Differential Bandwidth Allocation with Multiplexed TCP Connections

by

Hiroyoshi Iwashima

B.S., Massachusetts Institute of Technology, 2002

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

August 2003

Author . . . .
Department of Electrical Engineering and Computer Science
August 22, 2003

Certified by . . . .
Hari Balakrishnan
Associate Professor
Thesis Supervisor

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

Differential Bandwidth Allocation with Multiplexed TCP Connections

by

Hiroyoshi Iwashima

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents and evaluates the design and implementation of a user-level library that performs bandwidth allocation among multiple application flows using multiple TCP connections. This work is motivated by emerging trends in network overlay applications to send multiple flows of data between machines. Existing approaches to manage the network either do not offer the control over network resources that these applications need, or are difficult to adopt.

We present a novel user-level approach to this problem that uses multiple TCP connections, and we also present an evaluation of multiplexing strategies for the scheduler that multiplexes the multiple application flows onto the multiple TCP connections. We implemented and designed two scheduling algorithms, striping and pinning, to perform the multiplexing and evaluated them for various application behavior and various link characteristics through emulation.

We describe how striping is preferred for applications that are not delay dependent, as well as applications that are not order dependent, while pinning is preferred for applications that are heavily memory constrained. Our results show that, for delay-constrained applications, links with low loss and low number of cross traffic favor the striping scheduler, while high loss or high numbers of cross traffic favor the pinning scheduler. Delay-constrained applications that maintain high number of flows also favor the pinning scheduler, while those that do not favor the striping scheduler.

We conclude that a user-level approach is both feasible and preferred, and that both multiplexing approaches are viable options based on network characteristics and application behavior.

Thesis Supervisor: Hari Balakrishnan
Title: Associate Professor

Acknowledgments

First and foremost, I want to express my utmost gratitude to my advisor, Hari Balakrishnan, who has been a constant source of inspiration and direction. Because of Hari, I am proud of my thesis. He has taught me in the field of Computer Networks to be able to appreciate and understand my topic, and he has always demanded quality in my research. Thanks to Hari, I will leave MIT with a sense of academic accomplishment. I will always remember him as a role model for academic excellence.

I am also deeply indebted to my colleagues, Magdalena Balazinska and Jonathan Salz, who have been on the receiving end of my endless questions and pleas for help. They have been both patient and amazingly helpful, and they have made my thesis experience into a pleasant one.

Finally, I would like to especially thank a few of my closest friends, who have truly supported me emotionally and physically. To Jessie Chen, Tony Eng, and Kim Truong, I owe a lifetime of gratitude and faithful friendship. Thank you.

Contents

List of Figures

Chapter 1

Introduction

Emerging overlay network applications are changing the way they utilize the computer network. Instead of being simple applications that open up one connection between two computers in the system, these new applications open up multiple connections between computers. Applications like streaming databases, sensor networks, and even web clients are networked applications that exhibit this particular pattern of network usage. This new usage pattern raises the question of whether existing techniques for network management are the optimal approaches when dealing with applications that send multiple flows of data between machines.

To compare different approaches, we must first choose the criteria that we use to judge them. One way of judging that one approach does better than another is if the application obtains greater total throughput and lower application delay. These two metrics indicate the degree of efficiency of network usage.

Furthermore, we can judge approaches in how they utilize the throughput that they obtain. There is a growing need for such networked applications to divide the throughput among the different flows in a manner that the application can specify how much of the throughput a specific flow should receive. For example, in a sensor network, the network may decide that readings from one sensor are more important than readings from another, and hence, should be given comparatively higher throughput when possible. This notion of predefined allocation of throughput has become much more important as multiple application flows try to share the network resource.

Existing approaches to manage the network either do not meet acceptable levels of these two criteria, do not utilize the network efficiently, or are implemented in ways such that they are hard to adopt.

We therefore propose an approach that address this search for better network management. Our approach is to make a user-level library that uses TCP as the transfer mechanism. The library allows applications to open up multiple application flows, which we call channels. The data is then sent over several TCP connections to the destination. The library runs either a striping scheduler or a pinning scheduler to multiplex the multiple channels onto the multiple TCP connections. The striping scheduler takes the data packets from the channels and places the packets onto any available TCP connection. The pinning scheduler picks an assignment of channels to TCP connections and continues to send according to that assignment.

This thesis attempts to show that a user-level implementation using TCP is a viable and preferred method of efficiently managing the network. We also analyze the two approaches of multiplexing application channels onto multiple TCP connections. We show that striping allows applications to be more flexible in how they send data, but pinning gives applications more robustness with changing network characteristics. We show that applications that are not delay-dependent prefer to use striping, while heavily memory-constrained applications prefer to use pinning. Delay-intensive and non-memory-constrained applications look at the network characteristics and application behavior. If the network has both low packet loss rate and low numbers of cross-traffic, then the applications prefer striping. Conversely, high packet loss rate or high numbers of cross traffic favors pinning. If the application consistently maintains many flows between all its machines, then pinning is preferred, while if the application sometimes only has a few flows between its machines, then striping is preferred.

In this thesis, we first define the problem as well as a few terms (Chapter 2) and present a sample application to set the context of the work (Chapter 3). We then describe the design approach, as well as the two scheduling algorithms in detail (Chapter 5). Finally, we present an implementation of the ideas (Chapter 6), and evaluate the results (Chapter 7).

Chapter 2

Problem Definition

In this section, we describe a model that we use for the rest of the document to generally describe a series of networked applications. The kinds of applications that this thesis relates to are overlay network applications that multiplex many different **application channels**, such as sensor networks, streaming databases, and web servers. We use this model to analyze network usage and performance in these applications.

- A `channel` is a logically distinguishable application-layer virtual connection.

Our model is the simplest example of such a networked application, where an application runs on two nodes with many channels between the nodes. The findings for this model can be expanded to the scenario of a larger application with more nodes and more complex channel networks in the system. As we can see in Figure 2-1, Node A is the sender, and Node B the receiver. There are n channels of application data going from node A to node B.

The model also takes into account the desire for applications to perform **bandwidth allocation** among the different channels.

- `bandwidth allocation`: a division and reservation of the total available network bandwidth among the different channels

To specify the bandwidth allocation, each channel is associated with a relative weight $(w_1, w_2, ..., w_n)$ specifying the relative throughput desired for that channel. If we specify the total throughput of all the channels combined to be 1, then a channel i should obtain

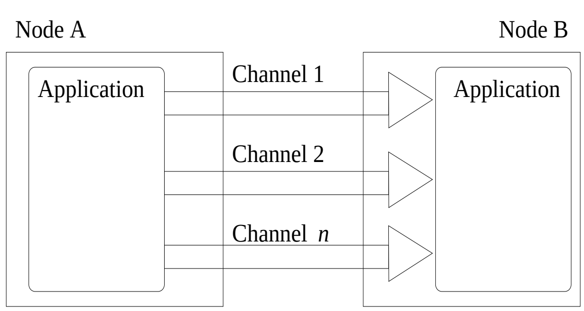


Figure 2-1: Simple Problem Model

$\frac{w_i}{\sum_{j=1..n} w_j}$ throughput. Comparing two channels i and j, channel i expects to obtain $\frac{w_i}{w_j}$ times as much throughput as channel j on average.

- `channel throughput`: the rate of transfer of data in bytes per second of an application channel

There are three goals in the mechanism that we want to create. The first goal is to have efficient usage of the network, which means having high total throughput and low average delay of data. The second goal is to achieve the desired bandwidth allocation specified by the weight distribution of the channels. Finally, we would like to run all this with low memory usage.

Chapter 3

Sample Application: Medusa

We present an example of a system whose network usage fits the description in the model. The system is **Medusa**, a distributed streaming database [6]. Medusa is designed as an application-level overlay network, where the inputs are streams of real-time sensor data, sent to any of the machines in the distributed Medusa system. A user can place continuous queries on this data and get back a stream of results.

Medusa defines a set of operators that can be applied to the streams of data. To satisfy a user query, Medusa creates a *network of operators* such that the resulting stream is the answer to the query. In Figure 3-1, we see an example of such a network with two data streams going through two operators, a filter operator and a join operator, to satisfy a user query. This boxes-and-arrows diagram is then mapped onto actual machines in the system.

Figure 3-2 shows a possible such mapping from virtual boxes to physical machines. It also shows that the physical mapping can span multiple machines if the input streams go to different machines. In this example, the boxes are put on two different nodes, and a channel is created between the two machines to transfer the output of the filter box to the input of the join box.

As a result, the application sets up a series of application channels between machines in the system to satisfy the user query. When many queries are simultaneously placed on the system, there are many such operator networks, so any two machines in the overlay can have multiple channels between them. Taking any two machines, then, gives us the model described previously.

Medusa is also a system that has semantic information about the user queries and input data. The system can distinguish between the different channels going between two machines, and could want to allocate bandwidth unevenly. For example, one channel could be part of an operator network that satisfies the user query for temperature readings in a building. During a fire emergency, the system may wish to give this channel more priority than another channel that is used to query energy conservation in the same building. As another example, the system may wish to give a query by a building administrator more resources than a query by a regular worker. In these cases, the system can place weights to represent the relative importance of the queries, thereby controlling the amount of network resources given to individual channels.

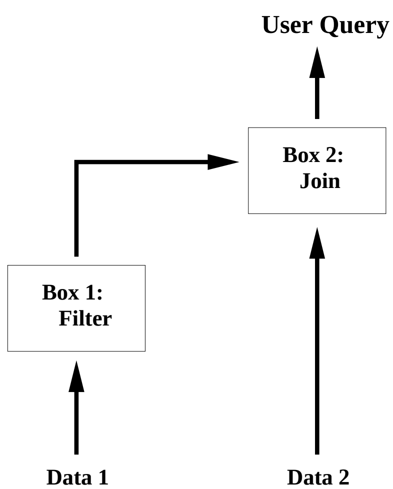


Figure 3-1: Medusa Query Diagram: two operators to fulfill a user query of two input streams

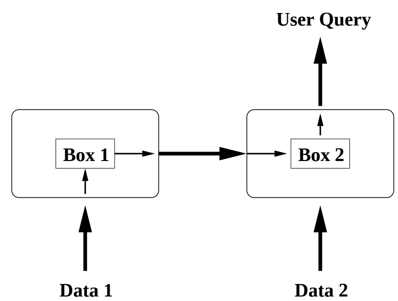


Figure 3-2: Medusa Node Diagram: the two input data streams go to different machines in the system, so the two operators are mapped onto two different nodes and a channel is created between them to transfer the data between the machines

Chapter 4

Background and Related Work

The problem of performing efficient data transfer with bandwidth allocation for multiple application channels has been explored previously with varying amounts of success. We take a look at some approaches people have taken. We also introduce some concepts that are relevant to this document. We assume that the reader has a basic understanding of computer networks and is familiar with terms like bandwidth, delay, and packet loss rate on network links.

4.1 Router-level Solutions

There are some implementations at the routers to try to perform bandwidth allocation among different flows. [17] is a survey on current approaches to QoS (Quality of Service) implementations at the router level, and shows that a variety of router implementations have been proposed and studied, but its application has been limited to VPN (Virtual Private Networks) and not in the general Internet. The Internet, which is dominated by IP, is a stateless routing protocol, making it hard to control state. IntServ [5] and DiffServ [9] are two attempts at this, but has been unsuccessful in widespread adoption.

Router-level solutions also perform sub-optimally because they cannot obtain application-level information readily. Computation is also expensive at the routers, leaving router-level solutions highly handicapped.

4.2 Kernel

A computer can run multiple programs simultaneously. The job of coordinating all the different programs and to present them with system resources is the job of the kernel. Applications that want to utilize system resources make system calls to the kernel. For example, the kernel handles the interface to the network card of the computer and presents an interface to the application to be able to utilize it.

4.3 TCP (Transmission Control Protocol)

One interface that the kernel presents to user programs is the TCP [18] interface. We look at the effects of TCP in the rest of the document, so we explain how TCP works and behaves.

TCP is a network protocol that implements a reliable, in order delivery of packets from the source to the destination, while probing for bandwidth and reacting to signs of congestion. To achieve this, TCP uses two main mechanisms, a sender/receiver sliding window and packet acknowledgements (acks). We present TCP to the extent that is relevant to this thesis in the following paragraphs. In [14] and [23], the authors more fully explain TCP mechanisms.

The basic mechanism starts with the sender assigning a sequence number to each packet that it sends out. The sender keeps track of all in-flight packets in its sender window. The receiver receives a packet and puts the packet in its receiver window. The receiver then slides the window as much as possible until it comes across a sequence number that it has not received, sending all the packets that it slides over to the application. It then sends back an acknowledgement for the last packet before the missing one to the sender. The sender, upon receiving an acknowledgement, slides its window up to the sequence number of the acknowledgement, and sends more packets to fill up the window.

4.3.1 Reliable In-order Delivery

TCP achieves reliable delivery through the use of the sender window and receiver acknowledgements. As we described before, whenever the receiver receives a packet, it sends back an acknowledgement with *the last in-order packet sequence number* that it has received. So, if the receiver receives packets with sequence numbers {1,2,3,5,6,4}, then it returns acknowledgements with sequence numbers {1,2,3,3,3,6}. The sender looks at the acks it has received, and if determines that a packet has been lost in transmission, then it retransmits that packet. It determines that a packet has been lost by the following two methods:

- **Timeout** is when the sender does not receive an acknowledgement for several estimated round-trip times. When no acks are received for this time, the sender assumes that a packet has been lost and the receiver has not received new data to send another ack. Therefore, the sender sends the packet following the last ack it has received.

- **Duplicate acks** are successive acks received by the sender that have the same sequence number. Duplicate acks are sent by the receiver for either a lost packet or packet reordering. TCP assumes that if duplicate acks are caused by reordering, then it only causes one or two duplicate acks. Therefore, when the sender receives *three or more duplicate acks*, then it assumes that a packet has been lost, and does a *fast retransmit* of the packet following the sequence number in the ack. Alternatively, there is a modification to TCP that allows for the sending of *SACK* (Selective acknowledgement), where the receiver can specify what packet it is missing, which allows for more efficient retransmission.

TCP also guarantees in-order delivery of packets and achieves this using the receiver window. The sliding receiver window does not slide unless the packets that it slides over have all been received. All out-of-order packets stay in the window until the missing packets arrive, after which the window slides past them and the packets are sent in order to the application.

4.3.2 Bandwidth Probing and Congestion Avoidance via AIMD

TCP also provides mechanisms to probe for bandwidth and respond to network congestion by changing its sender window sizes according to AIMD (Additive Increase Multiplicative Decrease) principles and by self-clocking using acknowledgements.

