SOFT QUALITY OF SERVICE VIA TCP CONGESTION SIGNAL DELEGATION ON AGGREGATED FLOWS

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Jason D. Valdez, B.S.

San Marcos, Texas

May 2008

To my soon-to-be-born nephew, may you be happy and healthy.

ACKNOWLEDGEMENTS

Foremost, I would like to thank my advisors Dr. Guirguis, Dr. Hazlewood, and Dr. Drissi of the Texas State University-San Marcos Computer Science Department. Dr. Guirguis has shown remarkable patience in guiding me through this research project. Dr. Hazelwood and Dr. Drissi were invaluable with the logistics of starting and completing this paper. Obviously, without their help this research could not have been completed.

A special thanks to my friends and family who helped and supported me during my years of university study. Your help has meant very much, and pretending not to seem exceedingly bored when I spoke about my research was appreciated.

This manuscript was submitted on March 21, 2008.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

THE FREE AND STUNTS

1.1 Introduction

Some classes of applications such as streaming real-time data, video, and audio require service guarantees in order to function properly Unfortunately, it is not always possible to provide those guarantees on an Internet scale due its inherent "best-effort" nature In particular, there are no built in methods to facilitate the delivery of different services that these applications require They are left with only the normal effort that the Transport protocols get from the underlying network.

Two major architectures exist to provide service guarantees over best-effort networks. The IntServ (Braden et al., 1994) and DiffServ (Blake et al., 1998) service models both have the ability to specify what kind of service is provided to a data flow The major difference being that IntServ utilizes a direct specification on service requirements through a Reservation Protocol, and DiffServ utilizes mostly traffic classification at the edges of networks The disadvantage is that both require changes to network routers in order to provide their services. These two architectures are discussed in section 2.2

This thesis proposes a new method entitled the "Free and Stunts" architecture. The goal of the architecture is to provide "Soft" throughput guarantees utilizing the

Transmission Control Protocol (TCP), at end-points, without any enhanced compliance from network devices. The Free/Stunt architecture takes advantage of a natural grouping of flows through an aggregation point that originate from the same sender A "Free" flow experiences enhanced service by having the ability to "target" a requested throughput rate. The others of the group are called the "Stunts." Stunt flows willingly reduce their bandwidth allocation to allow the Free flow to meet service requirements.

The Free flow's targeting abilities are accomplished by redirecting congestion signals to Stunt flows It is shown that a soft service model is possible by taking advantage of predictable behaviors of TCP. TCP controls the allocation of bandwidth to flows through the use of a Congestion Control Algorithm (Cerf and Kahn, 1974; Jacobson, 1988), which reacts predictably to congestion signals. The Free/Stunt architecture utilizes the Additive-Increase Multiplicative-Decrease (AIMD) phase of this algorithm to affect service guarantees.

During the AIMD phase congestion signals, such as dropped packets, require the flow experiencing them to slow down its rate by halving its congestion window on each congestion signal. When a Free flow experiences a congestion signal it may direct the action to a Stunt flow. With the ability to delegate congestion signals to other flows, the Free flow is able to meet service guarantees as requested by the application.

Deployment Examples:

Internet servers typically serve different forms of media to different clients. By employing this architecture, a server can give a particular flow (e.g. media stream) the freedom to match some requested Target throughput rate, while making other flows, such as bulky file transfers, behave as Stunts The existence of Stunt connections, from the server to other clients, is assumed. It is also assumed that they may be utilized by Free/Stunt. This architecture can also be used by an ISP to provide differentiated service among its clients, by having some behave as the Free connections, while others play the Stunt roles.

Thesis Organization:

Chapter two describes the background information necessary to understand the research. Chapter three will detail related work and how it applies to the new architecture. In chapter four and five the Free and Stunts architecture is detailed. Chapter six details the experimentation method. Finally, chapter seven and eight show the results of experiments in both the simulation and real-world implementation environments

This thesis shows the progression of the architecture through its conception as an abstract mathematical idea, simulation, and implementation as a Linux Congestion Control module. Thorough testing using a simulated environment was undertaken before moving onto implementation, which reinforces the simulation results by showing a remarkable correlation in behavior in a physical network environment. The research leaves no doubt to the validity of Free and Stunts architecture's claim that it can provide Soft service guarantees using only end-point control over the Internet.

CHAPTER 2

BACKGROUND

2.1 TCP's Congestion Control Algorithm

The Transmission Control Protocol offers reliable in-order transmission of data over a compatible network (Cerf and Kahn, 1974; Jacobson, 1988). There are multiple factors to consider for the design and behavior of such a protocol. The efficient and fair distribution of network bandwidth is one of those factors. In order to achieve this goal TCP implements its Congestion Control Algorithm.

Through the use of the Congestion Control Algorithm TCP has the ability to throttle its throughput over a network with the goal of allocating to each flow a "fair" share of the available bandwidth. The term "fair" suggest a roughly equivalent proportion of bandwidth for each flow through the bottleneck link, also referred to as "TCP fairness."

The protocol provides reliable transmission by ensuring that every packet sent is acknowledged (ACK) by the receiver. When a data transmission is received at the destination it is buffered and an acknowledgement is sent back through the network to the sending entity. A "timeout" event occurs when an ACK is not received within a certain amount of time. When multiple duplicate ACK's are received it is assumed that a packet was lost in the network. These definitions are the basis for the Congestion Control behaviors and will be explained in the detail description that follows.

All modern TCP variants support a standard set of Congestion Control actions including Additive-Increase Multiplicative-Decrease (AIMD), slow start, fast-recovery, and fast-retransmit (Kurose and Ross, 2008, Ch 3). TCP handles timeout events by putting itself in a "slow start" phase denoted by a drop in the *cwnd* to a minimum value followed by an exponential increase of the *cwnd* to some *ssthresh* value, or when another congestion event occurs. Timeout events occur when a data packet acknowledgement is not received within a certain expected variable window of time, as defined by the algorithm Fast-recovery and fast-retransmit are together intended to allow the TCP flow to retain bandwidth and also limit the amount of data that is sent more than once. These aspects of Congestion Control work together in order to both efficiently utilize available bandwidth while fairly distribute that resource among competing flows.

The TCP Congestion Control Algorithm maintains a set of variables that define the state of the flow and are also used for control Of those state variables this architecture utilizes the congestion window (*cwnd*), maximum segment size (MSS), slow start threshold (*ssthresh*), and smoothed round trip time ($SRTT$).

The *cwnd* defines a numerical limit of data that can be sent into the network. It is defined in terms of packets. The *cwnd* essentially determines at what rate the flow can transmit data. Its upper limit is governed by the "receive window," which is a value denoting how much buffer space the receiving entity has allocated to the flow The Congestion Algorithm both increases and decreases the *cwnd* in order to control the flow's throughput.

The *ssthresh* is a value, which defines the transcendent state between exponential and linear *cwnd* growth. When the $cwnd < ssthresh$ the congestion windows' growth is exponential.

The *SRTT* is a weighted average of the measured round trip times updated as packet acknowledgements are received. TCP uses the *SRTT* to estimate timeout intervals, while the Free/Stunt Architecture uses it to measure instantaneous data transmission rates.

Knowledge of the TCP Congestion Control Algorithm is required in order to understand where the Free/Stunt architecture operates within the algorithm. For the purposes of experimentation and testing, the TCP NewReno variant of the Congestion Control Algorithm was utilized. Other variations on TCP exist; however, NewReno was selected because of its wide use and the fact that Linux kernels natively support this congestion algorithm. This was a major factor in the implementation as a Linux pluggable congestion module, discussed in section 8. As stated, the choice was one of practicality, and the aspects of NewReno that make it unique do not necessarily make it more compatible with the Free/Stunt architecture than other variants.

2.2 Quality of Service

Quality of Service is usually defined in terms of time and priority. In actuality the defining factor has more to do with the needs of the entity receiving the data being sent. Some applications such as real-time video require that data arrive at the destination within some physically measurable time frame. Other applications may not specify a requirement of more than delivery of data as soon as possible, which the network provides in the form of best-effort transmission. Still, other applications may be able to subsist on very little resources. Quality of Service requirements range across all spectra, but it usually implies increased service over other flows or adherence to real-time requirements.

2.2.1 Soft Quality of Service

Soft QoS models generally refer to service that may not deliver the requested quality at all times. Even so, the application will not totally fail if a specific level of service is not achieved. Soft QoS architectures are usually defined by not directly interacting with hardware to provision resources. Many architectures have been designed that utilize both exploitation of TCP behaviors and entity coordination to provide enhanced service. The Free/Stunt architecture exploits the AIMD behavior of the TCP Congestion Control Algorithm. Congestion Manager, discussed in the literature section 3.1.1, is an example of a model utilizing coordination among many sending and receiving nodes. The common factor is that each method is at the mercy of network congestion and available link capacity.

These architectures cannot force the network to allocate more bandwidth or decrease queuing time at relays than normally possible. They can prioritize their own traffic enablng greater service proportionally compared to other flows in their domaın. The effect is a noticeable increase in quality in the frame of reference, but from the prospective of the entire network service quality is not better than any other traffic sharing that network.

That becomes the defining difference between "Hard" and "Soft" QoS models, as defined here. A more detailed overview of Soft QoS architectures is described in the section 3 Literature Survey.

2.2.2 Hard Quality of Service

Hard QoS modes generally refer to service that must deliver the requested quality at all times. In many cases, the applications will fail if the service is not delivered as required. It is possible to construct a network that provides guaranteed Quality of Service. That is if the network is compliant with a set of standard behavioral specifications with regard to data transmission. Judging by the currently available QoS architectures and research, it would appear that currently the only viable method of ensuring a specific level of QoS is to provision network resources appropriately at routers.

An early attempt at a unified architecture for Internet scale service guarantees was IntServ (Braden et al., 1994). The goal of Integrated Services (IS) is to provide real-time QoS for what was presumed to be the "next generation of traffic." Next generation traffic in the mind of IntServ developers would be composed of telecom applications such as teleconferencing, remote seminars, and other such voice and video data. These ideas are familiar today as video teleconferencing and voice over IP (VoIP) technologies.

The Integrated Services model includes aspects that provide QoS for best-effort, real-time, and "link-sharing" traffic. Standard flows, or packet relaying, with no additional QoS benefits fall in the best-effort category. Real-time applications are those that request a service guarantee with respect to delivery within some physical time, as measured in discrete increments. The term link-sharing encompasses service based on traffic classifications These three categories of service define the IntServ QoS model.

IntServ allocates bandwidth to flows through a Reservation Protocol, which ensures that routers on the path will carry the requested bandwidth with some Quality of Service as defined in the reservation. Senders are required to specify exactly what level of service they require using these Reservations before transmitting and are expected to adhere to their requested requirements as well.

The problem is that IntServ requires compliance with every device on the route, which is a major failing in regard to Internet scale communication. Differentiated Services, a later QoS model, contrast IntServ in that it does not require that every device on a route implement the DiffServ architecture In fact, the DiffServ RFC specifically states, "Sophisticated classification, marking, policing, and shaping operations need only be implemented at network boundaries or hosts" (Blake et al., 1998). DiffServ does not use anything analogous to a Reservation Protocol. The architectural instead relies on traffic classification to determine bandwidth allocation and priority, while still providing Soft reliable service guarantees.

2.3 Discussion

In general TCP and its Congestion Avoidance Algorithm, specifically Additive-Increase Multiplicative-Decrease (AIMD), are celebrated as a means of effectively sharing the network communication resource. Probably, the reasons for the accolades are its simplıcity and computational cost. It is decidedly an elegant solution to the complex problem of sharing the network's bandwidth.

Taking into consideration only the TCP Congestion Avoidance Algorithm it becomes apparent that effective transmıssion of messages over an inter-network of computers ıs a matter of serendipity A TCP flow's ability to achieve a target sending data rate is determined by many factors such as the bottleneck link capacity, the number of flows sharing that same link (i.e network congestıon), the receiver's ability to buffer incomıng packet data, and the sender's abılıty to supply data to be sent. All of these components vary with regard to the computers communicating and the dynamic nature of the network.

The Transmıssion Control Protocol alone has no built-in method of ensuring specıfic levels of Quality of Service. The ımplementation of Hard QoS often requires enhancements to network infrastructure such as routers. It becomes improbable to affect this type of service on the scale of the Internet. Communication on an Internet scale involves routing messages between Autonomous Systems (AS), which are indıvidually owned and operated networks. These AS are owned by dıfferent entities and have theır own QoS, security, and profit concerns effectively ensuring that cooperation on this scale is improbable at the very least. That is one of the reasons that soft QoS architectures such as the Free and Stunts are appealing. They utilize the ınfrastructure that is already in place to affect their service models.

CHAPTER 3

RELATED WORK

3.1 Literature Survey

There are many proposed architectures and methods that aim to provide some level of QoS usıng TCP. Specifically, many of these methods adhere to the End-to-End principle (Saltzer et al., 1984), which states that whenever possible communication protocols should be defined as close as possible to the end points. In many instances this varies from the communicating nodes to the edge of the network hosting those nodes.

The following architectures were studied prior to the start of this research. They fall in the same domain of Soft Quality of Service as the Free and Stunts architecture. They are described in the following sections. At the end of this chapter is a discussion of how these concepts relate to this research.

3.1.1 Congestion Manager

The Congestion Manager (CM), described in "An Integrated Congestion Management Architecture for Internet Hosts" (Balakrishnan et al., 1999), is an end-system framework designed to allow applications to adjust to network congestion. The CM allows applications to utilize it as a source of congestion information on the network.

Applications that utilize the CM's API can both retrieve the parameters it monitors as well as send information to the CM such as scheduled data transmissions.

It is important to note that the Congestion Manager does not buffer data. The design allows it to manage bandwidth allocation across the scope of all senders, but the developers purposefully decided to let the senders adjust their data rates as appropriate This is a consequence of the end-point design decision. The sendıng nodes are in control of when and how much data to send within the limits imposed by the overarching framework.

The end goal of the system is to allow more efficiently utilization of the available network communication resources. It provides a more tangible, by use of its API, Quality of Service while still allowing the actual data rate managed at the end-points.

3.1.2 Administrative Policies

In "Managing Soft QoS Requirements in Distributed Systems" (Molenkamp et al., 2000), a soft Quality of Service architecture is developed focusing on requırements for multimedia applications. This architecture seeks to have a very broad oversight and influence in the system

The framework is based on administrative policies governed by parameters such as network congestion, CPU, and memory usage. A "QoS Host Manager" uses the rules defined by policy to take corrective action at the application level. It alleviates behavior that adversely affects servıce to an entity ın the system by instructing "Component Managers." Those managers interact with "Instrumented Processes" to affect the corrective actıons.

Above all entities in their framework is a "QoS Domain Manager." This unit is essentially a conductor of sorts that can issue suggestions across system boundaries such as physical computers. Its actions are based on policy rules that guide its behavior towards other entities regarding their resource utilization. For example, this equates to requesting that a sending node slow applications processing in order to allow another node more bandwidth over the network.

