# HIGH-PERFORMANCE NETWORKS: OPTIMIZING QUALITY OF SERVICE USING TCP AUTO TUNING

By

**HITESH NIMBARK**

**(07MCE006)**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**AHMEDABAD-382481**

**May 2009**

# HIGH-PERFORMANCE NETWORKS: OPTIMIZING QUALITY OF SERVICE USING TCP AUTO TUNING

**Major Project**

Submitted in partial fulfillment of the requirements

For the degree of

**Master of Technology in Computer Science and Engineering**

By

**Hitesh Nimbark**

**(07MCE006)**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**AHMEDABAD-382481**

**May 2009**

# Certificate

This is to certify that the Major Project entitled ”High Performance Network: Optimizing Quality of Service Using TCP Auto Tuning” submitted by Hitesh Nimbark (07MCE006), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof Priyanka Sharma
Guide, Assistant Professor,
Department Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr S N Pradhan
Co-Guide, Professor,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Prof D J Patel
Professor and Head,
Department Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Dr K Kotecha
Director,
Institute of Technology,
Nirma University,
Ahmedabad

# Abstract

TCP has become the dominant protocol for all network data transport because it presents a simple uniform data delivery service. With the growth of high performance networking a single host may have simultaneous connections that vary in bandwidth. Consequently, connections on a single host can and will scale in bandwidth by several orders. TCP, now, requires queues proportional in size to a path's bandwidth. Traditional statically fixed mechanisms for allocating and limiting TCP queue space fall short on today's Internet, and often limit throughput only a small fraction of the available bandwidth specially for long- fat network (large bandwidths and high round-trip times)

Subsequent work on TCP has enabled the use of larger flow-control windows, yet the use of these options is still relatively rare, because manual tuning has been required. Other work has developed means for avoiding this manual tuning step, but those solutions lack generality and exhibit unfair characteristics.

We have identified requirements for an automatically tuning TCP to achieve maximum throughput across all connections simultaneously within the resource limits of the sender. The aim of the thesis work is to upgrade the TCP implementation of Linux kernel 2.6.26.5 to remove above barriers while using memory efficiently. This technique results in greatly improved performance, a decrease in packet loss under bottleneck conditions, and greater control of buffer utilization by the end hosts.

## Acknowledgements

With immense pleasure, I would like to present this report on the dissertation work related to "High-Performance Networks: Optimizing Quality of Service Using TCP Auto Tuning". I am very thankful to all those who helped me for the successful completion of the dissertation and for providing valuable guidance throughout the project work.

I would first of all like to thank Dr S N Pradhan, Professor In Charge M Tech(CSE), Institute of Technology, Nirma University, Ahmedabad whose keen interest and excellent knowledge base helped me to finalize the topic of the dissertation work. His constant support and interest in the subject equipped me with a great understanding of different aspects of the required architecture for the project work.

My kind and sincere thanks and gratitude to Dr Ketan Kotecha, Director Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and motivation throughout the Dissertation work. My sincere thanks and gratitude to Prof D J Patel, Professor and Head Computer Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and motivation throughout the Dissertation work.

I would also like to thank Prof Priyanka Sharma, Assistant Professor, CE Department, Nirma Institute of Technology, Nirma University, whose knowledge and expertise in Computer Networks has been a great pillar of support for my work. She has shown keen interest in this dissertation work right from beginning and has offered a great motivating factor in outlining the flow of my work.

The blessings of God, faculty members and my family members makes the way for completion of major project. I am very much grateful to them.

**- Hitesh Nimbark**
**(07MCE006)**

# Contents

**Certificate** **iii**

**Abstract** **iv**

**Acknowledgements** **v**

**List of Tables** **viii**

**List of Figures** **ix**

**Abbreviation** **x**

**1 Introduction** **1**
1.1 General Overview . . . . . . . . . . 1
1.2 Motivation . . . . . . . . . . 2
1.3 Objective . . . . . . . . . . 3
1.4 Scope of Work . . . . . . . . . . 3
1.5 Organization of Thesis . . . . . . . . . . 4

**2 Literature Survey** **5**
2.1 TCP's Reliable Delivery . . . . . . . . . . 5
2.2 Round Trip Time Components . . . . . . . . . . 6
2.2.1 Transmission time . . . . . . . . . . 6
2.2.2 Propagation delay . . . . . . . . . . 6
2.2.3 Routing queues . . . . . . . . . . 6
2.2.4 Processing time . . . . . . . . . . 6
2.3 Windows . . . . . . . . . . 7
2.3.1 Receive window . . . . . . . . . . 7
2.3.2 Retransmit queue . . . . . . . . . . 8
2.3.3 Congestion window . . . . . . . . . . 8
2.3.4 Sending application . . . . . . . . . . 8
2.4 The Bandwidth Delay Product . . . . . . . . . . 8
2.5 Flow control . . . . . . . . . . 9
2.5.1 Using Large sizing receive buffer . . . . . . . . . . 9

2.5.2 Appropriately sizing the receive buffer . . . . . . . . . . . . . 10
2.5.3 Actual TCP Bandwidth Delivered to the Application . . . . . 11
2.6 Uncooperative Network Application Behavior . . . . . . . . . . . . . 12
2.7 Tuning Methodology and other sources of poor Network Performance 13
2.8 Linux /proc file system . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.8.1 Networking info in /proc/net . . . . . . . . . . . . . . . . . . 17
2.9 Web100 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**3 TCP Tuning Domains 22**
3.1 Tuning Domains . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.2 TCP Queuing System Model . . . . . . . . . . . . . . . . . . . . . . 24
3.2.1 Purpose of TCP Tuning . . . . . . . . . . . . . . . . . . . . . 26
3.2.2 TCP Packet Processing . . . . . . . . . . . . . . . . . . . . . . 30
3.3 TCP STREAMS Module Tunable Parameters . . . . . . . . . . . . . 31
3.4 TCP TUNING SLIDING WINDOWS . . . . . . . . . . . . . . . . . . 33
3.5 TCP Tuning for Acknowledge Control . . . . . . . . . . . . . . . . . 35

**4 TCP Tuning State Model 37**
4.1 TCP State Engine . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.1.1 Connection Setup . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.1.2 Connection Established . . . . . . . . . . . . . . . . . . . . . . 39
4.1.3 Connection Shutdown . . . . . . . . . . . . . . . . . . . . . . 39
4.2 TCP Tuning on the Sending and Receiving Side . . . . . . . . . . . . 40

**5 Implementation 44**
5.1 User Mode Linux . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.2 Compiling Linux Kernel . . . . . . . . . . . . . . . . . . . . . . . . 46
5.3 TCP Extension for High Performance Network . . . . . . . . . . . . 48
5.4 Proposed approach . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.4.1 Proposed Algorithm . . . . . . . . . . . . . . . . . . . . . . . 52

**6 Testing and Analysis 56**
6.1 Testing and Analysis tools . . . . . . . . . . . . . . . . . . . . . . . 56
6.2 Test-bed setup and Experiments . . . . . . . . . . . . . . . . . . . . 57

**7 Conclusion and Future Scope 61**
7.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
7.2 Future Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

**References 63**

# List of Tables

I CPU Info in /proc/cpuinfo. . . . . . . . . . . . . . . . . . . . . . . . . . 17
II Network info in /proc/net. . . . . . . . . . . . . . . . . . . . . . . . . . . 18

# List of Figures

2.1 Web100 Implementation structure. . . . . . . . . . . . . . . . . . . . . . 20

3.1 Overview of Overlapping Tuning Domains. . . . . . . . . . . . . . . . . 23
3.2 Closed-Loop TCP System Model. . . . . . . . . . . . . . . . . . . . . . 25
3.3 Perfectly Tuned TCP System. . . . . . . . . . . . . . . . . . . . . . . 27
3.4 Tuning Required to Compensate for Faster Links. . . . . . . . . . . . 28
3.5 Tuning Required to Compensate for Slower Links. . . . . . . . . . . . 28
3.6 Complete TCP Stack on Computing Nodes. . . . . . . . . . . . . . . . 31
3.7 TCP and STREAM Head Data Structures Tunable Parameters. . . . 32
3.8 TCP Tuning for ACK Control. . . . . . . . . . . . . . . . . . . . . . . 36

4.1 TCP Tuning for ACK Control. . . . . . . . . . . . . . . . . . . . . . . 38
4.2 TCP Startup Phase. . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

5.1 User Mode Linux working stack. . . . . . . . . . . . . . . . . . . . . . 45
5.2 Implementation Scenario. . . . . . . . . . . . . . . . . . . . . . . . . 52

6.1 Test-bed setup. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.2 Throughput Vs Window size. . . . . . . . . . . . . . . . . . . . . . . . 59
6.3 Throughput before window tuning. . . . . . . . . . . . . . . . . . . . 59
6.4 Throughput after window tuning. . . . . . . . . . . . . . . . . . . . . 60
6.5 Tuned Vs Un-tuned connection window size. . . . . . . . . . . . . . . 60

# Abbreviation

| | |
|---|---|
| TCP | Transmission Control Protocol |
| FTP | File Transfer Protocol |
| Qos | Quality of Service |
| RTT | Round Trip Time |
| BW | Bandwidth |
| BDP | Bandwidth Delay Product |
| STREAMS | Data Flow in pipe |
| /proc | Linux Virtual File System |
| RED | Random Early Detection |
| MTU | Maximum Transmission Unit |
| MSS | Maximum Segment Size |
| ATM | Asynchronous Transmission Mode |
| KIS | Kernel Instrumentation Set |
| UML | User Mode Linux |
| SYN | Synchronization |
| ACK | Acknowledgement |
| API | Application Package Interface |
| DoS | Denial of Service |

# Chapter 1

# Introduction

This chapter covers general overview of the Thesis work. It include the motivation, objectives, and the scope of the Thesis work. It also guide about the organization of the Thesis report.

## 1.1 General Overview

The congestion control algorithms [1] and large window extensions in TCP permit a host running a single TCP stack to support concurrent connections across the entire range of bandwidth. In principle, all application programs that use TCP should be able to enjoy the appropriate share of available bandwidth on any path without involving manual configuration by the application, user, or system administrator. While most would agree with such a simple statement, in many circumstances TCP connections require manual tuning to obtain respectable performance.

For a given path, TCP requires at least one bandwidth-delay product of buffer space at each end of the connection. Because bandwidth-delay products in the Internet can span 4 orders of magnitude, it is impossible to configure default TCP parameters on a host to be optimal for all possible paths through the network. It is possible for someone to be connected to an FTP server via a 9600bps modem while someone else is connected through a 100Mbps bottleneck to the same server. An experienced system administrator can tune a system for a particular type of connection, but then performance suffers for connections that exceed the expected bandwidth-delay product. Since there are often many more "small" connections than "large" ones, system-wide tuning can easily cause buffer memory to be inefficiently utilized by more than an order of magnitude.

Tuning knobs are also available to applications for configuring individual connection parameters. However, use of the knobs requires the knowledge of a networking expert, and often must be completed prior to establishing a connection. Such expertise is not normally available to end users, and it is wrong to require it for applications and users. Further, even an expert cannot predict changes in conditions of the network path over the lifetime of a connection.

