6.042 ms
6 p4-0.core02.dca01.atlas.cogentco.com (66.28.4.81) 12.217 ms 11.871 ms 11.711 ms
7 p15-2.core01.iad01.atlas.cogentco.com (154.54.2.254) 13.318 ms 12.968 ms 13.194
ms
8 so-0-2-0.edge2.Washington1.Level3.net (4.68.127.9) 12.158 ms 12.279 ms 12.269 ms
9 so-1-1-0.bbr2.Washington1.Level3.net (64.159.3.65) 11.944 ms 12.051 ms 12.166 ms
10 4.68.128.145 (4.68.128.145) 12.269 ms 12.331 ms 12.303 ms
11 so-10-0.hsa2.Newark1.Level3.net (4.68.121.206) 12.405 ms 12.357 ms 12.306 ms
12 h0.verizon3.bbnplanet.net (4.25.108.46) 13.956 ms 13.976 ms 13.765 ms
13 66.sub-66-174-106.myvzw.com (66.174.106.66) 14.434 ms 14.820 ms 14.446 ms
14 14.sub-66-174-107.myvzw.com (66.174.107.14) 14.988 ms 14.288 ms 14.981 ms
15 163.sub-66-174-22.myvzw.com (66.174.22.163) 14.733 ms 15.246 ms 15.101 ms
16 117.sub-66-174-120.myvzw.com (66.174.120.117) 16.645 ms 16.707 ms 16.112 ms
17 * * *
18 78.sub-70-212-131.myvzw.com (70.212.131.78) 4061.946 ms 461.507 ms 278.753 ms
```

Figure A-4: `traceroute` from `W20-575-26.MIT.EDU` to the `CDMA2000` interface.

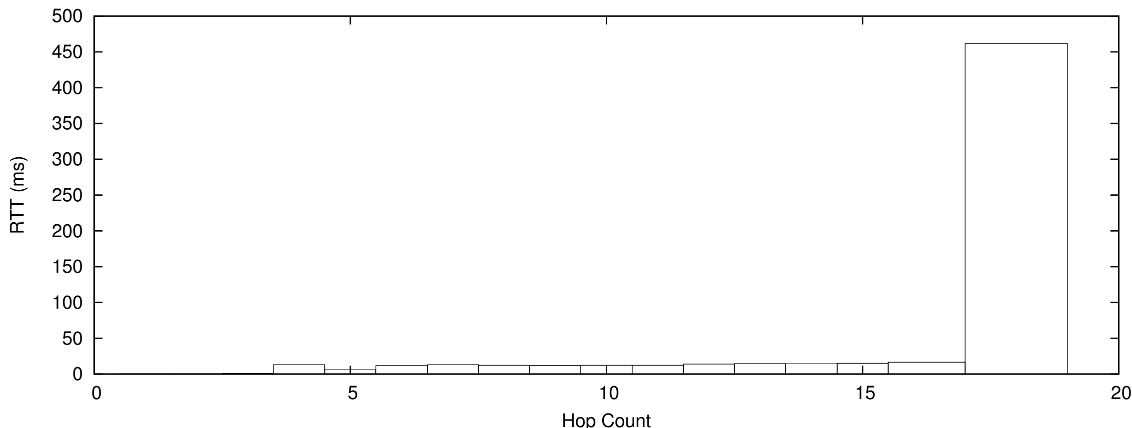


Figure A-5: Median packet round-trip-times for the traceroute to the `CDMA2000` interface from `W20-575-26.MIT.EDU`. The last IP hop's latency dominates.

it seems that the last few hops in the Verizon network dominate in determining packet round-trip-times along the path. From figure A-3, the median round-trip-time by hop 7 is already at $278ms$, within $22ms$ of its final value. From figure A-4, the median round-trip-time by hop 16 is only $16ms$, but spikes to $461ms$ by hop 18. Figure A-5 plots the median round-trip-times for each hop along the path traced to the interface, illustrating the dominating effect of the last hops.

It is unclear whether this last hop represents the actual wireless link between the Verizon base-station and our `CDMA2000` interface. It is likely that this single IP hop actually represents the entire `CDMA2000` network, as is the case in `GPRS` systems (§A.4.2). In any case, packet latency is overwhelmingly dominated by what the packet experiences inside the provider's network.

A.3.3 Link Capacity Estimates

To determine the capacity of the Verizon link, we used `pathload` and `pathrate` [29, 18]. Both utilities use packet pair/train dispersion techniques to estimate bandwidth.

`pathrate` reported an estimated upload bandwidth of $122-160kbits/sec$, close to what we expected based on the `CDMA2000` standard. Figure A-6 shows selected parts of the output from `pathrate` configured to use the `CDMA2000` interface as the sender and a MIT host as the receiver.

