Horde: Separating Network Striping Policy from Mechanism

Asfandyar Qureshi and John Guttag
MIT Computer Science and AI Laboratory
{asfand, guttag}@csail.mit.edu

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 an immutable packet scheduling policy and allow applications to be oblivious of the way in which packets are routed to specific network channels. Though this is appropriate for many applications, other applications can benefit from an approach that explicitly involves the application in the dynamic determination of the striping policy.

Horde is middleware that facilitates flexible striping in wireless environments for a diverse range of applications. Horde separates the striping policy from routing and scheduling. It allows applications to specify network quality-of-service objectives that the striping mechanism attempts to satisfy. Horde can be used by a set of application data streams, each with its own quality-of-service policy, to flexibly stripe data over a highly heterogeneous set of dynamically varying wireless network channels.

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.

1 Introduction

Horde is networking middleware that provides a simple and robust way for multi-stream applications to communicate over multiple channels with widely varying latency and bandwidth. The key problems it addresses are:

- providing applications with a way to influence the scheduling of packets over channels, without building into the applications knowledge about channel availability or characteristics, and
- providing a mechanism that uses this information to derive appropriate packet transmission schedules for these time-varying 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 (on the order of 300kbps), bi-directional audio, and uni-directional physiological data streams (EKG, blood pressure, etc) from a moving ambulance. By relaying real-time telemetry and video from ambulances, we hope to provide EMS teams with expert opinions on complex trauma injuries, and to aid the in-hospital teams in better preparing themselves for incoming patients. Our telemedicine system must be economically viable to build, deploy, and operate. We therefore will leverage existing communications infrastructure, instead of building our own network infrastructure.

In most urban areas, there are a large number of public carrier wireless channels providing mobile connectivity to the Internet (e.g., GPRS and CDMA). The upstream bandwidth offered by these Wireless Wide Area Network (WWAN) channels is typically rather limited and each channel provides little in the way of network Quality-of-Service (QoS) guarantees.

These issues led us to consider using inverse multiplexing, or network striping, to aggregate several of these WWAN channels to construct a virtual channel. Network striping takes data from the larger source channel and sends it in some order over the smaller channels, possibly reassembling the data in the correct order at the other end before passing it to the application. By taking advantage of service provider diversity, overlapping coverage, and network technology diversity, we can use striping to provide an application with the illusion of a reliable stable high-bandwidth channel.

	Mean (μ)	Sdev (σ)
GPRS upload bandwidth (stationary)	$25kbps$	1
(moving)	$19kbps$	5
CDMA upload bandwidth (stationary)	$130kbps$	5
(moving)	$120kbps$	22
GPRS small packet RTT (stationary)	$560ms$	100
(moving)	$760ms$	460
CDMA small packet RTT (stationary)	$460ms$	90
(moving)	$470ms$	120
768-byte packet RTT (CDMA)	$810ms$	
(GPRS)	$920ms$	

Figure 1: Summary of WWAN QoS characteristics.

1.1 WWAN Striping Challenges

A great deal of work has been done on network striping [1, 8, 9, 12, 15, 18]. Most of this work is aimed at providing improved scheduling algorithms under the assumption that the underlying links are relatively stable and homogeneous, and that application streams are similar (e.g., TCP). If these assumptions hold, 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 advantageous.

However, these assumptions of homogeneity and stability are unrealistic for the WWAN channels we are using [6, 14, 15]. The bandwidth/latency characteristics of the available WWAN channels can vary by as much as an order of magnitude. In addition to heterogeneity among channels, we also expect there to be a high degree of temporal variation in QoS on each WWAN channel. QoS varies in time, partly because of vehicular motion and partly because of competition with other users on the WWAN. Spatial variation of the QoS depends on the carrier's placement of cell-towers relative to the terminal.

Our experiments with existing WWANs (GSM/GPRS and CDMA2000 1xRTT) in the Boston area provide evidence of heterogeneity and high QoS variability [14]. Figure 1 summarizes our experimental observations.

Since the WWANs are neither stable nor homogeneous, the manner in which the middleware decides to schedule the transmission of application packets can have a large influence on data stream latencies, bandwidth, and loss rates. Furthermore, the data streams in our telemedicine system are heterogeneous with respect to which aspects of the network service they are sensitive to: some streams care about latency (e.g., video streams), some not (e.g., bulk-data transfers); some care about loss more than others (e.g., audio); and some care more about jitter than they do about latency (e.g., non-interactive video). Therefore we want to give the application some control over how striping is done.

1.2 Horde's Approach

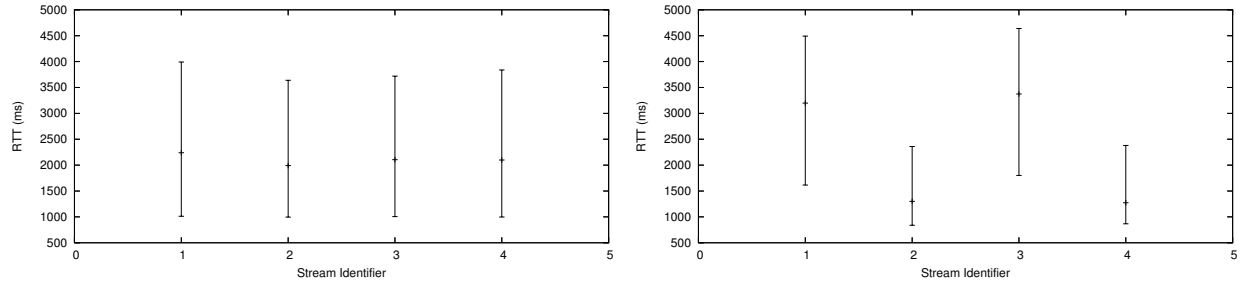
Horde separates the striping *mechanism* from the striping *policy*, the latter being specified by the application in an abstract manner. The key technical challenge in Horde is giving the application control over certain aspects of the data striping operation (e.g., an application may want urgent data 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.