TCP ensures that *packet conservation* is maintained, meaning that there are at most the sending window size number of packets in flight to the destination. This is maintained by self-clocking using acknowledgements, meaning that the sender only sends another packet when it receives an acknowledgement from the receiver. The receiver only sends an acknowledgement when a packet has left the network, so each acknowledgement signifies one less packet in the network. Therefore, in total, there are the same number of packets in flight on the network.

To probe for bandwidth, the sender increases its window size by one packet every round-trip-time (*rtt*), which is the additive increase component of AIMD. Increasing the window size increases the number of packets sent on the network, and the throughput for the connection is $\frac{senderwindowsize}{rtt}$, so this results in higher throughput.

When the sender sends data too fast, it causes congestion at the routers, causing a packet to be lost. As explained before, the sender notices a lost packet by either a timeout or duplicate acks. In either case, the window size is dropped to half (Multiplicative Decrease). This is the congestion avoidance algorithm. The multiplicative decrease, explained briefly, is necessary because the routers are congested to the point of losing a packet, so drastic measures are necessary to relieve the stress on the network.

4.3.3 Throughput constraints

TCP throughput in steady state for long-running flows is affected by packet loss rate and rtt of the connection and the maximum size of the sender and receiver windows. TCP uses AIMD for stability reasons, causing the packet loss rate to determine how frequently the windows get halved, thereby limiting throughput. TCP increases its window size every *rtt*, so this also affects throughput. Finally, memory constraints at the sender and receiver in terms of the maximum window size places a hard limitation on the throughput of the connection.

The analysis by Padhye et.al. [16] shows that TCP throughput for a long-running flow not bound by link bandwidth is:

$$Throughput \approx min\{\frac{W_{max}}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + ...}\} \tag{4.1}$$

W_{max} is the maximum size of the sender and receiver window, RTT is the round trip time, p is the packet loss rate, and b is the number of packets acknowledged by a received ack (which is usually two for modern TCP, called a cumulative ack).

This equation becomes important later in the document as we describe the importance of socket buffers while running the experiments, and also to describe the effects as we change RTT and p.

4.3.4 Multiple TCP Interaction

TCP flows perform AIMD to probe for bandwidth and to perform congestion avoidance. The result of the AIMD algorithm is that multiple TCP flows compete with each other for bandwidth. In the classic analysis of AIMD in [7], the authors show that, *under assumptions of synchronized feedback*, AIMD converges to an efficient and fair state, and that no other linear control does. By a fair state, the bandwidth at the bottleneck router is divided evenly among the TCP flows, assuming that all the flows through this router are TCP flows. The synchronized feedback requirement often translates to the requirement that different connections have the same RTT, so the statement states that TCP connections with the same RTT fairly share the bandwidth.

Other research has shown that TCP is unfair when this synchronized feedback assumption is removed, because TCP increases its window size as $\frac{1}{RTT}$ [12, 11]. These papers show that TCP favors connections with lower RTT. Furthermore, TCP is unfair against connections with multiple congested routers in its path [10], which affects the packet loss rate of the connection.

4.4 Kernel-level Solutions

Kernel-level solutions provide complete access to system resources, and hence, can be both complete and efficient. The majority of work done in the area of parallel connections has been to improve the quality of parallel TCP transfers [2]. In [2], the author presents a way of sharing TCP control information among multiple TCP connections for better congestion and loss recovery. This, however, does not give the sort of control necessary to perform bandwidth allocation, nor obtain as much throughput as possible.

The Congestion Manager [3, 4] is a kernel implementation that goes further to solve the problem described before. Similar to [2], the CM aggregates congestion information for concurrent flows, but it also exports an interface to the application in order to expose network conditions and allow programs to have control over network usage. The CM provides an asynchronous callback API so that the application is notified when it can send a certain number of bytes of data, allowing both the CM to schedule the sends and the application to choose what gets sent out. The CM can be configured to reserve a certain amount of bandwidth for each channel, and the CM uses these weights to schedule the application callback accordingly. Inside the CM, it aggregates all the flows to the same destination, shares the congestion information among those flows, and performs the application callback as desired.

While the CM solution provides an answer that works, the main downside to this is that it is a big application in the kernel, and for reasons described later, are not preferred to a user-level solution.

4.5 User Space

The user space is the realm of the application. A given program executes in the user space until a system call is made, at which point the execution is transferred to the kernel temporarily to perform some specific task.

4.6 User-level Solutions

The simplest of all solutions, and one that is the most commonly implemented, is the user-level solution of opening up one TCP connection per connection. This is the easiest solution, and offers reliable transfer of data, but it is both inefficient for short bursts of data and incapable of performing bandwidth allocation aside from the case where every channel gets the same bandwidth.

The majority of other work done in the user-level solution space are solutions for parallel channels over a single connection [13, 22]. The MUX and SCP protocols are ways of efficiently multiplexing channels onto one connection, such as a TCP connection. These solutions, though, both do not obtain as much throughput as desired, and also does not perform bandwidth allocation.

Chapter 5

Design

In this section, we present the design of the network manager as a solution to the problem as defined in Chapter 2. The two main elements of the network manager are the following:

- the mechanism is a *user level library* using *multiple TCP connections*
- there are two different scheduling algorithms, *striping* and *pinning*, that multiplex channels onto TCP connections

We first explain why we implement our **network manager** with a user space library using multiple TCP connections. We then describe how this situation leads us to a new model that multiplexes channels onto multiple TCP connections. Finally, we present the two scheduling algorithms.

5.1 User level library using TCP

Unlike other approaches that implement network management and bandwidth allocation mechanisms in the kernel, we implement the network manager in user space. We show that *implementations in the user space are preferable for development of a new idea.*

The advantage of kernel implementations is the complete control over the system resources, which lets designs in the kernel be optimally efficient. It is also flexible in the interface that it provides to the application, allowing it to be able to communicate information between the two.

However, there are some problems with kernel implementations. The biggest problem is *code maintainability.* Kernel versions change frequently, and with each new kernel, the code must be changed and thoroughly tested. This presents an obstacle to any kernel implementations, since most developing projects are unable to perform this level of maintenance. Another issue is that kernel implementations must be extremely careful, because they can have unbounded effects on the system. The care that must be taken to write kernel code slows down development. Related to this is that the risks make system administrators reluctant to install programs that modify the kernel. An example of this is the Congestion Manager [3, 4], which provides the functions but is not widely adopted. Once an idea has been widely accepted, thoroughly tested, and highly integrated into everyday usage, then it can be implemented in the kernel and added to every new version of the kernel.

Therefore, we chose an implementation in the user space rather than the kernel. The constraint of such an implementation is that it must rely on existing interfaces with the kernel to access the network. There are two main interfaces that kernels support: a low level network packet interface (UDP) and a higher level transmission control protocol (TCP).

We now look at the choice between UDP and TCP, and show why TCP is the preferred approach for reasons of performance and simplicity.

An implementation built using the UDP interface gives applications a lot of control. The implementation controls every packet sent to and received from the other machine, so it has a lot of flexibility in the management of the application data and the management of the network. However, having to control every packet has its downfalls. When dealing with high bandwidth connections, as is undoubtedly true in the future, the program has to constantly make decisions in the microsecond range. Programs, however, are context switched by the operating system, and are not guaranteed such granularities of processor control. This bounds the performance obtainable by implementations using UDP.

Implementations using UDP also has to implement bandwidth probing to utilize the network, congestion control algorithms to be network-friendly, as well as a reliable transfer mechanism to satisfy most applications. These mechanisms are hard to build correctly, as shown by difficulties in kernel implementations even today.

An implementation using the TCP interface, on the other hand, can gain from the fact that TCP already efficiently implements a lot of the mechanisms required. TCP implements reliable transfer, bandwidth probing, and congestion control in an efficient manner in the kernel, thereby reducing the amount of work that is required by the application. This also removes the throughput limitation as explained in the UDP evaluation.

A problem with TCP implementations, however, is that the user application only has limited control over how the kernel sends the data. We discuss the main limitation in the following section, and show how we can work around it.

5.2 Increasing Throughput : Multiple TCP Connections

As described in Section 4.3.3 and Section 4.3.4, the throughput obtained by a TCP connection is determined by the round-trip time, packet loss rate, and other cross traffic at the bottleneck router. This is due to the AIMD congestion avoidance algorithm in TCP. Applications that want to obtain more bandwidth cannot directly control how TCP changes its window sizes, so they cannot change how much bandwidth one TCP connection gets. An application can overcome this obstacle by opening up *multiple TCP connections*, thereby getting a higher percentage of the bandwidth at the bottleneck router.

For a simple explanation of this approach, if there are k other flows going through the bottleneck router, all TCP flows with the same RTT, opening up m connections allows the application to obtain $\frac{m}{k+m}$ fraction of the total bandwidth. As long as $k > 0$, opening up more connections gives higher total throughput. If $k = 0$, then multiple connections merely compete with each other, reducing the efficiency of the connections [2], so this scheme should not be used. However, the Internet today, and as we progress in the future, there is almost always cross traffic. If the application is run on such a controlled environment to have no cross traffic, then this approach is not necessary, and one TCP connection can be used.

For the purposes of this document, we disregard the question of whether opening up multiple connections to obtain a larger share of the bandwidth is an ethical usage of the

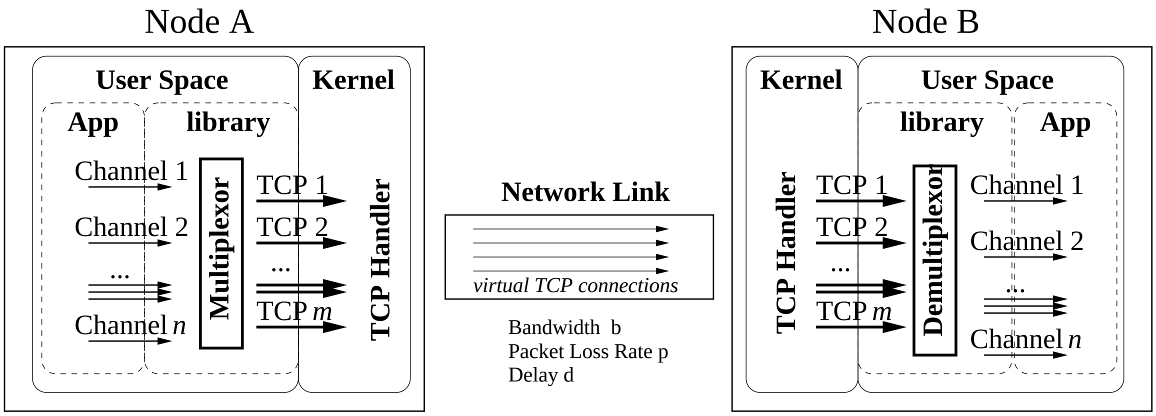


Figure 5-1: Refined Problem Model

network. The simple defense to this is that existing applications already use this approach to obtain more bandwidth, and TCP itself is unfair to connections with high RTT or those with multiple congested routers in its path.

5.3 Refined Model

We have so far shown why we have chosen to have a **user-level** library using **multiple TCP connections**. Let us then refine our model to include this decision.

As previously stated, we have two machines, Node A and Node B, running one application. The application uses a user-level library to send the data across the network and to receive it.

As we can see in Figure 5-1, applications open up n **channels** with **weights** (w_1, w_2, ..., w_n) associated with them. Again, comparing two channels i and j, channel i expects to obtain $\frac{w_i}{w_j}$ times as much bandwidth than channel j on average. The library then opens up m **TCP connections** (henceforth, just called connections). The library runs a multiplexor, or **scheduler**, to determine how to multiplex the n channels onto the m connections.

The goals of the scheduler are the following:

- **Maximize usage of the network**, meaning high total throughput and low average delay
- Achieve as close to a **good bandwidth allocation** as possible as defined by the weight distribution of the channels
- Run with a **minimal runtime memory** requirements

5.4 Multiplexing Schemes

Now that we have n channels and m connections, we need to determine how to multiplex the data onto the connections.

A non-invasive approach to the application would be to open up one connection per channel and send all the data from the channel onto the connection. This is analogous to not having the library at all and opening up individual TCP connections for each channel. The problem with this is that there is be only minimal bandwidth allocation, where all the channels get equal bandwidth regardless of their weights as described in Section 4.3.4, as well as only limited throughput for low values of n, as described in Section 5.2 about having multiple TCP connections.

We now look at two possible schemes, striping and pinning, and compare them.

5.4.1 Striping

One possible approach is to stripe the n connections onto the m connections. In striping, the scheduler independently picks the channel that sends data and the connection that data is sent on. The scheduler chooses the data to send, then *stripes* it to an available connection. We first look at how the scheduler picks the channels and connections at each iteration, then look at a side effect of this approach.

The scheduler chooses the channel to send in order to achieve the weight distribution of throughput using a *Deficit Round Robin algorithm*(DRR) [21]. This algorithm is an approximation of the *Weighted Fair Queueing*(WFQ) [8] algorithm, and achieves nearly perfect fairness in throughput while running in $O(1)$ time to process a packet rather than the $O(\log n)$ time required in WFQ.

We now describe how the DRR algorithm works. The DRR algorithm operates in rounds, and time is measured in rounds. To explain the algorithm, we assume that there are n channels waiting to send with channel i having weight w_i, and that the channels always have data to send. We define the fairness ratio for channel i as $f_i = \frac{w_i}{\sum_{1 \leq j \leq n} w_j}$. The algorithm maintains a state variable $DeficitScore_i$ for each channel that gets updated.

At each round, each channel tries to send up to $DeficitScore_i$ bytes of data, as long as all the packets sent are whole packets. Let $bytes_{i,k}$ be the number of bytes sent by channel i at round k. Then, the new $DeficitScore_i = DeficitScore_i - bytes_{i,k} + f_i * C$ where C is some constant that specifies how fast the algorithm moves along.

We choose the DRR algorithm to implement striping because this algorithm provides a good fast way of achieving the bandwidth allocation. If the throughput of channel i at time t is $Thru_{i,t}$, then the optimal bandwidth allocation is the following:

$$OptimalThroughput_{i,t} = f_i * \sum_{1 \leq j \leq n} Thru_{j,t} \tag{5.1}$$

DRR then claims the following:

$$\forall t \geq 0 : \qquad OptimalThroughput_{i,t} - Thru_{i,t} \leq Max \tag{5.2}$$

where *Max* is the maximum size of a packet [21]. This bounds the difference from the optimal distribution by *Max* bytes, which is very good as long as *Max* is relatively small compared to values of $Thru_{i,t}$.