3.1.3 Coordination Protocol

In "An Open Architecture for Transport-level Protocol Coordination in Distributed Multimedia Applications" (Ott and Mayer-Patel, 2007), the authors propose a Coordination Protocol (CP) aimed at enhancing "cluster-to-cluster" (C-to-C) communication between autonomous systems. The term cluster-to-cluster is a unique class of distributed multimedia applications. It is defined by groups of nodes that communicate over a network with a separate group of nodes. These two groups share a common routing path and aggregation point for the combined flows.

At the heart of their framework is the "Coordination Protocol." The goals of the CP are to inform C-to-C endpoints of the available bandwidth, setup an infrastructure for state information exchange, and allow coordination at the endpoints rather than the aggregation point.

To accomplish this task the CP inserts information into a new header that is positioned between the IP and Transport headers. Because CP information is gathered largely at the aggregation points, the use of this header provides an effective means of communicating with end points.

3.1.4 Elastic Tunnels

The Elastic Tunnel framework, described in "Providing Soft Bandwidth Guarantees Using Elastic TCP-based Tunnels" (Guirguis et al., 2004), is a novel approach to acquiring a fixed target of bandwidth between autonomous systems. It provides this effect by exploiting a property of the TCP congestion control behavior that is intended to provide for equal dispersion of network bandwidth.

It acquires a target bandwidth by opening and closing a group of connections. This method works as all flows through a bottleneck will attempt to acquire a "fair" share of the bandwidth resource no matter where the flows originate. This essentially means that the addition of flows between specific endpoints will increase their aggregate bandwidth through a route.

With the ability to add and subtract flows this method can effectively acquire and release bandwidth. It is described as "soft" because its growth is proportional to the number of flows across the link bottleneck. It is limited by the overall utilization of the network bottleneck.

3.2 Discussion

On reviewing related work, one finds that many of the approaches to Soft Quality of Service architectures rely on flow aggregation and cooperation to accomplish their goals. Those that are developed as overarching frameworks still rely on cooperation amongst senders and receivers to share resources effectively In the absence of any ability to force allocation of resources, as found in Hard QoS implementations, this seems to be the only viable option.

The Free/Stunt architecture shares many commonalities with the architectures described in this chapter. Like the Congestion Manager, cooperation at the end points is utilized to provide service. Much like Elastic Tunnels and the Coordination Protocol, it makes use of the concept of aggregated flows. Finally, like Administrative Policies, the Free/Stunt architecture has the ability to throttle a sender's throughput in favor of another.

However, the Free/Stunt architecture has a far smaller footprint. It also does not require sending applications be coded specifically to work with it, a characteristic of both the Congestion Manager and Administrative Policies architectures. Taken together, these aspects of the Free/Stunt architecture give it the ability to meet Soft QoS requirements while having a relatively simple computational model and acting only at the sending end-point.

CHAPTER 4

THE FREE/STUNT ARCHITECTURE

4.1 The Free/Stunt Service Model

This architecture is composed of a group of TCP flows aggregated together and treated as a single entity. A "Free" flow is defined as the component of the group that is given a target throughput request. The "Stunt" flows make up the remainder of the group that are not assigned target throughput request. These are the two entities that compose the Free/Stunt architecture.

The term "delegation" is used to refer to the assignment of a congestion signal from one flow to another flow. In order to carry out the delegation of congestion events, "distribution" of the congestion signals occurs. The Free/Stunt architecture uses both of these actions to meet Quality of Service requirements.

The motivation comes from the realization that a delegation of loss from the Free TCP flow to the aggregated Stunt flows will both allow the Free flow to achieve a desired sending throughput as well as not violate TCP fairness, defined in section 2.1. Meeting a designated throughput is achieved by carefully controlling when the Free flow reacts to a packet loss signal versus when it delegates that signal.

TCP fairness is defined globally across the group of Free and Stunt flows, as they are abstractly considered a single entity. As a rule, the group of Free plus Stunts should not together be more aggressive than an equivalent number of ordinary TCP flows when increasing their throughput.

4.2 Architecture Overview

The service model in section 4.1 describes what the architecture does to affect the service requirements. The feedback control loop in figure 4.1 depicts the architecture. It shows where each of the components described in this chapter fits into the Free and Stunts as a whole.

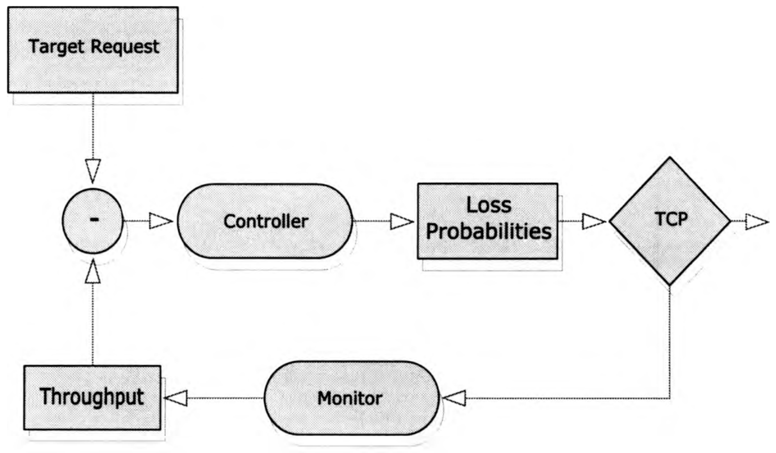


Fig. 4.1: *Architecture feedback control loop.*

Figure 4.1 represents the basic construct underlying the Free and the Stunts architecture. First, a Target request is subtracted from a monitored throughput measurement to produce an "error." This error is the amount that the throughput is off from the requested Target. From there the error is fed into a "Controller," which produces a loss probability that, in turn, is fed into the TCP Congestion Control Algorithm. At that point the probability is used to determine if a congestion signal (e g packet loss) is delegated to the Stunt flows or accepted at the Free flow. This process is repeated for each detected congestion signal.

4.3 Delegation of Congestion Signals

TCP sending side flows experience two types of congestion events These events are duplicate acknowledgements and timeouts. Timeouts are not distributed to the Stunt flows for several reasons having to do mostly with complexıty and rareness of that particular event. Contrary to the rareness of the timeout event, duplicate acknowledgements are far more common. TCP congestion control is discussed ın the background section 2.1.

When the Free flow detects congestion due to a duplicate acknowledgement it is required by TCP Congestion Control to reduce its throughput rate. It does this by halving its congestion window. In order to delegate this loss the Free flow can impose an equal or greater amount of congestion window reductions across the group of Stunt flows. After havıng the Stunt flows reduce theır rates, the Free flow can continue sending without having to react to that particular congestion event. This is how the Free/Stunt archıtecture functıons and the means by which it provides a servıce model softly ensurıng a specıfic throughput.

Delegation of loss during the AIMD phase of TCP Congestion Control Algorithm ıs not too complex The loss from a Free flow to the Stunt flows can be accomplished easily due to the fact that the TCP flow undergoes a relatively linear growth during the congestion avoidance phase. Simply, a connection with linear grown that is sending at a data rate x can distribute its loss to several Stunt flows sending at lower data rates y_i The resultant effect is that the Stunt flow's combined loss is greater-than or equal to the necessary throughput drop at the Free flow

That works because the Stunt flows combined together to send the same amount of data in the short term and experience the same or more loss comparatively. The "extra loss" is due to inefficiencies in the distributing method, discussed in section 7.7 On average there is always a bit of extra loss that is distributed to the Stunt connections. The loss distribution method is discussed in the sections on the Round-Robin Distribution Algorithm (4.3.2) and Controllers (4.4).

Two goals exist when implementing a method of congestion signals delegate. The first is that the method must retain the natural TCP fairness of the standard Congestion Control algorithm. This implies that the aggregated Free and Stunts flows should not gain more bandwidth than they would have had without delegation of congestion signals. The second aspect is implied in the first. The method is implemented in such a way as to not affect the overall network congestion in an abnormal manner.

4.3.1 Stunt-to-Free Delegation

The upward progression of the Free flow's data rate is determined by the Round-Trip Time and the linear increase in the congestion window during AIMD. The Free flow can delegate loss until the requested throughput rate is achieved. Once the target data rate is achieved sustaining that rate without staying over or under is a matter of loss probability, which is totally governed by network congestion or possibly any Active Queue Management scheme at the bottleneck.

Recall that the Free/Stunt service model is driven on TCP congestion events. Actions are taken only when a packet loss is detected. In order to maintain the requested target rate the Free flow reacts to packet loss in the usual manner while above the Adjusted Target rate discussed in section 4.4.2. This action allows it to oscillate about the Target rate with an average rate of the requested target.

In times of heavy congestion the Free flow will have a higher probability of loss while above the target. In turn, in times of relatively low congestion the Free flow will have a lower probability of loss. In order to add additional loss probability while above the target the Stunt flows can attempt to delegate their loss to the Free flow when it has surpassed its requested target rate.

The Free flow is better able to fix a target by accepting "reverse" delegation of packet loss events. Experimentation has shown that with this method enabled the Free flow is better able to meet the requested Target. The predictability lends itself to the use of the $1.3\bar{3}$ target used for matching, which is defined in section 4.4 2. Of course, all the rules of delegation apply to the Reverse Delegation as well.

4.3.2 Round-Robin Distribution Algorithm

In order to distribute the loss from the Free flow to the Stunts a Round-Robin algorithm was chosen for experimentation. The choice was based on one key assumption: the data rates of each Stunt should be about the same throughout the life of the connections. A Round-Robin approach seemed to be the most straight forward and efficient way to distribute the loss across the group.

Figure 4.2 depicts a single distribution of loss from the Free flow to the com-

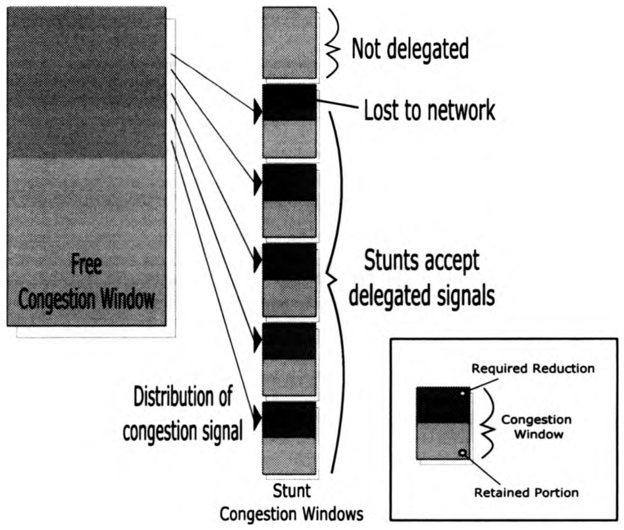


Fig. 4.2: *An overview of the congestion signal distribution method.*

bined Stunt flows. The windows are divided into loss portions and retained portions. The algorithm has to first ensure that there is enough combined congestion window across the group of Stunts to enable a distribution of loss. After determining that there is a suitable amount of window, it begins distributing AMID losses to the Stunt connections by cutting their congestion windows to half their instantaneous values, depicted as "required reduction" in figure 4.2. This effectively reduces the Stunts sending rate by half.

Of course, due to the lack of optimization, there exists a bit of extra window on the last Stunt that experiences a distribution event, shown on figure 4.2 as "lost to network." This is bandwidth essentially given up to the network for use by all the flows. The loss is diminished over time as the Free and Stunt flows with the cross-traffic distribute their shares of the bottleneck capacity fairly. However, constant delegation signals results in an average extra loss to the Free and Stunts as a whole, see section 7.7 for details.

4.3.3 The Free and Stunts do not utilize Slow Start

Slow start, discussed in the background section 2.1, delegation is slightly more complex than delegation in the AIMD phase of Congestion Control, which was one of the reasons that it was not implemented. Determining exactly how much the combined throughput of the Stunts should be slowed can be done through the normal delegation method. However, the problem with slow start delegation is in determining how fast the Stunt flows should be allowed to regain throughput. This determination requires guessing about future dynamics of the network congestion and–it was thought–too much overhead in general. Note, that this discussion is in regard to the actual slow start phase and not the complete timeout event.

Above all else, the reasoning for the decision to only utilize the AIMD phase for loss delegation has to do with the differences inherent in the two types of packet loss. Firstly, flows are intended to exist within the AIMD state while they attempt to send data through the network. This implies that this phase is longer than other phase of Congestion Control. Secondly, slow start is actually a type of disaster recovery option. The assumption is that network congestion is so acute that the flow must lower its data rate to a minimum value, and from that point start probing for available bandwidth. The Free/Stunt service model makes the same assumption and unconditionally allows flows to enter and leave the slow start phase when necessary.

4.4 Controllers

Two controller types were tested. Both have their strengths and weaknesses. Either can function as efficiently, with regard to meetıng the target, as the other given the right conditions. In order to understand the controllers better this section will begin by describing the method used to calculate the throughput rate of TCP. The two controllers are described in the following sections 4.4.3 for the On/Off controller and 4.4.4 for the PI controller.

4.4.1 Measuring TCP Data Rate

It is known from (Mathis et al., 1997) that the average flow rate can be described by using equation 4.1 where MSS is the Maximum Segment Size, RTT is the round trip time, C is a constant collected during the derivation of the formula, and p is the probability of packet loss for the flow being examined.

$$Throughput = \frac{C \times MSS}{RTT \times \sqrt{p}} \tag{4.1}$$

Equation 4.1 shows that the loss probability has a great affect on the data rate of a TCP connection. Loss probability is one of the factors that define how well the architecture can meet service requirements. A higher probability of loss means that the opportunity to shape the Free flow using delegation of congestion signals occurs more often. Notice, as well, that a loss probability would also result in a lower throughput for normal flows like the Stunts.

Equation 4.1 is valid for the macroscopic measurements where the rate is essentially flat The equation does calculate the data rate with accuracy, but it does not provide precise enough values within the time-domain necessary to implement the Free/Stunt architecture.

$$\textit{Instantaneous Rate} = \frac{cwnd \times MSS}{RTT} \tag{4.2}$$

Equation 4.2 is contrasted with equation 4.1 in that it is both less computationally expensive and a measurement of instantaneous rate. It is the instantaneous rate that is used for the purposes of testing and implementation

4.4.2 Target Rate Adjustments

The controllers use the instantaneous calculated data rate at the source to determine when to accept or delegate packet loss. If the loss is accepted at the free-flow then the data rate drops roughly by half. As mentioned before, this repeated behavior results is a actual data rate that is about the average of the data rate values before and after a loss, which we can calculate (Kurose and Ross, 2008, Ch 3).

$$\textit{Average Data Rate} = \frac{W + \frac{W}{2}}{2} \tag{4.3}$$

Equation 4 3 resolves to $0.75 \times W$ where W is the data rate before a loss. All things being the same, this would seem to be a valid average throughput measurement with the exception that the data rate drops as a result of packet loss would be slightly larger at a higher rate. It is necessary to find the adjustment on the target since this average throughput will be used to match the target requirements.

$$W = \frac{1.3\bar{3}W + \frac{1.3\bar{3} \times W}{2}}{2} \tag{4.4}$$

Luckily, the solution can be found in the original equation 4.3. One only needs to determine the constant factor that will result in an average data rate that is equal to the desired target. As can be seen from equation 4.4, that factor is $\frac{4}{3}$.