In addition to aggregating bandwidth, Horde allows an application to modulate network QoS for its streams. Horde allows an application to express its policy goals as succinct network QoS *objectives*. Each objective says something, relatively simple, about the *utility* an application gains from some aspect of network QoS on a stream. Objectives can take into account such things as expected latencies, observed loss-rates, and expected loss correlations. Using the set of expressed application objectives, Horde attempts to schedule packets at the sender so as to maximize the expected utility derived by the application from the resulting sequence of packet receptions.

Allowing an application to actively influence the striping operation can be beneficial. By allowing the application to express its desired goals, Horde can arrive at efficient transmission schedules that provide high utility.

Consider, for example, a simplified version of our telemedicine application. There are four data streams: EKG physiological data (stream 1), video (stream 2), additional physiological data (stream 3), and audio (stream 4). Figure 2a shows how packet latencies were distributed on these streams when striping over three WWAN channels, using a packet scheduler that did not distinguish between the data streams. All streams received roughly the same QoS. However, two of these streams (video and audio) are latency sensitive. An objective can be used that expresses that smaller packet latencies on the video and audio streams give the application more utility than larger latencies. Figure 2b shows that, with this single objective, Horde was able to provide lower-latency for the latency sensitive streams.

The telemedicine system contains more complex examples where QoS modulation is advantageous. An encoded video stream may contain both reference frames (I-frames) and delta 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. An application therefore derives more utility from an I-frame than it does from



(a) Randomized Round Robin

(b) Objective Driven Scheduling

Figure 2: Packet round-trip-time distributions with different schedulers, striping over three channels. The two schedulers were run over the same packet traces, collected from three existing WWAN channels (one CDMA and two GPRS channels). The graphs show the median and the upper and lower quartile packet latencies for the streams.

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 so that I-frame losses are uncorrelated. 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.

The Horde middleware is designed to ease the development of applications that care about network QoS and need to use WWAN striping. Application-specific striping code has been used in the past. Many multi-path video streaming applications [3, 5, 16] exemplify this approach. In these systems, 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.

Horde is not meant to be general networking middleware. Existing applications would have to be rewritten to take advantage of Horde's QoS modulation framework. Further, 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 [1, 10, 18] may be more appropriate.

Horde is most useful when dealing with:

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.

Heterogeneous/Time-varying Network Channels With such channels, the scheduler has an opportunity to significantly modulate QoS. Modulation can be more accurate when channel characteristics are predictable in (at least) the short term.

Bandwidth-Limited Application Applications that want to send more data than individual physical channels can support justify both the network striping operation and the additional processing cost of Horde's QoS modulation framework.

Single Application Our model is that of a single application with multiple streams. While the application may be split across multiple processes, we assume a consistent system-wide policy in this paper. We therefore ignore the effects of adversarial behaviour.

1.3 Paper Organization

We have introduced the problem domain and outlined our approach and motivations above. Section 2 presents an overview of the Horde architecture. Section 3 discusses our approach to separating application policy from the striping mechanism. Section 4 presents some experimental results. Section 5 describes related research. Finally, section 6 presents a summary and conclusion.

2 Horde Architecture

Horde provides to an application the ability to:

- Stripe data streams over a dynamically varying set of heterogeneous network channels;
- Abstractly define striping policy;
- Per-network-channel congestion control;
- Explicit flow control feedback for each stream.

This section presents a high-level overview of Horde's application interface and internal structure. We also discuss selected aspects of Horde. We defer discussion of the scheduler and policy interface to section 3.

2.1 Overview

Application Interface

Horde uses a connection-based model. The applications on the source and destination nodes must negotiate a connection, or *stream*, before they start sending data to each other. An application can register event callbacks for each data stream it creates. When sending/receiving data on a stream, applications communicate with Horde at the granularity of ADU's. Our design draws from previous arguments for application level framing [7, 2].

Horde manages stream flow-control. Streams request and receive bandwidth allocations. Additionally, for each stream, Horde sends *throttle* events to the application to notify it about changes in allocated bandwidth.

If the application is sensitive to some QoS aspects on any stream, it can inject *objectives* into the middleware to modulate the network QoS for those streams.

Internal Structure

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

Applications use the interface provided by the highest layer to inject and remove ADU's and policies. Horde also delivers ADU's and invokes stream event handler callbacks using this interface. Our implementation of Horde is in user-space. In our implementation, the highest layer provides a simple IPC interface to Horde[1].

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

In the lowest layer, the channel managers deal directly with the network channels. The channel pool manager monitors network connectivity, making sure there is an appropriate manager for each active channel.

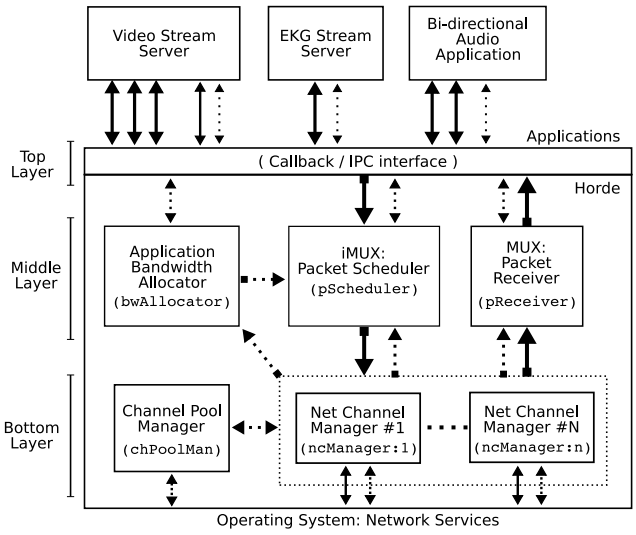


Figure 3: Modular breakdown of Horde. Solid arrows show data flow; dashed arrows represent control signals.