After the channel is chosen in this manner, the scheduler then picks a connection to send the data on by picking the first available connection. Therefore, the complete scheduling algorithm is the following:

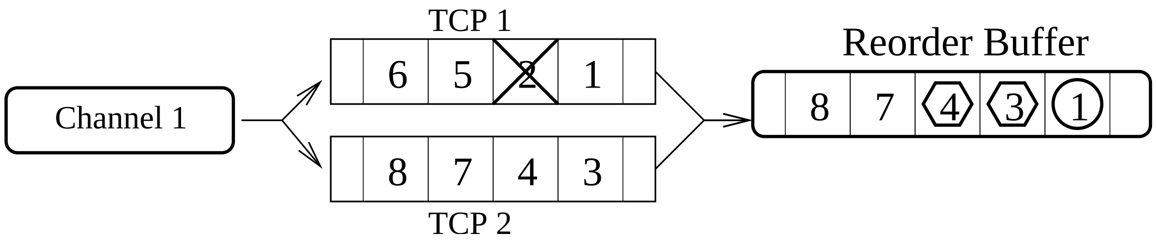


Figure 5-2: Channel Reordering Example: One channel sends on two connections. TCP 1 loses packet 2, causing packets 2,5, and 6 to be delayed. At the same time, TCP 2 sends the other packets, causing them to arrive at the receiver out of order. The receiver puts all the packets into the reorder buffer. The circled packet, packet 1, can be sent to the application in order. When packet 2 arrives after a TCP retransmit, the packets marked with a hexagon, packets 3 and 4, can be sent to the application along with packet 2.

1. Run the deficit round robing algorithm until a channel sends data

2. Find the next connection that is ready to send

3. Send a packet from the channel to the connection

4. Go to step 1

Let us now look at a side effect of this approach: **channel reordering**. In this approach, a channel can send data in parallel across multiple connections. Each connection, however, can independently lose packets and time out in the TCP layer, and hence, the receiver can experience different delays on the connections. While each connection sends the data in order to the receiver, the packets within a channel may arrive out of order.

For example, as shown in Figure 5-2, if there is one channel sending to two connections in parallel, then a packet loss on the first connection can cause the packets in the second channel to arrive out of order. In such cases, the receiver must place the packets in a reorder buffer. Packets that are ready to send to the application in order can be sent, while others wait until this can happen. In the figure, we see that packet 2 is lost by connection 1. Therefore, packets 3,4,7, and 8 arrive out of order at the receiver. Packet 1 is ready to be sent to the application, but the others must wait. When TCP 1 finally retransmits packet 2, then packets 2,3, and 4 can be sent to the application in order.

If the application cares about preserving the order of the packets in a channel, then the receiver must implement such a reordering scheme. This is a side effect of this design, but can be accounted for. However, channel reordering affects the performance of this design, as we see in the following sections.

In summary, the sender opens up m *connections and stripes the* n *channels according to the deficit round robin algorithm onto the first available connection. The receiver demultiplexes the packets received on the* m *connections back into the* n *channels and reorders the packets to send them to the receiving application in order.*

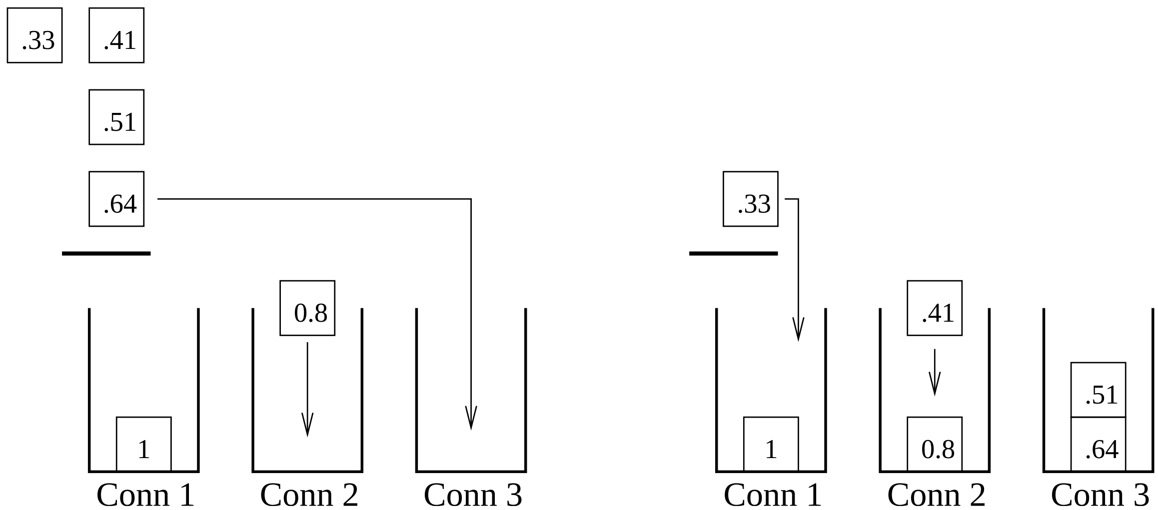


Figure 5-3: Pinning Algorithm Example: The left side shows the greedy best-fit algorithm after making 2 pinning assignments. The channels are sorted in decreasing weight order and placed into the bins one by one. The right side is after a few more iterations have occurred. Each new channel is pinned to the connection with the least total weight already pinned to it.

5.4.2 Pinning

Another way of sending the packets from the n channels onto the m connections is to pin each of the channels onto a connection. The difference between this and the non-invasive approach is that multiple channels can be pinned to a specific connection. We first look at how the scheduler makes the pinning assignments from channels to connections, then look at how the scheduler steps through execution.

The algorithm tries to pin the channels to the connections in a way to try to obtain the appropriate bandwidth allocation as specified by the weights of the channels. TCP behavior dictates that each of the connections with the same RTT gets the same bandwidth, so each connection gets $\frac{1}{m}$ portion of the total bandwidth. If we can find a pinning assignment such that the sums of the weight of the channels in each connection are the same, then we reduce the problem into finding the appropriate allocation within each of the connections.

This problem is analogous to two traditional problems. The first is the **multiprocessor scheduling problem**, where there are a list of jobs with associated time costs and a list of processors that can compute the jobs. The goal is to create a scheduling of jobs to minimize the total time required to finish all the jobs. We concentrate on the other traditional problem because it is easier to picture. The problem is a **ball and bin packing problem**, where the balls have different weights, there are fixed number of bins, and the goal is to place the balls in the bins in a way to get as close to equal weight bins as possible. We can also further specify "closeness" by stating that we want to minimize the weight of the heaviest bin.

A simple yet powerful approximation algorithm was first described by Hochbaum and Shmoys [1], which is a *greedy algorithm that places the balls in a decreasingly ordered best fit fashion.* The balls are ordered in decreasing weight order, then a ball is taken one at a time and is placed into an empty bin if possible, else onto the lightest bin. To go back to our specific problem, as shown in Figure 5-3, we order the channels by decreasing weight, then pin them one by one to the connection that has the least sum of the weights of the channels already pinned to it. The left picture shows the algorithm after two iterations, where the channels are pinned to the first empty connection. The right picture shows the algorithm after several more iterations, showing that the channels are pinned to the connection with the least total sum weight.

This algorithm was shown to have a maximum weight bin that is at most two times the optimal configuration, so this is considered a *dual approximation* to the problem. We choose to use this approximation algorithm because the original problem is NP-complete, and this algorithm is simple to implement while being a dual approximation.

Now that we understand the pinning algorithm, we look at how the scheduler uses this pinning assignment to send the data. On an iteration of the scheduler, the scheduler first chooses a connection to be able to send, then sends on that connection with data from one of the channels pinned to it. Each connection can have multiple channels pinned to it, so the connection sends according to the same deficit round robin algorithm as described before, but only among the channels pinned to it. More concretely, the scheduler does the following:

1. Wait for a connection that is ready to send
2. Look at all the channels pinned to this connection, and keep rerunning the steps of the deficit round robin algorithm until a channel is chosen to send
3. Send the packet from the channel to the connection
4. Go to step 1

In summary, the pinning algorithm uses the greedy best fit approximation algorithm to make a pinning assignment of the n *channels onto the* m *connections. Then, as a connection is able to send, the connection sends according to the DRR algorithm applied to the channels pinned to it.*

Chapter 6

Implementation

In this section, we describe an implementation of the network manager as described by the design in Chapter 5. We start by describing the program architecture, followed by a description of program execution, and end by taking a closer look at the implementations of the two multiplexing schedulers.

6.1 Environment

The library was written in C++ for the UNIX platform. The code was compiled using gcc-3.2.2 on both RedHat 8.0 and FreeBSD 4.7 operating systems. The code extensively used NMSTL, an asynchronous library for network sockets, written by Jon Salz. NMSTL can be downloaded from http://nmstl.sourceforge.net/.

6.2 Program Architecture

The network manager is implemented as a user library, and is responsible for handling the calls from the application to open up channels and for asynchronously sending the data. The library runs in a separate thread from the user application and uses one asynchronous **EventLoop** with event listeners to handle each of the network connections.

As we can see in Figure 6-1, the library consists of three main components, the **Channel Queue Manager** (CQM), **Scheduler**, and **Connection Manager**(CM), with the CM containing a NetHandler object for each TCP connection. Applications make calls to open, configure, and send data to channels through the interface provided by the CQM. The Scheduler then performs the multiplexing of the data from the n channels that are available at the CQM to the m connections maintained by the CM. The CM is responsible for instantiating a NetHandler object for each TCP connection. The NetHandler queues data for each connection and register itself with the EventLoop to send when the kernel signals that the connection is available.

We now describe in detail the EventLoop and each of the components.

6.2.1 Event Loop and Program Execution

The method of coordinating all the different components is role of the asynchronous EventLoop. The EventLoop allows for handlers to *wait on read/write availability on file descrip-*

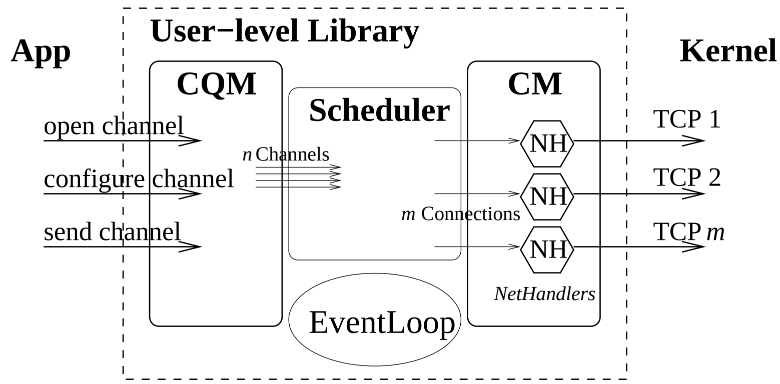


Figure 6-1: Overview of Program Architecture: Applications talk to the library through the Channel Queue Manager (CQM). The Scheduler performs the multiplexing between the channels and connections. Finally, the Connection Manager (CM) sends data on the TCP connections by putting data on a NetHandler object and having them wait on the asynchronous EventLoop to send data to the kernel.

tors. The EventLoop waits until an event occurs, as defined by the `select` statement in UNIX. Examples of events that can wake up the library are write availability on a TCP socket, read availability on a TCP socket, and read availability on a pipe used to message within the library.

The library uses this event signaling mechanism to synchronize the sending of all the different NetHandler objects. Each one of the NetHandler objects wait for the availability of their specific TCP connections to send data. When the write availability event is triggered, the NetHandler sends more data to the kernel.

The library also uses the EventLoop to signal to the Scheduler to wake up and schedule more packets to send on the connections. This happens through a chain of events. First, a NetHandler object is woken up by write availability on a connection. The NetHandler then sends some data on the connection. If the NetHandler needs more data to send, then it sends a signal registered by the CM to the Eventloop to queue more data. The CM is then woken up by the EventLoop, after which it scans through all connections to find the ones that need more data. Finally, it sends a signal registered by the Scheduler to wake up and send data to these connections.

The Scheduler is also woken up by the CQM when there is new data available for a channel, because it is possible that all the NetHandlers are empty and are waiting idly for more data.

6.2.2 Channel Queue Manager (CQM)

The **Channel Queue Manager** is the component responsible for managing the channels. It has three tasks:

1. provide an interface to the application to access the channels
2. provide an interface to the Scheduler to give it data as required
3. Queue the data for the application while the data waits to be scheduled to send

```
struct ChannelInfo {
  ChannelID  id;
  DestinationAddress  destination;
  ApplicationQueue  queue;
  int  current_sequence_number;
};
```

Figure 6-2: Channel Information for CQM: the CQM keeps data for each channel created by the application: id handle, the destination, the application queue to hold the data, and the current sequence number of the packets

```
class NetHeader {
  unsigned int channel_id;
  unsigned int seq_num;
  unsigned int request_type;
  unsigned int length;
};

class NetPacket {
    NetHeader head;
    string packet;
};
```

Figure 6-3: Packet and Packet Header Format

To the application, the CQM provides an interface to open channels, configure channels, and send data on channels. The usage pattern for an application is to 1)open a channel, 2) configure the channel to have weight w, and 3) send data on the channel.

When the application opens a channel to a destination, the CQM creates a data structure as shown in Figure 6-2. The CQM registers the channel for the destination, creates an application queue for that channel, and returns a handle to the application that can be used to reference the channel, much like what the kernel does in giving a file descriptor to an application. The application then sends the data together with the handle, and the CQM puts the data in the appropriate application queue.

To the Scheduler, the CQM provides an interface to provide the necessary data when sending data to a connection. The CQM is responsible for preparing the data as a packet that contains a library-specific header necessary for operation. As Figure 6-3 shows, the NetPacket contains the data to send in a string as well as a NetHeader class that provides the library with the necessary meta-data to send the packet appropriately. The NetHeader class contains a channel id that identifies the application channel, a sequence number to order the packets in sequence at the receiver, a request type to distinguish control messages from data packets, and the length of the packet to be able to correctly parse the NetPacket at the receiver. After the CQM sends a packet for a channel, it increments the current_sequence_number for the channel by one to continue the sequence.

6.2.3 Connection Manager (CM)

The **Connection Manager** is the component that is responsible for sending the data on the open connections with the kernel as fast as possible. The CM divides the work for each connection into a subcomponent, the NetHandler.

```
class NetHandlerData {
  DestinationInfo  destination;
  TCPSocket socket;
  DataQueue queue;
  unsigned int threshold;
  IOHandle notify_queue;
};
```

Figure 6-4: NetHandler Information for CM: Each NetHandler has its own data queue to store a buffer of data for efficient throughput. It also has a threshold value so that it can request for more data before the queue becomes empty. The IOHandle is the handle to the CM to wake up and fill up the queue.

Each NetHandler object is responsible for exactly one connection to send data to. When instantiated with a destination address, it makes a series of system calls to the kernel to open a TCP socket to that destination. As Figure 6-4 shows, this data is stored for each NetHandler. Also, in order to send data efficiently, each NetHandler keeps a FIFO data queue. This ensures that the NetHandler always has more data to send, because otherwise, the CM can have an empty queue and have to wait for the application to give it more data when connections are available. In order to keep the queue full, the NetHandler has a *threshold* value, and when the queue size gets below the *threshold*, then the NetHandler sends a request to the CM to fill up the queue to at least $queue\_fill\_factor * threshold$, which is some multiple of the threshold value as determined by the Scheduler.