Therefore, the simulation target is set to $1.3\bar{3} \times T$. This assumption works in the simulated world, but may not be the best target adjustment with the Linux implementation. There is a need for an adjustment, but on a real network the imperfections mean that the value needed is most likely not the one list here.

4.4.3 On/Off Controller

The On/Off controller functions as a switch that indicates when a loss should be taken as apposed to when a loss should be distributed. When the calculated data rate at the source node is greater than the target rate the probability of a loss distribution is 1 (100%). Conversely, when the data rate is less-than the target rate the probability of a loss distribution is 0. This equates to always delegating loss when below the Adjusted Target rate and always accepting the loss when above the Adjusted Target rate. Refer to section 4.2 for the architectural overview describing where the controllers function.

$$g_i = \begin{cases} 0 & x_i < T_i \\ 1 & x_i \geq T_i \end{cases} \tag{4.5}$$

The general controller can be described in equation 4.5 where x_i is the calculated throughput, T_i is requested target, and g_i is the probability that a congestion signal will be delegated.

4.4.4 Proportional-Integral Controller

The Proportional Integral controller can be thought of as the next best thing to the On/Off switch. This controller uses the calculated data rate error from the target (proportion) along with a summation of the previous errors (integration). Its parameters are the Target T_i, calculated data rate sx_i, and the K constant. Refer to section 4.2 for the architectural overview describing where the controllers function.

$$g_i = g_{i-1} + K \times (sx_i - 1.3\bar{3}T_i) \tag{4.6}$$

A few of the components of equation 4.6 need explaining. The calculated throughput sx_i is the instantaneous throughput measurement from equation 4.2 and smoothed by equation 7.1, updated at each acknowledgement received. The K constant is a value that is used to control the aggressiveness of the integration. The controller relates the error from the target (in bytes) to the probability of loss. The error oscillates around the target. Therefore, the K constant will be a very small value.

For the purposes of loss delegation the PI controller (g_i) is defined between $[-1, 1]$. While the actual position in that range is noted as negative or positive, whether the value is positive or negative denotes different behaviors. When the PI controller is in $[-1, 0)$ its absolute value represents the probability of a loss delegation. Alternatively, while the PI controller is in $[0, 1]$ the flow is allowed to behave normally. The value is always interpreted as a probability.

The PI control is only updated during loss events to limit its exposure to irrelevant input. The reasoning behind this decision is evident when considering that the Free/Stunt service model can only take action when a packet loss event occurs.

CHAPTER 5

MATHEMATICAL DERIVATION

5.1 The Nonlinear Fluid Model

The original concept testing for the Free/Stunt architecture is based on a mathematical model of the Transmission Control Protocol's AIMD congestion control behavior. The model is presented here in much the original form as presented in "Liberatıng TCP: The Free and the Stunts" (Valdez and Guirguis, 2008).

5.1.1 Model Derivations

The nonlinear fluid model, similar to those proposed in (Hollot et al., 2001; Kelly, 2001; Low et al., 2002; Shenker, 1990), Is utilized to capture the performance of m TCP flows traversing a bottleneck of capacity C, where m is equal to $(1 + s + n)$ as depicted in Figure 6.1.

The round trip time $r_i(t)$ at time t for connection i is equal to the round-trip propagatıon delay D_i between the sender and the receiver for connection i, plus the queuing delay at the bottleneck router. Thus $r_i(t)$ can be expressed by:

$$r_i(t) \quad = \quad D_i + \frac{b(t)}{C} \tag{5.1}$$

where $b(t)$ is the backlog buffer size at time t at the bottleneck router. The propagation delay is denoted from sender i to the bottleneck by $D_{s_i b}$, which is a fraction α_i of the total propagation delay.

$$D_{s_i b} = \alpha_i D_i \tag{5.2}$$

The backlog buffer $b(t)$ evolves according to the equation:

$$\dot{b}(t) = \sum_{i=1}^{m} x_i(t - D_{s_i b}) - C \tag{5.3}$$

which is equal to the input rate $x_i(.)$ from the m connections minus the output link rate. Notice that the input rates are delayed by the propagation delay from the senders to the bottleneck $D_{s_i b}$.

It is assumed that RED (Floyd and Jacobson, 1993) is employed at the bottleneck link Thus, the congestion loss probability $p_c(t)$ is given by:

$$p_c(t) = \begin{cases} 0 & v(t) \leq B_{min} \\ \sigma(v(t) - \varsigma) & B_{min} < v(t) < B_{max} \\ 1 & v(t) \geq B_{max} \end{cases} \tag{5.4}$$

where σ and ς are the RED parameters given by $\frac{P_{max}}{B_{max} - B_{min}}$ and B_{min}, respectively, and $v(t)$ is the average queue size, which evolves according to the equation:

$$\dot{v}(t) = -\beta C(v(t) - b(t)), \qquad 0 < \beta < 1 \tag{5.5}$$

Notice that in the above relationship, C is multiplied by β since RED updates the average queue length at every packet arrival, whereas our model is a fluid model (Hollot et al., 2001; Low et al., 2002).

The loss delegation between the Free and the Stunts causes them to pick up different congestion signals than those set by RED. In particular, the Free connection, upon delegating $g(t)$ of its congestion signals, would pick up:

$$q(t) = p_c(t) - g(t) \tag{5.6}$$

Each Stunt connection would pick up:

$$q(t) = p_c(t) + \frac{g(t)}{s} \tag{5.7}$$

The normal cross-traffic are not affected and will simply pick up

$$q(t) = p_c(t) \tag{5.8}$$

The throughput of TCP, $x_i(t)$ is given by

$$x_i(t) = \frac{w_i(t)}{r_i(t)} \tag{5.9}$$

where $w_i(t)$ is the size of the TCP congestion window for sender i .

According to the TCP Additive-Increase Multiplicative-Decrease (AIMD) rule, the dynamics of TCP throughput for each of the m connections can be described by the following differential equations:

$$\begin{aligned} \dot{x}_i(t) &= \frac{x_i(t - r_i(t))}{r_i^2(t)x_i(t)}(1 - q(t - D_{bs_i}(t))) - \frac{x_i(t)x_i(t - r_i(t))}{2}(q(t - D_{bs_i}(t))) \\ i &= 1, 2, .., m \end{aligned} \tag{5.10}$$

where $q(.)$ is the congestion signals observed by each connection based on its type. The first term represents the additive increase rule, whereas the second term represents the multiplicative decrease rule. Both sides are multiplied by the rate of the acknowledgments coming back due to the last window of packets $x_i(t - r_i(t))$. In the above equations, the time delay from the bottleneck to sender i, passing through the receiver i, is given by

$$D_{bs_i}(t) = r_i(t) - D_{s_ib} \tag{5.11}$$

The model above makes the following assumptions: (1) It ignores the effect of slow-start and timeout mechanisms of TCP, since our main focus is on the AIMD. (2) The delegation of some losses can be distributed in a linear fashion among the Stunts (as indicated in Equation 5.7). In general, this does not hold except for small value of losses, since the throughput is inversely proportional to the square-root of the loss probability. Despite these assumptions, however, the model above still captures the main dynamics as illustrate below.

5.1.2 Numerical Results

The model above is instantiated with specific parameters and solved iteratively. It is assumed that there is 1 Free connection, 4 Stunts and 15 cross-traffic, for a total of 20 connections. The bottleneck has a capacity 2000 packets/sec. The RTT for each connection is chosen at random around 100 msec.

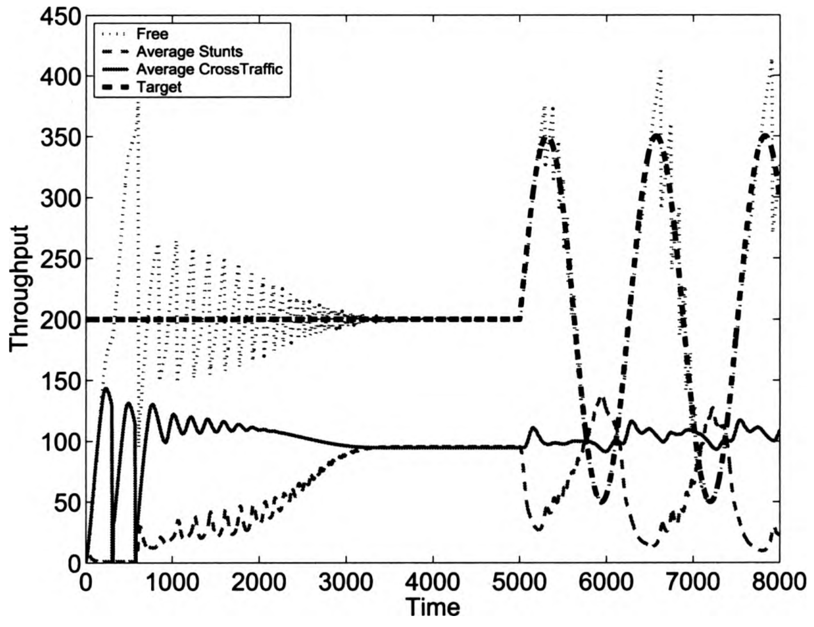


Fig. 5.1: *Numerical Results.*

Figure 5.1 illustrates the performance of the Free connection in matching a target trace that starts with constant throughput at 200 packets/sec and then follows a sin wave. The figure also shows the average throughput across the stunts as well as the average throughput across the cross-traffic connections. One can observe how the Stunt connections make room for the Free connection to match the target throughput.

Notice also, how little the normal cross-traffic is affected, except for the initial startup time (the first 3 seconds) where the whole system is still in a transient behavior. One can also see the impact of reverse delegation around time 6000. Since the target throughput drops below the fair-share (100 packets/sec), the Stunts can delegate congestion signals to the Free connection and thus they are able to increase their throughput a bit above their fair-share.

CHAPTER 6

EXPERIMENT DESIGN AND METHODOLOGY

6.1 Discrete Event Simulation with ns-2

The Network Simulator or ns-2 (E. Amir, 2007) is an open source discrete event simulator popularly used for academic research. It was necessary to change the network simulator's underlying code in order to utilize it for the purposes of this research. The ns-2 simulator is designed to adhere to standard behavior. In that way it implements a wide range of protocols and can simulate differing types of data transmission over any number of defined networks as created using the TCL scripting language.

As mentioned in section 4.3, the Free/Stunt architecture works on Congestion Control events such as packet loss. That requires interflow cooperation; this is a function that is not a behavior in any standard TCP implementation.

The ns simulator comes with a TCP class, which carries out the necessary functions of sending, receiving, and Congestion Control among others. The majority of changes to implement the necessary functionality are seated in the TCP class. Rather than edit this class, it was decided that an additional class should be created based of off the original TCP NewReno implementation provided with ns.

All of the ns-2 TCP implementations inherit from the TCP Tahoe variant class, which contains the majority of the congestion control functionality. The TCP

Tahoe and NewReno classes were cloned and then added to the build process to be incorporated into the simulator.

They were tested afterward by running a group of control simulation experiments using the standard TCP NewReno. The cloned classes were then simulated under the same topology and then the output of the experiments was examined for differences. Because the simulations utilized the same seed and the classes were identical the simulation output was identical as well. This ensured that the cloning of the classes was successful.

After ensuring that the class was functioning correctly the Free/Stunt architecture was implemented. The same simulations were run over the same topology with the Free/Stunt delegation turned off. The simulation output was exactly the same as in the previous two experiments. This ensured that there were no side-effects present for the Free/Stunt architectural additions to the class.

6.2 Data Collection and Use

The ns simulations were created using the TCL scripting language. The script allows for the creation of the network topology, the addition of data flows, and the parameterization that define the behavior. It is also necessary to specify what data to collect during the simulation and where it should be saved. See Appendix A for the scripts used for the collection and processing of simulation data.

Data collection in a discrete simulation involves writing system state information to a file at moments during simulation. State variables were measured with a sample rate defined for the specific simulation. Information such as the throughput rates, queue sizes, and a running report of congestion events were collected. Table 6.1 provides a summary of the metrics collected.

In addition, graphical output was produced to better examine the qualitative behavior of the simulated flows Most of the data plots in this paper were created automatically as part of the finalization steps of each simulation run These can be found were the experiment results are given and analyzed, chapter 7.

Table 6.1: *Collected metrics summarization table.*

Monitored	**Measurement Delta**	**Definition or Use**
Sender rate	usually measured every 0.1 sec	Sender instantaneous calculated throughput measurement
Sink rate	measured every 1.0 sec	Receiver throughput measurement
Sink error	Calculated from simulation output for 1.0 sec intervals	Measurement of the distance from the Target, these are the variances when computing the weighted metric (section 6.3).
Sender error	measured using aggregated sender rate in 1.0 second intervals	Measurement of the distance from the Target, these are the variances when computing the weighted metric (section 6.3).
RED average	weighted average of past values	parameters adjusted in an attempt to keep a level value
RED size	usually measured every 0.1 sec	Used to compare the congestion levels to the behavior of the Free flow
per-Stunt rate	measured every 1.0 sec	not used for singular purpose
Stunt average	updated every 1.0 sec using per-Stunt values	Used for comparison to Free throughput and as a aggregate value in calculations.

6.3 The Performance Metrics

Metrics exist that can be used to measure the health and performance of a flow. Amongst others, these include the data rate, jitter, and timeouts per some time delta. These types of measurements are important, but what is important for the Free/Stunt architecture is how often and to what degree the Free flow is meetıng the requested target.

The effectiveness of matching the requested target throughput to the actual data rate is measured with a weighted variant of the standard "sum-of-squared errors" method. The "sum-of-squared errors" does not work precisely enough due to the oscillation of the Free flow's data rate around the target. There is no differentiation between the cases where the achieved throughput is above the target versus the case where the achieved throughput is below the target.

In order to get some bearing on this matter the positive variance V^+ is defined as:

$$V^+ = \frac{\sum(x_i - T_i)^2}{C^+} \quad \forall \; x_i > T_i \tag{6.1}$$

where C^+ is the number of sample points that the achieved throughput is above target. Conversely, the negative variance V^- is defined to be:

$$V^- = \frac{\sum(x_i - T_i)^2}{C^-} \quad \forall \; x_i < T_i \tag{6 2}$$

where C^- is the number of sample points that the achieved throughput is below target.