```
pathrate run from 70.212.131.78 to ajrak.mit.edu on Tue Aug 24 00:54:26 2004
--> Average round-trip time:  355.8ms
...
--> Capacity Resolution:  38 kbps
-- Phase I: Detect possible capacity modes --
...
-- Local modes :  In Phase I --
* Mode:  22 kbps to 59 kbps - 742 measurements
Modal bell:  775 measurements - low :  7 kbps - high :  91 kbps
* Mode:  122 kbps to 160 kbps - 97 measurements
Modal bell:  243 measurements - low :  53 kbps - high :  311 kbps
-- Phase II: Estimate Asymptotic Dispersion Rate (ADR) --
...
-- Local modes :  In Phase II --
* Mode:  56 kbps to 94 kbps - 427 measurements
Modal bell:  497 measurements - low :  21 kbps - high :  178 kbps
--> Asymptotic Dispersion Rate (ADR) estimate:  88 kbps
--> Possible capacity values:
122 kbps to 160 kbps - Figure of merit:  91.60
-------------------------------------------------
Final capacity estimate :  122 kbps to 160 kbps
-------------------------------------------------
```

Figure A-6: Extracts from the output produced by `pathrate` using the `CDMA2000` interface as the sender and an MIT host as the receiver.

`pathrate` reported a download capacity estimate of 131-133 kbps. In this downstream case `pathrate` was able to increase its capacity resolution to 2 kbps and terminate quickly because of low measurement noise. `pathload` reported available bandwidth to be 0.14 Mbps and a MTU of 1472 bytes.

A.3.4 Upload Bandwidth

We configured the `CDMA2000` interface to send as many UDP packets with 1400 byte payloads as possible to a host on the MIT network. At the receiver, we logged the arrivals and used a 5-second sliding window to average the arrival rate. This arrival rate is an estimate for the link's available raw UDP bandwidth, or goodput. The plots in figure A-7 show how the arrival rate varied during two different experiments.

Stationary When stationary, the `CDMA2000` link provided a relatively stable bandwidth. The mean was around $129 kbits/sec$, with a standard deviation that was about 3.5% of the mean. Figure A-7a shows how the available bandwidth varied with time.

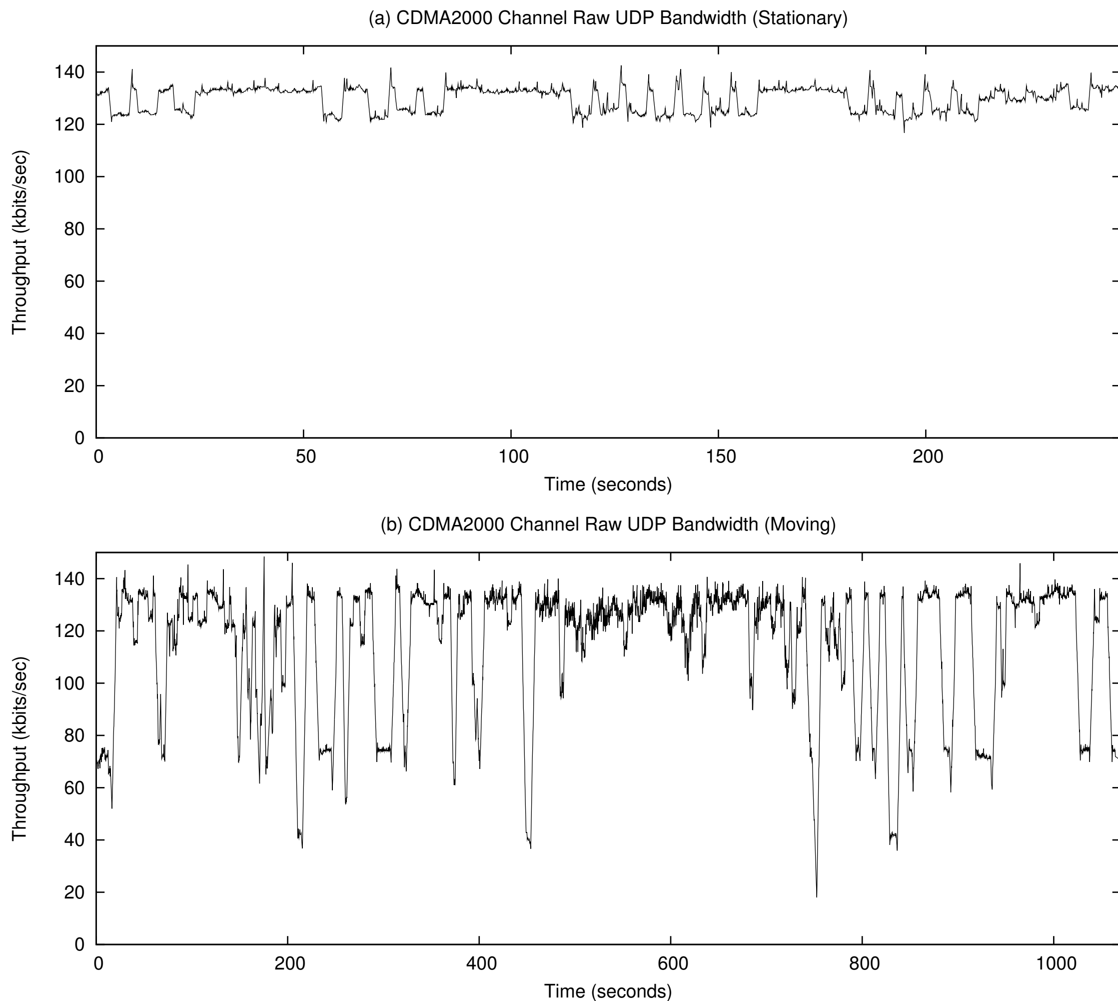


Figure A-7: Measured raw UDP upload bandwidth on a `CDMA2000` interface when (a) stationary ($\mu = 129.59, \sigma = 4.57$); and (b) moving ($\mu = 119.45, \sigma = 21.49$).

Moving Figure A-7b shows that the `CDMA2000` link behaved very differently when the interface was moving. Motion causes the standard deviation to be multiplied almost by a factor of 5 while the average bandwidth dropped by around $10kbits/sec$.

CDMA2000 1xEV-DO Unfortunately, Verizon does not offer its high data rate `1xEV-DO` service in Boston. Some relatively informal bandwidth experiments with the `1xEV-DO` networks—while on vacation in New York and Washington DC—revealed that using the Verizon `1xEV-DO` networks more than doubled the download bandwidth over `1xRTT`, but did not provide any perceptible increase in upload bandwidth.

A.3.5 Packet Round-Trip-Times

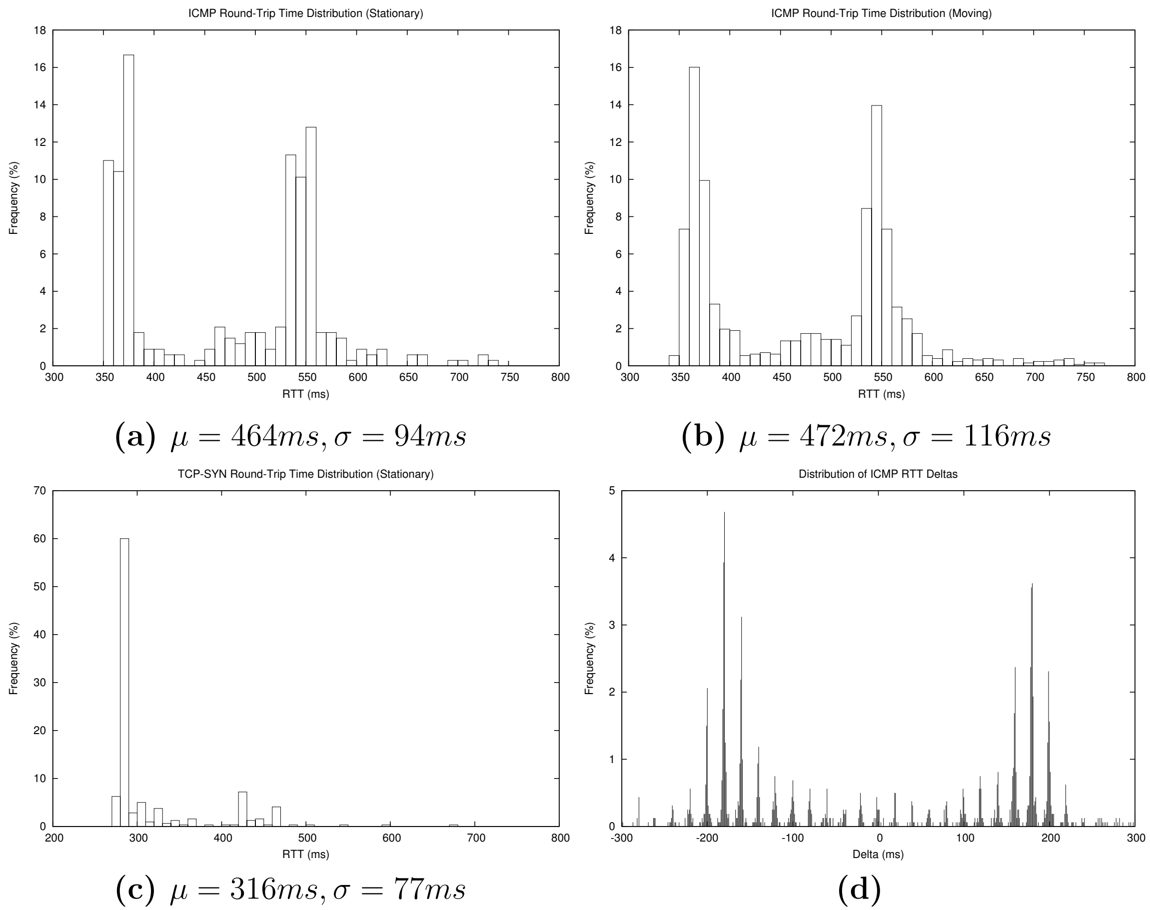
We conducted a number of `CDMA2000` latency related experiments whose results are documented below. The average packet round-trip-times measured on the `CDMA2000` link were high (the minimum measured average was around 315ms). The round-trip-time distribution did not seem to be affected by vehicular motion. Finally, the kind of dynamic variation that was observed in the packet latencies implies that the channel is optimized for bursty traffic.

ICMP Pings We started by measuring the round-trip-times for single, small ICMP packets using the `ping` utility. A small 56-byte ICMP packet was sent to a host on the MIT network, echoed back, and when it returned the round-trip-time was logged and another packet was sent out.

Figures A-8a and A-8b show how the round-trip-times we measured were distributed when the transmitter was (a) stationary and (b) when it was moving. Motion did not seem to affect the round-trip-times: the two distributions are similar; and the moderate differences may be because these are different experiments. Figure A-9 shows how round-trip-times varied with time in these two experiments.

It is notable that the distributions are bi-modal, the first mode occurring around 375ms and the second mode at around 550ms. For the overwhelming majority of the round-trip-time samples, if the packet i had a low round-trip-time (around 350ms), then the packet $i + 1$ had a high round-trip-time (around 550ms). This behaviour is visible in figure A-9. Figure A-8d shows the distribution of the inter-arrival times of the pings as they return to the `CDMA2000` interface. The secondary peaks in the distribution of figure A-8d are equally spaced at 20ms intervals, probably because the smallest frame size used for data by `CDMA2000` is 20ms in duration (§A.3.1).

Figures A-9a and A-9b plot how round-trip-times varied with time in these experiments. The inter-arrival time distribution is bi-modal because of the saw-tooth pattern seen in these figures. We consider the causes for this saw-tooth pattern later, when we examine the latencies observed by sustained UDP packet trains.



(a) $\mu = 464ms, \sigma = 94ms$ **(b)** $\mu = 472ms, \sigma = 116ms$

(c) $\mu = 316ms, \sigma = 77ms$ **(d)**

Figure A-8: Observed distributions of single packet round-trip-times for small packets on a `CDMA2000` link when (a) the transmitter is stationary; (b) when the transmitter is moving; (c) when the transmitter is stationary, but the packets; are TCP-SYN packets instead of ICMP packets; and (d) The distribution of $\Delta_{rtt} = (rtt_{i+1} - rtt_i)$ for all ICMP round-trip-times from (a) and (b).

TCP-SYN Pings using the `tcpping` utility we measured the round-trip-times for single, small TCP-SYN packets . A small SYN packet was sent to a host on the MIT network, ACK'd, and when the SYN-ACK returned the round-trip-time was logged and another SYN packet was sent out.

The distribution of round-trip-times measured using TCP-SYN packets, in figure A-8c, is very different from the earlier ICMP distributions. Notably, there is only a single mode. Furthermore, the mean round-trip-time is over 150ms shorter than it was for the ICMP pings. Since the ICMP and TCP-SYN packets are around the same size, the Verizon network is prioritizing the delivery of TCP-SYN packets.

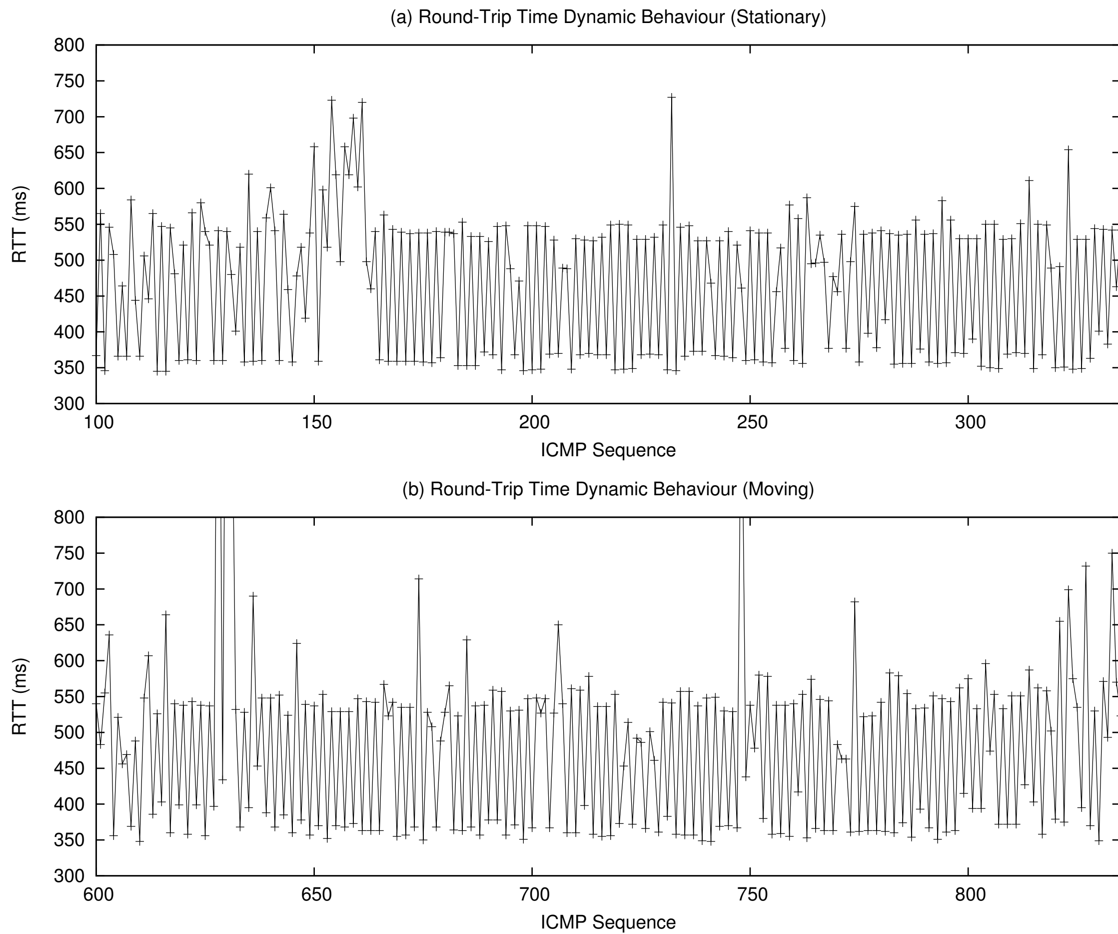
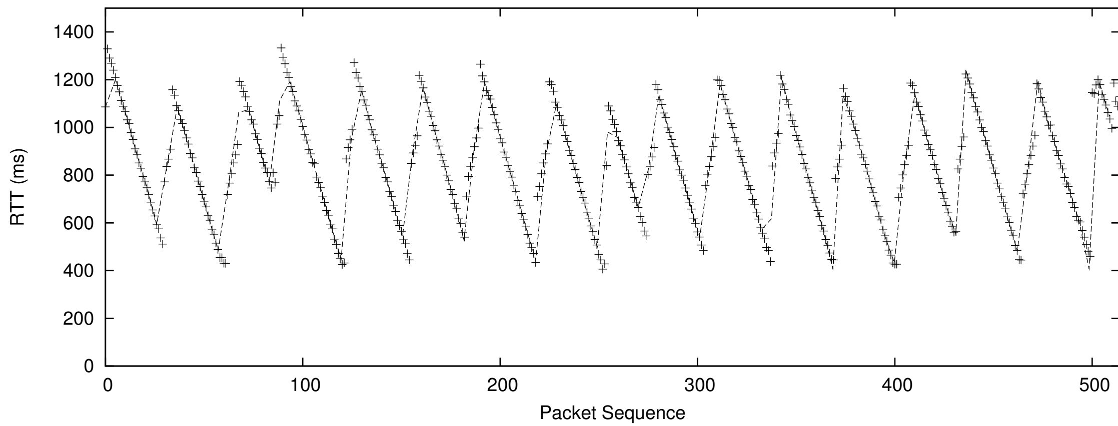


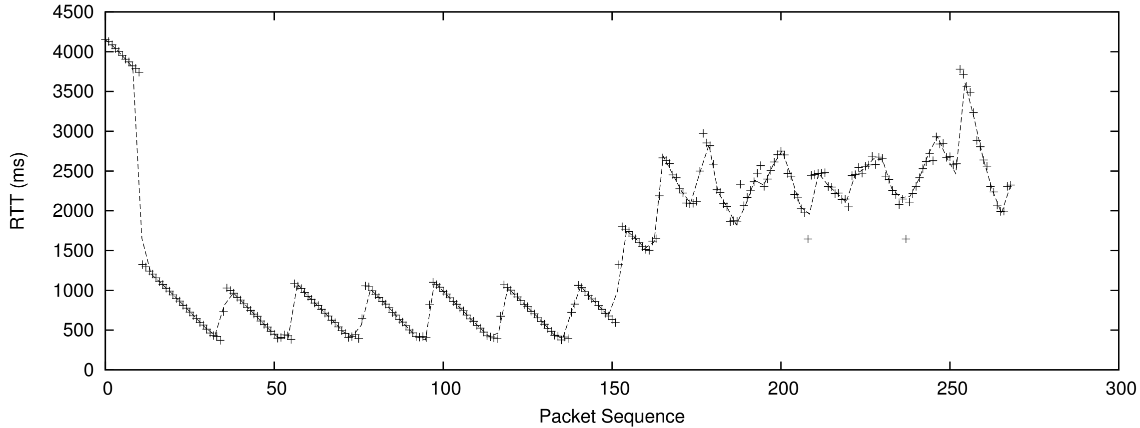
Figure A-9: Dynamic behaviour of single packet round-trip-times for small ICMP packets on a `CDMA2000` link when the transmitter is: (a) stationary and (b) moving.

UDP Packet Trains We generated UDP packets at a constant rate—spacing each packet pair equally far apart in time. These packets were sent over the `CDMA2000` interface to a host on the MIT network, and the MIT host acknowledged each received packet. Using these acknowledgments we logged round-trip-times. We experimented with multiple packet sizes and packet generation rates. As we discuss later, increased packet sizes tend to elevate round-trip-times.

Figure A-10a shows how the round-trip-times varied in a typical experiment. The sawtooth pattern from the `ping` experiments appears again, but the round-trip-time distribution (figure A-11a) is no longer bi-modal. Now the latencies gradually decrease until a minima and then increase quickly to the maxima, instead of swinging rapidly.



(a) 1152 byte payloads, $\mu = 850, \sigma = 221$



(b) 768 byte payloads, $\mu = 1570, \sigma = 990$

Figure A-10: Observed dynamic behaviour of round-trip-times when UDP packet trains are sent using a high rate packet generator, under varying network conditions: (a) normal network conditions; (b) abnormal network conditions can triple times.

The packets were generated with a constant temporal spacing, so figure A-10a shows a queuing effect causing the round-trip-time oscillation: the gradually decreasing round-trip-times can be explained by a draining queue. We are not sure why this happens; there are a number of possible explanations. It may be that the `CDMA2000` network is using TCP to tunnel data packets between the base-station and the gateway into the provider's data network (`GPRS` implementations are known to do this). Alternatively, it may be that the radio link protocol or driver on the Airprime card waits for a queue of packets to build up before reserving the `SCH` channel with the base station to send the data. Finally, the queuing effect could be the result of the schedul-

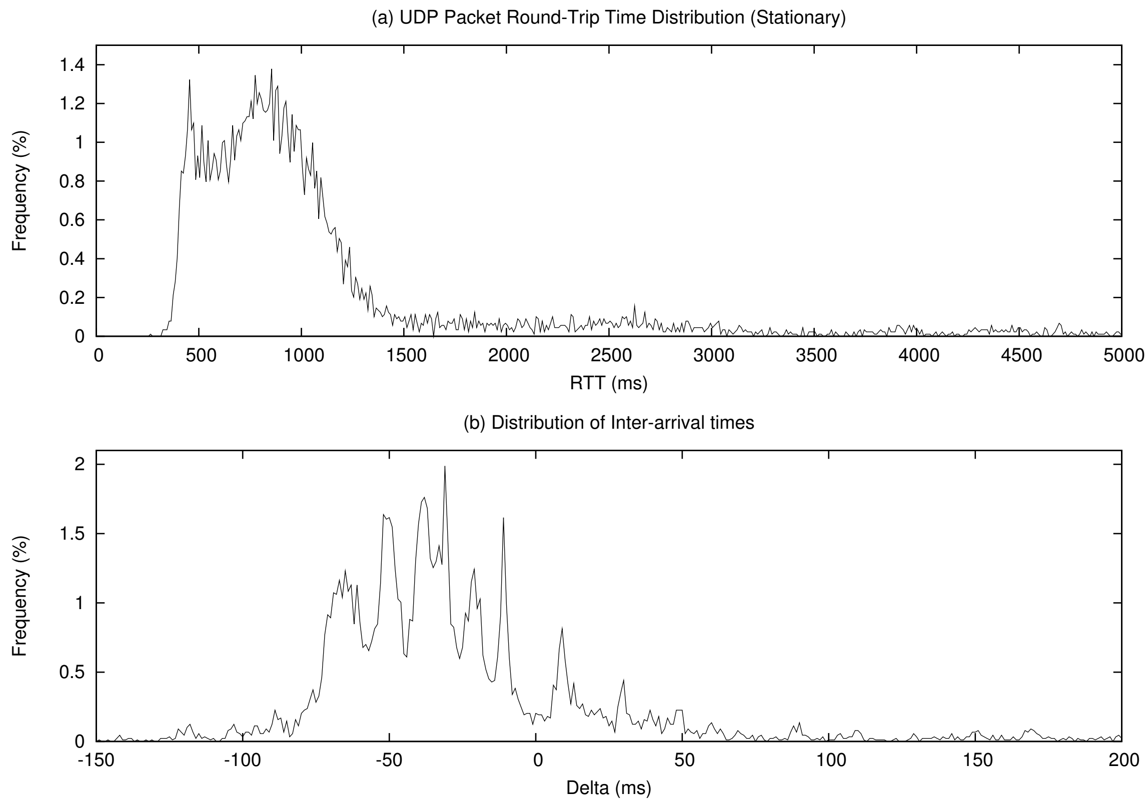


Figure A-11: Observed distribution of round-trip-times for packets generated at a high constant rates, with various packet sizes, over the CDMA2000 link: (a) distribution of round-trip-times; (b) distribution of inter-arrival times.

ing algorithms used at the base-station to manage the traffic from different senders. This periodic round-trip-time variation would probably not affect the performance of TCP, because of its usage of moving averages.

Figure A-11b shows the distribution of inter-arrival times for the packet acknowledgments. Most of the inter-arrival times are below zero; the majority of the packets were on the downward edge of the sawtooth. This makes sense for the packets being buffered and sent out in bursts: the packet at the head of the burst will have the highest round-trip-time, being delayed the longest, but since the queuing will have compressed the spacing between this packet and the others later in the burst, round-trip-times will decrease as the burst progresses.

A.3.6 Packet Size

We conducted a number of experiments (summarized in figure A-12) to investigate the impact of picking a specific packet size on the performance of the `CDMA2000` link. The `CDMA2000` interface allows a maximum transmission unit of 1472 bytes.

UDP Throughput Figure A-12a shows the achievable goodput when UDP packets were sent at a constant rate defined by the following formula:

$$rate_{sender} = \frac{bitrate_{max}}{size_{packet}} \approx \frac{130kbits/sec}{size_{packet}}$$

Where $size_{packet}$ is the size of the packet payload; and $bitrate_{max}$ is based on the maximum observed UDP goodput on the channel from our earlier experiments.

The least squares fit obtained by considering only small packets (payloads smaller than 384 bytes) implies that the goodput rises linearly as packet size is increased. The trend probably exists because small data packets are less efficient in throughput terms than medium or large ones.