Finally, static tuning configurations do not account for changes in the number of simultaneous connections. As more connections are added, more total memory is used until memory buffer exhaustion occurs, which can ultimately cause the operating system to crash. The Thesis work proposes a system for adaptive tuning of buffer sizes based upon network conditions and system memory availability. It is intended to operate transparently without modifying existing applications. Since it does not change TCP's Congestion Avoidance characteristics, it does not change TCP's basic interactions with the Internet or other Internet applications.

## 1.2 Motivation

Fire up FTP on a greater then 100-Mbps national network between two supercomputers or other high-performance hosts and you get 10-Mbps if we are lucky. Networks aren't usually the problem. Now a days Universities are frequently connected at least at T3 (45-Mbps) or higher Campus LANs and host connections for researchers are often capable of 1-Gbps. The problems are usually in the host-software. This is the so-called "bandwidth-delay-product" issue. Poor and inconsistent TCP implementations among operating system vendors and obscure programmatic interfaces (APIs) such as sockets that are difficult to use and lead to error/bad-performance prone user-codes, poor network application performance and changing in bandwidths.

The Market hasn't worked on the same because E-commerce development soaking up venture capita, vendors are focused on making money now, and often don't take the long-term view. The mass-market is often the major driver of high-technology and not vice-versa, so the solutions require uniformity to work well, and vendors tend to compete and not cooperate when strong standards don't exist.

A comprehensive solution need an approach that fixes all host-software problems. Simply fixing this problem or that in isolation won't yield high-performance results, Network researchers need to agree on what a comprehensive host-software solution is. Operating system and other host-software vendors need to be induced to consistently install this comprehensive solution, Need a common vision that researchers, policy makers, and vendors can all relate to demonstrate success with applications that everyone can relate too.

## 1.3 Objective

The main objective of the Thesis work is to work on the architectural and Quality of Service issues with high performance networks. It also summarized as follow.

- The QoS has to suffice the requirements of the real time traffic.
- The QoS issues covered will be for both wired and wireless networks.
- Various kinds QoS supports available and extensible from Linux kernels.
- Every enhancement should be based on widely existing TCP stack environment

## 1.4 Scope of Work

High performance networking is the need for today's growing demand to support large volumes of real time IP traffic on networks. The area addresses both wired and wireless networking environment. The detail about the thesis work we have discussed in sub coming chapters.

## 1.5 Organization of Thesis

The Thesis covers the TCP and Quality of Service for High Performance Network from understanding level to the research level.

**Chapter 2**, *Literature Survey* , This Chapter covers the literature survey on BDP (Bandwidth delay product), TCP Windowing, and the benefit of the optimization of the network connection buffering. It also cover the congestion control behavior regarding the tuning. This is not just an introductory chapter on TCP terminology.

**Chapter 3**, *TCP Tuning Domains*, This Chapter covers the detail of TCP tuning domains and dependability among them. It not only gives the TCP tuning domain concept but also provides scope of tuning TCP.

**Chapter 4**, *TCP Tuning State Model*, This Chapter covers the TCP state model and explain at each and every state the scope of TCP tuning.

**Chapter 5**, *Implementation*, This is an important chapter. It covers the proposed approach to achieve the maximum network throughput. The chapter also covers the brief explanation of working environment setup, analysis tools, proposed methodology and proposed algorithm of the Thesis work.

**Chapter 6**, *Testing and Analysis*, This chapter gives the idea of Qos parameter measurement and analysis. It provides frame work for the proposed test-bed. Chapter cover the analysis and results of proposed experimental approach.

**Chapter 7**, *Conclusion and Future Scope*, This chapter includes conclusion and future scope of the Thesis work.

# Chapter 2

# Literature Survey

This chapter covers the brief explanation of the TCP relevant to the thesis work. It covers the TCP Windows and it's characteristic, Flow control, Virtual /proc file system, and web100. This is not just an overview of the TCP. This should be entirely reviewed for a reader familiar with the workings of TCP and the kernel role in the same.

## 2.1 TCP's Reliable Delivery

TCP's primary function as a transport protocol is ensuring reliable in-order delivery of data from one host to another over a packet network which may drop or reorder packets. In order to meet this requirement, TCP creates connections, sets of state kept at both end hosts containing information about the progress of the data transfer. A data stream sent through a TCP connection is assigned a sequence space so that each 8-bit byte is numbered sequentially. The stream is divided into segments so that it can be put in packets. A receiving host will cumulatively acknowledge data it has received by sending the sequence number of the last in-order byte it has received. A receiver should queue out-of-order data until the holes in the stream are filled. This data is stored in the reassembly queue. A sender must be prepared to retransmit any unacknowledged data since the network may have dropped any of its outstanding packets. This data is stored in the retransmit queue.

## 2.2 Round Trip Time Components

A round-trip time (RTT) between two hosts is defined as the length of time between when an end host sends a packet and when it receives a reply from the other end host. This total time consists of four parts.

### 2.2.1 Transmission time

The length of time the interface takes to send the packet. This will likely have a linear dependence on packet size.

### 2.2.2 Propagation delay

The time taken for a signal to propagate through the network medium. This is equal to distance between endpoints divided by propagation speed (bounded by and often close to the speed of light).

### 2.2.3 Routing queues

Time spent in routing queues along the path. This will fluctuate over time depending on congestion in the path.

### 2.2.4 Processing time

The amount of time spent by end hosts and routers doing processing necessary for packet delivery. On current hardware, this is usually in the sub-microsecond range.

It is important for the TCP's retransmission and congestion control algorithms to have a good estimate of a connection's RTT. It is sampled by measuring the length of time between transmission of a data segment and reception of its acknowledgment. A host whose end of a TCP connection does not send data will not have a measurement of the RTT.

## 2.3 Windows

A window of data is the amount of data transmitted on a connection during one round-trip time. A TCP sender will have one window of outstanding (unacknowledged) data. Throughput is equal to window/RTT. Since RTT should be relatively constant over a given path, the throughput is approximately proportional to the window size. There are a number of bounds on the window size.

### 2.3.1 Receive window

It may be that a receiving application is not able to process data as quickly as the sending application or the network can transmit it. In this case, it is necessary to have a flow control mechanism to slow the transmission rate of the sender. TCP implements flow control by announcing a receive window in each segment header. This window announced by the receiver is an upper bound on the sender's window size.

A classical TCP implementation maintains a buffer of a static size between itself and the receiving application. It then announces its receive window as the amount of space available in that buffer. To fully open the TCP the receive window should be larger then the congestion window. the receive window is limited by buffer call as rclWnd (receive limit window) at the kernel side. If the application reads slower than data arrives, the buffer will start to fill. As the buffer fills, the window size and therefore throughput will fall (eventually reaching zero when the buffer fills entirely). Throughput will quickly match the application's consumption rate.

### 2.3.2 Retransmit queue

As stated above, the sender must keep all outstanding data in a retransmit queue. If the amount of memory available for this queue is an upper bound on window size.

### 2.3.3 Congestion window

In many cases, the amount of traffic coming into a router is greater than its outgoing bandwidth. A finite quantity of packets may be queued in the router, but in a steady state a certain percentage must be dropped. In this case, the router is said to be a bottleneck and experiencing congestion. One of TCP's functions is to detect this congestion (usually by observing lost packets) and limit its transmission rate accordingly. A TCP sender maintains a congestion window(cWnd), an upper bound on window size based on the observed properties of the network.

### 2.3.4 Sending application

The amount of data the sender produces per round-trip time is yet another bound on window size. If a sender has little or no data to send, the window will be accordingly small. For example, an interactive application such as telnet will likely send between zero and a few segments per round-trip time.

## 2.4 The Bandwidth Delay Product

If a connection's throughput is limited only by the network, its window size will be limited by the congestion window (cwnd). In this case, we ideally have: window = cwnd and throughput = bandwidth. Since throughput = window/RTT, we have bandwidth = cwnd/RTT. Therefore, we have:

$$cwnd = bandwidth * RTT \tag{2.1}$$

This quantity is known as the bandwidth-delay product, or BDP. (It should be noted that this cwnd is the ideal congestion window. In practice, bandwidth is not known and may change over time, so the congestion control mechanism grows and shrinks cwnd as it drives the network into congestion)

The RTT between two endpoints is fundamentally limited by the distance between them and the speed of light. The highest bandwidths, however, have been growing exponentially with time (similar to the way processors follow Moore's "law"). Consequently, BDPs across the Internet are also increasing over time. Further, connection BDPs may vary by many orders of magnitude on a single host. For example, a host connected with 100 Mbps fast Ethernet may connect to a similar host at the same time as it connects to one with a 56 kbps modem

## 2.5 Flow control

The typical ordinary flow control mechanism suffers from one serious problem - the largest possible receive window is the less than the fixed size of the receive buffer. If this maximum window is less than the BDP, the connection can never reach the throughput of which the network is capable. The size of this receive buffer is set_table in most implementations both globally by an administrator, and per-connection by the application; as describe below.

### 2.5.1 Using Large sizing receive buffer

A buffer too big will have an undesirable effect in interactive applications. For example, a telnet user whose server starts to output a large amount of text may wish to hit Control-C to stop the output. The telnet application will have to display a full window of data sent after the Control-C is sent.

It might be possible to improve the large receive window approach by using the congestion control mechanism to do flow control as well. The receiver's buffer could be viewed as a router queue, and it could trigger congestion events (drop packets or use ECN) to pull down the sender's congestion window.

This method has some serious problems, however. For one, if the receiving application completely stops, there will be no way to announce a zero window. If the receiver drops all incoming packets, the sender will try to retransmit until the connection times out and resets. Another problem is that this method would not solve the interactive application problem. A third undesirable property is that it would not respond quickly if the receiving application changed its consumption rate quickly. The congestion control mechanism is designed to conservatively estimate network properties, which are not likely to change rapidly like an web application.

### 2.5.2 Appropriately sizing the receive buffer

If we are to use the receive window for flow control as intended, the announced receive window will ideally be the minimum of the amount of data a receiving application would like per round trip and the amount of data the sender may send per round trip. To calculate such a window, however, a receiver must know the connection's RTT, information not generally available to a receiver. RTT is traditionally observed by the sending side by measuring the time between when a data segment is sent and its corresponding acknowledgment is received. If one end host never sends data on a particular connection, it will have no knowledge of the RTT.

If no RTT sample is available from sending data, web100 [2] we propose a technique for estimating RTT. They use the fact that a sender may not send more than an announced window of data within one RTT of its announcement. Therefore, the time between when an acknowledgment for sequence number s announcing receive window w is sent and a data segment containing sequence number s + w + 1 is received is an upper bound on RTT. They keep a minimum of these sampled upper bounds. If a smoothed RTT from sending is available, that is used; otherwise, the minimum upper bound is used. This measures the amount of data received in a round trip and sizes the receive buffer to fit twice that amount. This measurement and resizing is done once per RTT.

Assuming an accurate RTT measurement, we will ensure that in a steady state, the receive window will not limit window size if the receiver consumes all data quickly. If the receive window limits window for one round trip, it will be doubled on the next round trip. (It is worth noting that this doubling matches ”slow start,” the congestion control state which most quickly grows the congestion window).