2.2 Network Channel Managers

Horde uses a set of Network Channel Manager modules (the `ncManagers`) to manage the available network channels. Each `ncManager` handles network I/O, maintains a predictive model for the QoS on the channel, and also performs congestion control on the channel.

The generic `ncManager` interface allows different channel models and congestion control schemes to coexist without complicating the packet scheduler. The behaviour of the last hop wireless link can often dominate in determining how QoS varies and how well a congestion control scheme works [4, 14, 17]. Therefore, there may be multiple implementations of the `ncManager` interface, each optimized for a different type of network (e.g., 802.11, CDMA2000, GPRS).

2.3 Network Congestion Control

Congestion control in a striping system should be implemented below the striping layer, independently for each channel. When data is being striped to a single destination, since there are multiple independent channels, there are multiple independent congestion domains. Thus each channel manager in Horde runs an independent congestion control session[2].

We do not believe that having a single congestion control session straddling channels—as previous striping systems have often done—is the right approach. An application, above the striping layer, does not have enough information to implement efficient congestion control,

unless the application can perfectly simulate the scheduling algorithm. For instance, approaches based on 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). Older systems [1, 18] realized the need for some form of minimal per-channel congestion control below the striping layer, even though TCP congestion control was active above the striping layer.

2.4 Stream Flow Control

Bandwidth allocation is separated from QoS modulation in Horde. Different scheduling strategies with a bandwidth allocation can modulate QoS in different ways. For example, n bytes of a stream's data can be sent on a fast channel or spread over fast and slow channels.

Conceptually, Horde's interface separates the quantity of service a stream receives (the number of slots) from the quality of that service (the latency/loss/etc of those slots). Of course, this is not always a clean separation: a QoS with high loss can lead to low goodput.

In Horde, the QoS modulation mechanism is constrained by the bandwidth allocation mechanism. When producing transmission schedules, the amount of data sent is restricted based on the stream allocations.

Every active Horde stream is allocated some fraction of the available bandwidth. For each stream, an application either specifies the maximum rate at which it wants to send data, 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. The present implementation of Horde uses a simple adaptive min-max fairness policy to allocate available bandwidth among streams. If a stream 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. Simple streams can ignore these events, always sending at their maximum rates, letting the middleware decide which ADU's to drop. Conversely, adaptive streams can use the information conveyed by the events to guide changes to their behaviour. Some streams may respond to these events by 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 less important ADU's, etc.

Enforced bandwidth allocations and explicit flow-control feedback are important in providing graceful degradation when independent data streams exist. Available bandwidth will often be outstripped by total demand. Since callbacks managing different streams can be independent of each other, a mediator is needed inside the middleware to divide the bandwidth *fairly*. This mediator transforms changes in total bandwidth into some set of changes in the bandwidth of each stream.

2.5 Transmission Slots

Horde uses the notion of transmission slots, in the form of `txSlot` objects, to provide an abstract interface between the channel manager modules and the packet scheduler. A `txSlot` is an abstract representation for the capability to transmit some data on a specific network channel at a given time, along with the expected QoS for that transmission. The fields of the `txSlot` abstraction are shown in figure 4. As we discuss later in section 3, the scheduler makes its decisions based solely on the information in the `txSlot` objects. This significantly simplifies the scheduling problem.

Transmission Capabilities Each `txSlot` object represents a *capability* that the `ncManager` grants to the scheduler, allowing the transmission of data on that channel. The generation of `txSlot`'s is governed by the `ncManager`'s congestion control logic. In a scheduling cycle, the scheduler acquires slots from all active channel managers, maps ADU fragments to those `txSlot`'s, and passes that mapping back to the managers, resulting in data being transmitted.

Latency/Loss Expectations For every `txSlot`, the parent `ncManager` uses a dynamically updated probabilistic model of the channel's behaviour to derive the expectations for that slot. A simple `ncManager` implementation could use a weighted moving average of previously seen round-trip-times to determine the expected latency and an average ($\frac{total\ delivered}{total\ lost}$) for loss[3].

Correlated Loss When two slots suffer losses, we say that the losses in those slots are *correlated*. There are two important types of correlated losses: burst losses on a single channel and correlated losses on different channels. We have observed that individual WWAN channels exhibited bursty losses. Correlated losses on different

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 $P(both\ lost \mid either\ one\ lost)$.	$correl \in [0, 1]$
`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: The main components of Horde's transmission slot (`txSlot`) abstraction.

channels also occur, because of signal fading or external factors that cause cross-channel correlation (e.g., when two channels have antennas on a tower that is occluded).

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 the streams.

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 (e.g., streaming multiple description video). As figure 4 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.

Transmission Costs There are often monetary costs associated with the transmission of data on a channel. A `ncManager` can maintain its own cost model, configured with provider-specific information. For instance, for a WWAN account with a data transmission quota, the cost for each slot 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.

Phantom Transmission Slots A channel manager can also be asked to *look-ahead* into the near future (e.g. the next second) to estimate how many more slots are likely to become available on that channel. Managers can provide *phantom* `txSlot` objects, each tagged with a confidence level, a measure of how sure the channel manager is of its prediction. Phantom slots are discussed more fully in section 3.3.

3 Policy and Scheduling

At the core of any striping system lies a packet scheduler that decides how to transmit packets from data streams on the multiple network channels. The packet scheduler does not determine the data-rate for a stream. In Horde, the bandwidth allocator's policy, in conjunction with the congestion control algorithms running within the channel managers, ultimately limit stream data-rates. The scheduler must then decide which transmission slot carries which ADU.