The NetHandler schedules itself to be woken up by the EventLoop when the socket is signalled as being available for write. When the NetHandler wakes up, it sends as many bytes as possible from the front of the queue to the connection. It removes as many bytes from the queue as was sent to the connection, and it reschedules itself with the Event Loop. Finally, if the queue drops below *threshold*, it signals to the CM via the IOHandle *notify_queue* to wake up and give it more data to fill up the queue. The IOHandle is one side of a pipe, and the CM is registered with the EventLoop to wait on the other side of the pipe.

The Connection Manager provides the Scheduler with an interface to the connections. Whenever an application opens a channel with a destination, the CM makes sure that there are m connections open with the destination. For the first channel, the CM instantiates m NetHandler objects and assigns to each NetHandler a handle, a connection id, that the Scheduler uses to identify this connection.

As mentioned before, the CM registers itself with the EventLoop to listen on a pipe, so the NetHandlers can wake up the CM when the data queue drops below *threshold*, at which point the CM writes to the pipe which signals the Scheduler to wake up and check the CM for the list of connnections that need more data.

Pseudocode for the Reorder Buffer:
begin
  $current\_sequence\_number = 0;$
  while $packet := receive\_data()$ **do**
    $buffer[packet.header.seq\_num] = packet;$
    while $buffer[current\_sequence\_number + 1] \neq empty$ **do**
      call $send\_to\_app(buffer[current\_sequence\_number + 1]);$
      $current\_sequence\_number + +;$
    od
  od
end

Figure 6-5: Reorder Buffer Pseudocode

6.2.4 Scheduler

The **Scheduler** is the component that is responsible for multiplexing the channels onto the connections in a specific manner to obtain the desired bandwidth allocation. It registers itself with the EventLoop to be woken up when it has been signalled by another component via writing to a pipe. When the Scheduler wakes up, it goes through the specific algorithm to fill the necessary connections with data to send.

The Scheduler uses the CM interface to access the list of connections to know how much data can be sent for each connection. The CM returns a list of connections, their queue sizes, and the *threshold* for each connection. The Scheduler keeps a value, *queue_fill_factor*, and for connections whose queue sizes are below *threshold*, it fills it up to $queue\_fill\_factor * threshold$.

To pick what channels send the data, the Scheduler follows the specific implementation algorithm, and it has access to the interface provided by the CQM. The CQM tells the Scheduler the size of the next packet on each application queue, which the Scheduler can then use to determine which channel gets to send. Also, the CQM provides the Scheduler with the relative weight of each connection, which it can use to achieve the appropriate bandwidth allocation.

The specific algorithms for the striping and pinning are described in Section 5.4.1 and Section 5.4.2. The implementation of these algorithms are shown in the next section.

6.2.5 Reorder Buffer

In the cases where a channel is sent over more than one connection, the packets for a specific channel may arrive at the destination in a different order than when they were sent. Applications often have a requirement that the data they send needs to arrive at the destination *while preserving order*. We therefore implement a Reorder Buffer to send packets to the application in the original order.

This Reorder Buffer acts much like the TCP receiving window. The pseudocode for this element is shown in Figure 6-5. The incoming packets are placed into a buffer indexed by their packet header sequence numbers. The Reorder Buffer maintains a counter that remembers the sequence number of the last packet sent to the application. If a packet with the next sequence number resides in the buffer, then that packet is sent out to the application, and the counter is incremented by one.

6.3 Scheduling Algorithm: Striping

The steps of the Striping algorithm, as described in Section 5.4.1, are to perform Deficit Round Robin (DRR) to achieve bandwidth allocation and send on a first-available basis to multiple connections to achieve high throughput. When the EventLoop wakes up the Scheduler, the striping scheduling algorithm does the following:

1. pick the channel to get data from according to deficit round robin

2. pick the first available connection and sends the data to it

We will explain the implementation of the algorithm in detail here. The pseudocode is available in Figure 6-6.

The Scheduler needs to maintain a state variable for each channel, $score_i$, to keep track of the deficit score as described in the DRR algorithm. The Scheduler first queries the CM to get the list of connections and their respective queue sizes and thresholds. From here, the Scheduler selects the subset of connections that have $queue\_size_i < threshold_i$ and puts it into the list *conns*, because these connections are the connections that can take more data.

The Scheduler then picks the channel to send via the deficit round robin algorithm. It searches for the channel that has the least $f_i = \frac{next\_packet\_size_i - score_i}{w_i}$, because this is the channel that has the least number of steps to go before being allowed to send a packet, according to DRR. The division by w_i reflects that each channel moves w_i steps for every global step, thereby giving channels with higher weights more bandwidth. Suppose we choose to send from channel k. The Scheduler proceeds f_k global steps forward in the DRR algorithm. The other channels are then updated to reflect that they were not able to send during this time, so they increase their score by $f_k * w_i$ (same reason for multiplying by w_i), except for channel k, the one that sent, which resets itself to 0. The Scheduler has, at this point, successfully performed an iteration of DRR.

The Scheduler has to then choose the connection to send the data. Among the connections in *conns*, it finds the one with the least *queue_size* and chooses this connection to send. If we choose channel k, and this results in $queue\_size_k + next\_packet\_size_k > threshold_k * queue\_fill\_factor$, then this connection is deemed full and is removed from *conns*. In this way, the Scheduler sends the data on a first-available basis onto the connections.

6.4 Scheduling Algorithm: Pinning

There are two goals of the pinning algorithm. The first goal is to *simplify sending* by preselecting a pinning assignment of channels to connections, so that a channel continues to send on the same connection. The second goal is to try to *achieve bandwidth allocation* by performing a decreasingly ordered best fit greedy algorithm, as described in Section 5.4.2, which is a dual approximation to Weighted Fair Queueing. The majority of this algorithm is implemented at the time of channel creation, so we start the detailed description by looking at how the Scheduler chooses the assignment. The pseudocode for this section is shown in Figure 6-7.

The Scheduler first orders the channels by decreasing weight order. It obtains the weights of the channels by querying the CQM, and sorts the list using an introsort algorithm [15] that runs in worst case complexity $O(n \log n)$ that is implemented in STL for C++.

```
Pseudocode for stripe scheduling at an iteration:
begin
  comment: sets up the conns variable for later usage
  conns := {};
  foreach conn ∈ CM.get_queue_status() do
    if conn.queue_size < conn.threshold
      then conns.insert(conn) fi od
  .
  while conns ≠ {} do
        comment: choose channel to send
        bestchannel := 1;
        for i := 1 to n do
            if steps_left(i) < steps_left(bestchannel)              defined below
              then bestchannel := i;
            fi od
        .
        comment: choose connection to send to
        bestconnection := 1;
        for i := 1 to conns.size() do
            if conns[i].queue_size < conns[bestconnection].queue_size
              then bestconnection := i;
            fi od
        .
        comment: Update status variables
        for i := 1 → n do
            channel[i].score+ = steps_left(bestchannel) * w_i;
        od
        conns[bestconnection].queue_size+ = channel[bestchannel].next_packet_size;
        channel[bestchannel].score := 0;
        channel[bestchannel].next_packet_size := new_packet_size;
        .
        comment: send from the channel bestchannel to the connection bestconnection
        send_data(bestchannel, bestconnection);
        .
        comment: remove connection if full
        if conn_full(bestconnection)                                defined below
          then conns.remove(bestconnection);  fi
  od
where
funct steps_left(i) ≡ (channel[i].next_packet_size − channel[i].score) * 1/w_i;
.
funct conn_full(i) ≡
  if (conns[i].queue_size > conns[i].threshold * queue_fill_factor)
    then true;
     else false;
  fi
.
end
```

Figure 6-6: Detailed pseudocode for striping scheduler

Pseudocode for establishing pinning assignments:

```
begin
   Channels := {channel[1], channel[2], ..., channel[n]};
   OrdChannels := sort(Channels, introsort, decreasing);
   .
   comment: Perform the assignments using the state variable bin[i]
   ∀i. bin[i] = 0;
   for i := 1 → n do
        minbin := 1;
        for j := 1 → m do
             if bin[j] < bin[minbin] then minbin := j; fi
        od
        MakeAssignment(OrdChannels[i], minbin);
        bin[minbin]+ = OrdChannels[i].weight;
   od
end
```

Figure 6-7: Pseudocode for Establishing Pinning Assignments: first makes the assignment of channels to connections, then secondly the pseudocode for runtime execution

The Scheduler then takes each channel and assigns it to a connection that has the least weight assigned to it so far. It maintains a state variable, bin_j, that keeps track of how much total weight each connection has been assigned. Channel i finds bin j with the minimum size and makes the pinning assignment of channel i to connection j, as well as adding its weight into that bin $bin_j = bin_j + weight_i$.

We will now take a look at what happens during runtime. The pseudocode for this section is shown in Figure 6-8. During execution, the CM wakes up the Scheduler when connections are available. The Scheduler chooses a connection, then follows the same DRR algorithm as the striping Scheduler, except that the range of channels to choose from is now not all the channels, but the channels that are pinned to that connection.

The pinning Scheduler first queries the CM and obtains the list *conns* as described in Section 6.3. It then chooses the connection in *conns* that has the least *queue_size*. The Scheduler then looks up all the channels that are pinned to this connection, and among these channels, finds the channel that has the least $f_i = \frac{next\_packet\_size_i - score_i}{w_i}$. It chooses to send this channel onto this connection, and makes the appropriate updates to the *next_packet_size*, *score*, and *conns* as described before for the striping Scheduler.

Pseudocode for Pinning Scheduler:

```
begin
  comment: sets up the conns variable for later usage
  same as striping Scheduler
  .
  while conns ≠ {} do
        comment: choose connection to send to
        same as striping Scheduler
        .
        comment: choose channel to send from
        PinnedChannels := {set of channels assigned to this connection};
        bestchannel := 1;
        for i := 1 to PinnedChannels.size() do
            if steps_left(i) < steps_left(bestchannel)                 defined below
              then bestchannel := i;
            fi od
        .
        comment: Update status variables
        do the same as in striping, but only to the channels in PinnedChannels
        comment: send and remove full connection
        same as striping Scheduler
  od
where
funct steps_left(i) ≡
  (PinnedChannels[i].next_packet_size − PinnedChannels[i].score) ∗ 1/w_i;
.
end
```

Figure 6-8: Pseudocode for Running the Pinning Scheduler: Very similar to the striping Scheduler pseudocode, except for order and the selection of channels to choose from for a given connection

Chapter 7

Evaluation

In this section, we evaluate the implementation described in the previous chapter. We first describe the methodology of the experiments and the measurements, then evaluate and compare each of the two schedulers.

7.1 Methodology

7.1.1 Netbed

In order to gain the benefits of an actual implementation with the flexibility of choosing link characteristics to test the range of different network environments, we ran our experiments on the Utah Netbed [24], which derived from the Utah Emulab Testbed. Netbed is an environment made to facilitate network and distributed system research through emulation. It offers a large number of heterogeneous nodes and configurable network, providing *virtual machines in a virtual network environment* for experimentation. It also provides *ease of use* through automated setup and configuration.

Netbed is comprised mainly by 168 PCs in University of Utah and 50 at University of Kentucky. Each machine has five 100Mb Ethernet interfaces, one used for control, and the other four used for any configuration in the experiment. The computers are connected via programmable high-end switches to the testbed backplane, which are themselves connected by 2 Gb/s links.

The network is properly emulated through isolation from other networks and emulation of network characteristics within the experiment. An experiment isolates itself from other experiments running on the Netbed through the use of Virtual LAN (VLAN) technology implemented in the routers. The experiments emulate network characteristics by using DummyNet [20], a configurable network simulator implemented in the kernel that simulates routers with bounded queue sizes and different queueing policies, as well as bandwidth and delay of communication pipes. Netbed inserts DummyNet nodes in between two experimental nodes, and by going through the DummyNet, it transparently emulates network link characteristics between any two machines. DummyNet allows the virtual network topologies to be implemented with two kinds of router queues, DropTail and Random Early Detection, with configurable parameters.

An experiment sets up a network topology by sending the system a configuration file of a virtual topology written in ns script [19]. The Netbed system automatically maps the virtual nodes to physical machines, loads the appropriate operating system on them, and

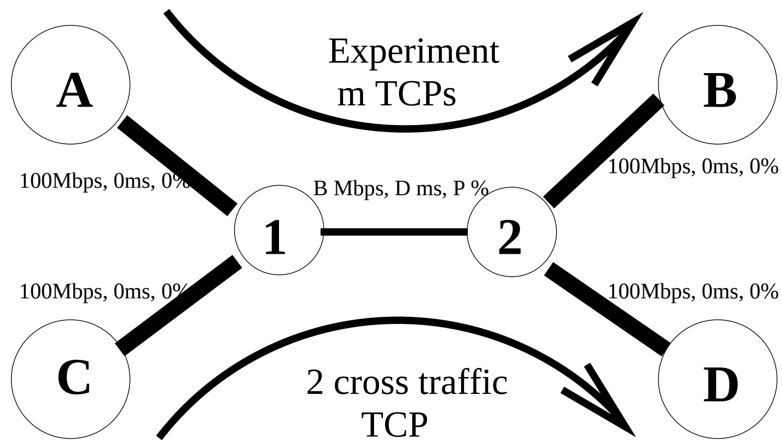


Figure 7-1: Network Topology: Experiments going from Node A to Node B through the bottleneck link between router 1 and router 2. There are 2 cross-traffic TCPs also going through this link. The routers are running RED.

configures the network to have the right network characteristics, the right routers, and the right routing information.

7.1.2 Network Topology

The network topology we configure in the emulation takes into consideration that we want to reduce the number of variables as well as having cross-traffic TCP connections to study the effects of cross traffic on the scheduling algorithms. As Figure 7-1 shows, we use a "dumbbell" network topology, where two network paths share the same bottleneck link. The first path is our experiment path, going from node A to node B through two routers, router 1 then router 2. The second path is our cross traffic path, going from node C to node D also through router 1 then router 2. This topology allows the experiment running on node A and node B to not be affected by the other applications in the emulation.

In order to isolate the bottleneck to be the link between router 1 and router 2, the connections from the nodes to the routers are high bandwidth low delay links (100 Mbps, 0 ms) with no packet loss. The link between the two routers changes depending on the experiments, but this link runs slower than the links between the routers and the nodes, ensuring that the bottleneck link remains the same. The routers implemented a RED queue with a maximum average size of 40 packets, so we avoid having synchronization issues between the multiple TCP connections..