In the case where the receiver consumes data slowly, flow control will work. As the receive buffer fills with unread data, the announced window size will shrink, so less data will be sent per RTT. With less data sent, the target buffer size will be calculated smaller, it is twice the size of the amount of data read by the receiver per round trip. Then, in the steady state, the receive window will be the amount of data read per round trip by the application, and an equal amount will be buffered to be read.

### 2.5.3 Actual TCP Bandwidth Delivered to the Application

If a host and application are properly tuned, effects outside the control of the host and application can adversely affect network performance. Limitations on TCP bandwidth arise from the effects of packet loss and packet round trip time on the network path between hosts. The TCP Slow Start and Congestion Control algorithms [3] probe the network path between the sender and receiver to both discover the maximum available transfer capacity of the network and at the same time minimize the effects of overloading the network and causing congestion.

Mathis [4] described the relationship between the upper bound of TCP bandwidth BW, packet round trip time RTT, and packet loss p with the Equation 2.2.

$$BW \leq \frac{(MSS * C)}{RTT * \sqrt{p}} \tag{2.2}$$

To achieve substantial network performance over a wide area network that has a relatively large RTT, the required maximum packet loss rate p must be very low. The relationship derived by Mathis [4] for the maximum packet loss rate required to achieve a target bandwidth is defined by the relationship Equation 2.3.

$$p < \sqrt{\left(\frac{MSS}{BW * RTT}\right)} \tag{2.3}$$

For example, if the minimum link bandwidth between two hosts is OC-12 (622 Mbps), and the average round trip time is 20 msec, the maximum packet loss rate necessary to achieve 66% of the link speed (411 Mbps) is approximately 0.00018%, which represents only two packets lost out of every 100000 packets. Current loss rates on the commercial Internet backbone [4] are on the order of 0.1%, which puts a hard upper limit on the potential bandwidth available to an application.

As the implementation and use of the Random Early Detection (RED) queuing mechanism becomes widely deployed across the Internet, the characteristic of drop-tail queuing mechanism in routers may change. Given the burst behavior of packet loss, obtaining significantly small packet loss rates can be very difficult. Looking at Equation 2.1, it is apparent that increasing the MTU from the usual default value of 1500 bytes to the "jumbo frame" size of 9000 bytes can increase the upper limit on TCP bandwidth by a factor of six. We experience that increasing the MTU by a factor of three from 1500 to 4470 bytes increased TCP throughput by a roughly equivalent factor.

## 2.6 Uncooperative Network Application Behavior

Application developers have learned to overcome poor TCP performance with a toolkit of "bad" (from the network administrator's perspective) behaviors.

The first approach usually taken is to abandon the TCP transport service and to rely on UDP along with a transport layer written for the application. In this approach, the application simply transmits packets as fast as it can. If any packets are lost, the application either drops them (as in the case of multimedia applications), or performs packet retransmission on an application level. This approach is considered "bad" for several reasons. If an application is injecting UDP packets into the network at a high rate, the network infrastructure has no way of signaling back to the application that the flow is congesting the network and affecting other users of the network. If the other users of the network are being "good" and using TCP for their connection, the UDP stream is able to take an unfair share of the available network bandwidth [5].

The second approach is to open parallel TCP network sockets between applications, and utilize software controlled striping of the data across the sockets, similar to disk striping [6]. This approach attempts to take more than the host's normal share of network bandwidth from other users of the network to deliver a higher aggregate network bandwidth to the end hosts. When there is a significant amount of random packet loss experienced by the end hosts that is not due to congestion, parallel TCP sockets can be a fair and effective method to improve aggregate throughput.

## 2.7 Tuning Methodology and other sources of poor Network Performance

Even if the end hosts are properly tuned and the network packet loss rate is acceptable, other factors may come into play that can adversely affect network performance. Each of these factors should be considered in turn when diagnosing poor network performance, since a fault at a lower layer will affect performance in all of the layers above it. The third novel method we proposed in chapter 5.

In the Physical Layer, network cables that are not within specification limits can be a significant source of poor performance. A general rule of thumb is that Cat-5 cables are good for 10-BaseT, Cat-5 enhanced cables (Cat-5e) are good for 100-BaseT, and Cat-6 cables are good for gigabit Ethernet over copper. Network adapters are very good at getting around bad cables by decreasing their throughput or using data-link layer CRC correction to compensate for a cable that is operating below specification. An additional source of problems are host network adapters that are configured to operate at half-duplex mode rather than full-duplex mode. If both the network switch and the network adapters support full-duplex transfers, both sides should be set to full duplex. If excessive losses are encountered in full-duplex mode, the cabling between the host and switch should be tested or replaced.

In the Data Link Layer, there are several potential sources of problems. First, if the maximum transmission unit size (MTU) for packets is set too low, TCP connections will suffer from poor performance. On 10 and 100 Mb/sec Ethernet, all adapter cards enforce a 1500 byte MTU limit. On some Gigabit Ethernet cards, the MTU can be set to a "jumbo size" 9000 byte frame. If we look back at Equation 2.1, it's apparent that increasing the MTU size (which is MSS + IP header) by a factor of six can increase TCP bandwidth by a factor of six! Unfortunately, most network switches and routers have a hard 1500 byte MTU limit that cannot be changed.

On the host side, another source of problems in the Data Link Layer is the number of CPU interrupts per second that are required to service the network adapter. If a transfer is occurring at gigabit Ethernet speeds, with a limited 1500 byte MTU, the network adapter and CPU must service over 83,000 packets per second. If the network adapter requires service from the CPU for a small number of packets, the CPU will be overwhelmed with servicing network adapter interrupts [7]. The device driver must be configured to permit an appropriate degree of packet coalescing to take advantage of the network adapter's packet buffer. Additionally, the size of the transmission queue in the operating system (txqueuelen in Linux) can affect the packet loss rate on the host. Finally, the PCI slot where the network adapter card is placed can have an impact on performance. Some motherboards, such as the Intel L440GX+[40], have a dual PCI bus architecture, with specialized PCI slots that are enhanced for specific functions (such as RAID adapters). If a host contains RAID adapters along with network adapters, improper adapter card placement can have an impact on the aggregate performance of the complete system.

In the Network Layer, there are several potential sources of problems. First, excessive packet loss and round trip time affects TCP bandwidth as described above in Equation 2.2. Second, there may be network configuration errors that forces traffic through an inadequate data link, or that adds unnecessary additional hops in the path between the hosts. This problem can be especially difficult to diagnose if IP encapsulation (such as AAL5 for IP over ATM) occurs on the network path, since IP based network tools (such as traceroute) do not have the ability to adequately penetrate an ATM cloud to diagnose ATM problems.

In the Transport Layer, mistimed host TCP options are a very common source of problems. we have mostly worked on same. Finally, the network I/O characteristics of the application can dramatically impact TCP performance. Application developers should consider multithreading their applications to decouple network I/O from computation. The process of examining the network from the Physical Layer up to the Application Layer represents an orderly methodology that should be followed when attempting to diagnose and correct network performance problems. It is important to note that if a problem exists at a lower layer in the network, such as the physical layer, efforts directed at tuning components at a higher layer to improve performance may not deliver the expected results. For example, if a physical link is improperly configured to operate at half duplex, attempts to increase performance by optimizing the end-to-end network path may yield little if any results. Thus, when diagnosing application network performance problems, it is important to make sure that tuning opportunities at each layer are explored.

## 2.8 Linux /proc file system

The Linux kernel has two primary functions: to control access to physical devices on the computer and to schedule when and how processes interact with these devices. The /proc/ directory contains a hierarchy of special files which represent the current state of the kernel allowing applications and users to peer into the kernel's view of the system.

Within the /proc/ directory, one can find a wealth of information detailing the system hardware and any processes currently running. In addition, some of the files within the /proc/ directory tree can be manipulated by users and applications to communicate configuration changes to the kernel.

It permits a novel approach for communication between the Linux kernel and user space. Many elements of the kernel use it both to report information and to enable dynamic runtime configuration. Loadable Kernel Modules (LKM) they're a novel way to dynamically add or remove code from the Linux kernel.

Under Linux, all data are stored as files. Most users are familiar with the two primary types of files: text and binary. But the /proc/ directory contains another type of file called a virtual file. It is for this reason that /proc/ is often referred to as a virtual file system.

These virtual files have unique qualities. Most of them are listed as zero bytes in size and yet when one is viewed, it can contain a large amount of information. In addition, most of the time and date settings on virtual files reflect the current time and date, indicative of the fact they are constantly updated.

Virtual files such as /proc/interrupts, /proc/meminfo, /proc/mounts, and /proc/partitions provide an up-to-the-moment glimpse of the system's hardware. Others, like /proc/filesystems and the /proc/sys/ directory provide system configuration information and interfaces.

For organizational purposes, files containing information on a similar topic are grouped into virtual directories and sub-directories. For instance, /proc/ide/ contains information for all physical IDE devices. Likewise, process directories contain information about each running process on the system.

By using the cat, more, or less commands on files within the /proc/ directory, users can immediately access an enormous amount of information about the system. For example, to display the type of CPU a computer has, type cat /proc/cpuinfo to receive output some text shown in Table I. When viewing different virtual files in the /proc/ file system, some of the information is easily understandable while some is not

| *File* | *Contents* |
|---|---|
| processor | 2 |
| vendor_id | AuthenticAMD |
| cpu family | 9 |
| model | 9 |
| model name | AMD-K6(tm) 3D+ Processor |
| stepping | 1 |
| cpu GHz | 2.0 |
| cache size | 512KB |
| fdiv_bug | No |
| fpu_exception | Yes |

Table I: CPU Info in /proc/cpuinfo.

human-readable. This is in part why utilities exist to pull data from virtual files and display it in a useful way. Examples of these utilities include lspci, apm, free, and top.

This is the special /proc file system. Notice that the first field, none, indicates that this file system isn't associated with a hardware device such as a disk drive. Instead, /proc is a window into the running Linux kernel. Files in the /proc file system don't correspond to actual files on a physical device. Instead, they are magic objects that behave like files but provide access to parameters, data structures, and statistics in the kernel. The "contents" of these files are not always fixed blocks of data, as ordinary file contents are. Instead, they are generated on the fly by the Linux kernel when you read from the file. You can also change the configuration of the running kernel by writing to certain files in the /proc file system. We are more interested in proc/net file system the some introductory text shown in Table II.

### 2.8.1 Networking info in /proc/net

One can use this information to see which network devices are available in their system and how much traffic was routed over those devices using following Linux command.

```
% cat /proc/net/dev
```

| *File* | *Contents* |
|---|---|
| arp | Kernel ARP table |
| dev | network devices with statistics |
| dev_stat | network device status |
| ip_fwchains | Firewall chain linkage |
| ip_fwnames | Firewall chain names |
| ip_masq | Directory containing the masquerading tables |
| netstat | Network statistics |
| raw | raw device statistics |
| route | Kernel routing table |
| rpc | Directory containing rpc info |
| rt_cache | Routing cache |
| tcp | TCP sockets |
| tr_rif | Token ring RIF routing table |
| igmp | IP multicast addresses, which this host joined |
| netlink | List of PF_NETLINK sockets |

Table II: Network info in /proc/net.

In addition, each Channel Bond interface has it's own directory. For example, the bond0 device will have a directory called /proc/net/bond0/. It will contain information that is specific to that bond, such as the current slaves of the bond, the link status of the slaves, and how many times the slaves link has failed.