The least squares fit obtained by considering only medium and large packets (384 bytes and above) shows that goodput falls, slightly, as packet size is increased. This weak trend may be an artifact. It could also be explained by the fact that larger packets may experience more wireless frame losses, increasing their round-trip-times as link-layer retransmits are used to mask these losses.

Latency Figure A-12b shows how packet round-trip-times were affected by different packet sizes, over a large number of experiments. For each packet size, the error bars plot the median, the 75th percentile and the 25th percentile round-trip-times from all the experiments. The curve shown on the figure is a function that approximates how the median round-trip-times vary with packet size. As packet sizes are decreased below 256 bytes, the round-trip-times also decrease. Round-trip-times are the same, on average, for packet sizes ranging from 256 to 1024 bytes. As packet sizes are increased beyond 1024 bytes, round-trip-times also increase.

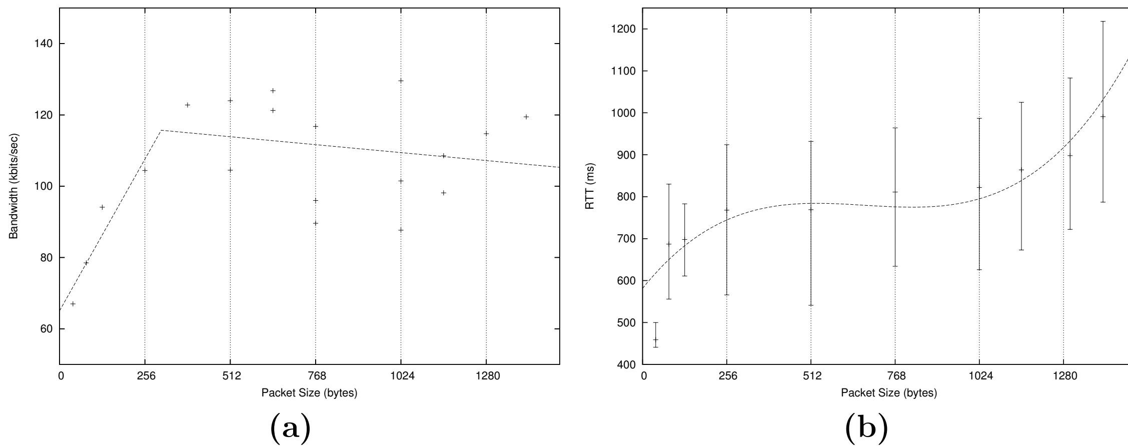


Figure A-12: Observed effects of changing UDP packet sizes on the performance of the `CDMA2000` interface: (a) other than for small packets, packet size does not significantly impact achievable throughput; but (b) as packet size is increased beyond a certain point, the round-trip-time increases with packet size.

A.3.7 Loss Characteristics

The packet losses observed on the `CDMA2000` link were much lower than would be expected on a raw wireless link, because of the effective use of forward error correction and link-layer acknowledgments. When AIMD-style flow control was used to avoid dropping packets at the sender, around 1% of the packets were lost[1].

Using constant-rate packet generation processes to send packets from the `CDMA2000` interface to an MIT host, as before, we ran a number of experiments with different rates to determine the nature of the packet losses. We were primarily interested in determining how bursty the losses on the `CDMA2000` interface were. We define a packet loss burst of length k as a contiguous sequence of k lost packets.

Figure A-13 shows how packet losses were distributed relative to burst length. The peak at burst-length 1 in figure A-13a is probably an artifact of our usage of constant-rate senders. The histogram shows that losses on the link were bursty in nature: 25% of the lost packets were lost in bursts ranging from 3 to 7 packets in length; 12% were lost in bursts of length 2; 18% were lost in bursts ranging from 8 to 20 packets in length; and larger bursts accounted for almost 10% of lost packets. Large burst

[1]These losses were probably unavoidable, a consequence of the fact that the AIMD flow control would continue to increase its sending rate, until it detected a loss.

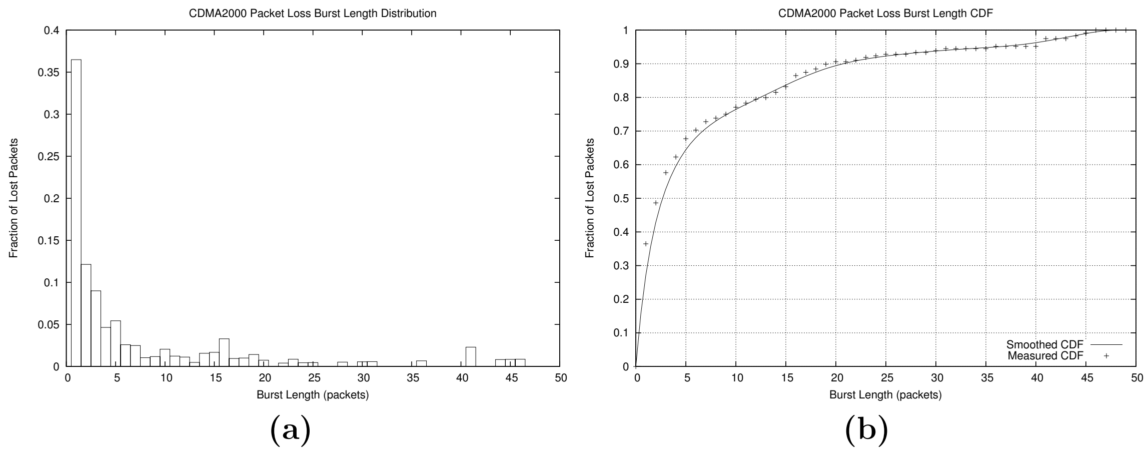


Figure A-13: Observed distribution of packet losses from a number of different experiments with CBR senders using a `CDMA2000` interface. For each packet loss, (a) the histogram tracks how many consecutive packets surrounding that lost packet were also lost; and (b) shows the cumulative density of this loss distribution.

losses (around 40 packets) were often observed at the start of an experiment (as can be seen in figure A-14a) and may be related to a startup transient on the `CDMA2000` channel, a time when network resources are being acquired..

Figure A-14 focuses on a single experiment where the sending rate was slightly larger than the available bandwidth. The burst-loss length histogram illustrates the burstiness of the losses during this particular experiment. Dark regions of the timeline indicate periods during which packet losses piled up. Frequent single-packet losses, because of the consistently high bit-rate, can also be seen in the timeline.

A.3.8 Disconnections

During our experiments we did not experience disconnections in our `pppd` sessions over the Verizon `CDMA2000` interface. This is in contrast to the behaviour observed with `GPRS` (§A.4.7) where disconnections were common. During the motion experiments we did once experience a short blackout: we could not transmit or receive data until the vehicle had moved on[2]. During the blackout the `pppd` session didn't terminate, and network service resumed as soon as we had passed that location.

[2]At the intersection of Dartmouth street and Commonwealth Avenue, on the Copley square side of Comm. Ave. The vehicle was stopped in the blackout location for less than 45 seconds.

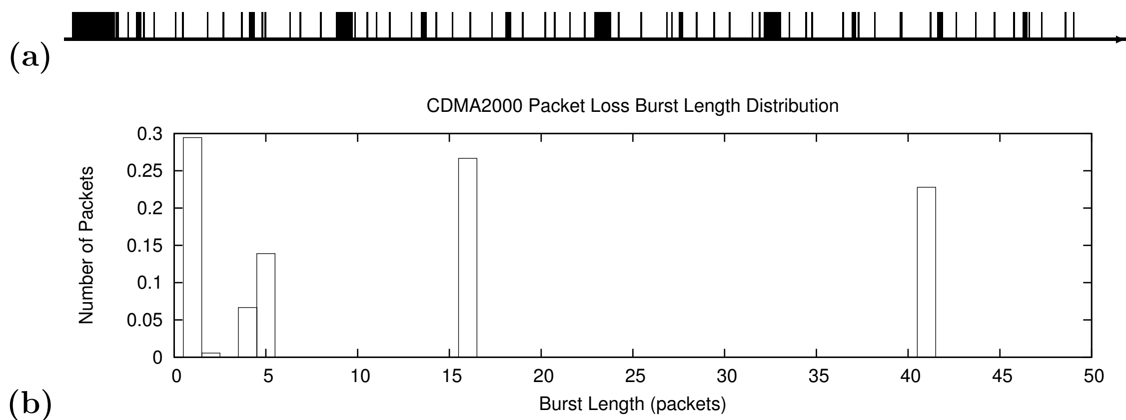


Figure A-14: Packet loss observations from a single experiment using the `CDMA2000` interface with 1400 byte packets and transmitting at a constant rate of around $120 kbits/sec$: (a) shows the loss timeline, each thin vertical line representing a single packet loss; (b) shows the packet burst-loss length distribution.

However, we encountered many periods of network abnormality in which available bandwidth plummeted and packet round-trip-times became excessively elevated. Figure A-10b shows one such period. At the start of the experiment shown in this figure, round-trip-times were high, but fell rapidly to their normal levels. Then, about 3 minutes into the experiment, round-trip-times became elevated again, rising above 2 seconds. Finally, about six minutes into the experiment, the round-trip-times rose to another plateau, averaging above 4 seconds. This experiment consisted of a constant-rate sender saturating the channel with as much data as it could send. We did not observe latency elevations in other experiments that used flow control. Stopping our senders and restarting them after a while also seemed to solve the problem.

It is possible that the round-trip-time elevation is caused by the fact that the sender does not back off when there is a degradation in channel quality. The channel uses link-layer retransmissions, the round-trip-time and elevation may be because of the delay caused by retransmissions for this or earlier packets. In any case, these abnormal periods imply that even though we are not likely to experience call drops, we may experience periods during which bandwidth becomes constrained and every video packet we send times out before it reaches the other end of the connection.

A.4 GPRS Link Characterization

A.4.1 GSM/GPRS Background

This section provides an overview of the GSM/GPRS standard. More detailed descriptions of the standard can be found elsewhere (e.g., [25]).

The Global System for Mobile communications (GSM) is a decade old digital cellular technology based on a hybrid Frequency Division Multiple Access (FDMA) / Time Division Multiple Access (TDMA) narrow-band technology. GSM is perhaps the most widespread cellular technology in the world: it is estimated that at the end of Jan 2004 there were over 1 billion GSM subscribers across more than 200 countries.

Using two frequency bands located near 900 MHz, GSM first divides its 25 MHz bandwidth into 124 carrier frequencies spaced 200 kHz apart. These carrier frequencies are then divided in time, using a TDMA scheme. The fundamental unit of time in the TDMA scheme is a burst period, lasting approximately 0.577 ms, eight of which are grouped together into a TDMA frame, lasting approximately 4.615ms. Burst periods form the basic unit for the definition of logical channels: one logical channel is one burst period per TDMA frame. Channels are defined by the position of their burst periods in the TDMA frame. Some channels are dedicated and allocated to a mobile base station; others are common, used by mobile stations in idle mode; and some carrier frequencies are reserved for use by base stations. The mobile and base station use slow frequency hopping to try to lessen the impact of multi-path fading.

A traffic channel (TCH) is used to carry speech and data traffic in the GSM scheme. Traffic channels are defined using a 26-frame multi-frame (a group of 26 TDMA frames) lasting 120 ms. Out of the 26 frames, 24 are used for traffic. TCH's for the uplink and downlink are separated in time by 3 burst periods.

The General Packet Radio Service (GPRS) uses an overlay on the basic GSM network to provide TCP/IP packet data services. The presently deployed version of GPRS is an evolution of the GSM Short Message Service (SMS) and is primarily meant to allow wireless providers to roll out basic TCP/IP services until they can make the more expensive technological shift to next generation GSM technologies like EDGE and

W-CDMA. Not surprisingly, GPRS does not aim for 3G compliance.

GPRS works by aggregating the GSM TCH's and using support nodes on the provider network to provide TCP/IP functionality. Theoretically, it is possible for GPRS to produce 115 kbps by using 8 GSM 14.4 kbps TCHs, but maximum speeds of 56 kbps are far more likely, given provider setups.

In a GSM/GPRS network, data packets from the wireless terminal are encapsulated and tunneled through the provider network using the GPRS Tunneling Protocol (GTP), which can be either IP or TCP based. Subsequently, the entire GPRS network shows up as a single IP hop. The Radio Link Control (RLC) protocol used for communications between the mobile and the base station uses a sliding window mechanism for flow control and uses link layer acknowledgments. Because of factors such as the use of GTP there is an added overhead of over 48 bytes in the headers of packets sent using GPRS.

A.4.2 Network Hops

Figure A-15 shows the result of running tcptraceroute[3] [41] over the GPRS interface to a host on the MIT network. As with the CDMA2000 link (§A.3.2) the path consists of a large number of hops. As before, the effects of the first hop dominate in determining the latency of the path: the median round-trip-time to the first hop is 85% of the median round-trip-time to the final hop. Because of the use of GTP, this single IP hop represents the entire GPRS network.

A.4.3 Upload Bandwidth

We configured the GPRS interface to try to send as many UDP packets with 1400 byte payloads as possible to a host on the MIT network, and measured the arrival rate of the packets at the receiver. Using a 5-second sliding window, we estimated the average UDP throughput thus obtained, the goodput. The plots in figure A-16 show the goodput through the course of two different experiments.

[3] tcptraceroute is a traceroute variant that uses TCP-SYN packets instead of ICMP packets.