7.1.3 Cross Traffic

We run cross traffic TCP connections to study the effects of the scheduling algorithm in shared network environments. Each of our experiments runs two cross traffic TCP connections from node C to node D. The programs run on 600 MHz Intel machines running FreeBSD 4.7. Node D runs a server that accepts TCP connections and throws away the data. Node C runs a client that opens up TCP connections, sets up the socket buffer to 200 KB, then sends data as fast as possible indefinitely. The programs run on different machines

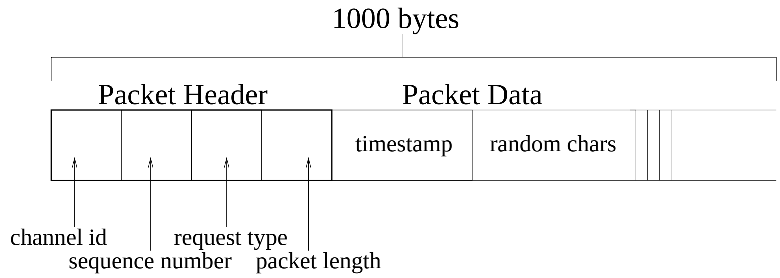


Figure 7-2: Packet Data

than those that run the experiments to minimize unwanted processor or bus effects on the experiment.

7.1.4 Sender and Receiver

The Sender and the Receiver are part of a sample application that uses the implementation of the network manager. Each of our experiments sends data from the Sender on node A to the Receiver on node B, both of which are 600 MHz Intel machines running FreeBSD 4.7.

The Sender can be configured to simulate many types of applications and many configurations of the library. The Sender takes in parameters specifying the number of channels n, the number of connections m, the size of the total socket buffer s (explained later), and the relative weights of each channel w. The Sender sets up the CQM and CM with these parameters.

One goal of the experiments is to observe how the different scheduling algorithms achieve proper bandwidth allocation. We choose not to specify all the weights of the channels individually, because this is tedious and hard to analyze, and we also choose not to make all the weights the same, because this is not as interesting of a case as different weight distribution. Therefore, we choose one parameter, the relative weight w, and combine it with the number of channels n to arrive at the weight distribution. The weight of each successive channel is w^{i-1}. So, the weights for the channels $\{channel_1, channel_2, ..., channel_n\}$ are $\{1, w, w^2, ..., w^{n-1}\}$.

The experiments assume that the application always has more data to send. The case when not all of the channels are sending adds unnecessary complexity to the problem. The Sender also timestamps the packet just before placing it in a small NetHandler queue, which almost immediately gets sent to the kernel buffer. The timestamp is useful for measurements as described later in the document

The Receiver, on the other hand, is a passive element that takes statistics on the experiment. It receives data sent on the m connections and demultiplexes them onto the n channels based on the channel id in the packet header. Since these packets can arrive out of order within a channel, each channel is connected to a Reorder Buffer as described in Section 6.2.5. The packets are removed from the Reorder Buffer whenever possible, measurements are taken, then the packets are discarded. These measurements are kept in a statistics log file and processed later for analysis.

7.1.5 Application Data

As shown in Figure 7-2, the application data packets are 1000 bytes each, and contain a 16 byte packet header consisting of the channel id, the sequence number, the request type, and the length of each packet. The rest of the packet is an 8-byte timestamp and 976 bytes of filler data.

Since the experiments run in batches, the experiments are carefully constructed to not interfere with one another. Each experiment starts by sending a start packet, distinguishable by the packet type in the packet header. The start packet signals the Receiver to start collecting a new set of measurements. Next, the Sender waits for 2 seconds to ensure delivery of the start packet, then sends data for 60 seconds. After 60 seconds, the Sender waits again for 2 seconds to make sure that all of the data is sent, then finally, it sends a stop packet to tell the receiver to stop taking this set of measurements.

7.1.6 Measurements and Evaluation Metrics

In order to quantify the effectiveness of the two schedulers, the analysis looks at the metrics for *sending performance*, *bandwidth allocation*, and *memory usage*. These metrics are calculated from experiment measurements of *throughput*, *delay*, and *reorder buffer size*. We first describe these metrics and how they are calculated. Then, we describe how the measurements for these metrics are collected by the Receiver.

The first metric is the **sending performance metric**, which is a measure of how well the network is being utilized. We calculate this by calculating $\frac{totalthroughput}{delay}$ for the experiment. This metric is meaningful because having high throughput and low delay means that the network is being utilized well. Inefficiencies in the system may cause throughput to drop, or delay to increase. Higher values for this metric mean better sending performance.

The second metric is the **bandwidth allocation metric**. The Receiver takes the distribution of average throughput measurements of each channel and compares this with two different distributions, the optimal distribution and the expected distribution. The analysis uses two ways of comparing distributions, Root-Mean-Squared (RMS) of the absolute difference and the RMS of the percentage difference.

The first distribution is the *optimal distribution*, which is used to calculate how far the observed distribution differed from the weight distribution. As described in Section 7.1.4, the sender picks a relative weight w, and chooses the weights for each connection by taking powers of w: $weight_i = w^{i-1}$. The optimal distribution is calculated by taking the total bandwidth of the experiment and dividing it in such a way to get relative weights of w for each channel. For example, if there are three channels, then the relative weights would be 1, w, and w^2. Therefore, the first channel should have $\frac{1}{1+w+w^2}$ fraction of the total bandwidth, the second $\frac{w}{1+w+w^2}$, and the third $\frac{w^2}{1+w+w^2}$. This is the optimal distribution that the implementation would like to achieve.

The second distribution is the *expected distribution*, which is computed for the pinning scheduler to calculate the difference between the observed distribution and the distribution that the pinning assignment algorithm should have given. This distribution shows whether the difference of the observed distribution from the optimal distribution is caused by the scheduler dynamics or by the ordered best fit pinning algorithm. The expected distribution is calculated by first stepping through the pinning assignment algorithm. For 3 channels and 2 connections, the weights would be 1, w, and w^2. The pinning assignment algorithm would assign channel 1 to connection 1, and channels 2 and 3 to connection 2. Since behavior of multiple TCP connections dictate that the connections are expected to have equal bandwidth, the expected bandwidth for each connection is $\frac{1}{2} totalbandwidth$. Channel 1 should then have $\frac{1}{2}$ of the total bandwidth. The second connection has two channels with weights w and w^2, so channel 2 should have $\frac{1}{2}\frac{w}{w+w^2}$ fraction of the total bandwidth, and channel 3 should have $\frac{1}{2}\frac{w^2}{w+w^2}$.

The actual bandwidth allocation metric is computed by calculating the Root-Mean-Squared (RMS) of the difference between the distributions. The RMS between the optimal and observed distribution gives one metric, and the RMS between the expected and observed distributions gives the other metric. A higher metric means that the distributions are farther away, hence the bandwidth allocation observed is worse. Similarly, the metrics are computed for the percentage difference between the distributions, rather than the absolute difference.

$$RMS(opt) = \sqrt{\frac{1}{n} \sum_{i=1..n} (throughput_i - opt\_throughput_i)^2} \tag{7.1}$$

$$RMS_{perc}(opt) = \sqrt{\frac{1}{n} \sum_{i=1..n} (\frac{throughput_i - opt\_throughput_i}{opt\_throughput_i})^2} \tag{7.2}$$

$$RMS(exp) = \sqrt{\frac{1}{n} \sum_{i=1..n} (throughput_i - exp\_throughput_i)^2} \tag{7.3}$$

$$RMS_{perc}(exp) = \sqrt{\frac{1}{n} \sum_{i=1..n} (\frac{throughput_i - exp\_throughput_i}{exp\_throughput_i})^2} \tag{7.4}$$

The final metric is the **memory usage metric**, which is a measure of how much memory the algorithms take to run. While memory is generally inexpensive compared to application throughput and delay, some systems are heavily memory constrained. These systems care about how much memory the scheduling algorithm takes up. This metric is primarily the size of the reorder buffer, because this value is determined by the behavior of the scheduling algorithm, while other data structures are just measurements of implementation efficiency and are also not very substantial. Lower value for this metric means better memory usage.

To calculate these metrics, the Receiver takes measurements for *throughput*, *delay*, and *memory usage* while receiving the data for each experiment. The measurements are taken by the Reorder Buffer component, starting when the start packet arrives, and ending when the stop packet arrives. These values are then used to compute the evaluation metrics for send performance, bandwidth allocation, and memory usage as described previously.

For **throughput**, the experiment keeps the throughput on a per-channel basis as well as the total throughput of all the channels combined. The Reorder Buffer sums the number of bytes received for the channel from the start packet to the stop packet and divides it by the length of the experiment to get the average throughput per second for the channel. The throughput for each channel is summed up to get the average throughput for the whole experiment.

For **delay**, the experiment keeps two different measurements: the TCP delay and the application delay. The Reorder Buffer reads the first 8 bytes of the packet data , giving it the send time of the packet. It then records the time of packet reception before putting the packet into the reorder buffer, calculating the time difference as the TCP delay. It records the time again when the packet is dequeued, calculating the difference with the send time as the application delay of the packet. The clocks on the two machines are synchronized using *ntptime*, a networked system time synchronizer, putting the two system clocks within a few ten microseconds apart. The individual delay times are averaged together with all the other packets, giving the average TCP delay and the average application delay for the experiment.

For **memory size**, the Reorder Buffer keeps track of the average size of its buffer. The size at any point in time is calculated by summing up the total number of bytes of all the packets in the reorder buffer. A sample point is taken after each packet is received, put in the buffer, then as many packets as possible are dequeued in order. The final value used is the average of all the sample points.

7.1.7 Experiments

We ran a series of experiments to study how varying the number of channels n and the number of connections m affects performance for each of the schedulers. We also varied the link characteristics (bandwidth B, delay D, and packet loss rate P) to determine how different network conditions affect performance of each of the schedulers.

Each experiment lasts for 60 seconds, which provides enough time to get past the initial startup behavior of the experiment to look at steady-state behavior.

As a default, we run experiments with 15 channels, 5 connections, and relative weight of 0.8 between the channels, on a 10 Mbps link with 25 ms one-way delay (50ms RTT) and packet loss rate of 0%. We then run experiments varying one variable at a time from the default values. We vary the number of channels n from 1, 2, 3, 4, 5, 7, 10, 13, 16, and 20, the number of connections m from 1, 2, 3, 4, 5, 7, 10, 13, 16, 18, 20, 23, 26, and 30. We also vary bandwidth B from 3Mbps, 5, 8, 11, 14, 18, 25, 35, and 50 Mbps, one-way delay $\frac{D}{2}$ from 0, 3, 5, 7, 10, 15, 20, 30, 40, 50, and 75 ms, and packet loss rate P from 0, 0.1, 0.2, 0.4, 0.8, 1.5, 2, 3, and 5 percent.

We produce graphs for each of these experiments. For every set of parameters, we run the experiments 15 times and plot both the mean and the standard deviation.

Variable	Units	Default	Range
m	connections	5	1, 2, 3, 4, 5, 7, 10, 13, 16, 18, 20, 23, 26, 30
n	channels	15	1, 2, 3, 4, 5, 7, 10, 13, 16, 20
B	Mbps	10	3, 5, 8, 11, 14, 18, 25, 35, 50
D	ms	25	0, 3, 5, 7, 10, 15, 20, 30, 40, 50
P	percent	0	0, 0.1, 0.2, 0.4, 0.8, 1.5, 2, 3, 5

7.1.8 Socket Buffer

One side effect of our choice to use TCP is that the application has to set the kernel socket buffer. Understanding and correctly setting the socket buffer is crucial to making sure that we are not tainting the measurements of the scheduling algorithms.

Let us first describe the role of the socket buffer. The socket buffer is the memory that the kernel uses to hold the runtime data structures for TCP. The main usage of this memory is to store the packets in the TCP sending window. There are several side effects for not setting the socket buffer size appropriately for a connection.

Setting the socket buffer size too low *limits the throughput* achieved by the connection. Since the socket buffer contains the sending window, bounding the socket buffer size also bounds the maximum sending window. TCP sending window normally undergoes AIMD, increasing additively for successful window transfers and decreasing multiplicatively for lost packets, and oscillates consistently under stable network conditions. However, if the socket buffer limit is less than peak of this oscillation, then the window size can no longer increase. The Kernel now sends at a constant rate, and does not probe for enough of the total bandwidth. Therefore, the socket buffer limit should be at least greater than the maximum sending window size.

The maximum sending window size is determined by how much data can be sent out before the acknowledgement of the data can be received. The acknowledgement takes at least a round-trip time to return, because the packet has to get to the destination and the ack has to come back. During this time, the sender can send $bandwidth * rtt$ bytes of data, which is called the bandwidth-delay product of the connection. Therefore, *the socket buffer size should be at least the bandwidth-delay product.*

Setting the socket buffer size below a few network packets *increases the delay and decreases the throughput* of the connection. This happens because if there are only a few packets in the sending window, then the receiver does not send enough duplicate acks for the sender to perform a fast retransmit as defined in Section 4.3.1. Therefore, the connection suffers from a timeout, which decreases throughput and increases delay of the packets.

However, setting the socket buffer size too high *increases the application delay* experienced by the packets. The kernel continues to accept packets from the application until the socket buffer is full. Therefore, if the socket buffer size is much larger than the sending window, then the buffer contains packets that cannot be sent yet because it is not in the sending window. These packets incur greater application delay, which is important because the application no longer has access to those packets that are sent to the kernel.

We adjust the size of the socket buffers for each of the experiments to match these constraints. Since we run m connections, we first look at the aggregated throughput of the connections, and set the aggregate socket buffer size for all the connections. The aggregate throughput of m connections with two cross traffic TCP connections is roughly $B * \frac{m}{m+2}$ where B is the bandwidth of the bottleneck link. This is because the connections all have the same RTT, and TCP connections with the same RTT fairly share the available bandwidth at the bottleneck router (Section 4.3.4). Therefore, the expected throughput for each individual connection is $\frac{B}{m+2}$.

We finally set the socket buffer size for each connection to be the maximum of 5000 bytes, which accounts for the minimum socket buffer size, and the *bandwidth-delay product*, which is $\frac{B}{m+2} * (2D)$, where the factor of two is because we want the round-trip delay rather than the one-way delay.

$$socket\ buffer\ size = max\{5000bytes, \frac{B * 2D}{m+2}\} \tag{7.5}$$

7.2 Results: Sending Performance

The sending performance, as described in Section 7.1.6, shows how well the Scheduler utilizes the available network resources. This metric looks at two measurements, the delay and throughput measurements. We compare the two schedulers by evaluating at how they react to the different parameters.