## 2.9 Web100

An application developer or systems administrator can make use of a combination of these tools to diagnose and correct host and application network problems, but there are inherent problems with the measurement methodologies within each tool that must be taken into account.

First, to make a fair estimate of the characteristics of the system under measurement, many measurements and data points must be collected, and systematic sources of error (such as time of day) need to be taken into account to eliminate artificial effects. Second, some of the tools (pchar, for example) require such a long time to run that the results of the measurement may not accurately reflect the current state of the system under measurement. Third, some components of the network path (such as switched ATM clouds) are resistant to IP based measurement techniques. Finally, a high degree of expertise in networking and operating systems is required to realize the full benefits from the use of these tools.

To address these problems, Web100 [2] was developed by a team at Pittsburgh Super computing Center to provide a window into the characteristics of a TCP connection for application developers and systems administrators, and to provide an integrated performance measurement and diagnosis tool. Web100 provides kernel level access to internal TCP protocol variables, settings, and performance characteristics for instantaneous feedback on TCP performance characteristics.

It is fundamentally important to the growth of the net that TCP hides all of the details of the lower layer that should be exposed but not. For example, packet loss is hidden by TCP's retransmission machinery. If packet loss is due to a flowed network, the only symptom will be reduced performance. The inability to easily observe TCP's inner working impairs our ability to conduct research in TCP behavior, test new TC algorithms, educate future protocol researches and detect bugs in TCP and the lower layers.

Many researchers exposed network detail by implementing ad-hoc TCP tools or specially instrumented alternate protocols. This can solve the information hiding problem, but it represents a significant duplication of effort to certify that the ad-hoc implementation are authentic.

The web100 project develop and advanced management interface for TCP. There are instruments for capturing common events such as segments sent and received, as well as many more subtle instrument such as those characterizing the protocol events that cause TCP to reduce its transmission rate. The instruments are collectively referred to as a Kernel Instrument Set(KIS).

There is a separate KIS structure and instance of the protocol stack for each connection. In Figure 2.1 we can see the network protocol stack, which exchanges information directly with KIS, as kernel memory is shared, KIS data is moved between the kernel and user space through the /proc file system interface. Diagnostic and tuning applications use a library to access the file system data. It is possible to implement as SNMP agent on top of the library for remote access to the KIS.

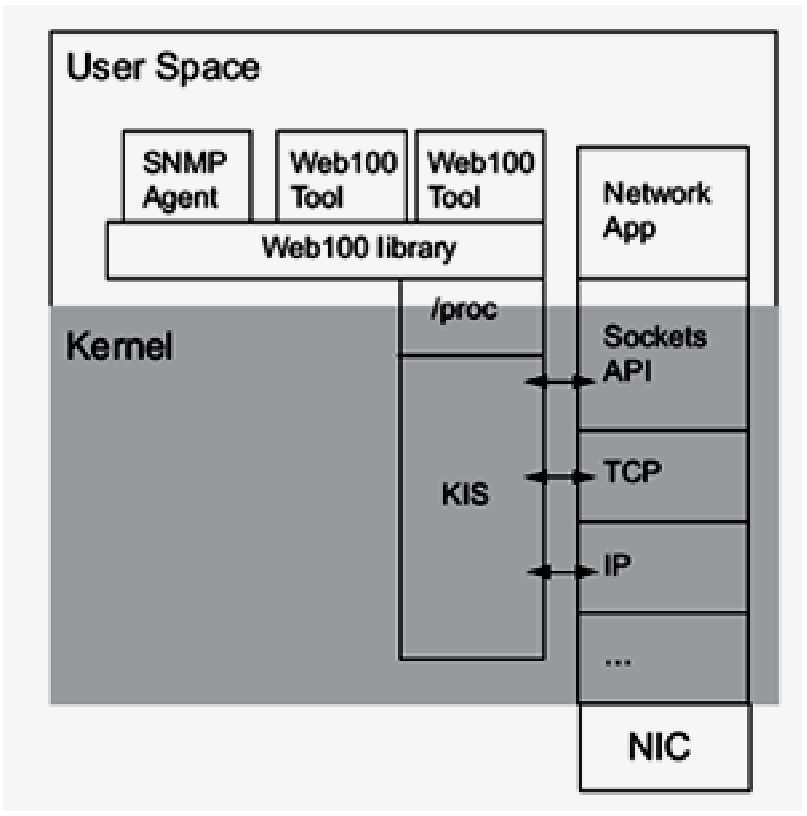


Figure 2.1: Web100 Implementation structure.

The web100 modifications to the kernel collect information about the state of a TCP transfer in a kernel data structure that is linked out of the ”sock” TCP structure in sock.h. First, kernel creates the /proc/web100 directory and the file /proc/web100/header at system boot time. Each new TCP connection is assigned a unique, unchanging number (similar to a pid), and its directory name is that number as ASCII decimal. These directories persist for about sixty seconds after the connection is terminated (goes into a CLOSED or TIME_WAIT state). The connection stats will not change after the connection is terminated. So a connection whose state variable is TIME_WAIT is not necessarily still in TIME WAIT. It should be noted that what is meant by a "connection" here is actually one side of a connection. If a connection is created from the local host to the local host, two connection ID's will be created. When writing an application to read from the /proc interface, it should be taken into consideration that the directories and their files can disappear at any time (they do so at an interrupt level). So if a file open fails on a file you just looked up (say, with glob), that's probably normal and the program should handle it gracefully.

Another seemingly strange thing that can happen is that stats for multiple connections with the same four-tuple can show up. No more than one of the connections may be in any state but CLOSED or TIME WAIT. This behavior is correct, and should be handled as such. The algorithms governing the connection numbers are not yet final. Currently, for simplification, it is only possible to have 32768 connections. Inside each connection directory is an identical set of files. One is spec-ascii, which contains the connection four-tuple in human-readable format.

The remaining files provide access to states of TCP-KIS variables in local host byte-order. Since the number, names, and contents of these files can and will change with releases, they are described in a header file – /proc/web100/header. A file named spec, which contains the variables describing the connection's four-tuple, should be present for any release. one can see the header file by typing

```
% cat /proc/web100/header
```

# Chapter 3

# TCP Tuning Domains

The thesis work describes some key TCP tunable parameters related to performance tuning. More importantly it describes how these tunables work, how they interact with each other, and how they impact network traffic when they are modified. Applications often recommend TCP settings for tunable parameters, but offer few details on the meaning of the parameters and adverse effects that might result from the recommended settings. This chapter is intended as a guide to understanding those recommendations. This chapter is intended for network architects and administrators who have an intermediate knowledge of networking and TCP. The concepts discussed in this chapter build on basic terminology concepts and definitions given by the reference book "Internetworking with TCP/IP Volume 1, Principles, Protocols, and Architectures" by Douglas Comer, Prentice Hall, New Jersey [8].

## 3.1 Tuning Domains

Network architects responsible for designing optimal backbone and distribution IP network architectures for the corporate infrastructure are primarily concerned with issues at or below the IP layer - network topology, routing, and so on. However, in data center networks, servers connect either to the corporate infrastructure or the service provider networks, which host applications. These applications provide net-

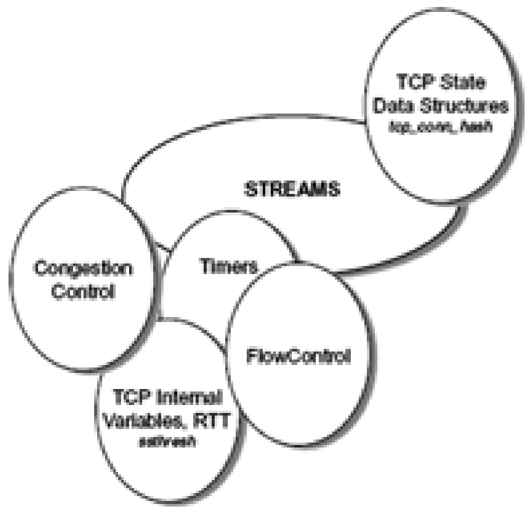


Figure 3.1: Overview of Overlapping Tuning Domains.

worked application services with additional requirements in the area of networking and computer systems, where the goal is to move data as fast as possible from the application out to the network interface card (NIC) and onto the network. Designing network architectures for performance at the data center includes looking at protocol processing above Layer 3, into the transport and application layers. Further, the problem becomes more complicated because many clients' stateful connections are aggregated onto one server. Each client connection might have vastly different characteristics, such as bandwidth, latencies, or probability of packet loss. One has to identify the predominant traffic characteristics and tune the protocol stack for optimal performance. Depending on the server hardware, operating system, and device driver implementations, there could be many possible tuning configurations and recommendations. However, tuning the connection-oriented transport layer protocol is often most challenging.

Transport Control Protocol (TCP) tuning is complicated because there are many algorithms running and controlling TCP data transmissions concurrently, each with slightly different purposes.

Figure 3.1 shows a high-level view of the different components that impact TCP processing and performance. While the components are interrelated, each has its own function and optimization strategy.

- The STREAMS framework looks at raw bytes flowing up and down the streams modules. It has no notion of TCP, congestion in the network, or the client load. It only looks at how congested the STREAMS queues are. It has its own flow control mechanisms.
- TCP-specific control mechanisms are not tunable, but they are computed based on algorithms that are tunable.
- Flow control mechanisms and congestion control mechanisms are functionally completely different. One is concerned with the endpoints, and the other is concerned with the network. Both impact how TCP data is transmitted.
- Tunable parameters control scalability. TCP requires certain static data structures that are backed by non-swappable kernel memory. Avoid the following two scenarios:
  - Allocating large amounts of memory. If the actual number of simultaneous connections is fewer than anticipated, memory that could have been used by other applications is wasted.
  - Allocating insufficient memory. If the actual number of connections exceed the anticipated TCP load, there will not be sufficient free TCP data structures to handle the peak load.

This class of tunable parameters directly impacts the number of simultaneous TCP connections a server can handle at peak load and control scalability.