```
1 m21ac9bd8.tmodns.net (216.155.172.33) 573.936 ms 496.636 ms 615.785 ms
2 * * *
3 * * *
4 * * *
5 sl-gw7-atl-6-0-0.sprintlink.net (144.228.83.57) 1312.060 ms 651.646 ms 660.789
ms
6 * * *
7 sl-bb22-fw-10-0.sprintlink.net (144.232.18.21) 1252.992 ms * 1288.253 ms
8 sl-bb27-fw-12-0.sprintlink.net (144.232.11.33) 680.744 ms 669.767 ms 720.771 ms
9 sl-st20-dal-1-0.sprintlink.net (144.232.9.136) 678.758 ms 629.699 ms *
10 interconnect-eng.Dallas1.Level3.net (64.158.168.73) 1208.258 ms 652.697 ms
660.721 ms
11 * so-1-2-0.bbr2.Dallas1.Level3.net (209.244.15.165) 1116.065 ms 652.678 ms
12 as-2-0.mp2.Boston1.Level3.net (64.159.4.181) 676.797 ms 670.705 ms 701.778 ms
13 ge-11-1.hsa1.Boston1.Level3.net (64.159.4.162) 678.716 ms 672.703 ms 703.851 ms
14 p0-0.mit3.bbnplanet.net (4.24.88.50) 675.698 ms 692.683 ms 676.007 ms
15 W92-RTR-1-BACKBONE.MIT.EDU (18.168.0.25) 733.528 ms 677.686 ms 678.752 ms
16 * * *
17 WEB.MIT.EDU (18.7.22.69) [open] 1262.767 ms 674.758 ms 699.498 ms
```

Figure A-15: `tcptraceroute` from the `GPRS` interface to `WEB.MIT.EDU`.

Stationary When stationary (figure A-16a), the `GPRS` link provides $25kbits/sec$ of upload bandwidth with a low variance. In another stationary experiments, we saw a little more dynamic variation in the available bandwidth, but on the whole it remained stable (excluding periods of abnormal network conditions discussed later in §A.4.7).

Moving Motion caused a great deal of variation in the available bandwidth. As figure A-16b shows, the average available bandwidth fell to around $19kbits/sec$ and the standard deviation of this bandwidth increased more than five fold. This motion induced variation is a large percentage of the total bandwidth of the `GPRS` link, making the `GPRS` channel far more sensitive to motion than the `CDMA2000` channel.

A.4.4 Packet Round-Trip-Times

We conducted a number of `GPRS` latency related experiments whose results are documented below. We also summarize some previously published work. The minimum average packet round-trip-times measured on stationary `GPRS` interfaces were around $550ms$ and much higher than those observed on the `CDMA2000` link. Furthermore, the average latency was seriously affected by vehicular motion (rising by $200ms$)

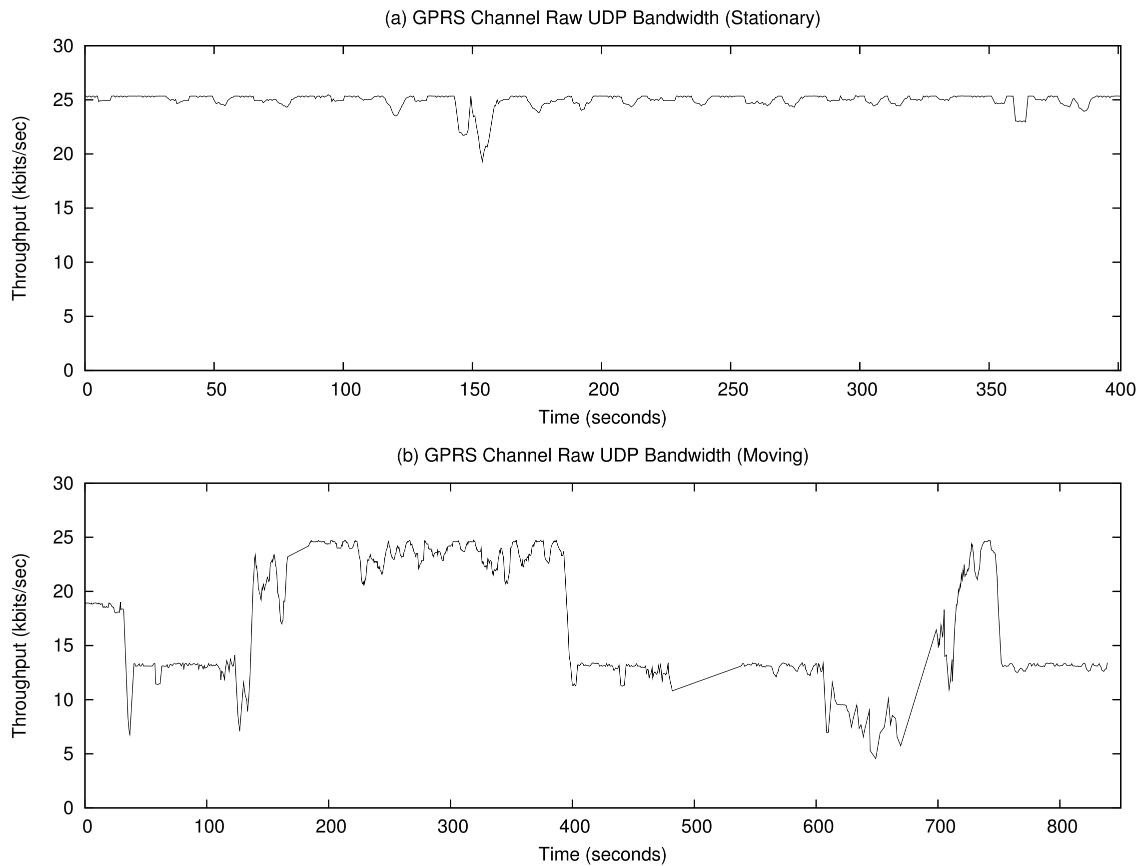
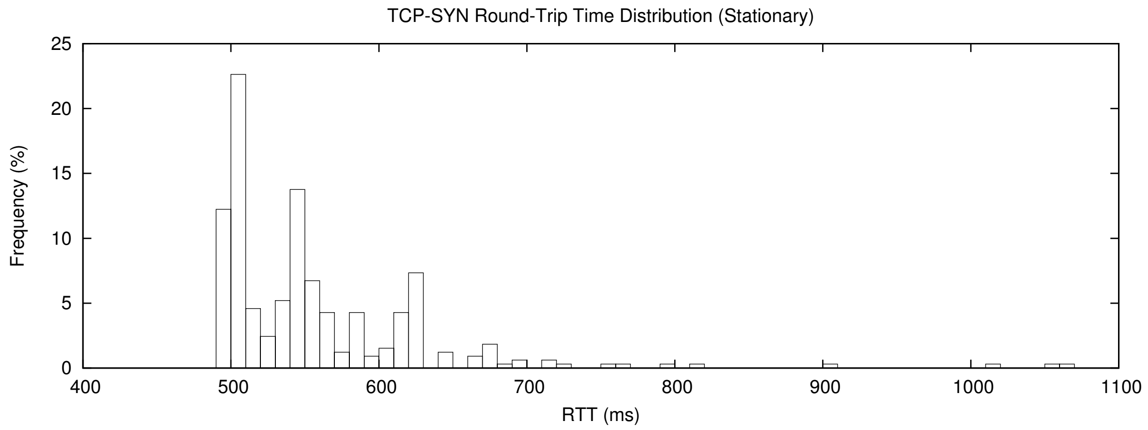


Figure A-16: Observed behaviour of raw UDP upload throughput on a GPRS interface, when (a) stationary ($\mu = 24.95$, $\sigma = 0.68$); and (b) moving ($\mu = 18.92$, $\sigma = 5.31$).

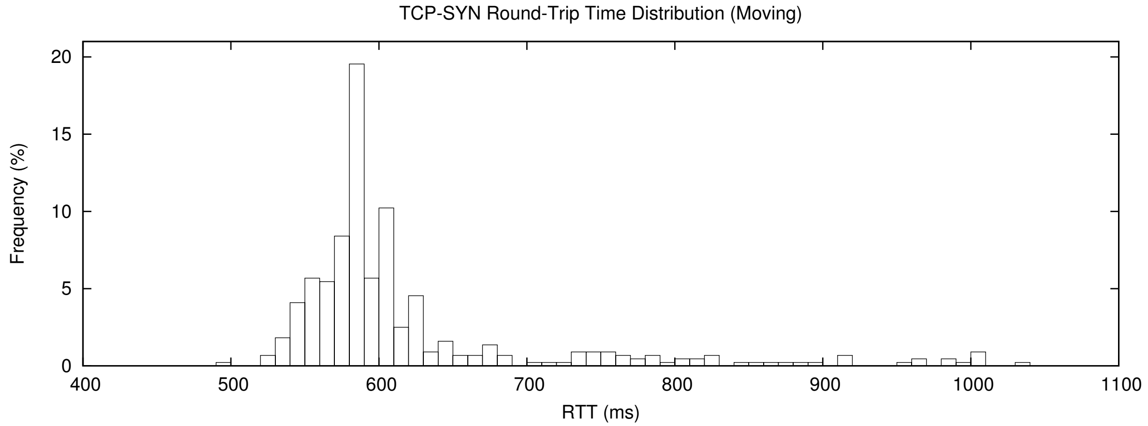
TCP-SYN Pings: Stationary We used the the `tcpping` utility to measure the round-trip-times for small TCP-SYN packets sent over a stationary GPRS interface[4]. Figure A-18a shows how measured round-trip-times varied during an experiment. Figure A-17a shows how these round-trip-times were distributed. Average TCP-SYN round-trip-times on GPRS links are almost $250ms$ higher than those on the CDMA2000 link (§A.3.5). The GPRS round-trip-time distribution is not bi-modal and the way in which round-trip-times vary on the GPRS link is not periodic like CDMA2000.

The secondary peaks in the round-trip-time distribution A-17b (at 545ms and 625ms) probably represent link level retransmissions, with the mode around 500ms representing the case where no retransmissions were needed for a packet.

[4] `tcpping` sends a TCP-SYN packet, waits for the SYN-ACK reply, calculates the round-trip-time, and then sends the next TCP-SYN packet.



(a) $\mu = 556ms$, $\sigma = 102ms$, $P(rtt > 1000ms) = 0.01$



(b) $\mu = 758ms$, $\sigma = 464ms$, $P(rtt > 1000ms) = 0.12$

Figure A-17: Observed distributions of single packet round-trip-times for TCP-SYN packets on a `GPRS` link when the transmitter is (a) stationary; and (b) moving.

TCP-SYN Pings: Moving As with available bandwidth (§A.4.3), motion seems to significantly impact the performance of the `GPRS` link. Figure A-17b shows the distribution of a moving `GPRS` interface's round-trip-times. Figure A-18b shows how measured packet round-trip-times varied during an experiment with a moving `GPRS` interface. The average latency when moving was $200ms$ larger than the average latency when stationary, and the standard deviation—at almost half a second—was over four times as large when moving. When moving, 12% of the sent packets experienced round-trip-times larger than a second.

Unlike `CDMA2000`, `GSM/GPRS` uses *hard hand-offs* and this could be a problem when moving. Changing base-stations would require the `GSM/GPRS` terminal to terminate

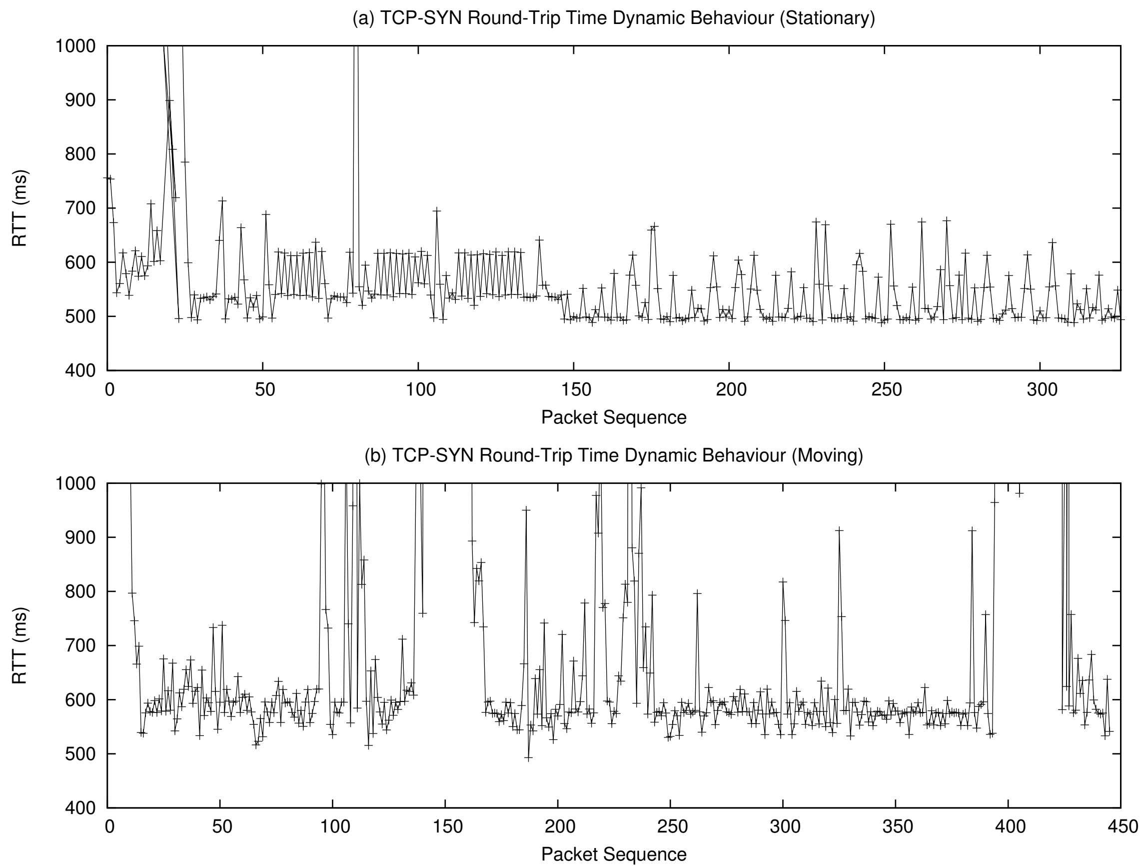


Figure A-18: Observed behaviour of single packet round-trip-times for TCP-SYN packets on a GPRS link when the transmitter is (a) stationary; and (b) moving.

the connection with the old base-station and to then negotiate a connection with the new base-station. The round-trip-time spikes in figure A-18b exceeding 1000ms could either be caused by base-station hand-offs or because lost packets needed many retransmissions, or both.

UDP Burst Latencies We did not look at UDP burst latencies because of the NAT on our GPRS path. However, other researchers studying GPRS links [14] have reported that when packets are sent in bursts, the round-trip-times in the burst are decreasing: the first packet usually experiences the largest latency, with others having progressively lower latencies. Furthermore, we know that the GPRS RLC uses sliding window flow control, and that our packets may be tunneled through TCP further downstream, so queue draining effects on the GPRS link should not be a surprise.

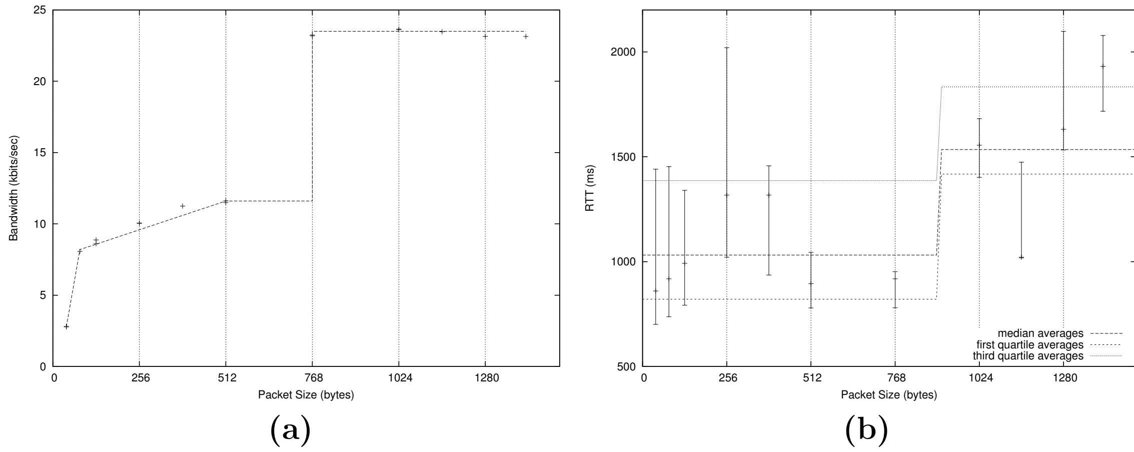


Figure A-19: Observed effects of changing packet sizes on the GPRS interface: (a) medium and small packet transmissions are highly inefficient on the link, but for packets larger than 768 bytes, packet size has no impact on goodput (b) the median round-trip-time for large packets is 1.5 times as large as the median round-trip-time for smaller packets.

A.4.5 Packet Size

We conducted a number of experiments to investigate the impact of picking a specific packet size on the performance of the GPRS link. Figure A-19 summarizes the observations from these experiments.

UDP Throughput Figure A-19a shows the achievable goodput obtained when UDP packets were sent using a GPRS sender sending packets at a constant rate defined by the following formula:

$$rate_{sender} = \frac{bitrate_{max}}{size_{packet}} \approx \frac{22.8kbits/sec}{size_{packet}}$$

Where $size_{packet}$ is the size of the packet payload; and $bitrate_{max}$ is based on the maximum observed UDP goodput on the channel from our earlier experiments.

From our observations, it seems that, as was the case with the CDMA2000 link (§A.3.6), the GPRS link is inefficient when dealing with small packets. In fact, for GPRS the effect is much more marked: instead of the gradual increase in efficiency

seen when packet sizes were increased on the CDMA2000 link, the efficiency of the GPRS link suddenly doubles when we start using 768 byte packets. For larger packets, goodput remains roughly constant.

Latency Figure A-19b shows how packet round-trip-times were distributed in experiments using TCP-SYN packets. The round-trip-time distribution of small and medium packets (768 bytes and smaller) is different from the round-trip-time distribution for larger packets (1024 bytes or more). The average quartiles for the two groups of packets are shown in the figure. The average median round-trip-time for large packets is over 500ms larger than the average median round-trip-time for smaller packets and the inter-quartile range for larger packets is $100ms$ less than this range for smaller packets.

Optimal Packet Size An interesting implication follows from our observations: 768 byte packets would allow us to reach the maximum goodput (figure A-19a), and also keep us in the lower round-trip-time group of packets (figure A-19b).

A.4.6 Loss Characteristics

As others have previously reported [14], packet loss rates observed on the GPRS link were much lower than would be expected on a raw wireless link. This is because of the effective use of link-layer forward error correction and link-layer acknowledgments. When AIMD-style flow control was used by us to avoid dropping packets at the sender, we observed that only around 1% of the packets were lost.

Using constant-rate packet generation processes to send packets from the GPRS interface to an MIT host, we ran a number of experiments with different rates to determine the nature of the packet losses. We were primarily interested in determining whether the losses on the interface were bursty in nature. Figure A-20 shows how packet losses were distributed relative to burst length on the GPRS link.

The bursty nature of the losses is illustrated by the fact that almost 75% of the packet losses occur in bursts of length 2 or greater. The mode of the burst-loss length

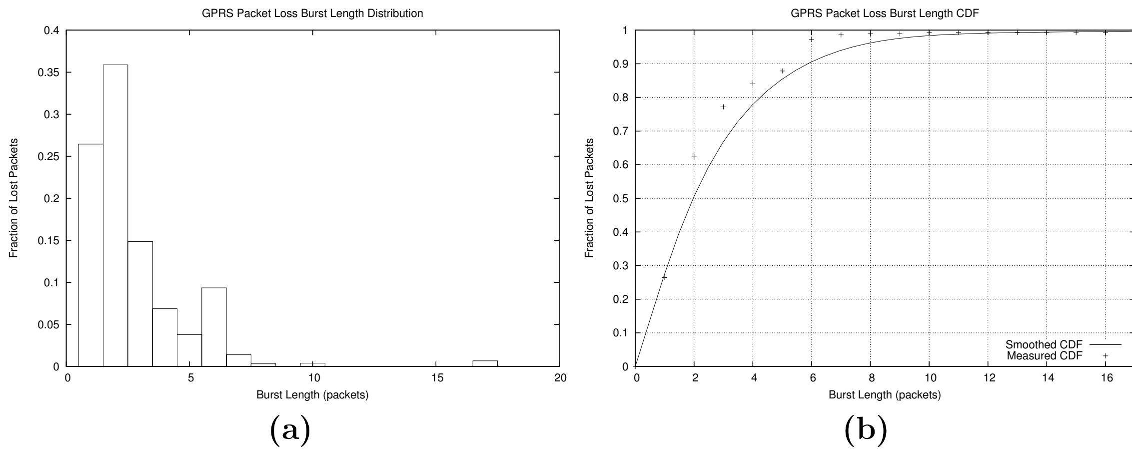


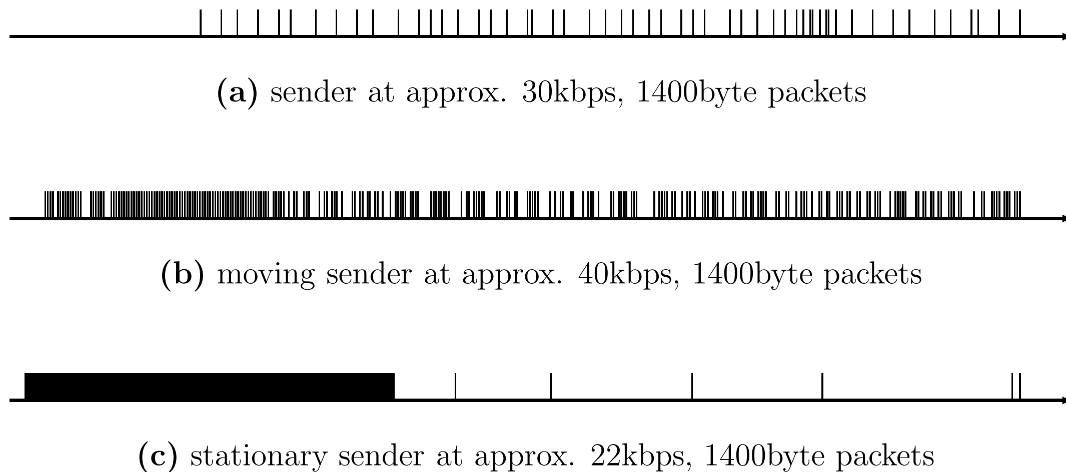
Figure A-20: Observed distribution of packet losses from many different experiments with constant-rate senders using a GPRS interface. For each packet loss, (a) the histogram tracks how many consecutive packets surrounding that lost packet were also lost; and (b) shows the cumulative density of this distribution.

distribution occurs at 2 (accounting for 35% of losses).

Figure A-21 shows the loss timelines for a few different experiments. The changing loss density seen in figure A-21b illustrates the fact that while the sender is sending at a constant rate, the available bandwidth is varying significantly due to motion. Figures A-21a and A-21c show the behaviour when the sender is sending at a bit-rate close to the average available bandwidth. The large startup transient burst loss in (a) is uncommon.

A.4.7 Disconnections

The GPRS interfaces seemed to have a relatively high rate of disconnection. In our experiments we noticed disconnections in our `pppd` sessions to the Nokia GPRS phones occurring on average every 15 minutes. In some locations, the disconnection seemed due to sparse cell tower coverage, and we could not effectively reconnect until we had moved. However, in the majority of the cases, we could reconnect quite quickly: all that we needed to do was to renegotiate the `pppd` session. Furthermore, when using multiple GPRS interfaces simultaneously, it was rare to see more than one of them disconnect at the same time. It is possible that these short-lived disconnections



(a) sender at approx. 30kbps, 1400byte packets

(b) moving sender at approx. 40kbps, 1400byte packets

(c) stationary sender at approx. 22kbps, 1400byte packets

Figure A-21: Packet loss timelines for constant-rate senders using a `GPRS` interface in three different experiments.

are the result of fundamental weaknesses within `GPRS`, or the provider's network setup, or the result of Linux driver problems. We believe the dropped calls were probably caused by signaling pathologies within the `GSM` network, problems such as the ping-pong effect experienced at cell-boundaries, in which a mobile continuously is handed-off between two base stations, until the call is finally dropped.