To capture the overall performance a weighted variance is used It is defined by:

$$V^* = \delta \times \frac{\sum(x_i - T_i)^2}{C^+ + C^-} \tag{6.3}$$

where δ is a ratio that is given by:

$$\delta = \frac{max(V^+, V^-)}{min(V^+, V^-)} \tag{6.4}$$

The δ is always greater than or equal to 1. Ideally, if a perfect oscillation of the Free flow occurs then δ would be 1. A larger value of δ indicates a bias in the matching, either above or below the target, and this would increase the weighted variance in turn. Examining both the V^+ and V^- metrics would indicate in which direction the bias existed.

The simulation runs are generally within the range of a few minutes. This range gives plenty of time to collect data and make variations to the requested target. Also, because of the small simulation time the metrics are computed over the entire simulation run. In an actual implementation these metrics would require a windowing or averaging with respect to time in order to be useful.

6.4 Experimentation Topology

Figure 6.1 depicts the general topology of the simulated network. It is composed of a single bottleneck link that is traversed by the Free, Stunts, and cross-traffic connections. It is assumes all connections have an infinite supply of data to transmit. However, to study the impact of differing dynamics that arise in practice, some of the cross-traffic connections are turned on and off at random time intervals during simulation. The number of Stunt flows is varied to demonstrate and examine the behavior of the architecture under changing congestion levels.

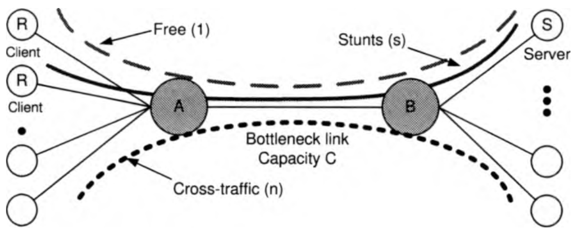


Fig. 6.1: *Topology used for experimentation.*

The bottleneck link replay is configured with RED (Floyd and Jacobson, 1993). The queue size at the bottleneck link is chosen to be $\frac{RTT \times C}{\sqrt{m}}$ as advocated in (Appenzeller et al., 2004), where C is the bottleneck link capacity and m is the total number of connections traversing the bottleneck. The RED parameter B_{min} is set to 0.25 the size of the queue. This results in a rule-of-thumb distance between B_{min} and B_{max} being three times B_{min}. Other parameters were chosen to encourage the stability of the average queue size. The appendix A has examples of the simulation scripts used for construction and simulation of the network.

It is important to note that on fully utilized simulation networks, those composed of only long lived TCP flows, the justification for dividing by the square root of m breaks down due to synchronization of flows (Appenzeller et al., 2004; Zhang et al., 1991), where synchronization is defined as flow loss happening in concert. However, when randomized cross-traffic flows are added, the premise is regained for the reduction in the buffer size due to the inability of flows to synchronize packet loss events.

CHAPTER 7

RESULTS AND ANALYSIS

7.1 PI Controller Parameters

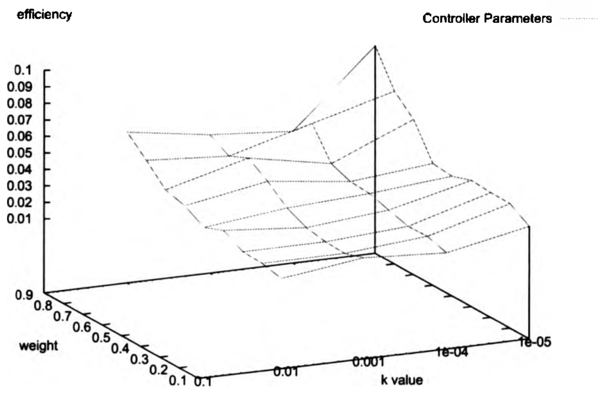


Fig. 7.1: *Testing for PI Controller Parameters*

In order to determine general parameters for the Proportional Integral Controller, discussed in section 4.4.4, experiments were run while varying the throughput smoothing $Weight$ and the PI's K value. The $Weight$ is used in a simple weighted average of the instantaneous calculated data rate, which is measured at each packet ACK (equation 7.1). This smoothed-throughput estimation is used with the PI Controller's "error" computation. The K value is the parameter of the PI Controller used to control how aggressive the integration will be. A higher K would result in a greater proportion of the error being integrated into the probability g_i (see section 4.4.4).

$$sx_i = (Weight) \times sx_i + (1 - Weight) \times x_i \tag{7.1}$$

The efficiency metrics in section 6.3 were used to gauge the relative effectiveness across simulation runs. The metric's values are used relative to each other with the goal of finding a minimum. Figure 7.1 combines the output with three degrees of specificity. From the data the general parameters of K and $Weight$ were chosen to be 0.001 and 0.333 respectively.

7.2 Ideal matching examined

The Free/Stunt architecture exploits aspects of congestion control and network dynamics to meet target requirements. Firstly, bandwidth must exist to support both the set of Free and Stunt flows along with the cross-traffic through the bottleneck link. That is there should be enough bandwidth to achieve the desired Free flow throughput, given the limitations due to increasing Stunts. Secondly, the flows must be limited by the Additive-Increase Multiplicative-Decrease (AIMD) Congestion Control behavior. Both timeout and TCP receive-window throttling will not allow the architecture to delegate congestion signals.

In order to get a general feel for the tools and method a haphazard-play with the parameters was done to observe the results. As apposed to the systematic effort taken with the formal experimentation, these tests took aspects like congestion,

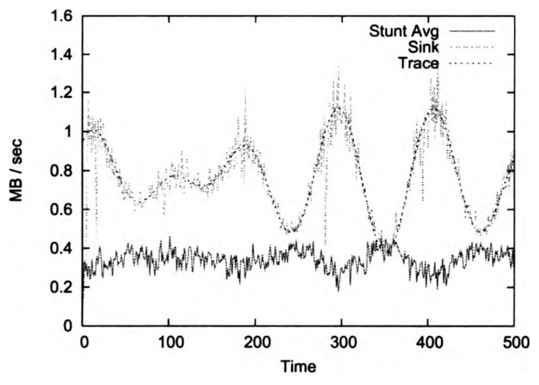


Fig. 7.2: *Tight matching to the requested target.*

topology construction, and cross-traffic dynamics to extremes in order to better grasp the possible ranges of the Free/Stunt architecture.

Figure 7.2 is a sample of a simulation in which the bottleneck link with a RED queuing discipline is configured to have a small queue size, while still allow the Random Early Detection mechanism to provide congestion signals. The networks were typically configured with a 50Mb Free/Stunt link, 80Mb bottleneck, and multiple varying numbers of 10Mb cross traffic links. For the experiment in figure 7.2 the RED queue parameters were a B_{min} of 74, B_{max} of 894, and a queue size 1100 (in packets). The fact that bandwidth contention is not an issue (timeouts) and that RED is providing plenty of opportunities to delegate loss (AIMD) results is a fairly good matching of the target.

7.3 Qualitative Analysis of Target Throughput

In order to assess the target matching behavior of the Free/Stunt architecture a standard sets of trace files were utilized. The trace files map target requests to time

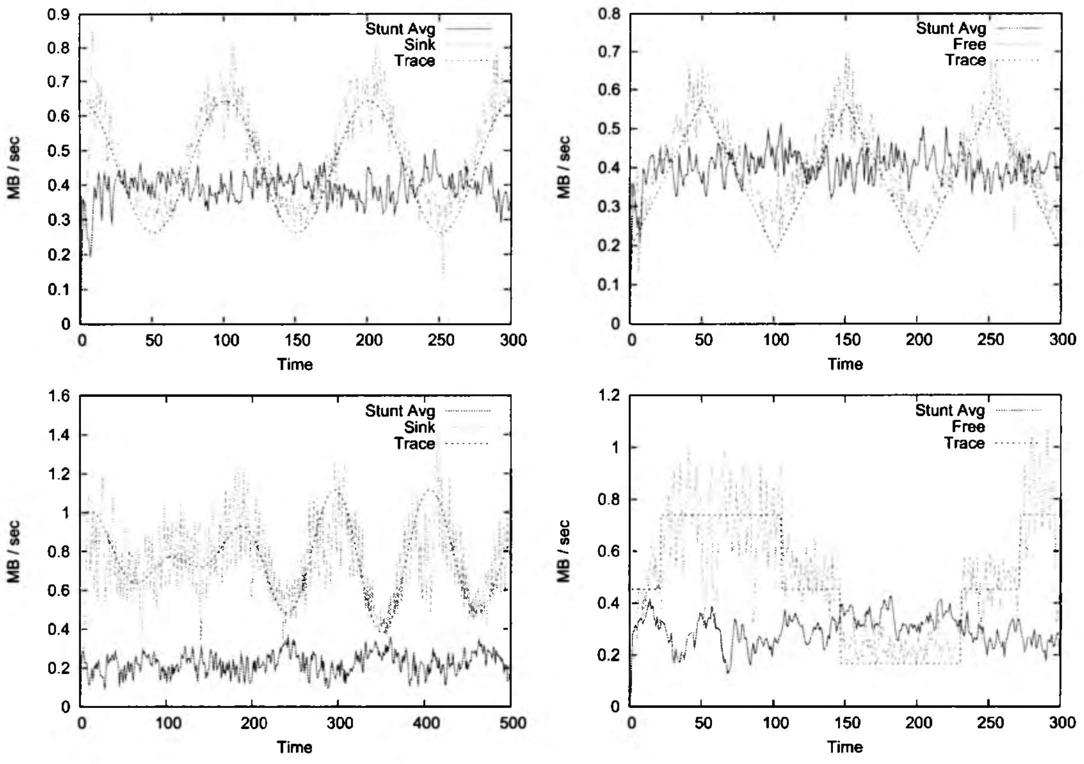


Fig. 7.3: *Representative simulation runs.*

during the simulation run. They were constructed to test specific aspects of the free flow's behavior. In particular, the ability of the flow to keep acquired bandwidth, acceptance of target rate variability, and high throughput rates were considered.

Figure 7.3 shows representative results using four different trace files. All results were obtained using a PI controller with 10 Stunt connections on networks with varying dynamics. In each simulation run the target trace, Free flow throughput, and Stunt average throughput are plotted.

The goal of these experiments is to gauge the versatility of the Free/Stunt architecture. The next few sections will examine them with more qualitative detail.

7.3.1 Target Trace with Large Variations

Figure 7.4 is obtained using a topology with 40 Mbps bottleneck link capacity. In order to provide some variability 8 of the cross-traffic connections through the bottleneck were randomized for an on/off behavior at intervals that varied in the range of 0 to 10 second. The exception was with 2 cross-traffic flows, which were instructed to send data continuously.

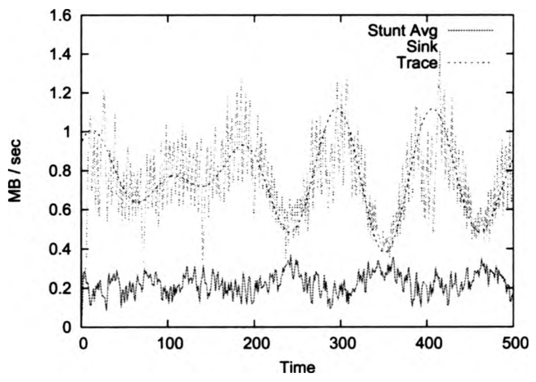


Fig. 7.4: *Target trace as sum of trigonometric functions.*

We see a great amount of oscillation around the target in the initial 200 seconds. This is to be expected due to the drop in data rate taken when the Free flow accepts a loss. This is the AIMD behavior of the congestion control algorithm. Recall that the Free flow accepts losses with some probability determined by the controller (see 4.4.4), and that loss is roughly a drop to about half its previous data rate. Also recall, the actual target for the purposes of control is $1.33 \times T$. The adjusted target allows the flow to average a rate that meets the requested target.

Moving onto the second interval past the 200 second mark we see that there seems to be an inability of the Free flow to fully target the next two local maxima values in the requested target trace function. The reason for this is due to network congestion, and possibly to a lack of reserve bandwidth at the Stunt flows. The architecture is designed to degrade gracefully rather than force acquisition of more bandwidth than is fair across the group of Free and Stunt flows.

Network congestion itself isn't the only reason that the Free flow cannot meet that target at that point in time. The other factor is that these drops are the result of a series of packet timeouts, noted by observing the instantaneous calculated throughput drop to a very low value at that time. Recall that timeout events are unconditionally accepted at the Free flow resulting in a slow start behavior (section 2.1). In effect, this is a feature of the architecture but not the service model.

7.3.2 Target Trace as Predictable Wave Forms

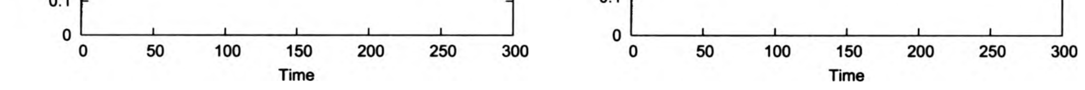


Fig. 7.5: *Target traces of repeating waveform functions.*

Figure 7.5 (top) is obtained using a topology with an 80 Mbps bottleneck link capacity. The number of cross-traffic links was kept constant at 15 connections. With figure 7.5 (bottom) the topology was created using an 80 Mbps bottleneck link capacity.

The number of cross-traffic links was kept constant at 20 connections.

Both target traces represent repeating waveform functions. This is of note because we are able to observe that the targeting is repeatable. For the most part it is also repeatable with what appears to be the same level accuracy.

We see that there is another observable consequence to the loss delegation mechanism of the Free/Stunt architecture. Notice, the average throughput of the stunts trends downward when the Free flow is above it and upward when it is below it. This is the direct result of loss delegation.

The most noticeable aspect of these plots is that the Free flow targeting is rather accurate no matter the value in relation to the stunts. We see that it displays a behavior described in section 4.3.1 as Stunt-to-Free Delegation. The Free flow would naturally have an average, fair share, throughput equal to that of the Stunts. In order to force the Free flow below this average the architecture allows Stunts to delegate packet loss to the Free flow while it is above the target.

7.3.3 Target Trace as a Stepping Function

Figure 7.6 is obtained using a topology with 80 Mbps bottleneck link capacity. The number of cross-traffic links was set at 20, randomize for on/off data transmission in the range of 0 to 30 seconds. Another four cross-traffic connections were configured to send data continuously.

This simulation shows that the Free/Stunt model quickly acquires the requested target within some error. To that end, it displays all of the behaviors as described in the other qualitative analyses. This is the case despite the network

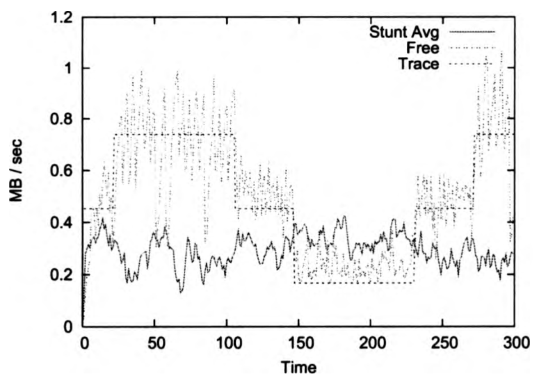


Fig. 7.6: *Target trace as a stepping function.*

dynamics being drastically different from the other simulations presented in the preceding sections.