The scheduler implements a *policy*. The scheduler transforms its policy into transmission schedules based on the offered load and using information from the `txSlot`'s, about the present and near-future behaviour of those channels. For example, a scheduler could have the policy of minimizing receiver-side ADU reordering in a stream. Such a scheduler may try to minimize reordering as it stripes the data, but—with imperfect predictions about the future—it may not be able to do so.

Most contemporary striping systems use a *static* scheduler policy [1, 10, 18] that cannot be modulated by applications. Often, the policy in these systems is inseparable from the scheduling mechanism itself.

With a heterogeneous and/or dynamically unstable set of channels, the transmission slots can vary considerably in terms of expected loss and latency characteristics, and so the manner in which the scheduler decides to transmit application packets can be crucial in determining the network QoS for a data stream. For example, a scheduler could consistently assign slots with low expected latencies to stream x, following a policy of minimizing the average latency on that stream.

In Horde, the striping policy is separated from the striping mechanism. Horde allows an application to define a time-varying striping policy at run-time, providing a generalized mechanism within the scheduler to facilitate many different policies. An Application expresses its policy goals as modular network QoS objectives, and these objectives drive the scheduler towards transmission

schedules valued highly by that application.

3.1 Application Utility

During a 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 function—in the same way microeconomics textbooks choose to model the utility consumers assign to goods they consume [13].

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. *Expected* net utility can be obtained by using latency and loss expectations as inputs to the utility function.

Utility is directly related to the number, identity and 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 `ack` arrival sequence. 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 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 $(adu, fragment, txslot)$; the `ack` reception history ($history_{rx}$) is a set of $(adu, fragment, time)$ triples; and f_{app} is an arbitrary application defined function.

$utility_{app}$ defines a partial order on possible transmission schedules, which can be used by the scheduler to rank its scheduling choices. $utility_{stream}$ (used below) is an analogous utility function for a stream.

Of course, the scheduler does not have perfect information about the future. Therefore, we work with the expected utility, using expectations from `txSlot`'s.

3.2 'Optimal' Scheduling

We define an *optimal* scheduler as one that picks the transmission schedules most valued by the application, over some time period, given the state of the network during that period and expectations about the future. These expectations are provided by Horde's channel managers.

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 ADU's that have been sent but not yet `ack`'d; expectations from unused `txSlot`'s; and expectations from phantom `txSlot`'s).

Over τ, an optimal scheduler would find a schedule that maximizes the sum of all streams' utility functions, using the provided expectations and constrained by the stream bandwidth allocations[4]:

$$maximize\Big[\sum_{\forall stream}\Big(utility_{stream}(\tau)\Big)\Big]$$

As posed above, the optimal scheduling problem is a computationally impractical, since scheduling decisions typically need to be made frequently. Consequently, our schedulers only attempt to find *high utility* transmission schedules instead of optimal schedules.

3.3 Objective Driven Scheduling

By providing a specialized scheduler and a sufficiently abstract policy expression interface, Horde allows applications to drive the striping operation. An important design issue was deciding how an application expresses its policy goals and how the Horde scheduler translates these goals into transmission schedules. Horde provides a specification language that allows the expression of application goals as succinct network-QoS *objectives*.

The modularity 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 (e.g., video I-frames), favour lower latency slots for some streams, and avoid correlated losses for certain sets of ADU's.

Objectives

Objectives represent the modular decomposition of an application's utility function. The set of expressed objectives drives the packet scheduler towards schedules that provide high utility to applications. Objectives can be injected and removed dynamically, supporting the specification of time-varying utility functions.

An objective defines a QoS goal and describes how the achievement of that goal adds to, or subtracts from, overall application utility.

A goal may be expressed in terms of a set of ADU's (e.g., a stream) or, more generally, in terms of a set of ADU sets (e.g., the sets of I-frame ADU's and P-frame ADU's). Objectives will usually be concerned with the policy of a single data stream but it is sometimes useful to define objectives that straddle multiple streams. For

instance, an objective for multi-description video streaming could express the application goal of minimizing correlated frame losses on the different video data streams.

In Horde, objectives describe value relationships between ADU's and `txSlot`'s. Objectives are independent of the number of channels being used to stripe the data. To simplify this discussion, we assume ADU's are never fragmented. A transmission schedule is then just a mapping of ADU's to `txSlot`'s. Each objective says something about how much an application values the assignment of a particular kind of `txSlot` to a particular kind of ADU.

In determining the value of an assignment, two things are important: the nature of the ADU and the nature of the `txSlot`. ADU's can be differentiated based on their header fields. Some of these are managed by Horde (e.g., `stream_id`) while others are application-specific annotations (e.g., `frame_type`). Slots can be differentiated based on the fields shown in figure 4.

More abstractly, the characteristics of a transmission slot can be seen as a collection of three random variables: latency (real), loss (0 or 1) and cost (real). The expectations (μ) and variances (σ^2) of these variables can be used in evaluating slot assignments.

Goals can be expressed over sets of ADU's and so may refer to the expectation or variance of the loss/latency/cost of the slots assigned to that set. With this view, correlated loss is not a special QoS metric, it can be viewed as the expected loss over a set of ADU's.

When an application does not specify an objective for some QoS aspect of a stream, it is implicitly assumed that the application does not care about that aspect of QoS on that stream. A stream with no associated objectives is likely to be assigned the transmission slots left over after existing objectives have claimed the good slots for their own streams. However, bandwidth allocations are enforced before schedules are constructed, so every stream always receives its fair share of slots[5].

3.4 Objective Specification Language

This section outlines a language whose purpose is to provide a flexible interface in which application objectives can be expressed and evaluated. This language represents a fairly straightforward formalization of the abstract notion of objectives we presented earlier.