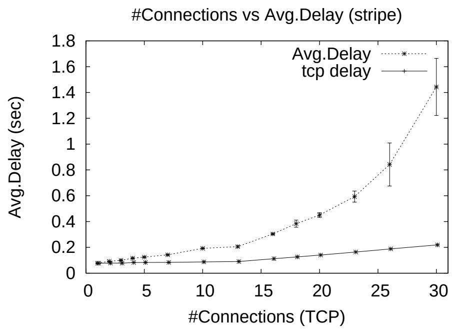


Figure 7-3: Striping: Connections vs. Delay

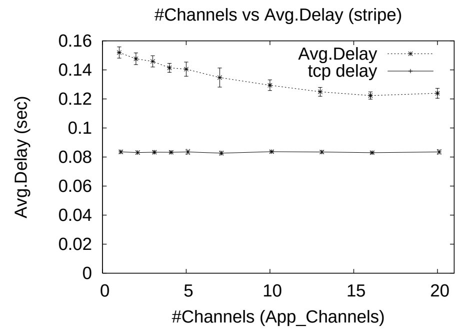


Figure 7-4: Striping: Channels vs. Delay

7.2.1 Striping Scheduler

The sending performance of the striping Scheduler is highly influenced by the number of connections m. We show that *there is an optimal choice for the number of connections* depending on the importance of throughput and delay to the application. We show that this optimal choice changes as the number of cross traffic connection and the packet loss rate P of the connection changes.

The delay experienced by the striping algorithm can be broken down to two distinct sources, the TCP delay and the reordering delay. Taking the TCP delay and the application delay measurements, as described in Section 7.1.6, the reordering delay is *application delay*$-$ *TCP delay*. This is the delay incurred by the packets while in the reorder buffer but are unable to be processed by the application due to a missing packet earlier in the sequence.

Figure 7-3 and Figure 7-4 show how the delay is affected by changing the number of connections m and the number of channels n respectively.

As we increase the number of connections, we find that the channels experience more reordering, causing the reordering delay to increase. This is because a lost packet in one connection does not prevent all the other connections from sending their data which may be out of order. With m connections, if one connection experiences a timeout and delays its packets for c rtt's, then the other connections send *total bandwidth* $* \frac{m-1}{m} * c * rtt$ bytes as long as they also do not lose packets. Also, opening up m connections causes the window increasing scheme to increase by m packets every rtt and decrease by $\frac{1}{m}$ when it loses packets, which is a more aggressive probing of the available bandwidth, and causes more packet losses than before. This further adds to the fact that increasing the number of connections increases the number of packets that get reordered, and hence increases the reordering delay. In Figure 7-3, we see that at $m = 14$, we have approximately 100ms of reordering delay, and at $m = 30$, we have 1.2s of reordering delay. The slight rise in TCP delay is the result of the minimum socket buffer size limit, which is explained in the pinning scheduler, but the main contributor to the application delay is the reordering delay. In either case, *more connections cause higher delay.*

We find that increasing the number of channels n causes reordering delay to decrease

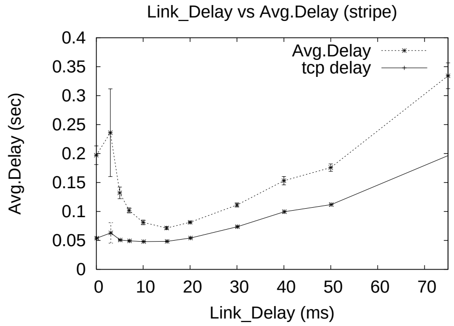


Figure 7-5: Striping: Link delay vs. Average Delay

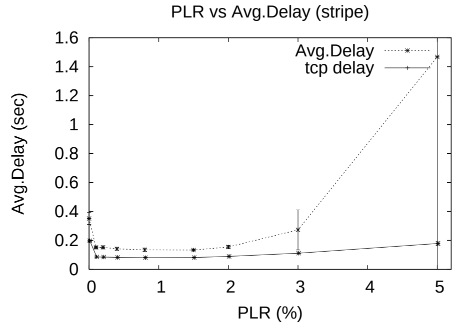


Figure 7-6: Striping: Packet Loss vs. Average Delay

slightly. Figure 7-4 shows this for the case when the number of channels $m = 5$. If a packet is lost and a channel delayed, hence causing a set of packets to be received out of order, then this affects only the channels that contain packets in this set. As n increases, the delayed packets affect a lower percentage of the total number of packets. If, in a period of time, P packets are sent and L of them are delayed by a connection, then this affects up to L channels. On average, every channel receives $\frac{P}{n}$ packets, so $L\frac{P}{n}$ packets are affected. While this is only a rough estimation, we can see how increasing n can cause the number of delayed packets to reduce, thereby decreasing the average reordering delay.

We also find that link delay causes a linear increase in TCP delay (Figure 7-5), and higher packet loss rates cause higher reordering delay (Figure 7-6). The link delay adds to the TCP delay, and does not cause an increase in the reordering delay. This shows that the effects of link delay on application delay are quite limited, and only affects it in a simple manner. Higher packet loss rates means that the connections are more likely to timeout, and each timeout causes more packets to arrive out of order, thereby increasing the reordering delay.

In conclusion, the main causes for high application delay are high number of connections, high link delay, and high link packet loss rate.

While delay has a negative effect on sending performance, higher throughput means better sending performance. As we change change the number of connections m, we find that throughput increases. We also find that increasing bandwidth, lowering delay, and lowering packet loss rate all positively affect throughput.

As the number of connections m increases, the experiment obtains more bandwidth from the cross traffic, causing the total throughput to increase. Figure 7-7 shows that the throughput roughly follows the $\frac{m}{m+2} * total\ bandwidth$ expectation as described in Section 5.2.

Changing the link characteristics follows the trends for TCP as described in Section 4.3.3. Increasing bandwidth changes a linear increase in throughput, while changing the delay and packet loss rate changes the throughput according to the $\frac{1}{RTT*\sqrt{P}}$ limitation of TCP throughput.

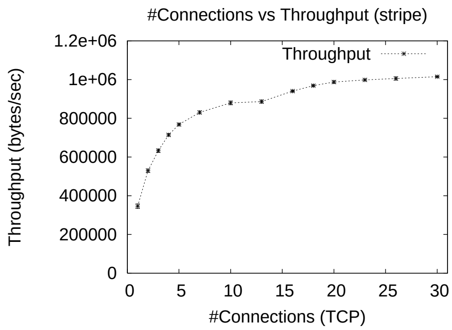


Figure 7-7: Striping: Connections vs. Throughput

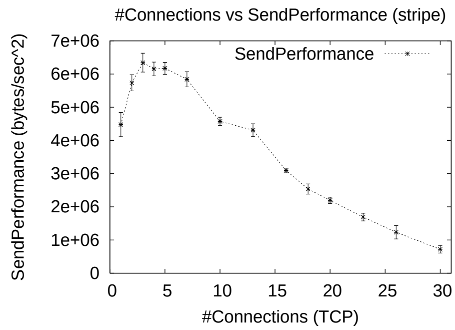


Figure 7-8: Striping: Connections vs. Sending Performance

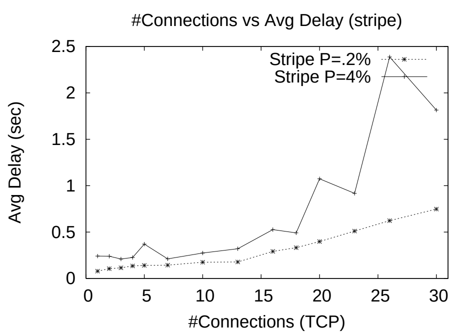


Figure 7-9: Striping: Connections vs. Delay 2

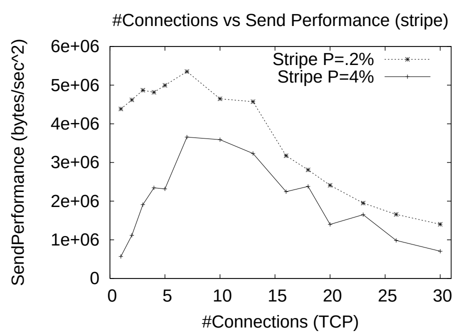


Figure 7-10: Striping: Connections vs. Sending Performance 2

The throughput, then, is affected negatively by low m, low B, high RTT, and high P.

The sending performance metric is a combination of the delay and throughput characteristics, computed as $\frac{delay}{throughput}$. The most meaningful variable is the number of connections m in the experiment. As Figure 7-8 shows, there is a *peak in sending performance*.

Initially, the sending performance increases because the experiment obtains more throughput from the cross traffic, while delay is staying the same. But, as the number of connections m increases, the rate of increasing throughput decreases, and the increasing delay caused by reordering takes over, causing the sending performance metric to go down. Therefore, the optimal value of m is somewhere in the middle.

This point of optimal m, however, shifts as link characteristic changes. The main cause of the shift is an increase in the packet loss rate, which causes the optimal point for m to shift to lower values. As Figure 7-9 and Figure 7-10 shows, higher values of packet loss rate causes delay to increase, thereby causing the optimal sending performance to decrease.

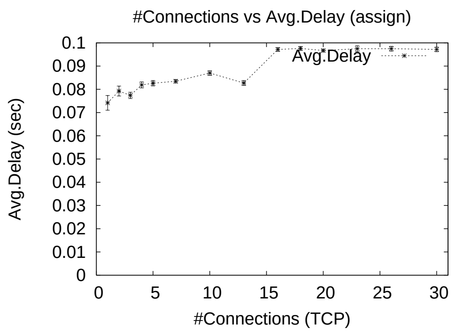


Figure 7-11: Pinning: Connections vs. Average Delay

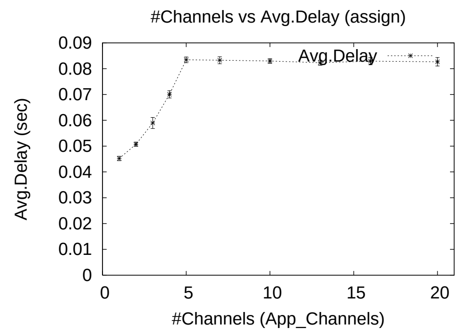


Figure 7-12: Pinning: Channels vs. Average Delay

This, as we shall see later, determines whether the striping approach is a good approach.

In conclusion, the striping Scheduler's sending performance peaks at a certain number of connections m*, which shifts both to lower values of* m *and lower sending performance as packet loss rate increases. The sending performance is independent on the number of channels.*

7.2.2 Pinning Scheduler

Like the striping Scheduler, the pinning Scheduler is also very influenced by the number of connections. We show that increasing the number of connections m increases sending performance, but once it exceeds the number of channels n, then sending performance does not change. Furthermore, if there are a lot of cross traffic connections, then the bandwidth obtained decreases, and higher values of n are required to be able to achieve the same bandwidth. This shows that the pinning scheduler is fit for applications that continue to maintain a high number of channels. We also show that the choice of an appropriate m for sending performance is relatively stable under changing network conditions.

Application delay for the pinning scheduler consists only of the TCP delay, and therefore, follows the behavior of delay for aggregated TCP connections. The Scheduler pins each channel to a connection, and since TCP sends packets to the library in order (Section 4.3.1, the library does not have to do any reordering. Therefore, the variables that affect application delay are the same as those that affect delay of aggregate TCP connections, which are m, link delay and link packet loss rate.

The TCP delay caused by an increasing number of connections m results from the minimum socket buffer size limitation as described in Section 7.1.8. As m increases, the bandwidth obtained by an individual connection decreases, thereby decreasing the bandwidth delay product for that connection. The socket buffer size, then, keeps on decreasing until the minimum buffer size limit is reached. After this point, the socket buffer size starts to be higher than the bandwidth-delay product, so the packets incur more delay while in the socket buffer and waiting to be sent. However, when $m > n$, increasing the number of con-

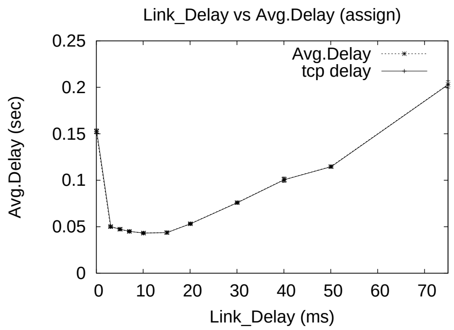


Figure 7-13: Pinning: Link delay vs. Average Delay

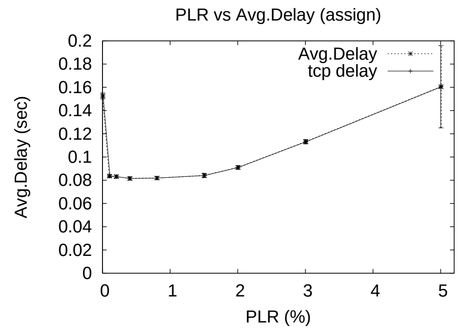


Figure 7-14: Pinning: Packet Loss vs. Average Delay

nections does not cause a change in the sending performance. This is because the pinning algorithm has no more channels to assign to those connections, hence those connections go unused and do not affect sending performance.

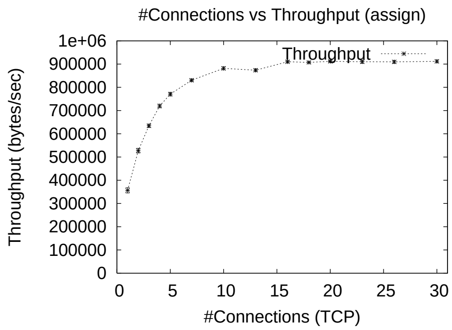
Figure 7-11 shows this effect. For these experiments, the socket buffer size is 80KB with two cross traffic connections, so the socket buffer size limit reaches when $5KB = \frac{80}{m+2}$ (Section 7.1.8), or $m = 14$ connections. As can be seen in Figure 7-11, the delay starts to have an increase at $m = 14$. Then, as $m > n$, which happens at $m = 15$ with 15 channels, the delay stays constant.

Changing the number of channels n affects the delay when $n < m$, where changing n changes the number of used connections. As Figure 7-12 shows, within this region of $n < m$ where $m = 5$ connections, delay increases as n increases.

Changing network conditions also affects TCP delay. As Figure 7-13 shows, increasing link delay causes packets to take longer to reach the other computer, causing the TCP delay to increase proportionally. Increasing packet loss rate also causes TCP delay to increase, as connections lose more packets and have to wait for packets to be retransmitted before being able to send them to the CM in order. Figure 7-14 shows this increase in TCP delay.

The analysis of throughput for the pinning Scheduler is exactly analogous to the analysis for the striping Scheduler, except for the fact that everything stays constant in the region where the number of connections exceeds the number of channels ($m > n$).