The waveforms show a greater amount of variation due to the need to delegate a greater amount of loss. This is a direct result the large amount of randomize cross-traffic. These randomize flows do not work in unison. The effect is that there are periods of time when the network becomes congested very quickly due to the random amount of time they are "on." The RED queuing discipline provides increased packet drop probability at these times. The fact that the Free/Stunt architecture is able to still affect targeting of the requested throughput rate is a noteworthy achievement.

7.4 Impact of the Number of Stunt Connections

The impact of the number of Stunt connections on the performance of the Free flow is of great importance. Clearly, if it is possible to reduce the number of stunts needed for a specific level of service it would be best to do so. For that reason a study was done of the impact the number of stunts has on the service model. In order to do this the number of stunts is varied while holding all other parameters constant. The weighted variance (as given in equation 6.3) versus the number of Stunts is plotted, and an analysis is given.

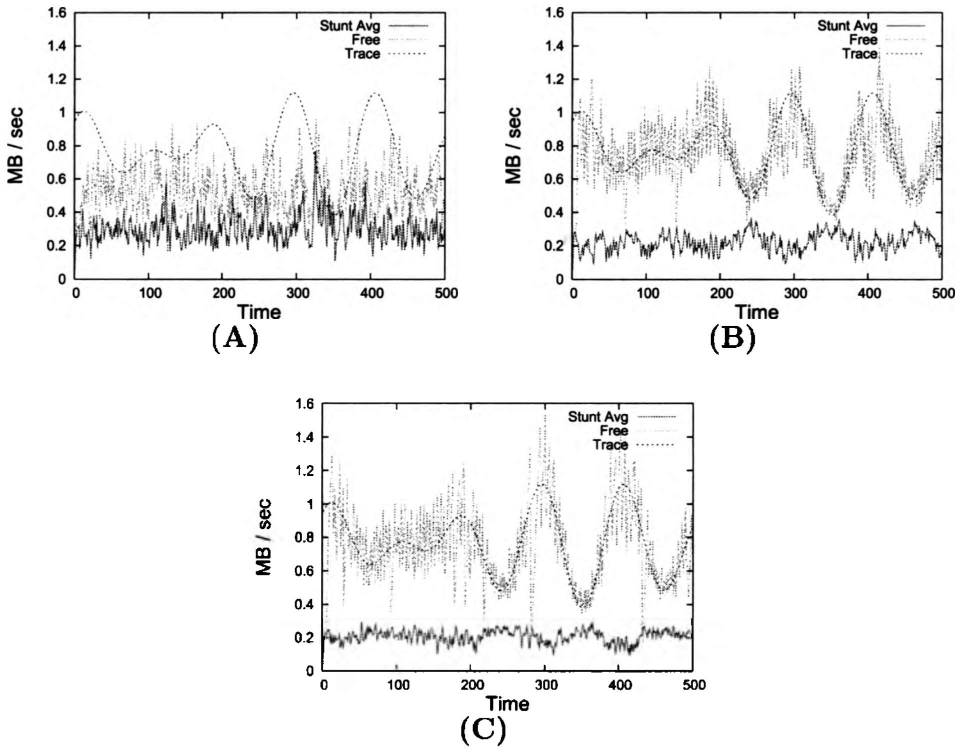


Fig. 7.7: *The impact of increasing numbers of Stunt flows on performance, from A to C the number of stunts is 2, 10, and 20 respectively.*

Figure 7.7 plots the free connection with the trace for 2, 10 and 20 Stunts. These plots were generated on a topology of an 80 Mbps bottleneck link with 20 cross-traffic connections. Notice, a very small number of Stunts (2) is unable to supply enough reserve bandwidth to meet the target rate. As the number of flows increases there is a noticeable improvement.

Figure 7.8 shows the results of several simulation runs where the number of Stunts is steadily increased. One can observe that there is an optimal number of

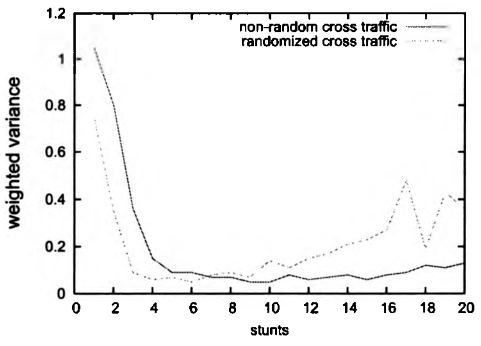


Fig. 7.8: *Impact of the number of Stunts on the weighted variance for non-random cross traffic and randomized cross-traffic.*

Stunts (about 5 or 6) that minimizes the weighted variance. As the number is increased there is an evident diminishing return.

The number of Stunts affects the overall efficiency of the method because they are more than a reservoir of bandwidth for the Free flow. In particular, they add to the level of congestion at the bottleneck proportionally to the number of flows passing through. If there number is very low, the Free flow cannot delegate losses since the fairness principles of TCP are strictly enforce on the aggregate of the Free and Stunt flows. If there numbers are very large, the network experiences a greater level of congested with an increase in the instances of timeout events.

7.5 Impact of Cross-traffic Dynamics

Several experiments were run to observe the behavior of the free flow when cross-traffic was randomized. Randomized traffic sends data for short periods of time, which turns on and off repeatedly. While experimenting with the randomized cross traffic the number of stunts was held constant while the dynamics of the cross-traffic was changed. The simulation was configured with 10 stunt connections for the network

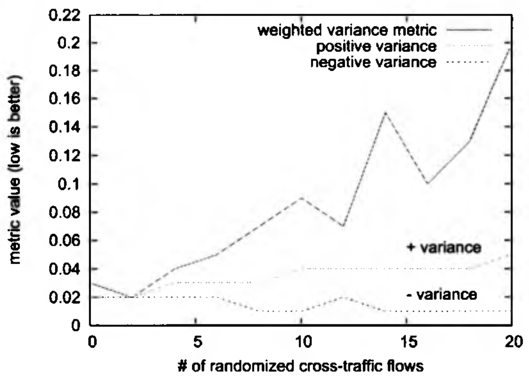


Fig. 7.9: *Variance metrics: weighted, positive, and negative.*

topology that was being simulated.

It is clear that, during this experiment, with the addition of the randomized cross-traffic comes a price paid with regard to staying on target. With respect to the increased randomization, it was observed that the free flow began to shifted upward rather than deformed. The two additional line plots in figure 7.9 represent the variance below and above the target, given in equations 6.2 and 6.1.

We see the positive and negative variances diverge away from their origin at about the same rate. The behavior across the simulation runs indicates a flow that is trending above the requested target. That observation was confirmed with an analysis of the graphical output of each simulation showing that was indeed the case. This accounts for the dramatic increase in the weighted efficiency metric as well.

Figure 7.10 shows a noticeable trending above the target when the randomized cross-traffic reaches 20 flows. Why exactly this occurs is due to a decrease in the utilization at the bottleneck. Randomizing cross-traffic flows with an on/off behavior effectively reduces the overall utilization of the bottleneck link in this instance.

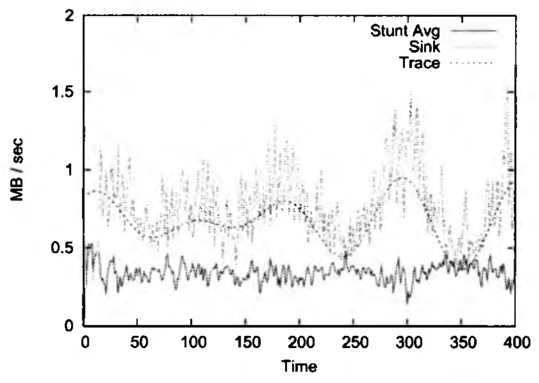


Fig. 7.10: *Free-Stunts with random cross-traffic.*

It should be noted that this particular problem can be alleviated with two distinct options. Firstly, the number of stunts can be increased with the effect of adding congestion at the bottleneck, or they could be decreased resulting in a lower achievable Free throughput. Secondly, the Free flow's AIMD losses could be increased to more than half the congestion window in order to better match the target. The first options is discussed in section 7.4, and the second options is left to future work.

7.6 Free and Stunts Achievable Range

A series of experiments were conducted to examine the relationship between the number of stunts and the Free flow's achievable throughput. Figure 7.11 is a plot of the data rate versus the number of stunts per simulation run. The network topology was configured to, again, have an 80Mbps bottleneck. The Free/Stunt link was 50Mbps, and the 15 cross-traffic flow links were 10Mbps each.

It was observed that the Free flow maxed out its link capacity due to an Elastic Tunneling effect (Guirguis et al., 2004). We see that as the number of stunts

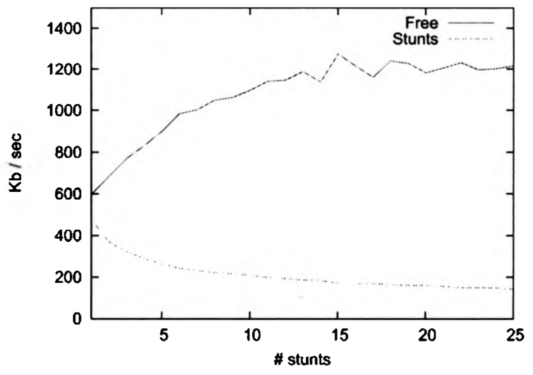


Fig. 7.11: *The Free flow's throughput as stunts are added.*

approaches 25 there is a leveling off at about the capacity of the Free/Stunt link.

Generally, the Free flow is limited by the amount of loss it can delegate to the Stunt flows. This limit is imposed by the size of the congestion windows. A simple rule is that the $cwnd_{Free} <= \sum_{Stunts=1}^{n} cwnd_{Stunts_n}$ for loss delegation to occur. There must be enough congestion "window" amongst the Stunt flows to absorb the loss Delegation from the Free flow.

Once the Stunt flows cannot accept delegation due to the lack of congestion "window," there is a brief time when all loss events will be accepted at the Free flow. That time is based on the average growth rate of the Stunts' congestion window. Increasing the number of stunts has the effect of replenishing the Stunts' combined window space faster.

$$G_f + E = (G_{fair} - G_s) \times C_s + G_{fair} \tag{7.2}$$

The Stunts have a lower limit that is defined by having a combined congestion window that is less than that of the Free flow. Network dynamic as well as the number of stunts affects the lower limit. The relation between the Stunts' lower limit and the maximum achievable throughput for the free flow is linear and predictable, to some degree, given by equation 7.2. Where G_{fair} is the TCP fair throughput rate, C_s is the stunt count, G_s is the average stunt throughput, E is the total "leakage" (section 7.7), and G_f is the average Free throughput.

7.7 Extra Loss Delegation as an Efficiency Tool

The extra term E is added to achievable Free flow throughput in equation 7.2 due to leakage from delegating too much loss. This is the result of the Round-Robin distribution algorithm described in section 4.3.2. This aspect is a matter of efficiency that can be resolved simply by forcing a smaller amount delegated loss onto the final Stunt in the distribution chain. This measure effectively gives the architecture the ability to control exactly how much of this extra loss it wishes to "leak" to the network.

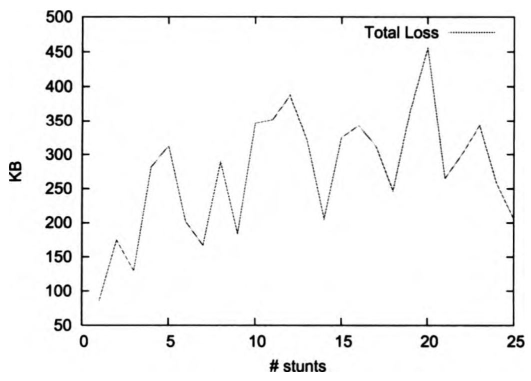


Fig. 7.12: *Total extra loss as Stunts increase.*

The data in the appendix B described here in figures 7.12 and 7.13 was obtained using the same topology as used in section 7.6. The numbers are high because the simulations were not designed to minimize the extra loss from distribution. The result is that these numbers are the maximum amount of extra loss that can be given up to the network being simulated. The appendix has the raw data for these experiments, and similar trending was observed in other test runs.

The amount of "leakage" is measured as a comparison to how much bandwidth the Free and Stunt flows would obtain without delegation of loss. Figure 7.12 shows the total bandwidth leaked to the network as the number of stunts is increased. The trending appears to be leveling off.

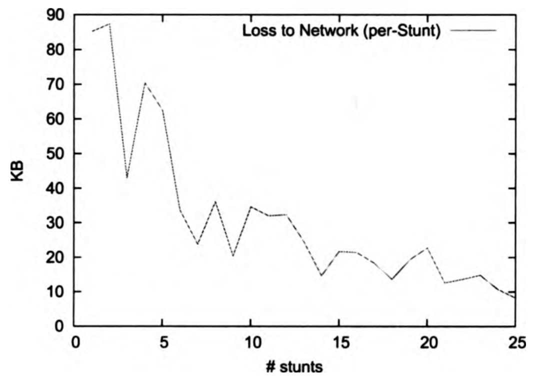


Fig. 7.13: *Extra loss per-Stunt as stunts increase.*

Figure 7.13 shows the per-Stunt extra loss as the number of stunts is increased. Again, the trend seems to be leveling off. It also shows a dramatic drop in the per-Stunt extra loss. The behavior seen here was predicted while designing the Round-Robin distribution method.

This would seem to be a rather crippling side-effect of the distribution method. However, as mentioned in the beginning of this section, this aspect can be negated by reducing the amount of loss distributed to the stunts. The architecture has the ability to control the amount of delegation to specific amounts on a per-Stunt basis. Impromptu testing has shown that it is possible to regain the "extra loss" described here. If one is not careful, it is also possible to make the distribution method behave unfairly by retaining too much bandwidth.

CHAPTER 8

FREE/STUNT IMPLEMENTATION IN LINUX

8.1 Experiment Setup

The Free/Stunt architecture is implemented as a pluggable module for the 2.6.20 Linux kernel. It is designed to behave as much like the simulation TCP Free/Stunt class as possible in the Linux environment. One notable exception is the controller, which is implemented as an on/off switch due to constraints of kernel programming. The Linux kernel does not natively support floating point operations, meaning that floating point math must be accomplished using fix-point arithmetic. In order to speed development the use of the On/Off Controller was chosen.