We have experimented with other possible interfaces. Our initial interface allowed applications to simply flag streams as being low-latency, low-loss, etc. This interface represents a short-hand, using a small number of objective *idioms*. It proved insufficiently flexible. We also tried allowing applications to export black-box callback functions that perform the $utility_{stream}$ calculation. In our experience, the callback approach made it harder to build an accurate scheduler, because the scheduler had very limited information about the objectives.

```
objective {
  context {
     adu:foo { (stream_id == "video1") &&
               (frame_type == "I")  }
     adu:bar { (stream_id == "video1") &&
               (frame_type != "I")  }
  }
  goal { prob(foo::lost?)
         < prob(bar::lost?) }
  utility { foo { 100 } }
}
```

Figure 5: An objective expressing the policy that, for stream `video1`, `txSlot`'s carrying I-frames should have lower loss probabilities than slots for other frames.

```
objective {
   context {
     stream:foo { stream_id == "audio1" }
   }
   goal { foo::latency_ave < 1000 }
   utility { foo { 100 } }
}
```

Figure 6: An objective expressing the policy that the average latency on a stream should be less than one second.

The language discussed here seems to provide an adequate level of abstraction, flexibility and programmability. Using literals, ADU's, streams, header fields, and latency and loss probability distributions as basic units, the language allows the construction of complex constraints.

The language is type-safe. Each expression has a well-defined type and there are specific rules governing how expressions 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.

An `objective` definition consists of three sections: a `context` (variable bindings); a `goal` (a predicate); and a `utility`. Figures 5, 6 and 7 show examples. When the `goal` predicate is `true`, the interpreter uses the `utility` section to determine how that `goal`'s fulfillment has affected the utility derived from the associated schedule. Active `objectives` are evaluated by the scheduler in an unspecified order.

```
objective {
  context {
     adu:x { (stream_id == "video1") ||
             (stream_id == "audio1")    }
  }
  goal { true }
  utility {
   x { 5 * (1000 - expected(x::latency)) }
  }
}
```

Figure 7: An objective expressing that the utility derived by an application from certain ADU's depends linearly on how close the round-trip-time is to one second.

Context An `objective`'s `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 `objective`'s `context` contains a filter predicate, expressed in terms of ADU header fields or stream identifiers. Fields like `stream_id` are predefined. Applications can arbitrarily define other fields to selectively activate `objectives` for marked ADU's.

The example objectives in figures 5 and 6 show different types of `context`s. Figure 6's objective applies to stream `7`. Figure 5 binds two variables: `foo` is always bound to an `I`-frame ADU from stream `17`; and `bar` to a non-`I`-frame ADU from stream `17`. `frame_type` is an example of a field being used to integrate application-specific ADU annotations into Horde's scheduler.

Goal The `goal` section specifies a predicate that can be evaluated on some schedule. `goal` predicates can be reasonably involved: simple probability, boolean and numerical expressions can be progressively composed to produce a boolean function. For example, figure 5'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 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.

Complex numeric `utility` expressions are possible. Figures 5 and 6 show examples that add a constant utility when their goal is met, but figure 7 uses a more involved `utility` section in which the utility gained from transmitting an ADU varies linearly with expected latency.

The language does not assign semantic meaning to numeric utility, other than the notion that higher utility is better than lower utility. Consequently, the impact of the constants `100` (figures 5 and 6) and `5` (figure 7) can only be gauged by looking at the constants in other active objectives. This represents a weakness in the language.

```
tx_schedule random_walk_scheduler() {
  // collect all slots w/ look-ahead
  slots = collect_slots(LOOKAHEAD_MSECS);
  // collect as many adus as 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 sched =
     create_random_schedule(slots, adus);

    // evaluate all objectives over
    // this schedule and get utility
    int util = evaluate_objectives(sched);

    // is this best schedule?
    if (util > max_util) {
      max_util = util;
      best_schedule = sched;
    }
  }
  return best_schedule;
}
```

Figure 8: Pseudo-C++ for `random-walk` scheduler.

3.5 Scheduler Implementation

In constructing the scheduler described here, we have consciously attempted to make it as simple as we could. Our goal for this implementation was to show that even very simple scheduling algorithms 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 a good schedule, given the set of active objectives. The scheduler creates k random transmission schedules, evaluates all objectives over each of these k schedules and picks the schedule with the highest aggregate utility. In the absence of any objectives, the `random-walk` scheduler works like a