## 3.2 TCP Queuing System Model

The goal of TCP tuning can be reduced to maximizing the throughput of a closed loop system, as shown in Figure 3.2. This system abstracts all the main components

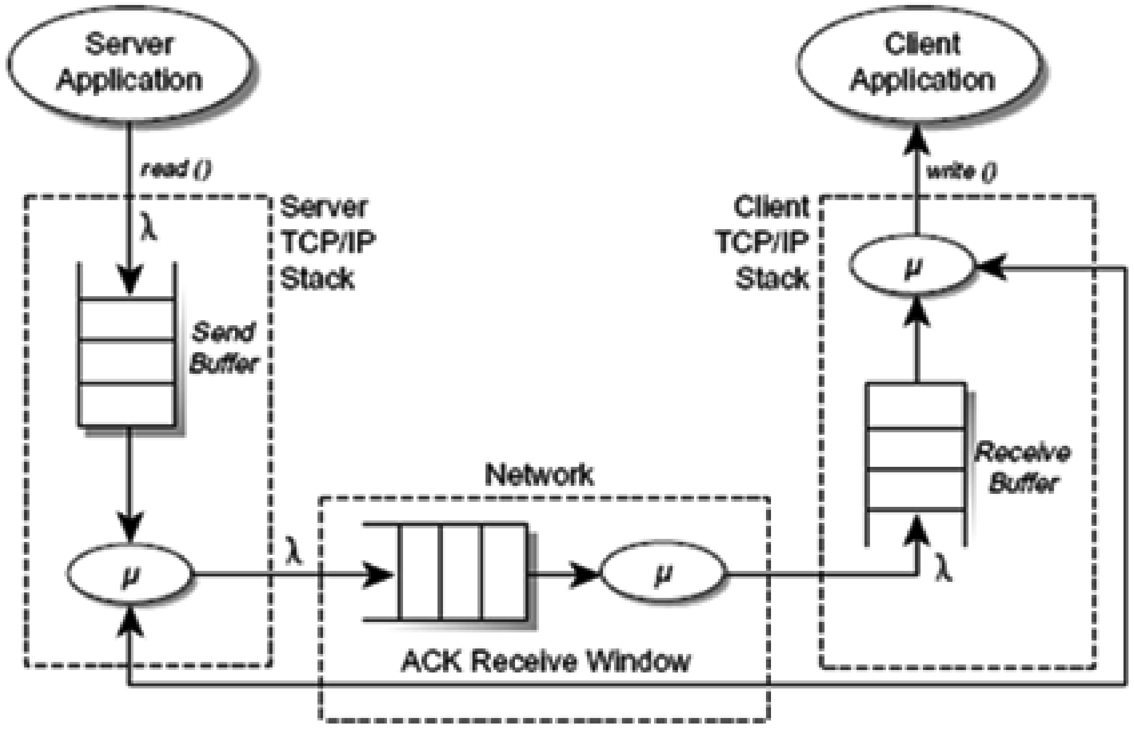


Figure 3.2: Closed-Loop TCP System Model.

of a complete TCP system, which consists of the following components:

- Server - The remote Server endpoint of the TCP connection
- Network - The endpoints can only infer the state of the network by measuring and computing various delays, such as round trip times, timers, receipt of acknowledgments, and so on.
- Client - The remote client endpoint of the TCP connection

This section requires basic background in queuing theory [9]. In Figure 3.2, we model each component as an M/M/1 queue. An M/M/1 queue is a simple queue that has packet arrivals at a certain speed, which we've designated as $\lambda$. At the other end of the queue, these packets are processed at a certain speed, which we've designated as $\mu$.

TCP is a full duplex protocol. For the sake of simplicity, only one side of the duplex communication process is shown. Starting from the server side on the left in Figure 3.2, the server application writes a byte stream to a TCP socket. This is modeled as messages arriving at the M/M/1 queue at the rate of 1. These messages are queued and processed by the TCP engine. The TCP engine implements the TCP protocol and consists of various timers, algorithms, retransmit queues, and so on, modeled as the server process (which is also controlled by the feedback loop as shown in Figure 3.2. The feedback loop represents acknowledgements (ACKs) from the client side and receive windows. The server process sends packets to the network, which is also modeled as an M/M/1 queue. The network can be congested, hence packets are queued up. This captures latency issues in the network, which are a result of propagation delays, bandwidth limitations, or congested routers. In Figure 3.2 the client side is also represented as an M/M/1 queue, which receives packets from the network and the client TCP stack, processes the packets as quickly as possible, forwards them to the client application process, and sends feedback information to the server. The feedback represents the ACK and receive window, which provide flow control capabilities to this system.

### 3.2.1 Purpose of TCP Tuning

Figure 3.3 shows a cross-section view of the sequence of packets sent from the server to the client of an ideally tuned system. Send window-sized packets are sent one after another in a pipelined fashion continuously to the client receiver. Simultaneously, the client sends back ACKs and receive windows in unison with the server. This is the goal we are trying to achieve by tuning TCP parameters. Problems crop up when delays vary because of network congestion, asymmetric network capacities, dropped packets, or asymmetric server/client processing capacities. Hence, tuning is required. In a perfectly tuned TCP system spanning several network links of varying distances and bandwidths, the clients send back ACKs to sender in perfect synchronization

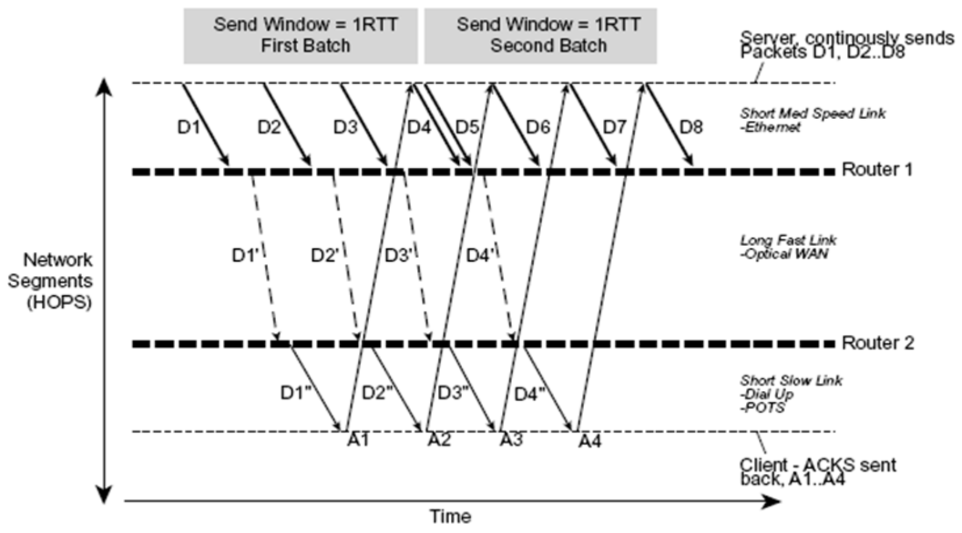

Send Window = 1RTT First Batch
Send Window = 1RTT Second Batch
Server, continously sends Packets D1, D2..D8
D1
D2
D3
D4
D5
D6
D7
D8
Short Med Speed Link -Ethernet
Router 1
D1'
D2'
D3'
D4'
Long Fast Link -Optical WAN
Network Segments (HOPS)
Router 2
D1"
D2"
D3"
D4"
Short Slow Link -Dial Up -POTS
A1
A2
A3
A4
Client - ACKS sent back, A1..A4
Time

Figure 3.3: Perfectly Tuned TCP System.

with the start of sending the next window. The objective of an optimal system is to maximize the throughput of the system. In the real world, asymmetric capacities require tuning on both the server and client side to achieve optimal throughput. For example, if the network latency is excessive, the amount of traffic injected into the network will be reduced to more closely maintain a flow that matches the capacity of the network. If the network is fast enough, but the client is slow, the feedback loop will be able to alert the sender TCP process to reduce the amount of traffic injected into the network. Later sections will build on these concepts to describe how to tune for wireless, high-speed wide area networks (WANs), and other types of networks that vary in bandwidth and distance.

Figure 3.4 shows the impact of the links increasing in bandwidth; therefore, tuning is needed to improve performance. The opposite case is shown in Figure 3.5, where the links are slower. Similarly, if the distances increase or decrease, delays attributed to propagation delays require tuning for optimal performance.

If a connection's throughput is limited only by the network, its window size will be limited by the congestion window (cwnd). In this case, we ideally have: window =

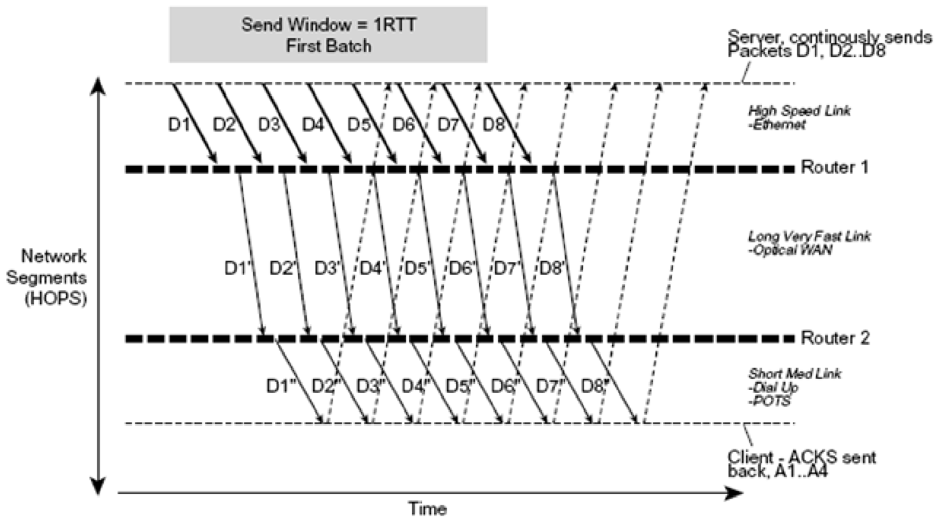

Send Window = 1RTT
First Batch
Server, continously sends Packets D1, D2..D8
D1 D2 D3 D4 D5 D6 D7 D8
High Speed Link -Ethernet
Router 1
Long Very Fast Link -Optical WAN
D1' D2' D3' D4' D5' D6' D7' D8'
Router 2
Short Med Link -Dial Up -POTS
D1" D2" D3" D4" D5" D6" D7" D8"
Client - ACKS sent back, A1..A4
Network Segments (HOPS)
Time

Figure 3.4: Tuning Required to Compensate for Faster Links.

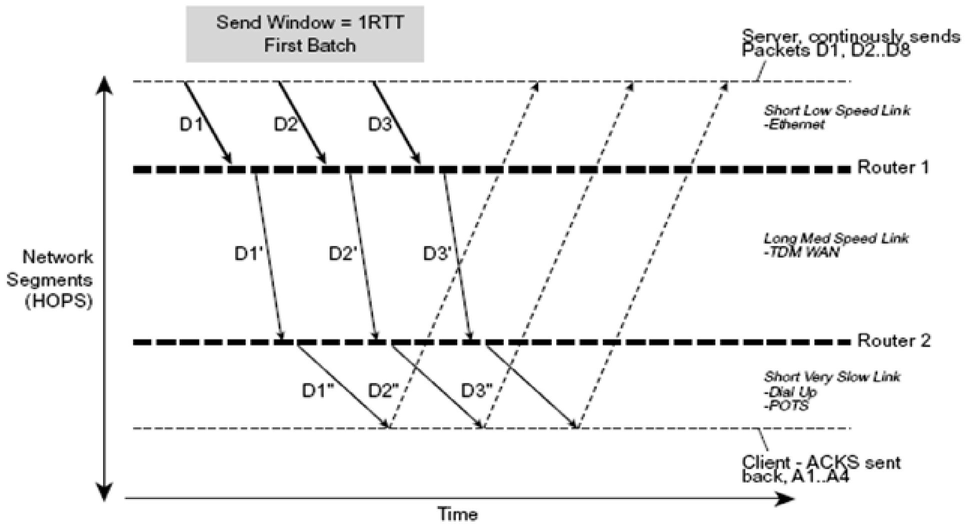

Send Window = 1RTT
First Batch
Server, continously sends Packets D1, D2..D8
D1 D2 D3
Short Low Speed Link -Ethernet
Router 1
Long Med Speed Link -TDM WAN
D1' D2' D3'
Router 2
Short Very Slow Link -Dial Up -POTS
D1" D2" D3"
Client - ACKS sent back, A1..A4
Network Segments (HOPS)
Time

Figure 3.5: Tuning Required to Compensate for Slower Links.

cwnd and throughput = bandwidth (BW). Since throughput = window/Round Trip Time (RTT), we have bandwidth = cwnd/RTT. Therefore, we have the Equation 2.1

This quantity is known as the bandwidth-delay product, or BDP (It should be noted that this cwnd is the ideal congestion window). In practice, bandwidth is not known and may change over time, so the congestion control mechanism grows and shrinks cwnd as it drives the network into congestion. The RTT between two endpoints is fundamentally limited by the distance between them and the speed of light. The highest bandwidths, however, have been growing exponentially with time. Consequently, BDPs across the Internet are also increasing over time. Further, connection BDPs may vary by many orders of magnitude on a single host. For example, a host connected with 100 Mbps. Fast ethernet may connect to a similar host at the same time as it connects to one with a 56 kbps modem. The relationship between the upper bound of TCP bandwidth, packet round trip time RTT , and packet loss p with As per the Mathis Equation 2.2.