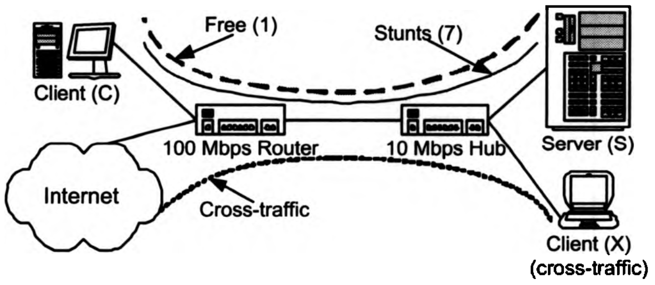


Fig. 8.1: *Linux implementation network setup.*

There are several differences between the simulated world and the physical test environment. Figure 8.1 shows the layout of the physical network test environment. The network bottleneck link C is a path composed of a 100Mbit switch A and a

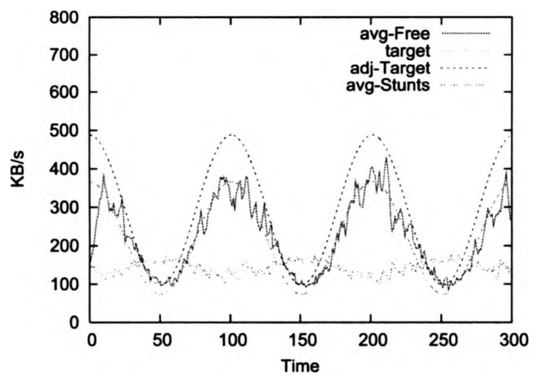
10Mbps hub B, which means that the bottleneck exist at the hub B. Recall, the simulated networks consisted of routers with a RED queuing discipline.

The server S was a Linux PC with the Free/Stunt architecture implemented as a pluggable Congestion Control module and a server application. The client R was a Windows XP PC running client software. All the experiments in this chapter were composed of 1 Free flow and 6 Stunt flows. The actual flow data was made up of alphanumeric character sequences and was continuous between the free and the stunts. However, cross traffic was randomly generated manually. These flows accessed websites and other downloadable content during the experiments.

The Free flow's Adjusted Target for these experiments is $1.33T$. Experimentation has shown that this adjustment is not optimal for the Linux implementation. Currently, there is no method in place to determine the optimal value. These experiments were composed with that target simply to conform to the standards set in the simulations.

8.2 Target Matching Across Experiments

The plots in figure 8.2 show the Free flow, Target, Adjusted-Target used by the module, and the average stunt flows. The Target is the requested service requirement. The Adjusted-Target is the value set in the module for matching throughput while executing the architectural method of delegating congestion signals. The hope is to use the Adjusted-Target to average to the requested Target. The average of the combined Stunts is plotted in order to show their combined behavior as the Free flow meets its requested service requirements.



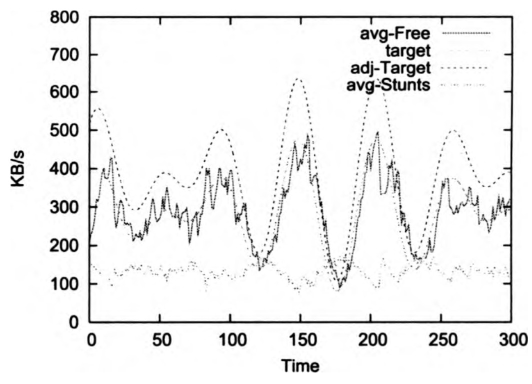


Fig. 8.2: *Average free flow shaping and target matching to a smooth trace request.*

In order to be of use the Free/Stunt architecture must be able to provide the requested level of quality on a relatively consistent basis. The Soft service model accepts that there are times when the requested service cannot be met, but it is best to attempt to meet service requirements whenever possible.

These experiments focus on the ability of the Free/Stunt architecture to consistently meet service requirements across multiple experiments. Each plot in figure 8.2 represents four experiments using the same Target trace. The Free flows were averaged together to exhibit a picture of where the targeting meets across experiments.

On average we see that the Free flow does shape to the requested service requirement. The shaping is rather impressive. More importantly, this experiment confirms that the Free and Stunt architecture has the ability to meet requirements with a consistent level of service, given the relatively consistent network dynamics. That is to say that one can expect the architecture to not behave erratically under slightly differing conditions.

8.3 More Efficient Distribution Methods

When performing reverse-delegation, described in section 4.3.1, the Free flow is instructed to perform a normal reduction of its congestion window. Stunt-to-Free delegation entails that the Free flow accept a loss in the usual manner, which is a drop of about $\frac{1}{2} \times cwnd$. The required reduction at the Free flow should only be half of the Stunt flow's congestion window value, but for simplicity the delegated signal was sent without making that distinction.

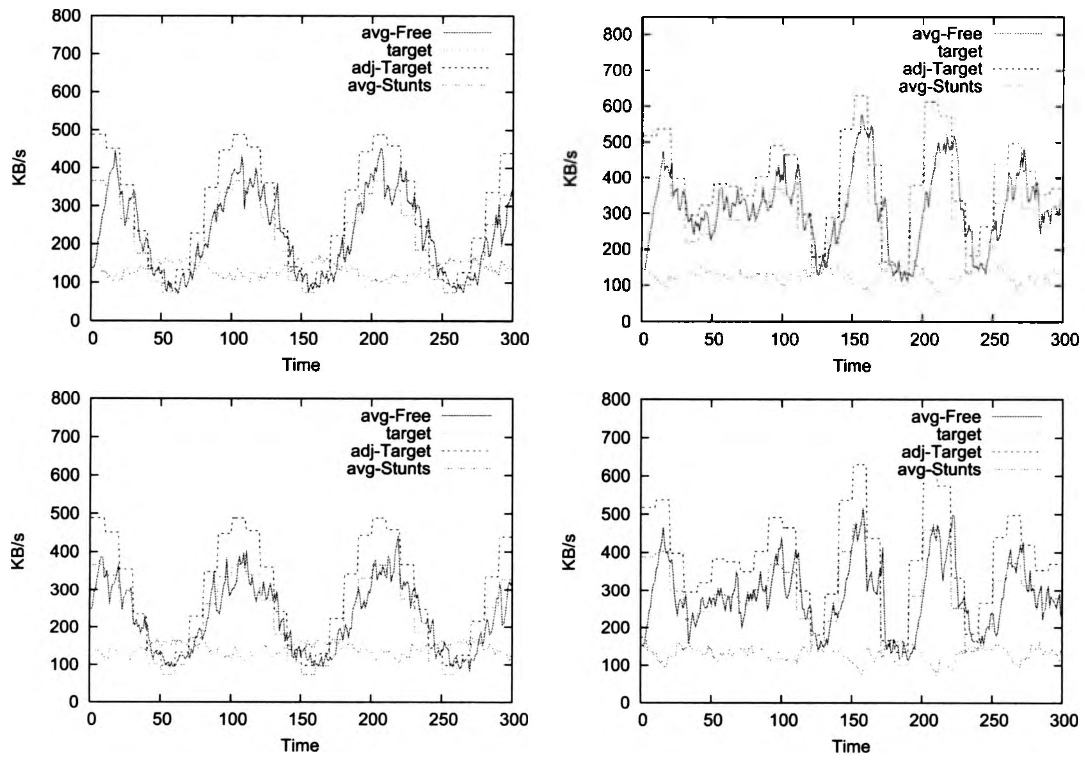


Fig. 8.3: *Multiple experiments showing the effects of changing the reverse distribution method.*

A change was made as a first step in making the targeting behavior more efficient. For this experiment the "reverse" delegation method was changed to only delegate the exact amount of loss necessary. It was assumed that this change would result in a closer targeting to the Adjusted Target rather than the Requested Target, which was the case.

Figure 8.3 (top two) is an average of four experiments using the more efficient reverse distribution method. Note, the stepping trace function was used as a matter of preference in these experiments. We see a higher trending than the same experiment run without the more efficient method (bottom two).

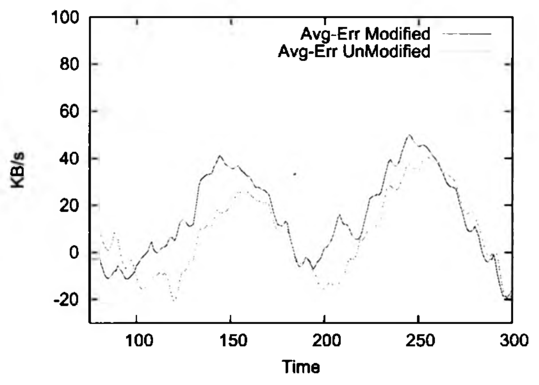


Fig. 8.4: *Modified reverse distribution results in higher error and closer trending to Adjusted Target.*

Figure 8.4 depicts the trending of the modified reverse delegation method versus the usual method, on the throughput error from the Requested Target. The plotted lines are the average error over all the experiments. They were constructed using the data from the cosine-wave experiments shown in figure 8.3. The plot time scale is truncated at 75 seconds because of the flow's random origination throughput at the beginning of the experiment.

The plots confirm the visual observation in figure 8.3 that the modified version trends higher on average. These experiments showed that the Free flow varied between the two target plots, with a trending more toward the Adjusted Target value when using the targeting efficiency methods described in this section.

The material results are give and take. There is a tradeoff in target matching efficiency between the modified method and the standard method. The original method has better target matching behavior due to more predictable throughput throttling on congestion events, while the modified version conserves bandwidth that would normally be lost to the network on delegation.

8.4 Application Level Resource Contention

One of the more significant issues with an actual network environment is that the clients and servers have Application level limitations on resource utilization. Each process has, among others, limits on the amount of memory and CPU utilization it will be allocated. Sometimes disturbances to those resource allocations can cause service problems. This issue affects the service level that the Free and Stunts can provide. To that end, these problems are not technically in the domain of the Free and Stunts architecture's service model, but there may be some way of limiting the impact on service. That specific method is left to future research endeavors. The purpose of this experiment was simply to observe this effect and note that it occurs.

Figure 8.5 shows the Free flow, Target, Adjusted-Target used by the module, and the average stunt flows. At around 340 seconds the client computer initiated a download of several videos from the site http://www.youtube.com/. The videos downloaded for approximately 50 seconds. This event added both network and process resource contention and correlated with the drop in data rate at that time.

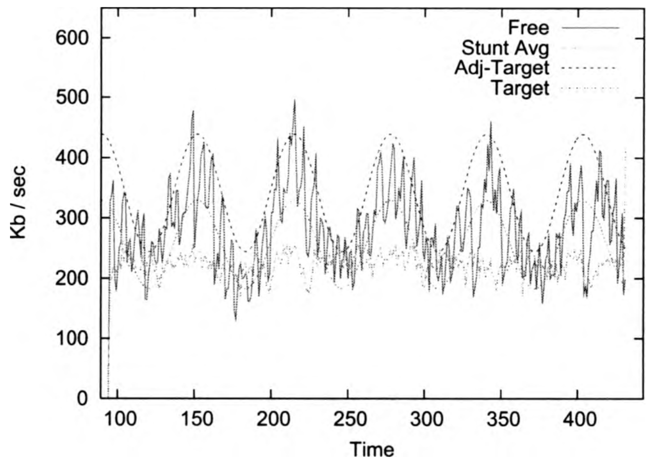


Fig. 8.5: *The effect of application resource contention on throughput as measured by the client software.*

Again, this more than likely was the application processing slowing while the browser loaded, executed, downloaded the page, and then started the videos. The Free/Stunt architecture works at the TCP level and does not have the ability to affect the Application. The application slowed its reading of data from the buffer for a short while due to lack of resources (e.g. CPU).

CHAPTER 9

CONCLUSIONS

9.1 The Results

In this paper a Soft Quality of Service architecture and service model were described. The Free/Stunt architecture is composed of a grouping of aggregated flows. In turn, that group of flows is composed of one or more Free flows and several Stunt flows. Using interflow resource sharing, described as Loss Delegation, the Free and Stunt flows are able to alter their sending throughput with a greater ability to meet a specific requested Target rate.

An analytical model was devised that was backed by a series of simulation experiments utilizing the ns2 discrete event simulator. Simulation results showed that the method was sound. Furthermore, metrics were defined to measure the target matching behavior. The effects of varying degrees of congestion, cross-traffic dynamics, and the necessary number of Stunt flows were examined from an efficiency standpoint. It was discovered that an optimal number of stunts existed for any given bottleneck configuration, and that target matching required attempting to average to the requested target rather than meet it continuously.

Building on the simulation results, a Linux Congestion Control Module was developed. The module showed the same ability to match a requested throughput rate as both the analytical model and the simulation results. However, the addition of Application level resource contention showed that other factors exist in a physical environment that affects service levels. These issues lay outside the domain of the Free and Stunts service model, but they may not be totally irresolvable with some alterations to targeting behavior.

9.2 Future Research

Future work will focus on better matching methods, which possibly do not only affect changes in congestion events. Also, control methods for setting the Adjusted Target during execution will need to be devised to compensate for the dynamic nature of the Application layer. Naturally, the majority of foreseeable research avenues lay in application of the method and increased targeting efficiency.

9.3 General Conclusion

This paper has presented a rather thorough examination of the Free/Stunt architecture. Among the results is the presentation of the concept of Loss Delegation, which is believed to be a unique method of interflow resource sharing. The Free/Stunt service model clearly shows the ability to provide a generally reliable service level between end-points without the need of network assistance. It is clear that the Free/Stunt architecture is usable in its current form but can be made more efficient for implementation in a real network environment, which is the goal of future research efforts.