Increasing the number of connections m causes an increase in throughput, which is the result of the connections in the experiment gaining a higher portion of the bandwidth at the bottleneck router. Figure 7-15 shows the increase according to the $B * \frac{m}{m+2}$ equation for this effect (Section 5.2). But, as m exceeds the number of channels, $n = 15$, we see that the throughput becomes constant. Any more connections that are added remain unused, so this does not affect throughput. By this point, however, the incremental gain of throughput for a connection is low enough that this limitation does not affect total throughput much. However, if the number of channels is low, or if there are many other cross traffic TCP connections, then this limitation hinders the library from obtaining a high percentage of the available bandwidth.



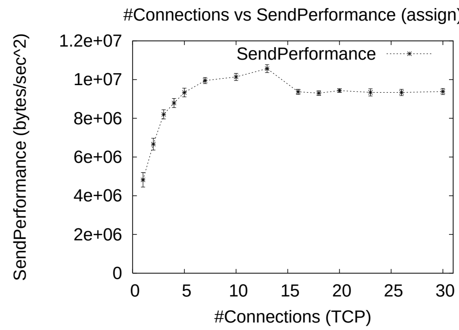
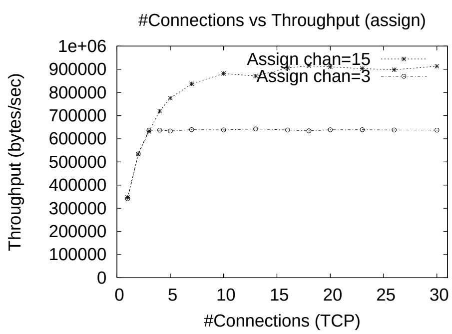


Figure 7-15: Pinning: Connections vs. Throughput

Figure 7-16: Pinning: Connections vs. Sending Performance



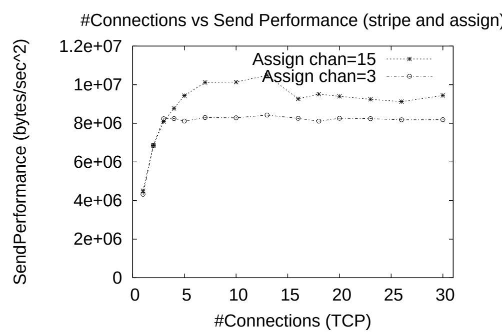


Figure 7-17: Pinning: Connections vs. Throughput 2

Figure 7-18: Pinning: Connections vs. Sending Performance 2

The effects of network characteristics on throughput parallel those with the striping Scheduler. Increasing bandwidth changes a linear increase in throughput, while changing the delay and packet loss rate changes the throughput according to the $\frac{1}{RTT*\sqrt{P}}$ limitation of TCP throughput.

Combining the behavior of delay and throughput, we see that sending performance generally increases as m increases, and that higher values of n relative to the number of cross traffic connections allows for better performance.

As Figure 7-16 shows, increase in m causes the sending performance to increase until one of two things happen. The first is the case that the minimum socket buffer is reached, which causes the delay to go up, causing the send performance to go down. This can be seen in Figure 7-16 when $m = 14$, as described in the delay section. The second is the case that $m > n$, in which the delay and throughput remains constant, and takes precedence over the first case. Before these limits, the sending performance initially increases due to rising throughput, while delay stays relatively constant. Then, the rate of increase of throughput decreases, while delay starts to increase at the minimum socket buffer size limit. But finally, as $m > n$, the connections go unused, and hence, does not affect performance.

Sending performance is better when there are more active connections open. However, if the number of channels n is low, then this limits the number of active connections, thereby limiting the sending performance. Figure 7-17 and Figure 7-18 shows this trend. The throughput and sending performance decreases as n decreases. Throughput also decreases as the number of cross traffic connections rise, so more cross traffic leads to lower performance, and higher values of n are required to be able to achieve the same performance level.

In conclusion, the pinning Scheduler's sending performance does better as the number of connections increases, but is adversely affected by low number of channels.

7.2.3 Comparison

The previous two subsections show how each of the schedulers, striping and pinning, perform according to the sending performance metric. We now compare the two to show that the pinning algorithm is able to have better sending performance when the number of channels n is high. We also show that, for links with a low packet loss rate, the striping scheduler is still able to obtain good throughput with reasonable delays, but as the number of connections m increases, the delay becomes much greater than that for the pinning scheduler. On the other hand, we show that when n is low, the pinning scheduler is unable to obtain a full capacity of throughput, and hence, the striping scheduler performs better for sending performance.

We have drawn these two conclusions from above:

- The striping Scheduler's sending performance peaks at a certain number of connections m, which shifts both to lower values of m and lower sending performance as packet loss rate increases. The performance is also independent of the number of channels.

- The pinning Scheduler's sending performance does better as the number of connections increases, but is adversely affected by low number of channels.

Looking at the application delay for the two schedulers in Figure 7-19, we see that the striping Scheduler has more delay, and the difference grows as the number of connections m increases. While the striping Scheduler grows with m, the pinning Scheduler maintains a constant delay, because it does not have to suffer from reordering delay. We also see that, as packet loss rate increases, the delay worsens further for the striping Scheduler.

Figure 7-20 compares the throughput for the two schedulers, and it shows that the throughput for the two are equivalent, until the pinning Scheduler becomes bounded in the case where there are more connections m than channels n, in this case when $m = 15$. Striping Scheduler has at least as much throughput as the pinning Scheduler, and for cases of low n, the striping Scheduler obtains more throughput than the pinning Scheduler.

Finally, Figure 7-21 shows the sending performance metric, as computed by $\frac{throughput}{delay}$ for the two Schedulers. The exact numbers for this graph are only meaningful in showing the trends, since the equation for the metric is chosen only to describe that high throughput and low delay is better for sending performance. The throughput for the two schedulers are the same for the most part, since the number of channels n for the pinning algorithm is high ($n = 15$). However, the delay for the striping Scheduler grows as m grows. This is

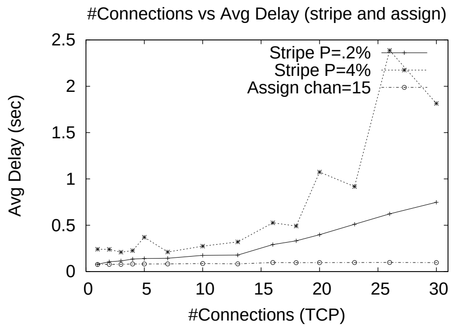


Figure 7-19: Comparison: Connections vs Average Delay

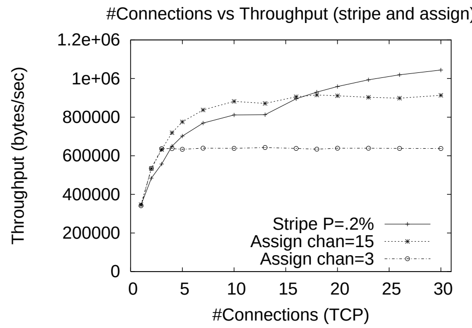


Figure 7-20: Comparison: Connections vs Throughput

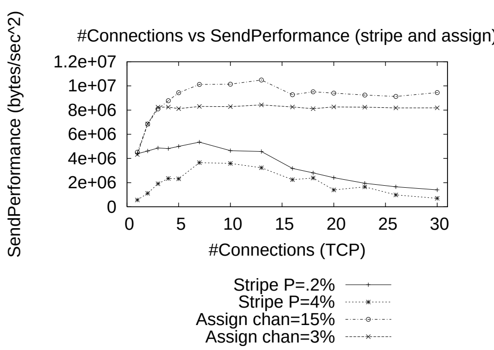
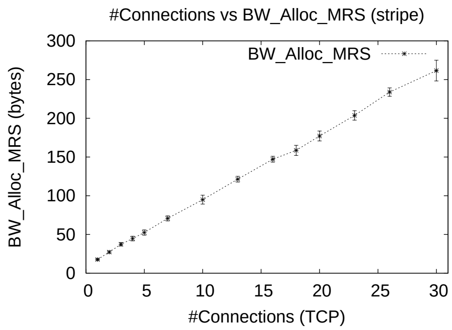


Figure 7-21: Comparison: Connections vs Sending Performance



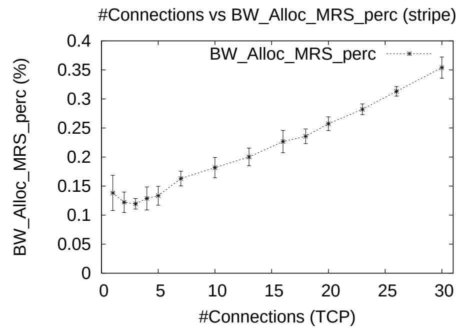


Figure 7-22: Striping: Connections vs. Bandwidth Allocation (optimum)

Figure 7-23: Striping: Connections vs. Bandwidth Allocation (optimum %)

shown by the decreasing sending performance when the number of connections m increases. However, for the appropriate choice of m, we see that the striping Scheduler is still able to obtain relatively good sending performance. As long as the application delay is acceptable, the striping Scheduler performs as well as the pinning Scheduler.

As the number of channels n decreases, the pinning Scheduler performance decreases, because any extra connections of $m > n$ remain unused instead of increase sending performance. As packet loss rate increases, the striping Scheduler performance decreases, which is caused by increasing delay.

In conclusion, the pinning Scheduler has better sending performance than the striping Scheduler when the number of channels n *is high. However, the striping Scheduler, for an appropriate choice of the number of connections* m*, is still able to obtain high throughput with reasonable delay, and remains a viable option. For low* n*, the pinning Scheduler becomes limited in performance, and the striping Scheduler is able to obtain better sending performance.*

7.3 Results: Bandwidth Allocation

The bandwidth allocation metric, as described in Section 7.1.6, is a measurement of how well the scheduler achieves the throughput ratio between the channels. We compute the RMS as well as the RMS_{perc} between the observed throughput and both the optimal and expected distributions. We show that the striping Scheduler achieves the optimal allocation to within one packet per channel. The pinning Scheduler, on the other hand, performs well if the number of channels n is much greater than m, but poorly otherwise.

7.3.1 Striping Scheduler

The nature of striping gives near-optimal bandwidth allocation. As described in Section 5.4.1 and in Equation 5.2, the deviation of throughput for any channel is within one application packet off of the optimal if all the packets in transit at the end of the experi-

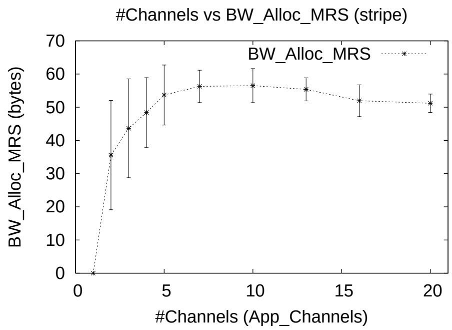


Figure 7-24: Striping: Channels vs. Bandwidth Allocation (optimum)

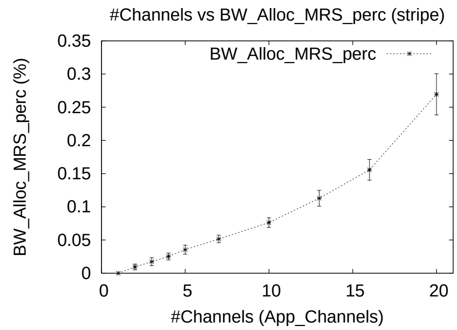


Figure 7-25: Striping: Channels vs. Bandwidth Allocation (optimum %)

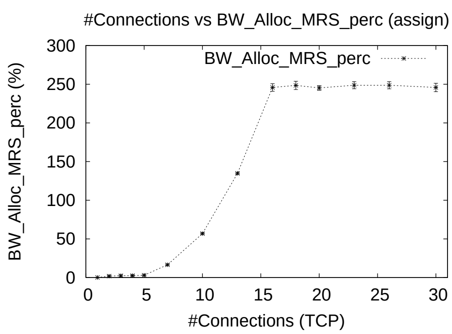


Figure 7-26: Pinning: Connections vs. Bandwidth Allocation (optimum %)

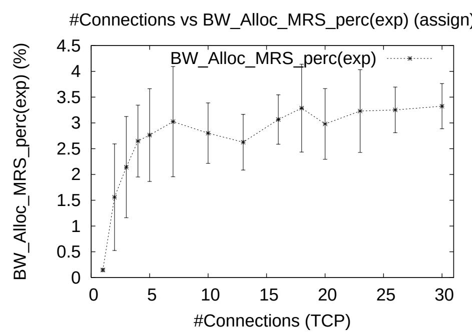


Figure 7-27: Pinning: Connections vs. Bandwidth Allocation (expected %)

ment are to be included. For our experiments, our packet sizes are all at 1000 bytes, so the maximum difference for any configuration of m and n for a specific channel is within 1000 bytes. This is shown for differing values of m and n in Figures 7-22 and 7-24.

7.3.2 Pinning Scheduler

The pinning Scheduler has less flexibility in how it sends data, because it pins the channels onto connections. Because of this, the pinning Scheduler is not able to achieve optimal bandwidth allocation. However, for the case when there are many more channels than connections ($n >> m$), the pinning Scheduler is still able to obtain good bandwidth allocation. We also show that the deviations are caused by the nature of the chosen algorithm, as opposed to being caused by runtime dynamics.

From Figure 7-26, we can see that increasing values of m causes the $RMS_{perc}(optimum)$ to increase. Initially, when there are more channels than connections (n = 15 channels),

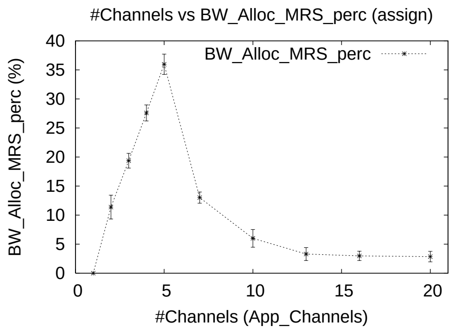


Figure 7-28: Pinning: Channels vs. Bandwidth Allocation (optimum %)

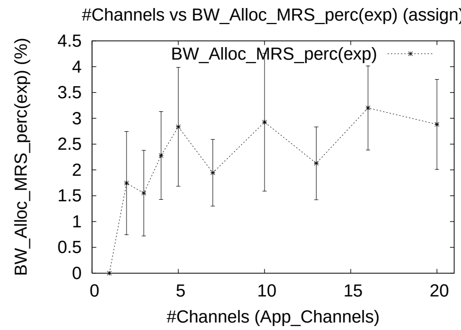


Figure 7-29: Pinning: Channels vs. Bandwidth Allocation (expected %)

then the assigning algorithm does well in obtaining good bandwidth allocation. However, as $m \to n$, the bandwidth allocation gets worse. This is because many channels are assigned to their own connection, and all the connections get approximately the same throughput. Therefore, these channels get the same throughput regardless of their weights. This can be shown with the case of $m = n$, where each channel is assigned its own connection, so all the channels get the same throughput. This, clearly, is not optimal. As $m > n$, the bandwidth allocation metric does not change, since the extra connections are not used.