A.5 Multi-Channel Effects

This thesis is primarily concerned with the ability to simultaneously use multiple WWAN channels in order to aggregate their bandwidth. We are interested in whether the behaviour of multiple WWAN transmitters placed in close proximity exhibits any kind of correlation. We would like to know if a fall in available bandwidth on one channel would mirror a fall in another channel's bandwidth; predict a rise in the other's available bandwidth; or provide no information about the other WWAN channel whatsoever.

We conducted multiple experiments, both stationary and mobile, to compare the behaviour observed across multiple WWAN transmitters. We used five WWAN inter-

faces: one CDMA2000 1xRTT interface (Verizon), and 4 identical GSM/GPRS interfaces connected to the same service provider (T-mobile). The interfaces were placed in close spatial proximity, in no particular order. This section describes a few representative experiments.

A.5.1 Aggregate Bandwidth

As before, we tried to upload data at a rate close to the channels estimated capacity. We used a bit-rate of $22.4 kbits/sec$ for the GPRS interfaces, a bit-rate of $128 kbits/sec$ for the CDMA2000 interface, and a packet size of 1024 bytes for both. We measured the data arrival rate at the receiver in MIT. Table A-24, figure A-22 and figure A-23 show how the observed aggregate bandwidth behaved in two such experiments.

Stationary When stationary (figure A-22) the aggregated available bandwidth was relatively stable with a mean of around $179 kbits/sec$. Nonetheless figure A-22 shows two network abnormalities: the dip in the CDMA link's bandwidth; and the disconnection of the GPRS-1 interface a little over four minutes into the experiment.

After the loss of the higher bandwidth GPRS-1 channel, the available bandwidths on the other GPRS channels do not rise. In our single-channel GPRS experiments, we measured a sustainable GPRS upload throughput of around $24 kbits/sec$. The disconnection of GPRS-1 frees up bandwidth for the other phones, which are all operating well below $22 kbits/sec$. With four GPRS channels, only one of them is able to achieve $22 kbits/sec$, presumably because all available bandwidth is being used. We found that with T-mobile even three Nokia 3650 phones captured all available upload bandwidth. A fourth phone *steals* bandwidth away from some of the other phones. The throughputs probably stayed low after the disconnection because the experiment ended before the phones had had a chance to renegotiate with the base-station.

Moving Motion causes large variations in the available bandwidth (figure A-23), causing the standard deviation of the aggregate bandwidth to treble. In spite of the loss of GPRS-4 eight minutes into the experiment, the aggregate average remains

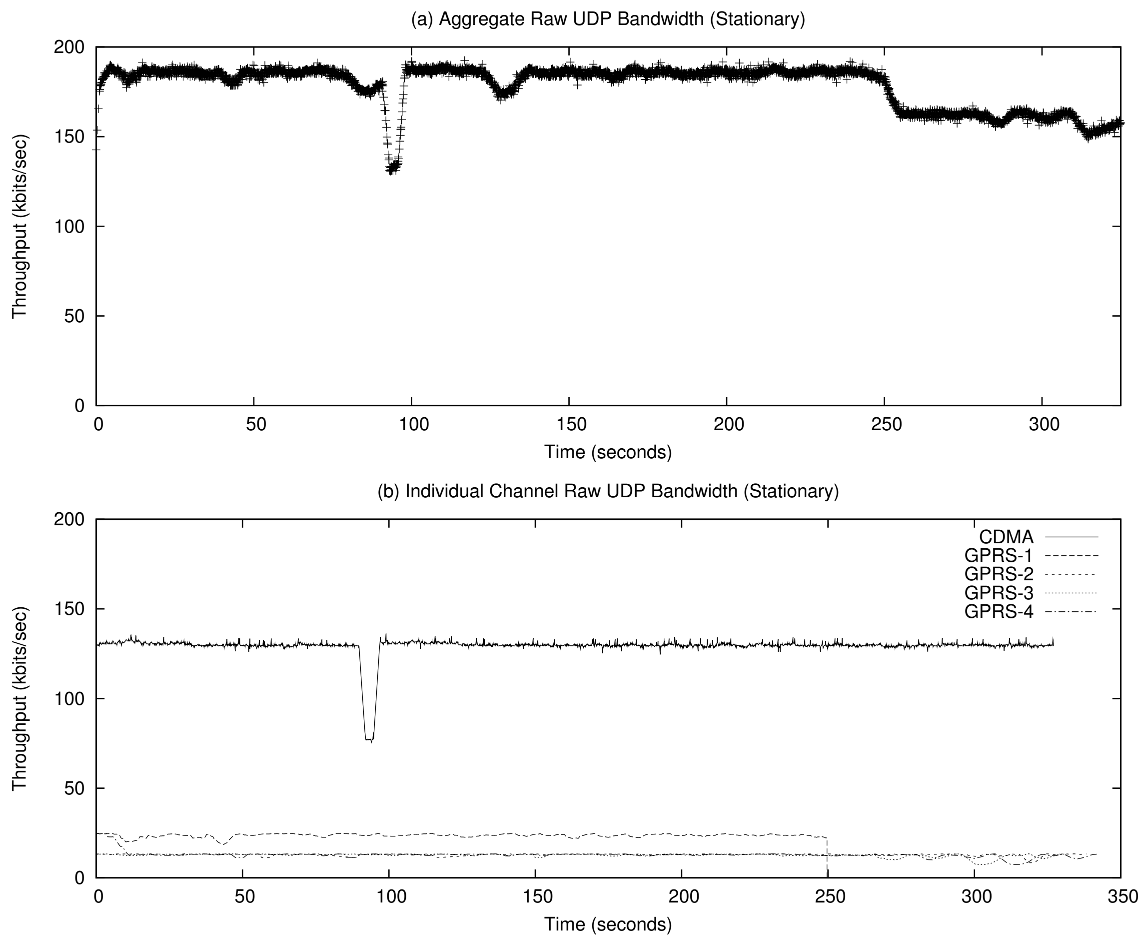


Figure A-22: Observed dynamic behaviour of available raw UDP upload bandwidth when multiple stationary interfaces are being used simultaneously: (a) aggregate upload bandwidth (b) individual channel bandwidths.

around $150kbits/sec$ throughout the experiment. The other GPRS channels are able to increase their bandwidths to compensate for the loss of GPRS-4. The fact that they are able to do this when moving and not when stationary, may be related to the issue of base-station hand-offs due to the motion. When a hand-off occurs, the GPRS interfaces can negotiate for more bandwidth from the new base-station. Furthermore, the fact that the throughput of the aggregate remains steady even when a single channel fails shows one way in which the aggregate channel has the potential to be more robust than individual channels.

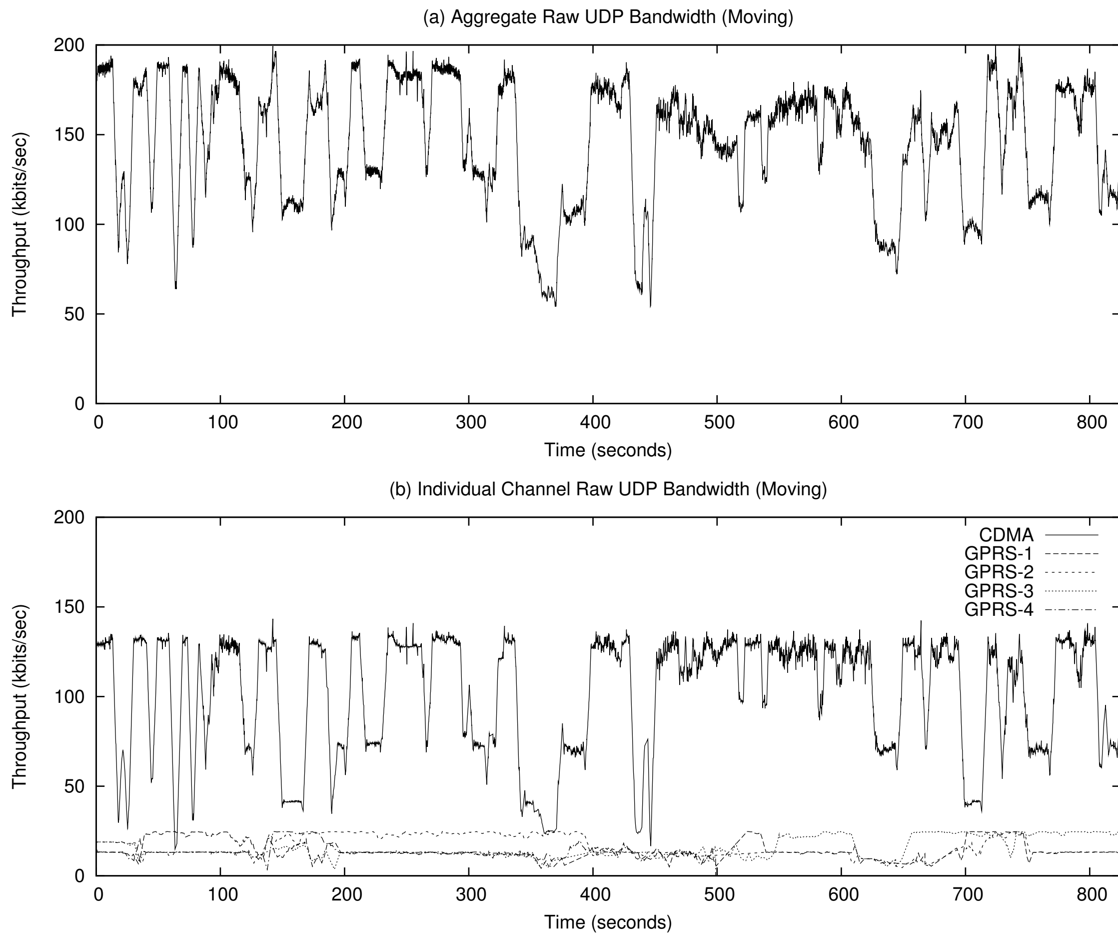


Figure A-23: Observed dynamic behaviour of available raw UDP upload bandwidth when multiple interfaces are being simultaneously used on a moving vehicle: (a) aggregate upload bandwidth (b) individual channel bandwidths.

	$\mu_{stationary}$	$\sigma_{stationary}$	μ_{moving}	σ_{moving}
CDMA	129.56	4.75	111.36	27.82
GPRS-1	23.64	1.33	16.68	5.50
GPRS-2	13.04	0.56	18.92	5.31
GPRS-3	12.71	0.83	18.31	5.54
GPRS-4	13.14	2.20	15.08	4.52
Aggregate	178.84	11.68	152.94	31.31

Figure A-24: Observed raw UDP bandwidths, for both moving and stationary multi-interface experiments. Averages are only for as long as the channel was active; they do not count periods of disconnection as zero.

A.5.2 Bandwidth Correlation

We want to know whether changes in bandwidth in observed on one WWAN channel are correlated with changes observed on another channel. We revisit the experiments in figures A-22 and A-23. By processing the data from one experiment, we can produce six discrete random variables, one for each row in table A-24. The throughput signals from the figures are all sampled at the same frequency, interpolating where necessary[5], and the sampled values give us the discrete random variables.

The correlation coefficients for these random variables can be calculated[6] and are shown in figure A-25. The coefficients in the two tables were calculated using around 500 samples from each experiment. Along with the coefficient matrix, we also calculated the matrix of corresponding p-values. p-values can be used for testing if the coefficients are statistically significant[7]. All coefficients that had an absolute value above 0.08 had p-values that indicated that correlation was significant for the related pair of variables.

The results in figure A-25 imply that, although some correlation exists, the `CDMA` and `GPRS` channels are weakly correlated at best, and the `GPRS` channels are never strongly correlated. In the stationary case (figure A-25a), the only significant correlation exists among the `GPRS` channel pairs. `CDMA-1` has no significant correlation with any other channel. This makes sense since the `CDMA-1` radio is very different from the others. The `GPRS` channels are correlated, but this correlation is weak, at best. In the mobile case (figure A-25a), the strongest correlations are again among `GPRS` channels. `GPRS-2` does correlate somewhat negatively with `CDMA`, but this is a very weak correlation. In both cases (figure A-25) the largest correlation coefficient involves the channel that failed during that experiment (`GPRS-4` in the mobile case, and `GPRS-1` in the stationary experiment).

[5]We used the `spline` function in `matlab` to interpolate, and sampled at one second intervals.

[6]We used the `corrcoef` function in `matlab` to calculate the coefficients.

[7]p-values from `corrcoef` can be used for testing the hypothesis of no correlation. Each p-value is the probability getting a correlation as large as the observed value by random chance, when the true correlation is zero. If $p_{i,j}$ is small, say less than 0.05, then the correlation for (i, j) is significant.

	CDMA	GPRS-1	GPRS-2	GPRS-3	GPRS-4	Aggregate
CDMA	1.0000	−0.0552	−0.0475	−0.0401	−0.0015	0.4015
GPRS-1		1.0000	0.1406	0.5619	0.3568	0.7724
GPRS-2			1.0000	0.0036	0.2218	0.2181
GPRS-3				1.0000	0.1690	0.4420
GPRS-4					1.0000	0.2413
Aggregate						1.0000

(a) stationary experiment (figure A-22)

	CDMA	GPRS-1	GPRS-2	GPRS-3	GPRS-4	Aggregate
CDMA	1.0000	0.0135	−0.2228	0.0850	−0.1330	0.8727
GPRS-1		1.0000	−0.0065	−0.0392	0.0298	0.1768
GPRS-2			1.0000	−0.1720	0.5166	0.0416
GPRS-3				1.0000	−0.5400	0.1091
GPRS-4					1.0000	0.0951
Aggregate						1.0000

(b) moving experiment (figure A-23)

Figure A-25: Channel bandwidth correlation coefficient matrices. Redundant values are omitted for clarity; the matrices are symmetric.

Appendix B

HOSE: the Horde Objective Specification Environment

This appendix describes the abstract interface Horde provides to allow applications to describe their network quality-of-service (QoS) *objectives*. Horde applications express QoS policy goals for streams in the form of succinct objectives. Objectives represent the modular decomposition of an application's utility function (§6.1), allowing the scheduler to determine which transmission schedules provide higher utility than others. Chapter 6 discussed the high-level semantic issues, this appendix covers the other language details.

The set of objectives describes the application policy regarding network quality-of-service. Each objective says something, relatively simple, about the sort of network QoS an application would like for some stream(s). The set of objectives describes how much an application values the assignment of a given type of `txSlot` to some ADU. An application injects one or more objectives into the library. Objectives can be injected and removed dynamically, allowing for the specification of time-varying utility functions. When an application does not specify an objective for some QoS aspect along some stream, it is implicitly assumed that the application does not care about that aspect of QoS on that stream.

The remainder of this appendix describes in detail the syntax and semantics of the `hose` language, used by applications to express their objectives.

B.1 Augmented BNF Notation Overview

The rules for hose syntax are defined in this appendix using a variant of Backus Naur Form (BNF) notation. We provide a brief overview of the notation.

<rule> ::= definition

> The ::= here means 'is defined as'. The definition specifies the BNF expansion for the rule. The definition can include other rules; definitions that recursively use the rule being defined are also possible.
>
> The angle brackets distinguish syntax rule names (also called non-terminal symbols) in the BNF from terminal symbols. The later are written exactly as they are to be represented in the final syntax. When special BNF tokens (::=, |, <, >) are intended to be used as terminal symbols, they are quoted as literals.
>
> By recursively expanding each non-terminal symbol in a rule definition, one can determine all the possible terminal symbol sequences that are valid instances of that rule in the final syntax this BNF rule is describing.

'literal'

> Quotation marks surround literal (case-sensitive) text, e.g., '<literal>' implies that this BNF fragment is to be viewed as defining the character string <literal> in the final syntax, even if a BNF rule named literal is defined.

rule1 | rule2

> Elements separated by a bar (|) are alternatives, e.g., yes | no implies that either yes or no are valid in the final syntax.

*element

> The character '*' preceding a BNF element indicates repetition. The full form is N*Melement indicating at least N and at most M occurrences of element in the expansion of the BNF. The default values are 0 for N and infinity for M. Therefore, *(element) allows any number of instances of element, including zero; 1*element requires at least one; and 1*2element allows one or two.

`+element`

Short-hand for `1*element`.

`[element]`

Square brackets enclose optional elements. The definitions `[element]` and `(0*1element)` are equivalent.

`(rule1 rule2)`

Elements enclosed in parentheses are grouped together as a single element.