APPENDIX A

PROCESSING SCRIPTS

Tcl Simulation Script Template

```
#
# _simtemplate3 tcl
# - This file is a template file used for running batch
#   simulations It is intended for use with the shell
#   script 'batchsim2'
#
# Author Jason Valdez
# Date 03/18/08
# Copyright 2008, Jason Valdez
#

set ns [new Simulator]

global defaultRNG
$defaultRNG seed <[randomseed]>

#simulation constants
set tcpfreetrace_file " /traces/var_tcpet tr"
set redqtrace_file " /traces/var_redqueue tr"
set tcpetsinkmon_file " /traces/mon_atsink txt"
set simtracefile_file <[simtracefile]>
set stuntdir " /stunt_traces"
set runsimfor <[runsimfor]>
set stuntcount <[stuntcount]>
set normalcount <[normalcount]>
set tcpetpacketsize <[tcpetpacketsize]>
set sr_datarate <[srdatarate]>
set sr_picontroller <[srpicontroller]>
set sr_piconstk <[srpiconstk]>
set sr_rateunder <[srrateunder]>
set sr_rateoverunder <[srrateoverunder]>

set redq_bmin <[redqbmin]>
set redq_bmax <[redqbmax]>
set redq_size <[redqsize]>

#used to monitor bytes at sink for free connection
set last_bytes 0
set last_value 0
set recordsink_refresh 1 0
set tcpetsinkmon_ [open "$tcpetsinkmon_file" w]

#this is the trace file
set simtracefile_ [open "$simtracefile_file" r]

#this samples the req q
set redqtrace_ [open "$redqtrace_file" w]

#sink average
set stuntSink_avg [open "$stuntdir/sink_stunt_avg tr" w]

#setup defaults for RED queue
#Queue/RED set setbit_ false, # use ECN bit
Queue/RED set bytes_ false
Queue/RED set queue_in_bytes_ false
Queue/RED set thresh_ $redq_bmin
Queue/RED set maxthresh_ $redq_bmax
Queue/RED set q_weight_ 0 000002
#Queue/RED set wait_ true
#Queue/RED set linterm_ 10
#Queue/RED set mark_p_ 0 1
#Queue/RED set use_mark_p_ true

set node_(s1) [$ns node]
set node_(r1) [$ns node]
set node_(r2) [$ns node]
set node_(s2) [$ns node]

$ns duplex-link $node_(s1) $node_(r1) 50Mb 4ms DropTail
$ns duplex-link $node_(r1) $node_(r2) 80Mb 20ms RED
$ns queue-limit $node_(r1) $node_(r2) $redq_size
$ns queue-limit $node_(r2) $node_(r1) $redq_size
$ns duplex-link $node_(s2) $node_(r2) 50Mb 4ms DropTail

$ns duplex-link-op $node_(s1) $node_(r1) orient right-down
$ns duplex-link-op $node_(r1) $node_(r2) orient right
$ns duplex-link-op $node_(r1) $node_(r2) queuePos 0
$ns duplex-link-op $node_(r2) $node_(r1) queuePos 0
$ns duplex-link-op $node_(s2) $node_(r2) orient left-down

set normalRGN [new RandomVariable/Uniform]
$normalRGN set min_ 0 0
$normalRGN set max_ 30 0
proc randstart {starttime parobj} {
  global ns normalRGN
  set st [expr $starttime + [$normalRGN value] ]
  set et [expr $st + [$normalRGN value] ]
  $ns at $st "$parobj_start"
  $ns at $et "$parobj_stop"
  $ns at $et "randstart_$et_$parobj"
}

set lmon [new Agent/TCPSink]    ,# tcp free connection sink
set tcp1 [new Agent/TCPET/Newreno] ,# free connection
$ns attach-agent $node_(s1) $tcp1
$ns attach-agent $node_(s2) $lmon
$ns connect $tcp1 $lmon
$lmon set window_ 15000000
$tcp1 set window_ 15000000
$tcp1 set packetSize_ $tcpetpacketsize
$tcp1 set fid_ 1

set t1 [new Application/FTP]
$t1 set type_ FTP

$t1 attach-agent $tcp1

#mark tcp1 as the free connection
#and setup the sink monitor (this is not used by the tcp class)
$tcp1 isfree true
# enable the PI Controller
$tcp1 enablepi true
$tcp1 pi_datarate_period 3 0
$tcp1 pi_datarate_alpha <[picontrolleralpha]>
#$tcp1 stuntdisttofree true  ,needs to be done after stunts are setup
#$tcp1 mon-sink $lmon
$tcp1 setSampleRate datarate $sr_datarate
$tcp1 setSampleRate picontroller $sr_picontroller
$tcp1 picontroller_const_k $sr_piconstk
$tcp1 setSampleRate rateunder $sr_rateunder
$tcp1 setSampleRate rateoverunder $sr_rateoverunder
# stunt connections
for {set i 1} {$i <= $stuntcount} {incr i 1} {
  set stunt$i [new Agent/TCPET/Newreno]
  set stuntSink$i [new Agent/TCPSink]
  #set stunt$i [$ns create-connection TCPET/Newreno $node_(s1) TCPSink $node_(s2) $i]
  $ns attach-agent $node_(s1) [set stunt$i]
  $ns attach-agent $node_(s2) [set stuntSink$i]
  $ns connect [set stunt$i] [set stuntSink$i]
  [set stunt$i] set window_ 15000000
  [set stunt$i] set packetSize_ $tcpetpacketsize
  [set stunt$i] isfree false
  [set stunt$i] set fid_ $i
  $tcp1 add-stunt [set stunt$i]

  set stuntSinktr_$i [open "$stuntdir/tr_stunt_sink_$i tr" w]
  set lastbytes_stuntSink$i 0 0

  [set stunt$i] setSampleRate datarate 0 1

  set tstunt_$i [open "$stuntdir/tr_stunt_$i tr" w]
  #[set stunt$i] trace cwnd_
  #[set stunt$i] trace droptimedelta_
  #[set stunt$i] trace rateunder_
  #[set stunt$i] trace rtt_
  [set stunt$i] trace datarate_
  #[set stunt$i] trace rateoverunder_
  [set stunt$i] attach [set tstunt_$i]

  set ftp1$i [[set stunt$i] attach-source FTP]

  $ns at 0 0 "[set_ftp1$i]_start"
}

# this will either turn on or off stunt loss distribution
# to the free connection
# NOTE - need to turn on and off after stunts have been created and linked
$tcp1 stuntdisttofree false
```

```
#this can be used to break the
#link between the free and stunt connections
#after they have been configured
#$tcp1 isfree false

#regular connections, optional random on off
for {set i [expr $stuntcount + 1]} {$i < [expr $stuntcount + 1 + $normalcount]} {incr i 1} {
  puts "SETUP -----Normal_Connection_$i_created"
  set ssource$i [$ns node]
  set ssink$i [$ns node]
  $ns duplex-link [set ssource$i] $node_(r1) 10Mb 4ms DropTail
  $ns duplex-link [set ssink$i]   $node_(r2) 10Mb 4ms DropTail
  set tcp_$i [$ns create-connection TCP/Newreno [set ssource$i] TCPSink [set ssink$i] $i]
  [set tcp_$i] set window_ 15000000
  [set tcp_$i] set packetSize_ 512
  set ftp_$i [[set tcp_$i] attach-source FTP]

  if {$i > [expr ($stuntcount + 1 + $normalcount) - <[normalrandom]>] } {
     randstart 0 0 [set ftp_$i]
  } else {
     $ns at 0 0 "[set_ftp_$i]_start"
        }
}

#define tcl var for queue
set redq [[$ns link $node_(r1) $node_(r2)] queue]

set tcpfreetrace_ [open "$tcpfreetrace_file" w]
#$tcp1 trace cwnd_
$tcp1 trace droptimedelta_
#$tcp1 trace rateunder_
#$tcp1 trace rtt_
$tcp1 trace datarate_
#$tcp1 trace picontroller_
#$tcp1 trace rateoverunder_
$tcp1 attach $tcpfreetrace_

$ns at 0 0 "$t1_start"
$ns at 0 0 "readtrace"
$ns at 0 0 "recordsink"
$ns at 0 0 "recordredq"
$ns at 0 0 "recordstuntsink"
$ns at $runsimfor "finish"

proc finish {} {
   global ns
  $ns flush-trace

  close_files

  exit 0
}

proc close_files {} {
  global ns tcpfreetrace_ tcpetsinkmon_ simtracefile_ redqtrace_
  $ns flush-trace

  if { [info exists tcpfreetrace_] } {
    close $tcpfreetrace_
  }

  if { [info exists tcpetsinkmon_] } {
    close $tcpetsinkmon_
  }

  if { [info exists simtracefile_] } {
    close $simtracefile_
  }

  if { [info exists redqtrace_] } {
    close $redqtrace_
  }

  closeStuntTraces
}

proc recordstuntsink {} {
  global ns stuntcount stuntSink_avg

  for {set i 1} {$i <= $stuntcount} {incr i 1} {
    global stuntSink$i
    global lastbytes_stuntSink$i
    global stuntSinktr_$i
  }

  set sinksum 0 0

  for {set i 1} {$i <= $stuntcount} {incr i 1} {
    set bytes [[set stuntSink$i] set bytes_]
    set last_bytes [set lastbytes_stuntSink$i]
    set diff [expr $bytes - $last_bytes ]
    puts [set stuntSinktr_$i] "[$ns␣now]␣$diff␣␣"
    set lastbytes_stuntSink$i $bytes
    set sinksum [expr $sinksum + $diff]
  }
  puts $stuntSink_avg "[$ns␣now]␣␣[expr␣$sinksum␣/␣$stuntcount␣]␣"
  set sinksum -1 0
  $ns at [expr [$ns now] + 1] "recordstuntsink"
}

proc recordredq {} {
  global ns redqtrace_ redq
  puts $redqtrace_ "[$ns␣now]␣␣[$redq␣set␣curq_]␣␣[$redq␣set␣ave_]"
  $ns at [expr [$ns now] + 0 10] "recordredq"
}

proc recordsink {} {
  global lmon ns tcpetsinkmon_ last_bytes tcp1 recordsink_refresh
  set bw [$lmon set bytes_]
  set diff [expr ($bw - $last_bytes)]
  set last_bytes $bw
  puts $tcpetsinkmon_ "[$ns␣now]␣$diff"
  $ns at [expr [$ns now] + $recordsink_refresh] "recordsink"
}

proc readtrace {} {
  global ns simtracefile_ tcp1
  if {[gets $simtracefile_ line] != -1} {
    set items [split $line]
    set time  [lindex $items 0]
    set value [lindex $items 1]
    $tcp1 freewindow $value
    $ns at [expr $time] "readtrace"
  } else {
    close $simtracefile_

  }
}

proc closeStuntTraces {} {
        global stuntcount stuntSink_avg

  if { [info exists stuntSink_avg  ] } {
    close [set stuntSink_avg]
  }

  for {set i 0} {$i < $stuntcount} {incr i 1} {
    global tstunt_$i
    if { [info exists tstunt_$i] } {
      close [set tstunt_$i]
    }
    global stuntSinktr_$i
    if { [info exists stuntSinktr_$i] } {
      close [set stuntSinktr_$i]
    }
  }
}

# start simulation
$ns run
```