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

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

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

  return random;
}
```

Figure 9: Pseudo-C++ for random schedule constructor.

randomized-round-robin scheduler: any valid mapping of slots to ADU's is equally likely.

A scheduling cycle runs every T milliseconds. During each cycle, the scheduler acquires from each 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 ADU's can be collected from each stream. Streams are treated as a FIFO queues.

Figures 8 and 9 describe the scheduler in pseudo-code. In practice, the scheduler is slightly 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 specification 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. To differentiate good schedules from bad ones, `evaluate_objectives` should define a partial order on the set of possible schedules.

The `random-walk` scheduler has parameters that can be adjusted to make trade-offs between scheduler accuracy and processing cost. With a `WALK_LENGTH` of 1, it becomes the randomized-round-robin scheduler. Whether accuracy is more important than cost is is likely to vary depending on the overall system.

Scheduling with Phantom Slots

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 efficiently predicted by a channel model—to make good scheduling decisions. 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 demonstrate that even relatively trivial look-ahead logic 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 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[6]. 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.

4 Experimental Evaluation

We have implemented a very preliminary version of the Horde middleware, and are working on building a mobile video streaming system using this implementation. In this section we report on some experimental results.

WWAN Channels

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

The experiments reported here used three stationary WWAN interfaces. They consisted of sending as many 768 byte packets[7] as Horde's flow control layer allowed, from the laptop to a host on the MIT ethernet. We used a generic `ncManager` implementation for all channels. This implementation used AIMD for congestion control,

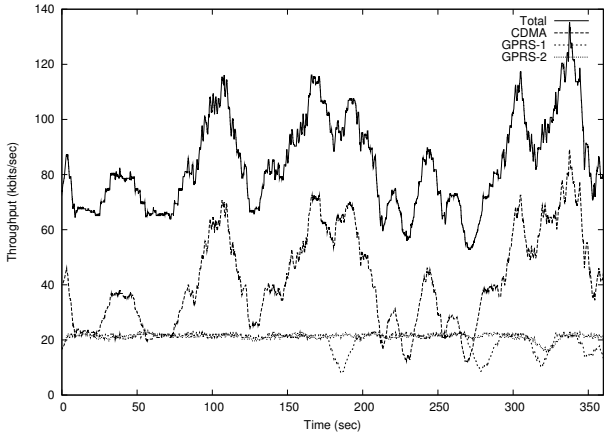


Figure 10: Throughput provided by Horde in a stationary experiment, using three colocated interfaces.

weighted-moving-averages to predict expected round-trip-times and loss rates in `txSlot`'s, and averages of past sending rates to generate phantoms.

Throughput Figure 10 shows the throughput provided in an experiment. The throughputs were calculated using a 10-second sliding window. The GPRS bandwidths are relatively stable in this experiment. This is consistent with older experiments measuring raw UDP throughput [14], which did not use Horde. However, the experiments in [14] also indicate that vehicular motion leads to significant variability in throughput. The variability in CDMA throughput may be caused by contention with other users, or due to the AIMD congestion control. Experiments in [14], when the network was—presumably—lightly loaded, have shown the CDMA interface providing upload bandwidth of up to 120 kbits/sec, close to its theoretical maximum. Generally, bandwidths on these WWAN interfaces varied greatly, both over short time scales (seen in the figure) and over longer time scales.

Packet Latency Figure 11 shows the ADU 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 GPRS packet round-trip-time distributions. Typically, those are longer and more variable.

Scheduling The existence of marked QoS differences among co-located WWAN interfaces highlights the potential impact of scheduling decisions on stream QoS. Figure 11a and 11c exemplify the very large differences

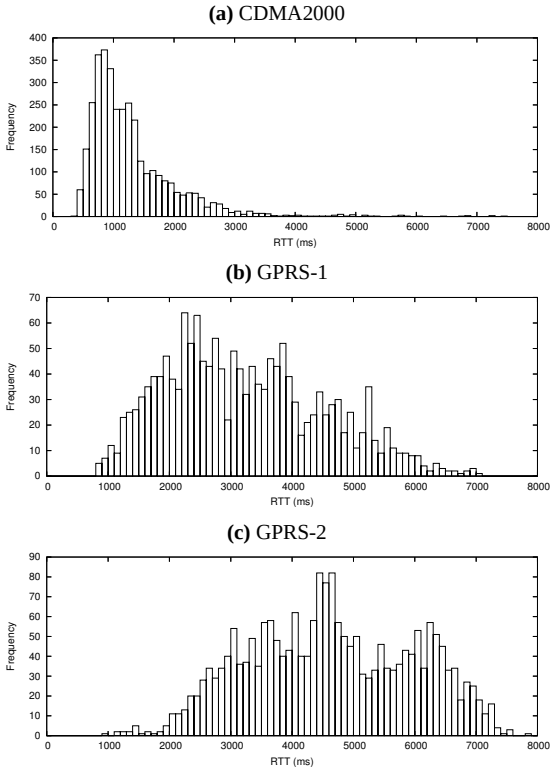


Figure 11: Observed WWAN packet RTT distributions.

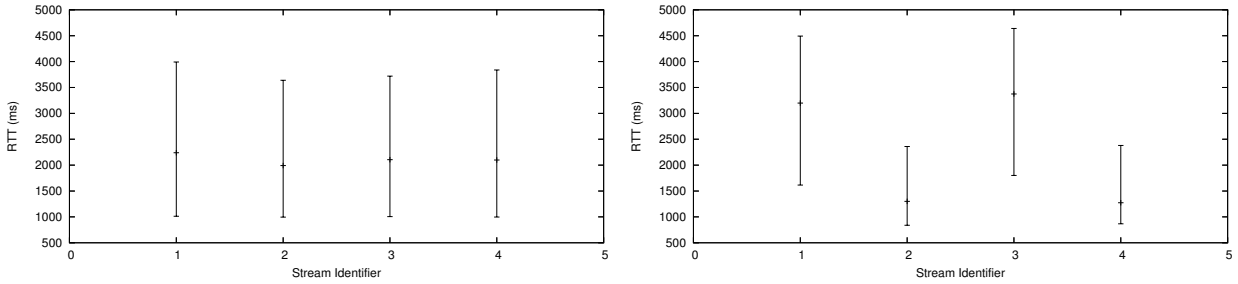
that exist between two colocated WWAN channels using different technologies and providers. Figures 11b and 11c show that even WWAN channels from the same provider using the same technology can provide very different QoS.

In order to examine the impact of different scheduling policies on QoS, we simulated the Horde scheduler on the packet traces, using simple channel models in the simulated channel managers[8]. In this way, we could compare the effects of using different schedulers and policies under identical network conditions.

Figure 12 compares a randomized round robin scheduler to Horde's `random-walk` scheduler with objectives that imply that low latency is more important for some streams. We set up four data streams, each of which demanded and received the same bandwidth.

Figure 12a shows the distribution of the latencies observed by each stream when a randomized round robin scheduler was used. As expected, there were only small variations in the latency distributions across the streams.

Figure 2b shows the distribution of observed latencies when an objective driven scheduler was used. No objective was specified for streams 1 and 3. Streams 2 and 4 were given the objective in figure 7. This objective



(a) Randomized Round Robin

(b) Objective Driven Scheduling

Figure 12: Measured round-trip-time distributions with different schedulers, striping over three existing WWANs (one CDMA and two GPRS). The graphs show the median and the upper and lower quartile packet latencies for the streams.