```
<fubar> ::= elem (foo | bar) elem
```

Therefore, the above rule allows '`elem foo elem`' and '`elem bar elem`' to be valid in the final syntax, but not the expression '`elem foo bar elem`'.

`@TYPE(element)`

If the given rule can take on multiple run-time-types, the `@TYPE` preceding it specifies that we only mean those instances of the rule which are of the specified type. Some of the types we use here are: `@BOOLEAN`, `@NUMERIC` and `@STRING`.

`<letter>`

Convenient rule. Matches characters A-Z and a-z.

`<digit>`

Convenient rule. Matches characters 0-9.

`<whitespace>`

Convenient rule matching horizontal (space and tab characters) and vertical (linefeed and carriage return characters) whitespace.

The following sample BNF rule uses many of the constructs defined above.

```
<variable_name> ::= +(<letter> | '_' | <digit>)
```

According to this definition a variable name must consist of one or more characters: either letters, underscores or numerical digits. No whitespace can be part of the name.

B.2 The HOSE Language

The purpose of the hose language is to provide a flexible mechanism using which application objectives can be expressed and evaluated. Using literals, ADU's, stream's, header fields, and ADU latency and loss probability distributions as the basic units, the language allows the specification of complex constraints.

hose is similar to functional languages like LISP. The order in which objectives are evaluated is undefined. No mechanism exists for declaring arbitrary variables; although, a mechanism for binding variables to existing ADU's and streams is provided. The language does not provide any mutators, nor are there any other side-effects.

The language is type-safe. Every expression in hose has a well-defined type and there are specific rules governing how expressions of the same or different types can be composed and what the type of the composition will be. hose recognizes the following types: ADU, stream, boolean, numeric, string and probability distributions over boolean and numeric values. For simplicity, the numeric type in this language definition only supports integers. Consequently, probabilities are represented as percentage values ranging from 0 to 100.

The syntax of hose is not sensitive to either horizontal or vertical whitespace, except when that whitespace is used to delimit tokens (e.g. variable names and reserved words), or when it occurs inside string literals. The way we use whitespace in the BNF definitions requires some clarification. Except when a BNF fragment contains the OR (|) operator or hose delimiters (e.g., {, (, etc), the reader should implicitly assume that whitespace from the BNF maps to whitespace in the final syntax; whitespace surrounding a BNF OR operator does not imply whitespace in the final syntax is allowed, unless the <whitespace> rule is explicitly used.

hose is case-sensitive. ADU variable names, field names, reserved words, and the contents of string literals are all identified in a case-sensitive manner.

B.2.1 Defining Objectives

The hose language defines a single top-level construct: an objective.