To achieve substantial network performance over a wide area network that has a relatively large RTT, the required maximum packet loss rate p must be very low. The relationship derived by Mathis for the maximum packet loss rate required to achieve a target bandwidth.

For example, if the minimum link bandwidth between two hosts is OC-12 (622 Mbps), and the average round trip time is 20 msec, the maximum packet loss rate necessary to achieve 66% of the link speed (411 Mbps) is approximately 0.00018%, which represents only two packets lost out of every 100000 packets. Current loss rates on the commercial Internet backbone [6] are on the order of 0.1 %, which puts a hard upper limit on the potential bandwidth available to an application.

We need to ensure our buffers are large enough to ”fill the pipe”- That is, they must be able to hold a bandwidth-delay product’s worth of data. Suppose on a 1Gbps link with 100ms delay that’s 1Gbps x 100ms = 12.5 MB, But stock buffers are far too small - only 8KB by default for window and 16KB for Linux so stock flows will use: 16KB/12.5MB $\approx$ 0.0013 which is 1/1000th of available bandwidth. We could see the big size of available bandwidth being west in case of high performance long-fat network.

### 3.2.2 TCP Packet Processing

Now, let's take a look at the internals of the TCP stack inside the computing node. We will limit the scope to the server on the data center side for TCP tuning purposes. Since the clients are symmetrical, we can tune them using the exact same concepts. In a large enterprise data center, there could be thousands of clients, each with a diverse set of characteristics that impact network performance. Each characteristic has a direct impact on TCP tuning and hence on overall network performance. By focusing on the server, and considering different network deployment technologies, we essentially cover the most common cases.

Figure 3.6 shows the internals of the server and client nodes in more detail. To gain a better understanding of TCP protocol processing, we will describe how a packet is sent up and down a typical STREAMS-based TCP implementation. Consider the server application on the left side of Figure 3.6 as a starting point. The following describes how data is moved from the server to the client on the right.

- The server application opens a socket. (This triggers the operating system to set up the STREAMS stack, as shown.) The server then binds to a transport layer port, executes listen, and waits for a client to connect. Once the client connects, the server completes the TCP three-way handshake, establishes the socket, and both server and client can communicate.

- Server sends a message by filling a buffer, then writing to the socket.

- The message is broken up and packets are created, sent down the stream head, down the read side of each STREAMS module, by invoking the rput routine. If the module is congested, the packets are placed on the service routine for deferred processing. Each network module will prepend the packet with an

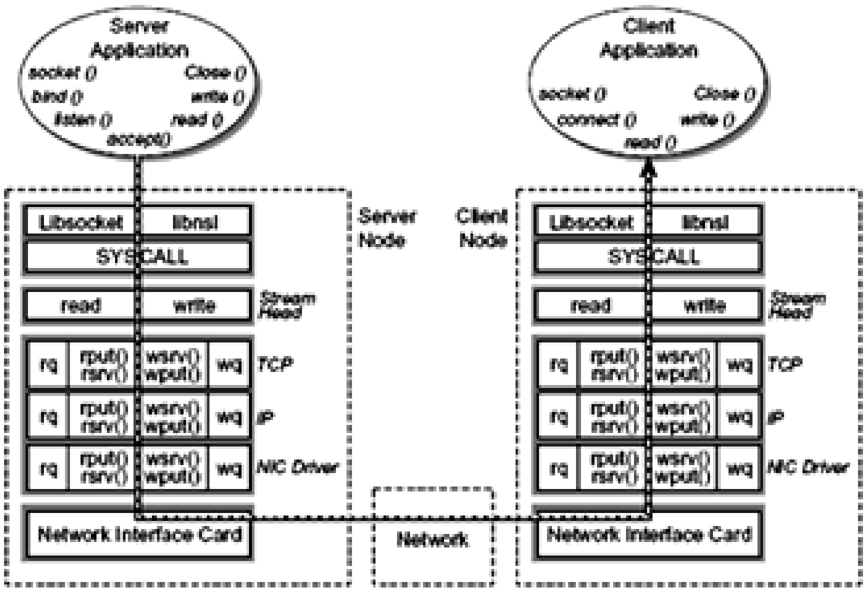


Figure 3.6: Complete TCP Stack on Computing Nodes.

appropriate header.

## 3.3 TCP STREAMS Module Tunable Parameters

The TCP stack is implemented using existing operating system application programming interfaces (APIs). The Linux OS offers a STREAMS framework. which was originally designed to allow a flexible modular software framework for network protocols. The STREAMS framework has its own tunable parameters, for example, sq_max_size, which controls the depth of a STREAMS syncq. This impacts how raw data messages are processed for TCP. Figure 3.7 provides a more detailed view of the facilities provided by the Linux STREAMS framework. Figure 3.7 shows some key tunable parameters for the TCP-related data path. At the top is the stream head, which has a separate queue for TCP traffic, where an application reads data. STREAMS flow control starts here. If the operating system is sending up the stack to the application and the application cannot read data as fast as the sender is sending

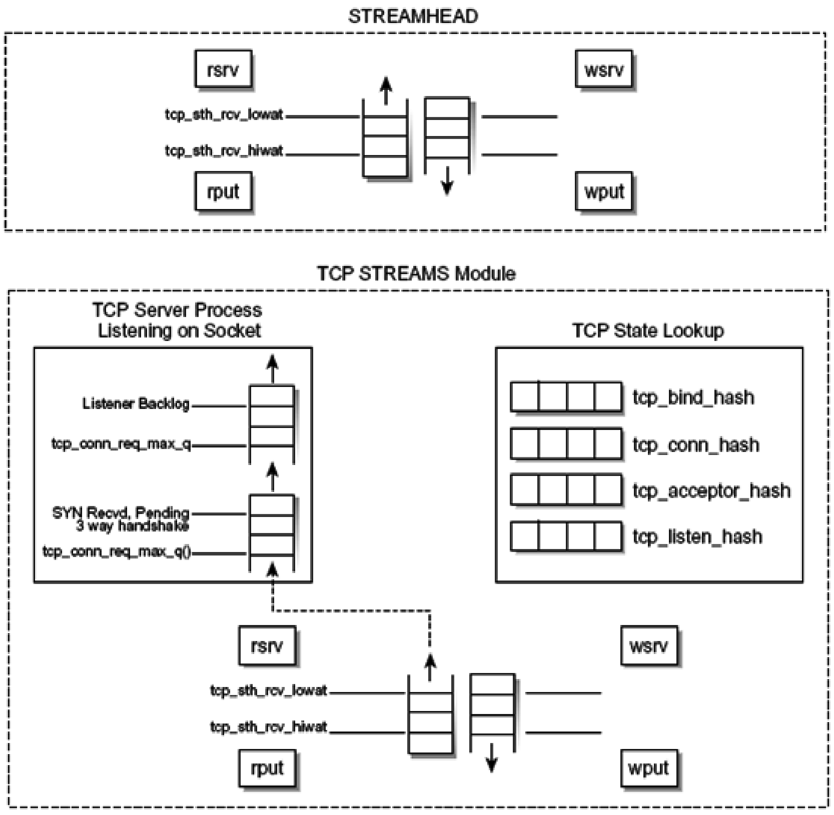


Figure 3.7: TCP and STREAM Head Data Structures Tunable Parameters.

it, the stream read queue starts to fill. Once the number of packets in the queue exceeds the high-water mark, tcp_sth_recv_hiwat, streams-based flow control triggers and prevents the TCP module from sending any more packets up to the stream head. The TCP module will be flow controlled as long as the number of packets is above tcp_sth_recv_lowat.

In other words, the stream head queue must drain below the low-water mark to reactivate TCP to forward data messages destined for the application. Note that the write side of the stream head does not require any high-water or low-water marks because it is injecting packets into the downstream, and TCP will flow control the stream head write side by its high-water and low-water marks tcp_xmit_hiwat and tcp_xmit_lowat. TCP has a set of hash tables. These tables are used to search for the associated TCP socket state information on each incoming TCP packet to maintain state engine for each socket and perform other TCP tasks to maintain that connection, such as update sequence numbers, update windows, round trip time (RTT), timers, and so on.

The TCP module has two new queues for server processes. The first queue, shown on the left in Figure 3.7, is the set of packets belonging to sockets that have not yet established a connection. The server side has not yet received and processed a client-side ACK. If the client does not send an ACK within a certain window of time, then the packet will be dropped. This was designed to prevent synchronization (SYN) flood attacks, where a bunch of unacknowledged client SYN requests caused servers to be overwhelmed and prevented valid client connections from being processed. The next queue is the listen backlog queue, where the client has sent back the final ACK, thus completing the three-way handshake. The server socket for this client will move the connection from LISTEN to ACCEPT. But the server has not yet processed this packet. If the server is slow, then this queue will fill up. The server can override this queue size with the listen backlog parameter. TCP will flow control on IP on the read side with its parameters tcp_recv_lowat, tcp_recv_hiwat, similar to the stream head read side. We refer it here as rcv_lim_wnd and rcv_max_buffer.

## 3.4 TCP TUNING SLIDING WINDOWS

One of the main principles for congestion control is avoidance. TCP tries to detect signs of congestion before it happens, and reduce or increase the load into the network accordingly. The alternative of waiting for congestion and then reacting is much worse because once a network saturates, it does so at an exponential growth rate and reduces overall throughput enormously. It takes a long time for the queues to drain, and then all senders again repeat this cycle. By taking a proactive congestion avoidance approach, the pipe is kept as full as possible without the danger of network saturation. The key is for the sender to understand the state of the network and client and to control the amount of traffic injected into the system. Flow control is accomplished by the receiver sending back a window to the sender. The size of this window, called the receive window, tells the sender how much data to send. Often, when the client is saturated it might not be able to send back a receive window to the sender, signaling it to slow down transmission. However, the sliding windows protocol is designed to let the sender know, before reaching a meltdown, to start slowing down transmission by a steadily decreasing window size. At the same time these flow control windows are going back and forth, the speed at which ACKs come back from the receiver to the sender provides additional information to the sender which caps the amount of data to send to the client. This is computed indirectly.