We can show that these effects are due to the properties of the assignment algorithm rather than the dynamics at runtime by looking at the expected bandwidth assignment metric (Figure 7-27). We can see that the RMS_{perc} is now within 10%, which means that the bandwidth allocation was not too far away from the expected distribution.

We can observe the same reasoning behind the trends as we fix the number of connections m and increase the number of channels n. Figure 7-28 shows the RMS_{perc} as we fix $m = 5$. Initially, as $n < m$, increasing the number of channels n causes the bandwidth allocation to become worse. This is because each channel is pinned to its own connection, so there is no bandwidth allocation occurring aside from the equal distribution obtained by the multiple TCP connections. Since the channels are given weights $w_i = (w_{initial})^{i-1}$ (Section 5.3), if all the channels are given the same bandwidth, then the deviation from the optimal increases as the number of channels increases. Then, as $n > m$, the algorithm is better able to pin the channels to connections in a way to even out the total weight assigned to each connection, which makes bandwidth allocation better (Section 5.4.2).

Again, we can see that these are effects of the pinning algorithm rather than on the runtime behavior by looking at the RMS_{perc} of the expected distribution. Figure 7-29 shows the RMS_{perc} within a few percent, which shows that the experiment does not deviate much from the distribution expected by the algorithm.

Figure 7-30 shows that bandwidth allocation with lower numbers of channels n is not achievable. When $n = 3$, any value of $m > 12$ suffers from lots of reordering, until it flattens off when $m > n$.

In conclusion, we see that the pinning scheduler is able to achieve good bandwidth allo-

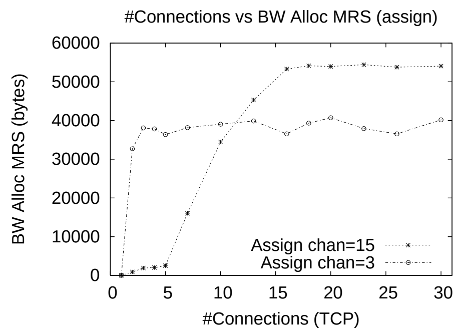


Figure 7-30: Pinning: Connections vs. Bandwidth Allocation (optimum) 2

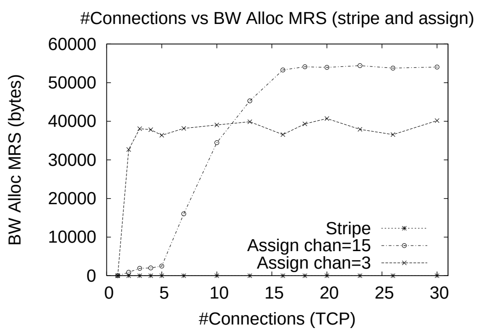


Figure 7-31: Compare: Connections vs. Bandwidth Allocation (optimum)

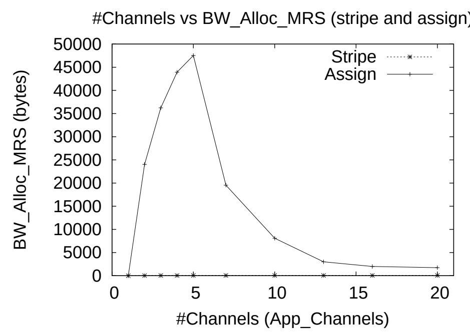


Figure 7-32: Compare: Channels vs. Bandwidth Allocation (optimum)

cation when $n >> m$, but otherwise, is unable to achieve good bandwidth allocation.

7.3.3 Comparison

The evaluations of the bandwidth allocation for each of the schedulers shows the following:

- Striping gives near optimal bandwidth allocation
- The pinning scheduler is able to achieve good bandwidth allocation when $n >> m$, but otherwise, is unable to achieve good bandwidth allocation.

We now compare the two schedulers to show that the striping Scheduler always has better bandwidth allocation than the pinning Scheduler. For the pinning Scheduler to obtain better bandwidth allocation, then the number of channels n must be much larger than the number of connections m ($n >> m$).

As Figure 7-31 shows, striping Scheduler achieves near-perfect bandwidth allocation, while the pinning Scheduler performs poorly when the number of connections m increases. In this case of the number of channels $n = 15$, as $m \rightarrow n$, the bandwidth allocation for the

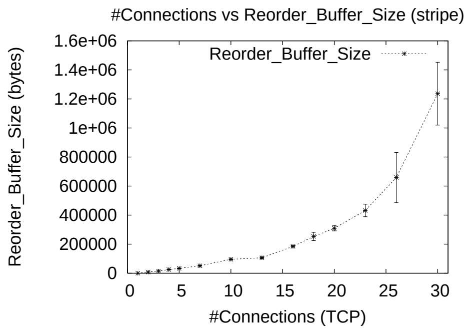


Figure 7-33: Striping: Connections vs. Memory Usage

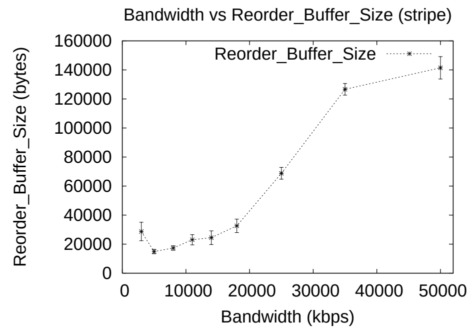


Figure 7-34: Striping: Bandwidth vs. Memory Usage

pinning Scheduler gets worse. Then, as n decreases, we see that bandwidth allocation gets even worse, because $n >> m$ never happens.

Figure 7-32 shows the same dynamics. The striping Scheduler gives good bandwidth allocation and the pinning Scheduler performs better as $n >> m$ where $m = 5$ in this case.

In conclusion, the striping Scheduler always has better bandwidth allocation, being near-perfect. The pinning Scheduler can achieve good bandwidth allocation as $n >> m$, but as $m \to n$, the bandwidth allocation gets worse.

7.4 Results: Memory Usage

The memory usage metric shows us how much memory is required by the schedulers to operate. As explained in Section 7.1.6, we look at the size of the reorder buffer to calculate the memory efficiency of the Schedulers. We show that the striping Scheduler requires more memory as the number of connections increases, as well as when bandwidth, packet loss, or delay increases. On the other hand, the pinning Scheduler has near-constant memory requirements for all conditions. Therefore, systems that are heavily memory constrained should use the pinning Scheduler.

7.4.1 Striping Scheduler

The size of the reorder buffer for the striping Scheduler parallels the reordering delay metric. Reordering delay is caused by more packets being in the reordering buffer, as is the size of the reorder buffer. Therefore, like the trends for reordering delay, the reorder buffer size grows as the number of connections m increases, as well as bandwidth, packet loss, and delay increase.

Figure 7-33 shows the size of the reorder buffer as the number of connections m increases. This parallels the increasing reorder delay in Figure 7-3. Similarly, Figure 7-34 shows rising reorder buffer size for increasing values of bandwidth. The packet loss rate and link delay cause similar effects as the reordering delay, which are shown in Figure 7-6 and Figure 7-5.

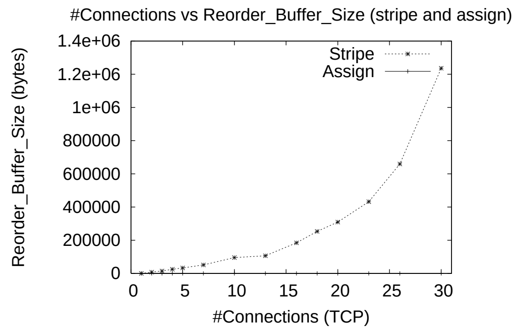


Figure 7-35: Compare: Connections vs. Memory Usage

7.4.2 Pinning Scheduler

Memory size in the Scheduler primarily comes from the reordering buffers, but since TCP already does the reordering for us and there is no reordering in the reorder buffer, memory usage stays very low. Therefore, the pinning Scheduler is very memory-efficient, and is suited for systems with heavy memory constraints.

7.4.3 Comparison

The previous two sections show us how the memory usage for the two Schedulers behave with changing parameters. We find that:

- The striping Scheduler reorder buffer size grows as the number of connections m increases, as well as bandwidth, packet loss, and delay increase.
- The pinning Scheduler is very memory-efficient

We now compare the two to highlight that the pinning Scheduler always uses less memory than the striping Scheduler, so the pinning Scheduler will be preferred for applications with heavy memory constraints.

Figure 7-35 shows that the pinning Scheduler reorder buffer size stays constant at zero, while the striping Scheduler reorder buffer size increases as m increases. Furthermore, we have shown previously that the striping Scheduler reorder buffer size grows as the bandwidth, link delay, and packet loss rate increase. Under changing network conditions, then, the memory size of the striping Scheduler can fluctuate.

In conclusion, we have shown that the pinning Scheduler has better memory size performance than the striping Scheduler, and is well suited for applications with heavy memory constraints, even under changing network conditions.

7.5 Summary: Striping versus Pinning

We have so far individually evaluated the three metrics, sending performance, bandwidth allocation, and memory size. In this section, we will summarize the findings and develop an overall evaluation of the two Schedulers.

We have shown that the pinning Scheduler has better sending performance for high numbers of connections n, but the performance becomes limited as n decreases toward the number of connections m. The striping Scheduler, although with more delay, can achieve comparable sending performance under low packet loss rate conditions, and is not affected by the choice of the number of channels n.

We have also shown that the striping Scheduler has near-perfect bandwidth allocation, and the pinning Scheduler can also achieve good bandwidth allocation if $n >> m$, but as $n \to m$, good bandwidth allocation becomes impossible.

Finally, we have shown that the pinning Scheduler has constant memory usage, regardless of network conditions.

We now evaluate the two Schedulers from an application perspective to find the appropriate Scheduler for an application.

If the application is **not delay dependent**, then the striping Scheduler provides good throughput and good bandwidth allocation, and is versatile to both program behavior in the number of connections that the application keeps open, as well as to changing network characteristics like changing available bandwidth, packet loss rate, and delay. The pinning Scheduler does not achieve as good bandwidth allocation and is not versatile to application behavior.

If the application is heavily **memory constrained**, then the application prefers the pinning Scheduler for its constant memory size under changing network conditions. The striping Scheduler will fluctuate depending on bandwidth, packet loss rate, and delay of the network.

If the application is **delay dependent and not memory constrained**, then the choice depends on the normal characteristic of the network, the normal application behavior, and the variables that are likely to change.

If the network has *low packet loss rate and a low number of expected cross traffic*, then the striping Scheduler is able to obtain good sending performance, while having good bandwidth allocation. Conversely, if the network has *high packet loss rate or high numbers of cross traffic*, then the striping Scheduler is unable to obtain good sending performance, and the pinning Scheduler is preferred.

If the application has *low number of channels* at one time, then the striping Scheduler is preferred, because the pinning Scheduler sending performance and bandwidth allocation both perform poorly. If the application has *high number of channels*, then the pinning Scheduler achieves better sending performance with good bandwidth allocation.

Therefore, if the application can have either stable network characteristics or stable application behavior, then the choice is as stated above. Otherwise, the application chooses the scenarios in which it accepts worse performance.

Chapter 8

Conclusion

In this thesis, we concentrate on the problem of achieving good network utilization and bandwidth allocation for overlay network applications that send multiple flows of data between its machines. We present a new user-level approach with multiple TCP connections, and show that this design leverages the code maintainability of a user-level implementation with the simplicity and efficiency of an implementation that uses TCP.

We then look at the problem of picking a multiplexing strategy to multiplex the multiple application flows onto the multiple TCP connections. This thesis presents and analyzes two multiplexing strategies, striping and pinning. The striping strategy aggregates all the flows and stripes them across the TCP connections according to a deficit round robin algorithm. The pinning strategy pins flows to TCP connections according to a ordered best fit algorithm that is a dual approximation of the analogous multiple processor scheduling problem.

This work goes further to implement and evaluate the two scheduling strategies to determine how the two strategies compare. We find the following:

- If the application is *not delay dependent*, then the application prefers striping Scheduler
- If the application is *heavily memory constrained*, then the application prefers pinning Scheduler
- If the application is *delay dependent and not memory constrained*, then the choice depends on the normal characteristic of the network and the normal application behavior.
  - If the network has *low packet loss rate and a low number of expected cross traffic*, then the striping Scheduler is preferred
  - If the network has *high packet loss rate or high numbers of cross traffic*, then the pinning Scheduler is preferred.
  - If the application has *low number of flows at one time*, then the striping Scheduler is preferred
  - If the application consistently has *high number of flows*, then the pinning Scheduler is preferred

We have shown that a user-level implementation using TCP is both viable and highly attractive, and that both striping and assigning multiplexing schemes are possible choices depending on the network conditions, application constraints, and application behavior.

Bibliography

[1] Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *Journal of ACM*, 34(1):144–162, January 1987.

[2] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, Mark Stemm, and Randy H. Katz. TCP behavior of a busy internet server: Analysis and improvements. In *INFOCOM (1)*, pages 252–262, 1998.

[3] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for internet hosts. In *SIGCOMM*, pages 175–187, 1999.

[4] Hari Balakrishnan and S. Seshan. The congestion manager. *Internet RFC 3124*, 2001.

[5] R. Braden, D. Clark, and S. Shenker. Integrated services in the Internet architecture: an overview. Technical Report 1633, 1994.

[6] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[7] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, pages 1–12, 1989.

[9] Y. Bernet et. al. A framework for differentiated services. *Internet draft*, 1999.

[10] S. Floyd. Connections with Multiple Congested Gateways in Packet-Switched Networks Part 1: One-way Traffic. *Computer Communications Review*, 21(5):30–47, October 1991.

[11] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[12] Sally Floyd and Van Jacobson. Traffic phase effects in packet-switched gateways. *Journal of Internetworking:Practice and Experience*, 3(3):115–156, September, 1992.

[13] Gettys, J. Mux protocol specification, wd-mux-961023. *http://www.w3.org/pub/WWW/Protocols/MUX/WD-mux-961023.html*, 1996.

[14] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[15] D. R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–, 1997.

[16] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe. Modeling TCP throughput: A simple model and its empirical validation. *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314, 1998.

[17] Pragyansmita Paul and S V Raghavan. Survey of qos routing.

[18] J. Postel. Transmission control protocol. *Internet RFC 793*, 1981.

[19] The VINT Project. The ns manual, 2002. http://www.isi.edu/nsnam/ns/ns-documentation.html.

[20] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[21] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM*, pages 231–242, 1995.

[22] Spero, S. Session Control Protocol (SCP). *http://www.w3.org/pub/WWW/Protocols/HTTP-NG/http-ng-scp.html*, 1996.

[23] W. Stevens. Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. *Internet RFC 2001*, 1997.

[24] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.