```
<objective> ::= objective {
                  context  { <context> }
                  goal     { <goal>    }
                  utility  { <utility> }
                }
```

All valid fragments of hose code must start with an objective definition. An objective consists of three sections: a context (variable bindings); a goal (a boolean predicate); and a utility section (how application utility is affected when the goal predicate is true). For a schedule in which the goal predicate is true, the hose interpreter uses the utility section of the objective to determine how that goal's fulfillment has affected the utility derived from the associated transmission schedule.

B.2.2 Objective: Context

The context section of an objective provides the variable bindings that can be used inside the goal and utility sections.

```
<context> ::= *(<adu_binding> | <stream_binding>)
```

Each binding in the context section binds a variable of type ADU to an actual ADU, or a variable of type stream to an actual stream. Each binding expression is a filter defined over all the ADU's in the system; whichever ADU's pass through the filter are possible values for the given variable. The variable can be bound, and the objective evaluated, for each of these possible values. When there are multiple variables, there can be many possible sets of bindings.

```
<stream_binding> ::= stream:<variable> { stream_id == +<digit> }
<adu_binding>    ::= adu:<variable> { <adu_filter> }
```

```
<adu_filter> ::=  <adu_predicate>
                | ([‘!’] ‘(‘ <adu_filter> ‘)’)
                | (<adu_filter> (‘&&’ | ‘||’) <adu_filter>)
<variable> ::= +(<letter> | ‘_’ | <digit>)
```

As specified above, each `stream` variable is bound to the stream with the given numeric identifier, and `ADU` variable binding filters are defined as a series of simple boolean predicates composed using boolean `AND` and `OR` operations. Predicates are comparisons of an `ADU` header field with a simple expression that evaluates to the same type as that field (see the definition below). An ADU filter is just a specialized boolean expression, so the usual precedence rules apply (§B.2.5).

```
<field> ::= +(<letter> | ‘_’ | <digit>)
<adu_predicate> ::=  true
   | (@NUMERIC(<field>) <numeric_compare_op> <context_numeric_expression>)
   | (@BOOLEAN(<field>) <boolean_compare_op> <context_boolean_expression>)
   | (@STRING(<field>) <string_compare_op>  <string_literal>)

<numeric_compare_op> ::= ‘==’ | ‘!=’ | ‘<=’ | ‘>=’ | ‘<’ | ‘>’
<boolean_compare_op> ::= ‘==’ | ‘!=’
<string_compare_op>  ::= ‘==’ | ‘!=’

<context_boolean_expression> ::= <boolean_literal>
   | @BOOLEAN(@ADU(<variable>)::<field>)
   | [‘!’] ‘(’ <context_boolean_expression> ‘)’

<context_numeric_expression> ::= <numeric_literal>
   | @NUMERIC(@ADU(<variable>)::<field>) ‘+’ <numeric_literal>
   | @NUMERIC(@ADU(<variable>)::<field>) ‘-’ <numeric_literal>
   | @NUMERIC(@ADU(<variable>)::<field>) ‘*’ <numeric_literal>
   | @NUMERIC(@ADU(<variable>)::<field>) ‘/’ <numeric_literal>
```

```
objective {
  context {
     adu:one   {  stream_id == 0  }
     adu:two   { (stream_id == 0) && (stream_seq == one::stream_seq+1) }
  }
  goal { expected(two::latency) >= expected(one::latency) }
  utility { one { 1 } two { 1 } }
}
```

Figure B-1: An objective to limit reordering. The `ADU` variables are bound to consecutive ADU's from stream 0.

Applications can define arbitrary header fields in the ADU's they inject. Fields like `stream_id` and `stream_seq` are predefined fields. The field names `lost?` and `latency` are reserved (see §B.2.7) for probability distributions and cannot be referenced in the `context` section.

`ADU` variables can be referenced immediately after they are bound. Therefore, if there are multiple bindings defined in the `context`, the fields of the `ADU` variable bound first can be used in the predicate expressions for the second variable onwards. Figure B-1 shows the `context` for an `objective` used to limit reordering which uses this immediate binding feature.

B.2.3 Objective: Goal

The `goal` section must contain a `boolean` valued expression (see §B.2.5 for detailed syntax). The constant `false` is not a valid `goal`. This expression can use any of the `ADU` and `stream` variables declared in the `context`.

```
<goal> ::=  <boolean_expression>
```

B.2.4 Objective: Utility

The `utility` section specifies how application utility is affected when the objective has been met—when the `goal` is `true`. The `utility` section contains a numeric valued expression for each `ADU` or `stream` variable whose utility is affected. The utility section has the syntax:

```
<utility> ::=  +(<variable> { <numeric_expression> })
```

Whenever the goal is met, the specified utility will be added to the overall utility derived from the schedule for each given ADU and stream. Negative utilities bias the scheduler against schedules with the property specified in the related goal; positive utilities promote such schedules. The base utility of transmitting an ADU is zero. Therefore, a scheduler implementation can decide not to transmit an ADU whenever the set of objectives specifies that all possible transmission slots give negative utilities for that ADU.

B.2.5 Numbers, Booleans and Strings

Expressions that represent definite values within a fragment of hose code can either have a numeric, boolean or string type. Indefinite numeric and boolean values, represented as probability distributions, are also part of the language. ADU and stream variables are containers for fields. Fields can be of either definite or indefinite types.

Literals The simplest definite value expressions are literals:

```
<boolean_literal> ::= true | false
<numeric_literal> ::= [('+' | '-')]+<digit>
<string_literal>  ::= '''('*<character>)'''
```

boolean literals use the hose reserved words true and false. All numeric literals are integers. string literals can contain any characters other than quotation marks (") which are used to delimit string literals; there are no character escape sequences.

Arithmetic Operations numeric expressions can be composed using arithmetic operators, with the usual semantics, forming more complex numeric expressions:

```
<arithmetic_operation> ::=  <arithmetic_add> | <arithmetic_subtract>
                          | <arithmetic_multiply>
                          | <arithmetic_divide> | <unary_minus>
```

```
<arithmetic_add> ::= <numeric_expression> ‘+’ <numeric_expression>
<arithmetic_subtract>
        ::= <numeric_expression> ‘-’ <numeric_expression>
<arithmetic_multiply>
        ::= <numeric_expression> ‘*’ <numeric_expression>
<arithmetic_divide>
        ::= <numeric_expression> ‘/’ <numeric_expression>
<unary_minus> ::= ‘-’<numeric_expression>
```

Logical Operations `boolean` expressions can be composed using logical operators, with the usual semantics, forming more complex `boolean` expressions:

```
<logical_operation>
       ::= (<boolean_and> | <boolean_or> | <boolean_not>)
<boolean_and>
       ::= <boolean_expression> ‘&&’ <boolean_expression>
<boolean_or>
       ::= <boolean_expression> ‘||’ <boolean_expression>
<boolean_not> ::= ‘!’<boolean_expression>
```

Comparisons Any two expressions of the same type (`numeric`, `boolean` or `string`) can be compared. Each such comparison forms a `boolean` valued expression. `boolean` and `string` expressions can only be compared for equality and inequality; `numeric` values can be compared in the usual ways.

```
<comparison> ::= <numeric_comparison> | <boolean_comparison>
                | <string_comparison>
<numeric_comparison> ::=
    <numeric_expression> <numeric_compare_op> <numeric_expression>
<numeric_compare_op> ::= ‘==’ | ‘!=’ | ‘<=’ | ‘>=’ | ‘<’ | ‘>’
<boolean_comparison> ::=
    <boolean_expression> <boolean_compare_op> <boolean_expression>
```

```
<boolean_compare_op> ::= ‘==’ | ‘!=’

<string_comparison> ::=
    <string_expression> <string_compare_op> <string_expression>

<string_compare_op> ::= ‘==’ | ‘!=’
```

Conditionals The language provides some special forms for simple conditionals:

```
<max> ::= max ‘(’ <numeric_expression> ‘,’ <numeric_expression> ‘)’
<min> ::= min ‘(’ <numeric_expression> ‘,’ <numeric_expression> ‘)’
<abs> ::= abs ‘(’ <numeric_expression> ‘)’
```

Summary Summarizing the various forms of definite value expressions:

```
<boolean_expression> ::=  <boolean_literal>
                      | @BOOLEAN(<field_value>)
                      | <logical_operation>
                      | <comparison>
                      | ( ‘(’ <boolean_expression> ‘)’ )
<numeric_expression> ::=  <numeric_literal>
                      | @NUMERIC(<field_value>)
                      | <arithmetic_operation>
                      | <expectation> | <probability>
                      | <max> | <min> | <abs>
                      | ( ‘(’ <numeric_expression> ‘)’ )
<string_expression> ::= <string_literal>
                      | @STRING(<field_value>)
```

Operator Precedence An operator with higher precedence is applied first to determine the semantics for an expression. As in other languages like `C`, parentheses can be used to overcome these precedence rules, to apply a lower precedence operator before a higher precedence one. In `hose` the operators defined above have the following semantic precedence:

<table>
<tr><td>arithmetic UNARY MINUS -
boolean NOT !</td><td>highest precedence</td></tr>
<tr><td>boolean AND &&
boolean OR ||
arithmetic MULTIPLY *
arithmetic DIVIDE /</td><td>second highest</td></tr>
<tr><td>arithmetic ADD +
arithmetic SUBTRACT -</td><td>third highest</td></tr>
<tr><td>ALL comparisons
==, !=, <=, >=, <, ></td><td>lowest precedence</td></tr>
</table>

B.2.6 ADU and Stream Variables

`ADU` and `stream` variables are bound to actual ADU's and streams in the `objective`'s `context` section. In `hose` these variables represent collections of simpler variables. `ADU` and `stream` variables have fields which can be accessed using the variable name and the scope operator (`::`). Some `ADU` fields are predefined and managed by Horde, and applications can define other arbitrary fields in the ADU headers. `stream` variables have a limited number of predefined numeric fields. References to these fields result in expressions that have the same typed value as the field:

The following `ADU` fields are predefined and managed by Horde:

```
stream_id    stream_seq    length    queue_delay          (NUMERIC)
lost?        latency                                      (PDF's)
```

`stream_id` and `stream_seq` define an ADU's parent stream and position in that stream. `length` is the size of the ADU payload in bytes. `latency` represents the elapsed time between when that ADU was inserted into Horde by the application and when the `ack` event for that ADU was delivered to the application. `queue_delay` is the local queuing delay for that ADU. `lost?` is `true` in the event that the ADU was dropped. `latency` and `lost?` are random variables (see §B.2.7).

stream variables have numeric fields for the stream loss rate, mean latency and the latency's standard deviation. The following stream fields are predefined:

```
stream_id    latency_ave    latency_sdev    loss_rate    (NUMERIC)
```

stream variables are included to allow the expression of certain types of constraints. stream variables are needed, for example, in objectives based on the ADU latency average[1] or jitter.

```
@STRING(<field_value>)  ::=  @STRING(@ADU(<variable>)::<field>)
@NUMERIC(<field_value>) ::=  @NUMERIC(@ADU(<variable>)::<field>)
                           | @NUMERIC(@STREAM(<variable>)::<field>)
@BOOLEAN(<field_value>) ::=  @BOOLEAN(@ADU(<variable>)::<field>)


@PDF_BOOLEAN(<field_value>) ::=  @ADU(<variable>)::lost?
@PDF_NUMERIC(<field_value>) ::=  @ADU(<variable>)::latency
```

B.2.7 Probability Distributions

Horde does not always have complete information during objective evaluation. For instance, Horde does not know the round-trip latency for a transmitted ADU until an ack has been received, nor does it know about a loss until a nack or a timeout. Horde can, however, derive expectations for latency and loss using its channel models. We incorporate this notion of incomplete information into the hose language as typed random variables. Expressions based on probability distributions can be referenced inside the goal and utility sections.

We do not provide a generalized random-variable (or PDF) type and generic PDF manipulation operators. The underlying network models used by Horde are not mature enough to generate probabilities for arbitrarily conditioned network events. Correlated ADU loss, single ADU loss, and ADU ack arrival time are well-defined network events, which Horde's network models can handle.

The ADU loss and latency random variables are accessible as special ADU fields:

[1] The average latency is different from the *expected* latency of a transmission

```
<adu_loss_expr>    ::= @ADU(<variable>)::lost?