The amount of data that is to be sent to the remote peer on a specific connection is controlled by two concurrent mechanisms:

- The congestion in the network - The degree of network congestion is inferred by the calculation of changes in Round Trip Time (RTT): that is the amount of delay attributed the network. This is measured by computing how long it takes a packet to go from sender to receiver and back to the client. This figure is actually calculated using a running smoothing algorithm due the large variances in time. The RTT value is an important value to determine the congestion window, which is used to control the amount of data sent out to the remote client. This provides information to the sender on how much traffic should be sent to this particular connection based on network congestion.

- Client load - The rate at which the client can receive and process incoming traffic. The client sends a receive window that provides information to the sender on how much traffic should be sent to this connection, based on client load.

## 3.5 TCP Tuning for Acknowledge Control

Figure 3.8 shows how senders and receivers control ACK (Acknowledge) waiting and generation. The general strategy is that clients want to reduce receiving many small packets. Receivers try to buffer up a bunch of received packets before sending back an acknowledgment (ACK) to the sender, which will trigger the sender to send more packets. The hope is that the sender will also buffer up more packets to send in one large chunk rather than many small chunks. The problem with small chunks is that the efficiency ratio or useful link ratio utilization is reduced. For example, a one-byte data packet requires 40 bytes of IP and TCP header information and 48 bytes of Ethernet header information. The ratio works out to be l/(88+l) = 1.1 percent utilization. When a 1500-byte packet is sent, however, the utilization can be 1500/(88+1500) = 94.6 percent. Now, consider many flows on the same Ethernet segment. If all flows are small packets, the overall throughput is low. Hence, any effort to bias the transmissions towards larger chunks, without incurring excessive delays is a good thing, especially interactive traffic such as Telnet. Figure 3.8 provides an overview of the various TCP parameters.

There are two mechanisms senders and receivers use to control performance:

- Senders - timeouts waiting for ACK. This class of tunable parameters controls various aspects of how long to wait for the receiver to send back an ACK of the data that was sent. If tuned too short, then excessive retransmissions occur. If tuned too long, then excess wasted idle time elapses before the sender realizes the packet was lost and retransmits.

- Receivers - timeouts and number of bytes received before sending an ACK to sender. This class of tunable parameters allows the receiver to control the rate at which the sender sends data. The receiver does not want to send an ACK for every packet received because the sender will send many small packets, increasing the ratio of overhead to actual useful data ratio and reducing the efficiency of the transmission. However, if the receiver waits to long, there

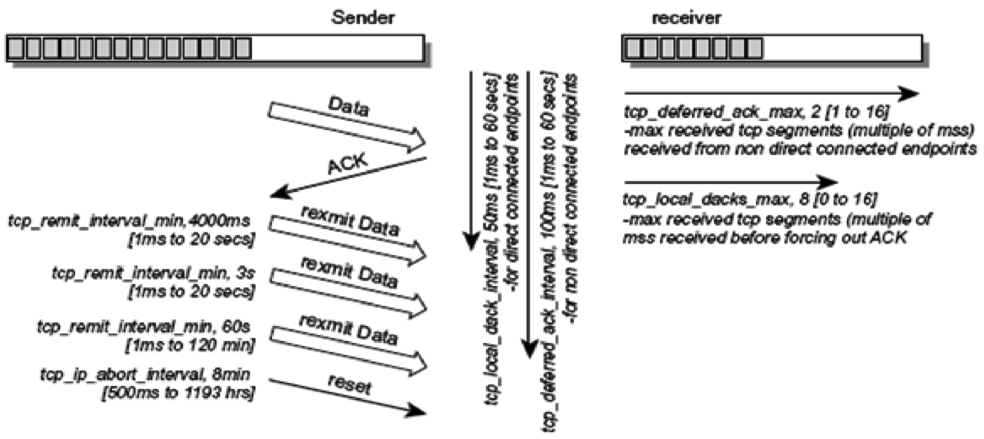

Sender
receiver
Data
ACK
tcp_remit_interval_min,4000ms [1ms to 20 secs]
rexmit Data
tcp_remit_interval_min, 3s [1ms to 20 secs]
rexmit Data
tcp_remit_interval_min, 60s [1ms to 120 min]
rexmit Data
tcp_ip_abort_interval, 8min [500ms to 1193 hrs]
reset
tcp_local_dack_interval, 50ms [1ms to 60 secs] -for direct connected endpoints
tcp_deferred_ack_interval, 100ms [1ms to 60 secs] -for non direct connected endpoints
tcp_deferred_ack_max, 2 [1 to 16] -max received tcp segments (multiple of mss) received from non direct connected endpoints
tcp_local_dacks_max, 8 [0 to 16] -max received tcp segments (multiple of mss received before forcing out ACK

Figure 3.8: TCP Tuning for ACK Control.

is excess latency which increases the burstiness of the communication. The receiver side can control ACKs with two overlapping mechanisms, based on timers and the number of bytes received.

# Chapter 4

# TCP Tuning State Model

TCP is a reliable transport layer protocol that offers a full duplex connection byte stream service. The bandwidth of TCP makes it appropriate for wide area IP networks where there is a higher chance of packet loss or reordering. What really complicates TCP are the flow control and congestion control mechanisms. These mechanisms often interfere with each other, so proper tuning is critical for high-performance networks.

In this chapter we start by explaining the TCP state machine, then describe in detail how to tune TCP, depending on the actual deployment. We also describe how to scale the TCP connection-handling capacity of servers by increasing the size of TCP connection state data structures.

## 4.1 TCP State Engine

This Figure 4.1 shows the server and client socket API at the top, and the TCP module with the following three main states:

### 4.1.1 Connection Setup

This includes the collection of substates that collectively set up the socket connection between the two peer nodes. In this phase, the set of tunable parameters includes

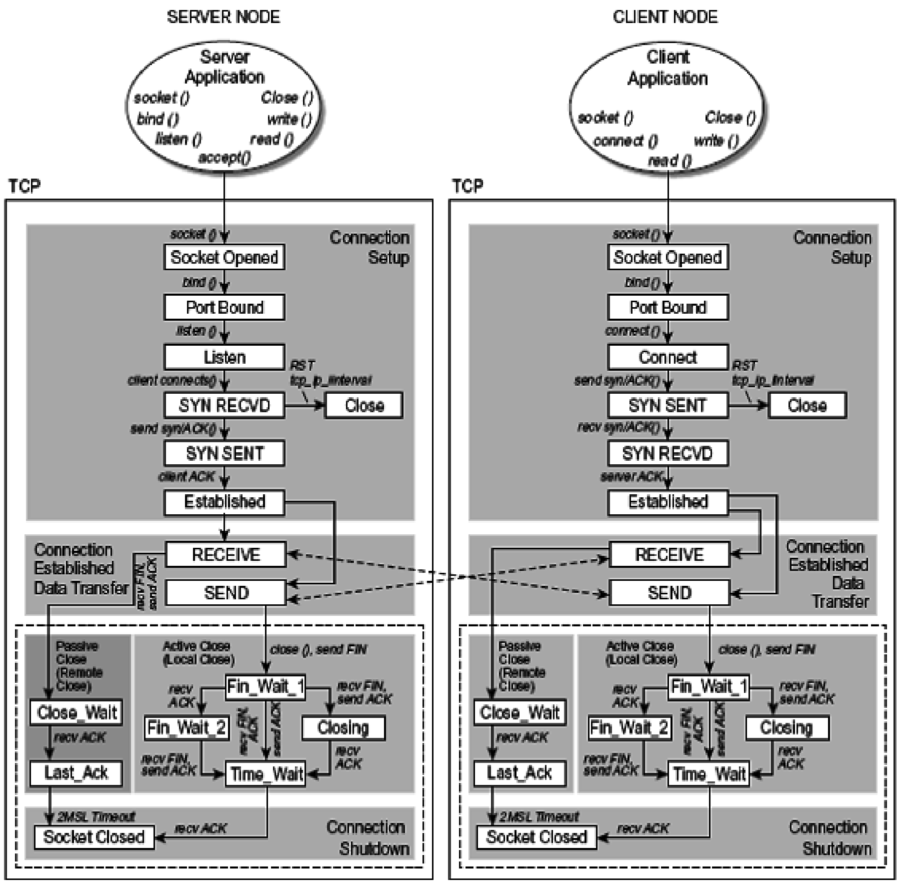


Figure 4.1: TCP Tuning for ACK Control.

- tcp_ip_abort_cinterval: the time a connection can remain in half-open state during the initial three-way handshake, just prior to entering an established state. This is used on the client connect side.

- tcp_ip_abort_linterval: the time a connection can remain in half open state during the initial three-way handshake, just prior to entering an established state. This is used on the server passive listen side.

For a server, there are two trade-offs to consider:

- Long Abort Intervals - The longer the abort interval, the longer the server will wait for the client to send information pertaining to the socket connection.

This might result in increased kernel consumption and possibly kernel memory exhaustion. The reason is that each client socket connection requires state information, using approximately 1-2 kilobytes of kernel memory. Remember that kernel memory is not swappable, and as the number of connections increases, the amount of consumed memory and time delays for lookups for connections increases. Hackers exploit this fact to initiate Denial of Service (DoS) attacks, where attacking clients constantly send only SYN packets to a server, eventually tying up all kernel memory, not allowing real clients to connect.

- Short Abort Intervals - If the interval is too short, valid clients that have a slow connection or go through slow proxies and firewalls could get aborted prematurely. This might help reduce chances of DoS attacks, but slow clients might also be mistakenly terminated.

### 4.1.2 Connection Established

This includes the main data transfer state (the focus of our tuning explanations in this article). The tuning parameters for congestion control, latency, and flow control will be described in more detail. Figure 4.1 shows two concurrent processes that read and write to the bidirectional full-duplex socket connection.

### 4.1.3 Connection Shutdown

This includes the set of sub states that work together to shut down the connection in an orderly fashion. We will see important tuning parameters related to memory. Tunable parameters include:

- tcp_time_wait_interval: how long the state of the TCP connection can wait for the 2MSL timeout before shutting down and freeing resources. If this value is too high, the socket holds up resources, and if it is a busy server, the port and memory may be desperately needed. The resources will not free up until this time has expired. However, if this value is too short and there have been many routing changes, lingering packets in the network, which might be lost.

- tcp_fin_wait2_flush_interval: how long this side will wait for the remote side to close its side of the connection and send a FIN packet to close the connection. There are cases where the remote side crashes and never sends a FIN. So to free up resources, this value puts a limit on the time the remote side has to close the socket. This means that half open sockets cannot remain open indefinitely.