Bash Batch Simulation Script

```
#!/bin/bash
# author Jason Valdez
# version 2 0
# this version of the batsim script uses a command input to create the trace files from the executable generator
# the executable files are located in the traces directory under source

function printReport {
  printf "$@" >> $REPORT
}

function graph_views {
# create scripts to view graphs
  PREVDIR=$(pwd)
  cd "$BATCHDIR/$WORKINGDIR/graphs"

  #interpolated and trace graph at sink
  printf "set␣terminal␣png␣nocrop␣enhanced␣size␣640,480␣\n" >> " /drate_sink_trace gplt"
  printf "plot␣\"  /traces/mon_atsink txt\"␣using␣1 2␣title␣\"Sink bytes/sec\"␣with␣lines ,␣" >> " /drate_sink_trace gplt"
  printf "\"  /traces/$TRFILENAME\"␣using␣1 2␣title␣\"Trace bytes/sec\"␣with␣lines␣\n" >> " /drate_sink_trace gplt"

  #calculated datarate and trace at sending free node
  printf "set␣terminal␣png␣nocrop␣enhanced␣size␣640,480␣\n" >> " /drate_calc_trace gplt"
  printf "plot␣\"  /traces/datarate_trace txt\"␣using␣1 2␣title␣\"Calc bytes/sec\"␣with␣lines ,␣" >> " /drate_calc_trace gplt"
  printf "\"  /traces/$TRFILENAME\"␣using␣1 2␣title␣\"Trace bytes/sec\"␣with␣lines␣\n" >> " /drate_calc_trace gplt"

  #sink line graph, trace and, calculated datarate at avgerage of stunt
  printf "set␣terminal␣png␣nocrop␣enhanced␣size␣640,480␣\n" >> " /drate_sink_stnt_trace gplt"
  printf "plot␣\"  /stunt_traces/sink_stunt_avg tr\"␣using␣1 2␣title␣\"Stunt Avg\"␣with␣lines ,␣" >> " /drate_sink_stnt_trace gplt"
  printf "\"  /traces/mon_atsink txt\"␣using␣1 2␣title␣\"Sink bytes/sec\"␣with␣lines ,␣" >> " /drate_sink_stnt_trace gplt"
  printf "\"  /traces/$TRFILENAME\"␣using␣1 2␣title␣\"Trace bytes/sec\"␣with␣lines\n" >> " /drate_sink_stnt_trace gplt"


  #redq
  printf "set␣terminal␣png␣nocrop␣enhanced␣size␣640,480␣\n" >> " /redq_trace gplt"
  printf "plot␣\"  /traces/var_redqueue tr\"␣using␣1 2␣title␣\"RED Q\"␣with␣lines ,␣" >> " /redq_trace gplt"
  printf "\"  /traces/var_redqueue tr\"␣using␣1 3␣title␣\"RED Q AVG\"␣with␣lines␣\n␣" >> " /redq_trace gplt"

  #redq
  printf "set␣terminal␣png␣nocrop␣enhanced␣size␣640,480␣\n" >> " /redq_trace_avg gplt"
  printf "plot␣\"  /traces/var_redqueue tr\"␣using␣1 3␣title␣\"RED Q Avg\"␣with␣lines␣\n␣" >> " /redq_trace_avg gplt"

  #pi controller
  printf "set␣terminal␣png␣nocrop␣enhanced␣size␣640,480␣\n" >> " /pi_trace_ gplt"
  printf "plot␣\"  /traces/picontroller_trace txt\"␣using␣1 2␣title␣\"PI Controller\"␣with␣lines␣\n␣" >> " /pi_trace_ gplt"

  gnuplot " /pi_trace_ gplt" > " /pi png"
  gnuplot " /redq_trace gplt" > " /redq png"
  gnuplot " /redq_trace_avg gplt" > " /redq_avg png"
  gnuplot " /drate_sink_stnt_trace gplt" > " /drate_sink_stunt png"
  gnuplot " /drate_sink_trace gplt" > " /drate_sink png"
  gnuplot " /drate_calc_trace gplt" > " /drate_calc png"
  cd $PREVDIR
}


if [ ! -f $1 ], then
{
  printf "File ␣$1␣ ␣Does␣not␣exist\n"
  exit 0
}
else if [ "$1" == "" ]; then
  printf "USAGE␣batchsim␣<source␣file>␣<source␣template>␣<sim␣id>\n"
  exit 0
   fi
fi

if [ ! -f $2 ]; then
{
  printf "File␣$2␣␣Does␣not␣exist\n"
  exit 0
}
else if [ "$2" == "" ]; then
  printf "USAGE␣batchsim␣<source␣file>␣<source␣template>␣<sim␣id>\n"
  exit 0
   fi
fi

if [ "$3" == "" ]; then
  printf "USAGE␣batchsim␣<source␣file>␣<source␣template>␣<sim␣id>\n"
  exit 0
fi

RUNID=$3
SIMSOURCE=$1
SIMTEMPLATE=$2
COUNT=0
WORKINGDIR=""
CPSIMTRDIR="./simtraces"
BATCHDIR=$(pwd)
REPORT="$BATCHDIR/runs/$RUNID/report.csv"

if [ ! -e "./runs" ]; then
  printf "Directory␣'./runs'␣does␣not␣exist␣-␣creating\n"
  mkdir "./runs"
fi
if [ ! -e "./runs/$RUNID" ]; then
{
  printf "Directory␣'./runs/$RUNID'␣does␣not␣exist␣-␣creating\n"
  mkdir "./runs/$RUNID"
}
else
{
  printf "Directory␣'./runs/$RUNID'␣does␣exist␣-␣clearing\n"
  rm -rf "./runs/$RUNID"
  mkdir "./runs/$RUNID"
}
fi

touch $REPORT

printReport "BAT-ID␣RUN-ID␣WORKING␣DIR␣TRACE␣PI␣K␣"
printReport "COUNTc␣AVGc(+)␣COUNTc␣AVGc(-)␣BOTHc␣T-COUNTc␣EFF-METRICc
␣␣␣␣␣␣␣␣␣␣␣␣␣COUNTm␣AVGm(+)␣COUNTm␣AVGm(-)␣BOTHm␣T-COUNTm␣EFF-METRICm␣\n"

cp "$SIMSOURCE" "./runs/$RUNID/$SIMSOURCE"

while read sourceline
do
   if [ "$sourceline" != "" ]; then  #don't process empty string
    SIMPARS=( $sourceline )
    if [ "${SIMPARS[0]}" != "#" ], then

     COUNT=$(($COUNT + 1))
     printf "Processing␣simulation␣$COUNT\n"

     if [ ${#SIMPARS[@]} != 19 ], then
      printf "Source␣file␣does␣not␣define␣all␣variable␣inputs\n"
      printf "Source ␣$sourceline\n"
      exit 0
     fi
     WORKINGDIR=" /runs/$RUNID/run␣$COUNT"
     if [ -e $WORKINGDIR ], then
      printf "Directory␣exist␣-␣clearing\n"
      rm -rf $WORKINGDIR
     fi
     # strip " from input
     SIMPARS[0]=$(printf "${SIMPARS[0]}" | sed 's/\"//g')
␣␣␣␣#␣create␣the␣test␣run␣directory
␣␣␣␣mkdir␣$WORKINGDIR
␣␣␣␣#␣create␣a␣directory␣for␣the␣stunt␣traces
␣␣␣␣mkdir␣"$WORKINGDIR/stunt␣traces"
␣␣␣␣mkdir␣"$WORKINGDIR/graphs"
␣␣␣␣mkdir␣"$WORKINGDIR/traces"
␣␣␣␣#␣copy␣the␣trace␣file␣to␣the␣simulation␣run␣directory
␣␣␣␣TRACECMD="$CPSIMTRDIR/source/${SIMPARS[0]} ${SIMPARS[1]} ${SIMPARS[2]} ${SIMPARS[3]}"
␣␣␣␣TRFILENAME="${SIMPARS[0]}␣${SIMPARS[1]}␣${SIMPARS[2]}␣${SIMPARS[3]} txt"
␣␣␣␣$TRACECMD␣>␣"$WORKINGDIR/traces/$TRFILENAME"

␣␣␣␣cat␣" /$SIMTEMPLATE"␣\
␣␣␣␣␣|␣sed␣-e␣"s/<\[simtracefile\]>/\"\ \/traces\/$TRFILENAME\"/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[runsimfor\]>/${SIMPARS[4]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[stuntcount\]>/${SIMPARS[5]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[normalcount\]>/${SIMPARS[6]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[tcpetpacketsize\]>/${SIMPARS[7]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[srdatarate\]>/${SIMPARS[8]}/"␣␣\
␣␣␣␣␣␣␣-e␣"s/<\[srpicontroller\]>/${SIMPARS[9]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[srpiconstk\]>/${SIMPARS[10]}/"␣␣\
␣␣␣␣␣␣␣-e␣"s/<\[srrateunder\]>/${SIMPARS[11]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[srrateoverunder\]>/${SIMPARS[12]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[redqbmin\]>/${SIMPARS[13]}/"␣␣\
␣␣␣␣␣␣␣-e␣"s/<\[redqbmax\]>/${SIMPARS[14]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[redqsize\]>/${SIMPARS[15]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[normalrandom\]>/${SIMPARS[16]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[picontrolleralpha\]>/${SIMPARS[17]}/"␣\
␣␣␣␣␣␣␣-e␣"s/<\[randomseed\]>/${SIMPARS[18]}/"␣\
␣␣␣␣␣>␣"$WORKINGDIR/simtemplate tcl"


␣␣␣␣printReport␣"$RUNID $COUNT \"$WORKINGDIR\" \"$TRFILENAME\" "
␣␣␣␣printReport␣"${SIMPARS[10]} "

␣␣␣␣cd␣$WORKINGDIR
␣␣␣␣#␣run␣simulation
␣␣␣␣printf␣"   running simulation\n"
␣␣␣␣$SHELL␣-c␣"ns ' /simtemplate tcl' > ' /output txt'"

␣␣␣␣#␣create␣trace␣deviation␣files
␣␣␣␣#␣-␣calcualted
␣␣␣␣printf␣"   processing output\n"
```

```
␣␣␣␣cat␣" /traces/var_tcpet tr"␣|␣$BATCHDIR/tools/awkTraceProcess
␣␣␣␣printf␣"data cleared for space\nTrace files should have been generated"␣>␣" /traces/var_tcpet tr"

␣␣␣␣$BATCHDIR/tools/errorFromTrace␣" /traces/$TRFILENAME"␣" /traces/datarate_trace txt"␣\
␣␣␣␣␣␣␣␣␣␣␣␣>␣" /traces/err_trace_tcpet txt"

␣␣␣␣printReport␣"$($BATCHDIR/tools/errorFromTrace_stat ' /traces/err_trace_tcpet txt') "

␣␣␣␣#␣--at␣sink
␣␣␣␣$BATCHDIR/tools/errorFromTrace␣" /traces/$TRFILENAME"␣" /traces/mon_atsink txt"␣\
␣␣␣␣␣␣␣␣␣␣␣␣>␣" /traces/err_trace_sink txt"
␣␣␣␣printReport␣"$($BATCHDIR/tools/errorFromTrace_stat ' /traces/err_trace_sink txt') \n"


␣␣␣␣graph_views

␣␣␣␣cd␣$BATCHDIR

␣␣␣␣printf␣"   done\n\n"

␣␣␣fi
␣␣fi
done␣<␣"$SIMSOURCE"

exit␣0
```

APPENDIX B

SELECT EXPERIMENT DATA

Loss to Network: Raw Data

Table B.1 shows the total and per-Stunt throughput lost to the network due to the Round-Robin distribution algorithm without correction for "extra loss." The extra loss is defined as reductions in throughput that are not necessary with regard to compensating for the Free flows required reduction on a delegation event. The delegation results in a lowering of the combined Free and Stunt throughput because loss is delegated to a greater degree than necessary.

Table B.1: *Total and Per-Stunt extra loss as the number of stunts is increased*

Stunts	Total KB	Per-Stunt KB	Stunts	Total KB	Per-Stunt KB
1	85.22632576	85.22632576	14	205.8129735	14.70092668
2	174.7433712	87.37168561	15	325.6590909	21.71060606
3	129.4076705	43.13589015	16	342.1576705	21.3848544
4	281.6193182	70.40482955	17	312.5800189	18.38705994
5	312.9230587	62.58461174	18	246.8252841	13.71251578
6	201.7291667	33.62152778	19	366.3011364	19.27900718
7	166.7391098	23.81987284	20	455.6657197	22.78328598
8	289.1879735	36.14849669	21	265.3385417	12.63516865
9	183.9346591	20.43718434	22	300.6171875	13.66441761
10	346.5300663	34.65300663	23	342.6912879	14.89962121
11	352.2473958	32.02249053	24	257.8297822	10.74290759
12	388.1740057	32.34783381	25	205.415483	8.216619318
13	320.0125473	24.6163498			

Throughput Comparisons: Raw Data

Table B.3 shows 25 experiments in which the delegation is turned off. Table B.2 shows 25 experiments in which delegation is turned on. In each group of experiments the number of stunts is increased from 1 to 25. The network topologies are exactly the same.

Table B.2: *Experiments with delegation as the number of stunts is increased*

Stunts	Free	Avg Stunts	Total Stunt	F/S Total
1	633389.7374	505370.3434	505370.3434	1138760.081
2	688628.7677	365382.5859	730765.1717	1419393.939
3	722936.404	342340.7677	1027022.303	1749958.707
4	782178.8283	286666.8283	1146667.313	1928846.141
5	790497.8586	310581.2525	1552906.263	2343404.121
6	812639.1919	316059.4343	1896356.606	2708995.798
7	813904.8889	303582.8802	2125080.162	2938985.051
8	789683.798	295946.2828	2367570.263	3157254.061
9	772845.0101	310646.303	2795816.727	3568661.737
10	806823.6768	287434.6101	2874346.101	3681169.778
11	816218.8283	280976.5142	3090741.657	3906960.485
12	824259.0707	267309.1919	3207710.303	4031969.374
13	768891.798	270742.4646	3519652.04	4288543.838
14	779864.8889	272270.2222	3811783.111	4591648
15	763778.8283	255177.5515	3827663.273	4591442.101
16	785357.0101	244197.3283	3907157.253	4692514.263
17	781392.6465	236568.3185	4021661.414	4803054.061
18	775292.7677	229238.3838	4126290.909	4901583.677
19	775616.1616	226643.2111	4306221.01	5081837.172
20	773720.404	209702.6949	4194053.899	4967774.303
21	699691.0707	217025.3506	4557532.364	5257223.434
22	717176.6465	209173.4288	4601815.434	5318992.081
23	759965.0101	198802.3434	4572453.899	5332418.909
24	651260.0404	201114.4949	4826747.879	5478007.919
25	753023.1919	189054.4937	4726362.343	5479385.535

Table B.3: *Experiments without distribution as the number of stunts is increased*

Stunts	Free	Avg Stunts	Total	Fair
1	659205.4949	566826.3434	1226031.838	613015.9192
2	563938.101	517196.5253	1598331.152	532777.0505
3	471101.7374	470456.8081	1882472.162	470618.0404
4	433967.1919	445814.2828	2217224.323	443444.8646
5	436504.1616	445466.6343	2663837.333	443972.8889
6	413448.404	417019.6768	2915566.465	416509.4949
7	388034.101	388813.114	3109725.899	388715.7374
8	410794.3434	380323.5253	3453382.545	383709.1717
9	377049.8586	375551.2189	3757010.828	375701.0828
10	332031.1919	370398.5374	4036016.566	366910.5969
11	347860.7677	356345.55	4267661.818	355638.4848
12	331021.9798	341536.4646	4429459.556	340727.6581
13	338214.7071	329078.6138	4616236.687	329731.1919
14	297573.0101	321773.3911	4802400.485	320160.0323
15	340935.6768	305598.7556	4924917.01	307807.3131
16	299769.8586	296444.6162	5042883.717	296640.2187
17	268417.3737	285571.6839	5123136	284618.6667
18	244502.9495	272768.3232	5154332.768	271280.672
19	278292.0404	272559.8682	5456929.535	272846.4768
20	248261.0101	259305.7495	5434376	258779.8095
21	267090.3434	250563.798	5528930.101	251315.0046
22	264062.7071	243761.8806	5626824.081	244644.5253
23	231154.5859	237051.3131	5683334.788	236805.6162
24	221224.1616	230033.3939	5742025.616	229681.0246
25	224039.9192	218627.6428	5689730.99	218835.8073

BIBLIOGRAPHY

Appenzeller, G., Keslassy, I., and McKeown, N. (2004). Sizing Router Buffers. In *Proceedings of ACM SIGCOMM'04*, Portland, Oregon.

Balakrishnan, H., Rahul, H., and Seshan, S. (1999). An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of ACM SIGCOMM'99*, Cambridge, MA.

Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and Weiss, W. (1998). An Architecture for Differentiated Services. *IETF RFC 2475.*

Braden, R., Clark, D., and Shenker, S. (1994). Integrated Services in the Internet Architecture: an Overview. *RFC 1633.*

Cerf, V. and Kahn, L. (1974). A Protocol for Packet Network Interconnections. *IEEE Transactions on Communications.*

E. Amir, e. a. (2007). UCB/LBNL/VINT Network Simulator - ns (version 2). Available at http://nsnam.isi.edu/nsnam/index.php/.

Floyd, S. and Jacobson, V. (1993). Random Early Detection Gateways for Congestion Avoidance. *Transactions on Networking*, 1(4):397–413.

Guirguis, M., Bestavros, A., Matta, I., Riga, N., Diamant, G., and Zhang, Y. (2004). Providing Soft Bandwidth Guarantees Using Elastic TCP–based Tunnels. In *In proceedings of the 9th IEEE Symposium on Computer and Communications (ISCC'2004)*, Alexandria, Egypt.

Hollot, C., Misra, V., Towsley, D., and Gong, W. (2001). A Control Theoretic Analysis of RED. In *Proceedings of IEEE INFOCOM 2001*, Anchorage, AL.

Jacobson, V. (1988). Congestion Avoidance and Control. In *Proceedings of ACM SIGCOMM'98*, Stanford, CA.

Kelly, F. (2001). Mathematical Modelling of the Internet. *Mathematics Unlimited - 2001 and Beyond*, pages 685–702.

Kurose, J. F. and Ross, K. W. (2008). *Computer Networking: A Top-Down Approach 4th Edition.* Addison-Wesley, Boston, MA, USA.

Low, S., Paganini, F., Wang, J., Adlakha, S., and Doyle, J. (2002). Dynamics of TCP/RED and a Scalable Control. In *Proceedings of IEEE INFOCOM 2002*, New York, NY.

Mathis, M., Semke, J., and Mahdavi, J. (1997). The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communications Review*, 27(3):67–82.

Molenkamp, G., Katchabaw, M , Lutfiyya, H., and Bauer, M. (2000). Managing Soft QoS Requirements in Distributed Systems. In *Proceedings of ICPP Workshop*, Toronto, Canada.

Ott, D. and Mayer-Patel, K. (2007). An Open Architecture for Transport-level Protocol Coordination in Distributed Multimedia Applications. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 3(3).

Saltzer, J. H., Reed, D. P., and Clark, D. D. (1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288.

Shenker, S. (1990). A Theoretical Analysis of Feedback Flow Control. In *Proceedings of ACM SIGCOMM'90*, Philadelphia, PA.

Valdez, J. and Guirguis, M. (2008). Liberating TCP: The Free and the Stunts. *Computer Science Department, Texas State University-San Marcos.*

Zhang, L., Shenker, S., and Clark, D. D. (1991). Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. In *Proceedings of ACM SIGCOMM'91*, Zurich, Switzerland.

VITA

Jason Valdez was born in Lubbock, Texas, to Thomas and Margie Valdez on January 4, 1981. He completed his undergraduate study at Midwestern State University in Wichita Falls, Texas, in late 2003. Moving to Austin entailed a year off while he found work and settled into what was a much larger city than he was used to. Graduate study at Texas State University-San Marcos began in January of 2005. Jason is currently residing in Austin, Texas and working for Unisys Technical Services.

Permanent Address: 2410 Robertson Dr
Abilene, Texas 79606

This thesis was typed by Jason D Valdez.