<adu_latency_expr> ::= @ADU(<variable>)::latency
```

The lost? field represents the likelihood of the event that the ADU was lost in the network. latency is the time in milliseconds that elapses between the injection of an ADU into Horde and the reception of an ack for that ADU. latency is the queue_time plus the round-trip-time.

A special form provides the correlated loss distribution over a pair of ADU's:

```
<correlated_loss_expr> ::=
        correlated_loss '(' @ADU(<variable>), @ADU(<variable>) ')'
```

The correlated_loss distribution represents the likelihood of the event that the losses for the specified ADU's were correlated. Correlated loss is the event:

$$P(correlated\ loss) = P(A\ lost\ \wedge\ B\ lost \mid A\ or\ B\ lost)$$

There are two types of random variables in hose:

```
<boolean_pdf_expression> ::= <adu_loss_expr> | <correlated_loss_expr>

<numeric_pdf_expression> ::= <adu_latency_expr>

<pdf_expression> ::=  <boolean_pdf_expression>
                    | <numeric_pdf_expression>
```

Both loss expressions yield values which are probability distributions over {true, false}. The latency of an ADU is a probability distribution over the set of natural numbers. If an objective for an ADU is evaluated after an ack has been received then—although the hose type of a loss or latency expression is still a PDF—the PDF will represent a definite value (only one value in that PDF will have a positive probability; that probability will be 1).

In order to manipulate an expression of a PDF type, it must first be transformed into a numeric expression[2]. boolean PDF's can be transformed into numeric expressions by taking the probability of true in that PDF. Since hose's numeric types are

[2]Because of this requirement, the possibility of entirely avoiding the notion of probability distributions within the language arises. a::lost? could be redefined as having the semantics of

integers, probabilities are represented as percentages—integers between 0 and 100—instead of as fractions in the range $[0, 1]$. `numeric` PDF's can be transformed into `numeric` expressions by taking the expected value over that PDF.

```
<probabilty>  ::= prob '(' <boolean_pdf_expression> ')'
<expectation> ::= expected '(' <numeric_pdf_expression> ')'
```

B.3 Sample Objectives

Example objectives are given in figures 6-1, 6-2, 6-3, 7-4, 7-8, 7-12, 7-19 and B-1.

B.4 Complete HOSE Grammar

```
; hose reserved tokens:
:     objective context goal utility adu stream true false correlated_loss
;     prob expected max min abs lost? latency stream_id stream_seq length
;     queue_delay loss_rate latency_ave latency_sdev

; variable types: @ADU, @STREAM
<variable> ::= +(<letter> | '_' | <digit>)
; fields have value types: @NUMERIC, @BOOLEAN, @STRING and the PDF's
<field>    ::= +(<letter> | '_' | <digit>)

; top-level language constructs
<objective> ::= objective {
                  context { <context> }
                  goal    { <goal>    }
                  utility { <utility> }
                }
```

`prob(a::lost?)` and `a::latency` could be redefined as `expected(a::latency)`. Our inclusion of the probability type is partly carried over from an earlier iteration of the `hose` language, in which distributions could be manipulated much more freely. That said, our inclusion of the probability type in the current language specification reflects our desire to keep explicit which values are indefinite.

```
; objective components
<context> ::= *(<adu_binding> | <stream_binding>)
<goal>    ::=  <boolean_expression>
<utility> ::=  +(<variable> { <numeric_expression> })

; context components
<stream_binding> ::= stream:<variable> { stream_id == +<digit> }
<adu_binding>    ::= adu:<variable> { <adu_filter> }
<adu_filter>     ::=  <adu_predicate>
                   | ([‘!’] ‘(‘ <adu_filter> ‘)’)
                   | (<adu_filter> (‘&&’ | ‘||’) <adu_filter>)
<adu_predicate> ::=  true
         | (@NUMERIC(<field>) <numeric_compare_op> <context_numeric_expression>)
         | (@BOOLEAN(<field>) <boolean_compare_op> <context_boolean_expression>)
         | (@STRING(<field>) <string_compare_op>  <string_literal>)

; expressions for the context section
<context_numeric_expression> ::= <numeric_literal>
         | @NUMERIC(@ADU(<variable>)::<field>) ‘+’ <numeric_literal>
         | @NUMERIC(@ADU(<variable>)::<field>) ‘-’ <numeric_literal>
         | @NUMERIC(@ADU(<variable>)::<field>) ‘*’ <numeric_literal>
         | @NUMERIC(@ADU(<variable>)::<field>) ‘/’ <numeric_literal>

<context_boolean_expression> ::= <boolean_literal>
         | @BOOLEAN(@ADU(<variable>)::<field>)
         | [‘!’] ‘(’ <context_boolean_expression> ‘)’

; probability distribution expressions
<boolean_pdf_expression> ::= <adu_loss_expr> | <correlated_loss_expr>
<numeric_pdf_expression> ::= <adu_latency_expr>
<pdf_expression> ::= <boolean_pdf_expression> | <numeric_pdf_expression>

<adu_loss_expr>    ::= @ADU(<variable>)::lost?
<adu_latency_expr> ::= @ADU(<variable>)::latency
<correlated_loss_expr> ::= correlated_loss ‘(’ @ADU(<variable>), @ADU(<variable>) ‘)’
```

```
; pdf->value conversions
<probabilty>  ::= prob ‘(’ <boolean_pdf_expression> ‘)’
<expectation> ::= expected ‘(’ <numeric_pdf_expression> ‘)’

; the typed value from an adu or stream field
@STRING(<field_value>)  ::=  @STRING(@ADU(<variable>)::<field>)
@NUMERIC(<field_value>) ::=  @NUMERIC(@ADU(<variable>)::<field>)
                           | @NUMERIC(@STREAM(<variable>)::<field>)
@BOOLEAN(<field_value>) ::=  @BOOLEAN(@ADU(<variable>)::<field>)

@PDF_BOOLEAN(<field_value>) ::=  @ADU(<variable>)::lost?
@PDF_NUMERIC(<field_value>) ::=  @ADU(<variable>)::latency

; string value expressions
<string_expression> ::= <string_literal> | @STRING(<field_value>)
; characters surrounded by quotes.
<string_literal>    ::= ‘‘‘’(*<character>)‘’’’

; numeric value expressions
<numeric_literal>    ::= [(‘+’ | ‘-’)]+<digit>
<numeric_expression> ::=  <numeric_literal>
                        | @NUMERIC(<field_value>)
                        | <arithmetic_operation>
                        | <expectation> | <probability>
                        | <max> | <min> | <abs>
                        | ( ‘(’ <numeric_expression> ‘)’ )

; arithmetic operator precedance:
;     {unary-} > {*, /} > {+, -} > {==, !=, <=, >=, <, >}
<arithmetic_operation> ::= <arithmetic_add> | <arithmetic_subtract> |
                           <arithmetic_multiply> | <arithmetic_divide> |
                           <unary_minus>
<arithmetic_add>      ::= <numeric_expression> ‘+’ <numeric_expression>
<arithmetic_subtract> ::= <numeric_expression> ‘-’ <numeric_expression>
<arithmetic_multiply> ::= <numeric_expression> ‘*’ <numeric_expression>
<arithmetic_divide>   ::= <numeric_expression> ‘/’ <numeric_expression>
```

```
<unary_minus>          ::= ‘-’<numeric_expression>

; conditionals
<max> ::= max ‘(’ <numeric_expression> ‘,’ <numeric_expression> ‘)’
<min> ::= min ‘(’ <numeric_expression> ‘,’ <numeric_expression> ‘)’
<abs> ::= abs ‘(’ <numeric_expression> ‘)’

; boolean value expressions
<boolean_literal>    ::= true | false
<boolean_expression> ::=  <boolean_literal>
                        | @BOOLEAN(<field_value>)
                        | <logical_operation>
                        | <comparison>
                        | ( ‘(’ <boolean_expression> ‘)’ )

; logical operator precedence: {!} > {&&, ||} > {==, !=}
<logical_operation> ::= <boolean_and> | <boolean_or> | <boolean_not>
<boolean_and> ::= <boolean_expression> ‘&&’ <boolean_expression>
<boolean_or>  ::= <boolean_expression> ‘||’ <boolean_expression>
<boolean_not> ::= ‘!’<boolean_expression>

; comparisons
<comparison>  ::=  <boolean_comparison>
                 | <numeric_comparison>
                 | <string_comparison>
<boolean_comparison>
      ::= <boolean_expression> <boolean_compare_op> <boolean_expression>
<boolean_compare_op> ::= ‘==’ | ‘!=’
<string_comparison>
      ::= <string_expression> <string_compare_op> <string_expression>
<string_compare_op>  ::= ‘==’ | ‘!=’
<numeric_comparison>
      ::= <numeric_expression> <numeric_compare_op> <numeric_expression>
<numeric_compare_op> ::= ‘==’ | ‘!=’ | ‘<=’ | ‘>=’ | ‘<’ | ‘>’

;---- end ----
```

Bibliography

[1] Advantages of CDMA2000.
http://www.cdg.org/technology/3g/advantages_cdma2000.asp.

[2] ASTRA project.
http://nms.lcs.mit.edu/.

[3] BARWAN: the Bay-Area Wireless Network.
http://daedalus.cs.berkeley.edu/.

[4] Massachusetts General Hospital Stroke Service.
http://www.stopstroke.org.

[5] MIT class: Discrete Time Signal Processing.
http://web.mit.edu/6.344/www/.

[6] MPLAYER: the linux video player.
http://www.mplayerhq.hu/.

[7] Hari Adiseshu, Guru M. Parulkar, and George Varghese. "A Reliable and Scalable Striping Protocol". In *SIGCOMM*, pages 131–141, 1996. http://citeseer.nj.nec.com/adiseshu96reliable.html.

[8] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. "System support for bandwidth management and content adaptation in Internet applications". In *Proceedings of 4th Symposium on Operating Systems Design and Implementation, USENIX*, pages 213–226, October 2000.

[9] J. Apostolopoulos and S. Wee. "Unbalanced Multiple Description Video Communication using Path Diversity". citeseer.ist.psu.edu/apostolopoulos01unbalanced.html.

[10] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. "A comparison of mechanisms for improving TCP performance over wireless links". *IEEE/ACM Transactions on Networking*, 5(6):756–769, 1997.

[11] Deepak Bansal and Hari Balakrishnan. Binomial Congestion Control Algorithms. In *IEEE Infocom 2001*, Anchorage, AK, April 2001.

[12] Ali Begen, Yucel Altunbasak, and Ozlem Ergun. "Multi-path Selection for Multiple Description Encoded Video Streaming". In *IEEE ICC*, 2003.

[13] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, pages 24–35, 1994.

[14] Joel Cartwright. "GPRS Link Characterisation". http://www.cl.cam.ac.uk/users/rc277/linkchar.html.

[15] R. Chakravorty, J. Cartwright, and I. Pratt. "Practical Experience With TCP over GPRS". In *IEEE GLOBECOM*, 2002.

[16] Benjamin A. Chambers. "The Grid Roofnet: a Rooftop Ad Hoc Wireless Network". http://citeseer.nj.nec.com/chambers02grid.html.

[17] D. Clark and D. Tennenhouse. "Architectural Consideration for a New Generation of Protocols". In *ACM SIGCOMM*, 1990.

[18] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. "Packet-dispersion techniques and a capacity-estimation methodology". *IEEE/ACM Trans. Netw.*, 12(6):963–977, 2004.

[19] J. Duncanson. "Inverse Multiplexing". *IEEE Communications Magazine*, pages 34–41, April 1994.

[20] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. "Exokernel: an operating system architecture for application level resource management". 1995.

[21] N. Feamster and H. Balakrishnan. "Packet Loss Recovery for Streaming Video". In *the 12th International Packet Video Workshop*, April 2002.

[22] Nicholas G. Feamster. "Adaptive Delivery of Real-Time Streaming Video". citeseer.ist.psu.edu/443706.html.

[23] Nick Feamster. Adaptive Delivery of Real-Time Streaming Video. MEng Thesis, Massachusetts Institute of Technology, May 2001.

[24] P. Fredette. "The Past, Present, and Future of Inverse Multiplexing". *IEEE Communications Magazine*, pages 42–46, April 1994.

[25] Vijay K. Garg. "Wireless Network Evolution: 2G to 3G". Prentice Hall Communications.

[26] Ekram Hossain and Nadim Parvez. Enhancing tcp performance in wide-area cellular wireless networks: transport level approaches. pages 241–289, 2004.

[27] H. Hsieh, K. Kim, and R. Sivakumar. "a Transport Layer Approach for Achieving Aggregate Bandwidths on Multi-homed Hosts". In *ACM MOBICOM*, pages 83–94, 2002.

[28] H. Hsieh, K. Kim, Y. Zhu, and R. Sivakumar. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. In *Mobicom*, pages 1–15, 2003.

[29] M. Jain and C. Dovrolis. "End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput", 2002.

[30] Nancy Joyce. "Building Stata : The Design and Construction of Frank O. Gehry's Stata Center at MIT". The MIT Press.

[31] D. Katabi, M. Handley, and C. Rohrs. "Internet Congestion Control for Future High Bandwidth-Delay Product Environments", August 2002.

[32] L. Magalhaes and R. Kravets. MMTP: Multimedia multiplexing transport protocol, 2001.

[33] Luiz Magalhaes and Robin Kravets. "Transport Level Mechanisms for Bandwidth Aggregation on Mobile Hosts". In *ICNP*, 2001.

[34] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall. "MPEG Video Compression Standard". 1996. Chapman and Hall, International Thomson Publishing.

[35] Jeffrey M. Perloff. "Microeconomics 3rd edition" . Addison Wesley Publishing Company.

[36] Pablo Rodriguez, Rajiv Chakravorty, Julian Chesterfield, Ian Pratt, and Suman Banerjee. MAR: A Commuter Router Infrastructure for the Mobile Internet. In *Mobisys*, 2004.

[37] E. Setton, Y. J. Liang, and B. Girod. Multiple description video streaming over multiple channels with active probing. In *IEEE International Conference on Multimedia and Expo*, 2003.

[38] P. Sinha, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. "WTCP: A Reliable Transport Protocol for Wireless Widea-Area Networks". University of Illinois at Urbana-Champaign.

[39] Alex Snoeren. "Adaptive Inverse Multiplexing for Wide-Area Wireless Networks". In *IEEE conference on Global Communications*, pages 1665–1672, 1999. http://citeseer.nj.nec.com/snoeren99adaptive.html.

[40] Joshua A. Tauber. Issues in building mobile-aware applications with the Rover toolkit. Master's thesis, Massachusetts Institute of Technology, May 1996.

[41] Michael C. Toren. tcptraceroute: an implementation of traceroute using TCP SYN packets. http://michael.toren.net/code/tcptraceroute/.

[42] Y. Xiao, D. Gagliano, M. P. LaMonte, P. Hu, W. Gaasch, R. Gunawadane, and C. Mackenzie. “Design and Evaluation of a Real-Time Mobile Telemedicine System for Ambulance Transport”. *Journal of High Speed Networks*, pages 47–56, September 2000. http://hrfp.umm.edu/Papers/2000/mobiletelemed.pdf.