states that added value is derived in proportion to the difference of the expected RTT and one second. 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 12b clearly shows that the scheduler recognized that QoS unfairness can be beneficial to the application.

Simulated Channels

We felt it was necessary to use simulation to fully evaluate the performance of the Horde scheduler. Our use of simulated channels was motivated by the fact that the set of real channels available to us 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 CDMA channel. A more diverse set of simulated channels allowed us to better investigate the performance of the scheduler.

For our simulations, we replaced the standard `ncManager` implementation with one that used pseudo-random variables to generate slots. Our simulated channels are based on measurements from real WWANs. We simulated two pairs of high-latency GPRS channels (each pair had a different mean latency); one medium-latency low-loss CDMA channel; and one low-latency medium-loss channel based on CDMA statistics.

We injected objectives for streams 1 and 4. Two objectives were defined for stream 4: one valuing low-latency and one valuing low-loss. A single low-latency objective was defined for stream 1. Figure 13 shows the resulting latency distributions for the streams. Streams 2 and 3 are spread over all channels, indicated by their large latency ranges; stream 4 mostly uses the medium-latency low-loss channel; and stream 1 is mostly spread over the medium-latency and low-latency channels.

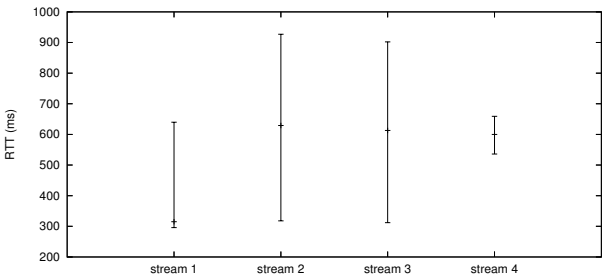
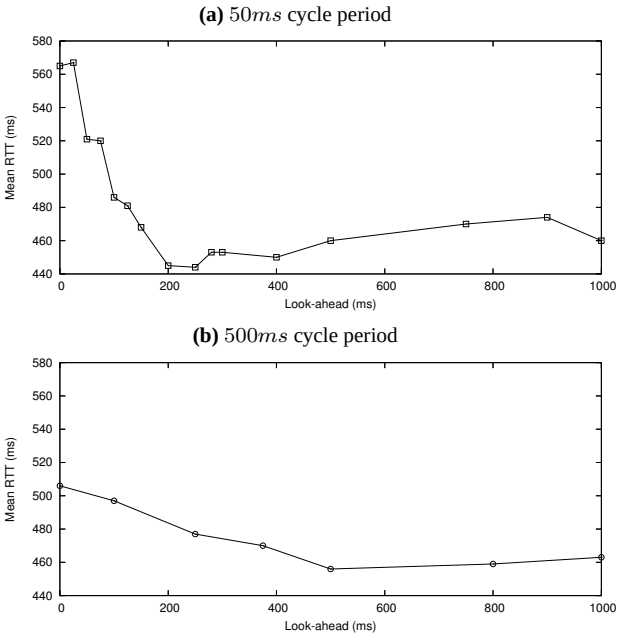


Figure 13: RTT distributions in a simulated experiment.



(a) $50ms$ cycle period

(b) $500ms$ cycle period

Figure 14: Impact of look-ahead on accuracy. The graphs show how the average latency on a latency sensitive stream varied with different look-ahead and scheduling cycle periods, on simulated channels.

Effects of Look-ahead on Accuracy

We were interested in how important look-ahead was for an accurate objective driven scheduler. To investigate this, we injected a single objective valuing a low network-latency for ADU's on a specific stream. We ran the scheduler with a number of different cycle periods (T) and different look-ahead periods (L), on simulated channels. In these experiments, a lower average network latency for the latency-sensitive stream implied a more accurate scheduler. Figure 14 shows how latency varied for some different values of T and L.

Scheduler accuracy improves by increasing L (up to some L). With a larger look-ahead period the scheduler can see, in each cycle, the future slots from both the low-latency high-rate channels (CDMA) and the high-latency low-rate channels (GPRS). Thus with a large enough L (around $200ms$) the scheduler works well.

Scheduler accuracy also improves with T (up to some value of T). Again the scheduler is able to see both types of slots in each cycle, improving its accuracy. However, with a higher T, ADU's are queued longer locally, so the overall ADU delay becomes quite large, even as the scheduler becomes more accurate.

5 Related Work

Schemes that provide link layer striping have been around for a while [9, 8, 1]. Most such schemes were pioneered on hosts connected to static sets of ISDN links. These schemes mostly assume stable underlying channels. Instabilities and packet losses can cause some protocols to behave badly [1]. 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.

Most link-layer striping systems are, quite reasonably, optimized for TCP. Mechanisms are chosen to minimize reordering, and packets may be buffered at the receiver to prevent the TCP layer from generating DUPACKS [1]. partial IP losses can be exposed to the higher layer by rewriting TCP headers within the striping code.

Implementing congestion control under the striping scheduler is not a new idea. The LQB scheme [18] extends the deficit-round-robin approach to reduce the load on channels with higher losses. The use of TCP-v [10], a transport layer protocol that provides network striping with the goal of achieving reliable in-order delivery, also results in per-channel congestion control. However, whereas Horde allows channel-specificity, others use the same mechanism on every type of channel.

Some recent proposals for network striping over wireless links have proposed mechanisms to adjust striping policy. Both MAR [15] and MMTP [11] allow the core packet scheduler to be replaced with a different scheduler implementing a different policy. This is the hard-coded policy case mentioned earlier.

We were motivated by the need to develop a multi-path video streaming application. Setton et al [16] 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 [5] describe a similar scheme. Horde allows more flexible scheduling of video packets over the available paths.

6 Summary and Conclusion

Horde is a library that facilitates flexible network striping in WWAN environments for a diverse range of applications. Horde separates the striping policy from routing and scheduling. It allows applications to describe QoS 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 policy, to flexibly stripe data over a highly heterogeneous set of dynamically varying network channels.

Horde is most useful when different streams gain value from different aspects of network performance and when the available network channels have dissimilar and/or time-varying characteristics.

Horde tackles many complex problems. We have presented techniques to separate striping policy from the mechanism used, and evaluated how well a generic striping mechanism is able to interpret simple expressed policies. Furthermore, 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

In this paper, we described the basic abstractions upon which Horde is built. The two key abstractions are transmission slots (`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.

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

In conclusion, more work is needed on the specification language and schedulers. Presently, the language allows too much complexity, making objective programming harder than it should be. Furthermore, some abstractions could be generalized more (e.g., streams, and correlated loss) as operations over sets of ADUs. We are working on addressing these issues. Finally, we are experimenting with more complex schedulers that use language semantics to derive better schedules.

Acknowledgments

The authors would like to thank Allen Miu, Godfrey Tan, Magdalena Balazinska, Vladimir Bychkovsky, Eugene Shih, Michel Goraczko and Dorothy Curtis. This research is supported by the National Library of Medicine, CIMIT, the Center for the Integration of Medicine and Innovative Technology, and Acer Inc., Delta Electronics Inc., HP Corp., NTT Inc., Nokia Research Center, and Philips Research under MIT Project Oxygen.

Notes

[1]The IPC overhead is acceptable in our system, since the aggregate data rates are low (300 kbits/sec). The IPC stubs can be easily replaced with more efficient mechanisms if needed, for a single-process Horde.

[2]With multiple destination hosts, a `ncManager` may need to maintain a unique congestion control session for every destination.

[3]Moving-averages seemed to work reasonably well in our experiments, though they were often wrong by over $100ms$. We found better estimators could be constructed using *a priori* knowledge about the WWAN technology [14]. Furthermore, more elaborate estimators plug easily into our modular 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 have more losses, could be incorporated without complications.

[4]If fairness is of concern, this definition can be modified to take that into account. In this paper, we restrict our consideration of fairness to bandwidth allocations.

[5]Nonetheless, since a stream may receive slots more likely to result in losses, that stream may not receive its fair share of goodput. Fair shares of slots do not imply that all streams are allocated equal goodput.

[6]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.

[7]Our UDP experiments on WWAN channels [14] show that packet size affects both available 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

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

References

[1] ADISESHU, H., PARULKAR, G. M., AND VARGHESE, G. "A Reliable and Scalable Striping Protocol". In *SIGCOMM* (1996), pp. 131–141.

[2] ANDERSEN, D., BANSAL, D., CURTIS, D., SESHAN, S., AND BALAKRISHNAN, H. "System support for bandwidth management and content adaptation in Internet applications". In *Proceedings of 4th Symposium on Operating Systems Design and Implementation, USENIX* (October 2000), pp. 213–226.

[3] APOSTOLOPOULOS, J., AND WEE, S. "Unbalanced Multiple Description Video Communication using Path Diversity". citeseer.ist.psu.edu/apostolopoulos01unbalanced.html.

[4] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., AND KATZ, R. H. "A comparison of mechanisms for improving TCP performance over wireless links". *IEEE/ACM Transactions on Networking 5*, 6 (1997), 756–769.

[5] BEGEN, A., ALTUNBASAK, Y., AND ERGUN, O. "Multi-path Selection for Multiple Description Encoded Video Streaming". In *IEEE ICC* (2003).

[6] CARTWRIGHT, J. "GPRS Link Characterisation". http://www.cl.cam.ac.uk/users/rc277/linkchar.html.

[7] CLARK, D., AND TENNENHOUSE, D. "Architectural Consideration for a New Generation of Protocols". In *ACM SIGCOMM* (1990).

[8] DUNCANSON, J. "Inverse Multiplexing". *IEEE Communications Magazine* (April 1994), 34–41.

[9] FREDETTE, P. "The Past, Present, and Future of Inverse Multiplexing". *IEEE Communications Magazine* (April 1994), 42–46.

[10] HSIEH, H., KIM, K., AND SIVAKUMAR, R. "a Transport Layer Approach for Achieving Aggregate Bandwidths on Multi-homed Hosts". In *ACM MOBICOM* (2002), pp. 83–94.

[11] MAGALHAES, L., AND KRAVETS, R. MMTP: Multimedia multiplexing transport protocol, 2001.

[12] MAGALHAES, L., AND KRAVETS, R. "Transport Level Mechanisms for Bandwidth Aggregation on Mobile Hosts". In *ICNP* (2001).

[13] PERLOFF, J. M. "Microeconomics 3rd edition" . Addison Wesley Publishing Company.

[14] QURESHI, A. Flexible Application Driven Network Striping over Wireless Wide Area Networks. MEng Thesis, Massachusetts Institute of Technology, March 2005.

[15] RODRIGUEZ, P., CHAKRAVORTY, R., CHESTERFIELD, J., PRATT, I., AND BANERJEE, S. MAR: A Commuter Router Infrastructure for the Mobile Internet. In *Mobisys* (2004).

[16] SETTON, E., LIANG, Y. J., AND GIROD, B. Multiple description video streaming over multiple channels with active probing. In *IEEE International Conference on Multimedia and Expo* (2003).

[17] SINHA, P., VENKITARAMAN, N., SIVAKUMAR, R., AND BHARGHAVAN, V. "WTCP: A Reliable Transport Protocol for Wireless Widea-Area Networks". University of Illinois at Urbana-Champaign.

[18] SNOEREN, A. "Adaptive Inverse Multiplexing for Wide-Area Wireless Networks". In *IEEE conference on Global Communications* (1999), pp. 1665–1672.