## 4.2 TCP Tuning on the Sending and Receiving Side

TCP tuning on the sender side controls how much data is injected in to the network and the remote client end. There are several concurrent schemes that complicate tuning. So to better understand, we will separate the various components then describe how these mechanisms work together. We will describe two phases: Startup and Steady State. Startup Phase TCP tuning is concerned with how fast we can ramp up sending packets into the network. Steady State Phase tuning is concerned about other facets of TCP communication such as tuning timers, maximum window sizes, and so on. **Startup Phase:** In Startup Phase tuning, we describe how the TCP sender starts to initially send data on a particular connection. One of the issues with a new connection is that there is no information about the capabilities of the network pipe. So we start by blindly injecting packets at a faster and faster rate until we understand the capabilities and adjust accordingly. Manual TCP tuning is required to change macro behavior, such as when we have very slow pipes as in wireless or very fast pipes such as 10 Gbit/sec. Sending an initial maximum burst has proven disastrous. It is better to slowly increase the rate at which traffic is injected, based on how well the traffic is absorbed. This is similar to starting from standstill on ice. If we initially floor the gas pedal, we will skid, and then it is hard to move at all. If on the other hand we start slowly and gradually increase speed, we can eventually reach a very fast speed. In networking, the key concept is that we do not want to fill buffers. We want to inject traffic as close as possible to the rate at which the network and target receiver can service the incoming traffic. also at receiving end we make set the tcp_rev_max double to the tcp_cwnd. During this phase, the congestion window is much smaller than the receive window. This means the sender controls the traffic injected into the receiver by computing the congestion window and capping the injected traffic amount by the size of the congestion window. Any minor bursts can be absorbed by queues. Figure 4.2 shows what happens during a typical TCP session starting from idle sender does not know the capacity of the network, so it starts to slowly send more and more packets into the network trying to estimate the state of the network by measuring the arrival time of the ACK and computed RTT times. This results in a self-clocking effect. In Figure 4.2, we see the congestion window initially starts with a minimum size of the maximum segment size (MSS), as negotiated in the three-way handshake during the socket connection phase. The congestion window is doubled every time an ACK is returned within the timeout. The congestion window is capped by the TCP tunable variable tcp_cwnd_max, or until a timeout occurs. At that point, the ssthresh internal variable is set to half of tcp_cwnd_max. ssthresh is the point where upon a retransmit, the congestion window grows exponentially. After this point it grows additively, as shown in Figure 4.2. Once a timeout occurs, the packet is retransmitted and the cycle repeats.

Figure 4.2 shows that there are three important TCP tunable parameters:

- tcp_rwnd_max: sets up the initial receiving window buffer just after the socket connection is established.

- tcp_slow_start_initial: sets up the initial congestion window just after the socket connection is established.

- tcp_slow_start_after_idle: initializes the congestion window after a period of inactivity. Since there is some knowledge now about the capabilities of the network,

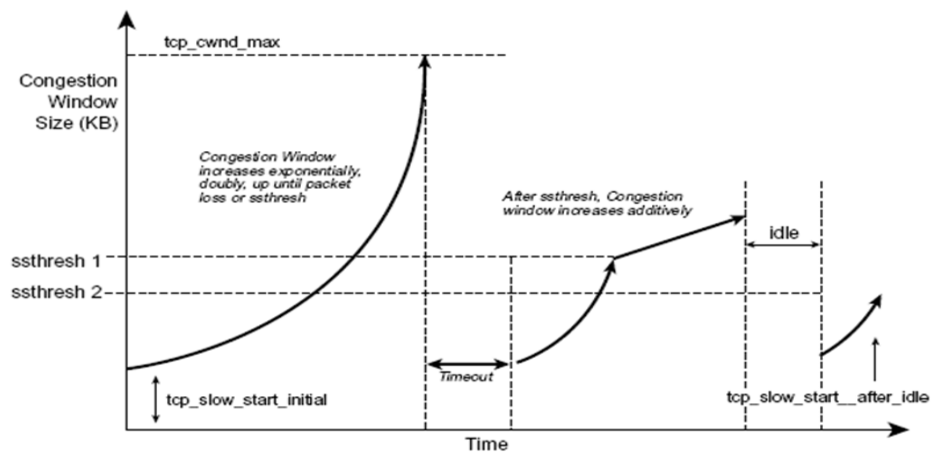


Figure 4.2: TCP Startup Phase.

we can take a short cut to grow the congestion window and not start from zero, which takes an unnecessarily conservative approach.

- tcp_cwnd_max: places a cap on the running maximum congestion window buffer. If the receive window grows, then tcp_cwnd_max grows to the receive window size.

In different types of networks, you can tune these values slightly to impact the rate at which you can ramp up. If you have a small network pipe, you want to reduce the packet flow, whereas if you have a large pipe, you can fill it up faster and inject packets more aggressively.

Steady State Phase: In Steady State Phase, after the connection has stabilized and completed the initial startup phase, the socket connection reaches a phase that is fairly steady and tuning is limited to reducing delays due network and client congestion. An average condition must be used because there are always some fluctuations in the network and client data that can be absorbed. Tuning TCP in this phase, we look at the following network properties:

- Propagation Delay - This is primarily influenced by distance. This is the time it takes one packet to traverse the network. In WANs, tuning is required to keep the pipe as full as possible, increasing the allowable outstanding packets.

- Link Speed - This is the bandwidth of the network pipe. Tuning guidelines for link speeds from 56kbit/sec dial-up connections differ from lOGbit/sec optical local area networks (LANs).

# Chapter 5

# Implementation

The implementation and result(s) of the thesis work is/are always important in the thesis work. This chapter covers the brief explanation of working environment setup, analysis tools, proposed methodology and algorithm of the thesis work.

## 5.1 User Mode Linux

This topic will take a quick look at the inside of a UML (User Mode Linux). We will concentrate on the relationship between the UML and the host. For many people, encountering a virtual machine for the first time can be confusing because it may not be clear where the host ends and the virtual machine starts.

For example, the virtual machine obviously is part of the host since it can't exist without the host. However, it is totally separate from the host in other ways. You can be root inside the UML and have no privileges whatsoever on the host. When UML is run, it is provided some host resources to use as its own. The root user within UML has absolute control over those, but no control, not even access, to anything else on the host. It's this extremely sharp distinction between what the UML has access to and what it doesn't that makes UML useful for a large number of applications.

In order to run a process, you obviously need some level of privilege on the system. However, a UML host can be set up such that the user that owns the UML processes

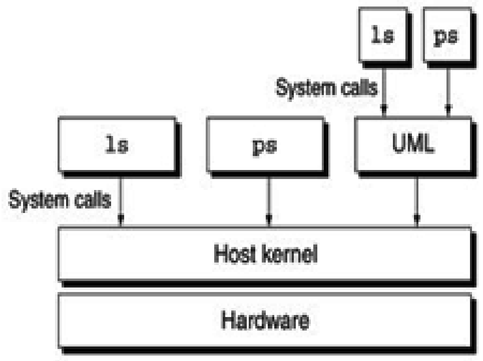


Figure 5.1: User Mode Linux working stack.

on the host can do nothing but run the UML process.

A second common source of confusion is the duality of UML. It is both a Linux kernel and a Linux process. It is useful, and instructive, to look at UML from both perspectives. However, to many people, a kernel and a process are two completely different things, and there can be no overlap between them. So, we will look at a UML from both inside and outside, on the host, in order to compare the two views to each other. We have different views of the same things. They will look different but will both be correct in their own ways.

The relationship among a UML instance, the host kernel, and UML processes. To the host kernel, the UML instance is a normal process. To the UML processes, the UML instance is a kernel. Processes interact with the kernel by making system calls, which are like procedure calls except that they request the kernel do something on their behalf.

Like all other processes on the host, UML makes system calls to the host kernel in order to do its work. Unlike the other host processes, UML has its own system call interface for its processes to use. This is the source of the duality of UML. It makes system calls to the host, which makes it a process, and it implements system calls for its own processes, making it a kernel as show in Figure 5.1.

## 5.2 Compiling Linux Kernel

Before starting check whether your /boot file system is ext3. If not you might end up with a lot of custom configurations, which we haven't mentioned here. We are here describing basic steps for the same.

a. Get latest Linux kernel code. Visit `http://kernel.org/`

b. Extract tar (.tar.bz2)

   ```
   % tar -xvjz linux-2.6.26.5.tar.bz2 -C /usr/src/linux-2.6.26.5
   ```

c. Configure kernel

   Before doing anything else, we need to have development tools on your system. If you are using a Debian distribution you need lib6c-dev and libncursesw5-dev Change the directory to `/usr/src/linux-2.6.26.5`

   We have more then three options to get started

   - `make menuconfig`: Text based color menus, radiolists & dialogs. This option also useful on remote server if you wanna compile kernel remotely
   - `make xconfig`: X windows (Qt) based configuration tool, works best under KDE desktop
   - `make gconfig`: X windows (Gtk) based configuration tool, works best under Gnome Dekstop

   we are interested in,

   ```
   make menuconfig
   ```

d. Compile kernel

   Start compiling to create a compressed kernel image

   ```
   % make
   ```

   Start compiling to kernel modules

   ```
   % make modules
   ```

Install kernel modules

```
% make modules_install
```

e. Install kernel

   So far we have compile module and installed kernel. Let's install kernel,

```
% make install
```

   After that change directory to /boot, you will be able to observer,

```
System.map-2.6.26.5
config-2.6.26.5
vmlinuz-2.6.26.5
```

f. Create initrd image

```
% cd /boot
% mkinitramfs -o initrd.img-2.6.26.5 2.6.26.5
```

g. Modify Grub configuration file - `/boot/grub/menu.lst`

   we have LILO and Boot from Floppy options also but we are interested in Grub.

```
% cd /boot/grub
```

h. update-grub

   Update grub is a cool way to edit the file automatically. One could use LILO instead of GRUB. Though the update utility do the job for us, it wont set the initrd. Hence,

```
% vi /boot/grub/menu.lst
```

   Add the following lines at the end of the file.

```
title Linux, kernel 2.6.26.6 Default
root (hd0,6)
kernel (hd0,6)/vmlinuz-2.6.26.5 ro root=/dev/VolGroup00/LogVol00
rhgb quiet initrd /initrd-2.6.26.5.img
savedefault
boot
```

Please note that "root" will change according to your configuration

i. Reboot computer and boot into your new kernel

A UML is both very similar to and very different from a physical machine. It is similar as long as you don't look at its hardware. When you do, it becomes clear that you are looking at a virtual machine with virtual hardware. However, as long as you stay away from the hardware, it is very hard to tell that you are inside a virtual machine.

Both the similarities and the differences have advantages. Obviously, having a UML run applications in exactly the same way as on the host is critical for it to be useful.

## 5.3 TCP Extension for High Performance Network

We have discuss the web100 tools in section 2.9. Here this section covers general instructions and advice on how to:

- Apply the web100 patches to a User mode Linux kernel
- Build the kernel, and
- Install the newly built User Mode Linux kernel on the system

Which is the TCP extension for high performance network, Here we are sure that we have a good repeatable procedure for building, installing and testing kernels before applying the web100 patch. If you are moving to a new base kernel, it is especially important that you verify that your new kernel is fully functional before you apply the web100 patch. We typically have far less difficulty installing web100 than getting new base kernels properly configured to support all devices and features.
Before we proceed we need to become root:

```
% su
```

```
Enter root password.
```

Before we proceed, it may be a good idea to take the following steps so that you can backtrack if you run into problems:

- Make a copy of the current kernel
- Make a copy of the .config file (usually in /usr/src/linux)
- tar up the files in the /lib/modules directory corresponding to the current working kernel

Extract the sources:

```
% tar -C /usr/src -xvf linux-2.6.26.5.tar.gz.
```

Make a symbolic link:

```
% rm -rf /usr/src/linux
% ln -s /usr/src/linux-2.6.26.5 /usr/src/linux
```

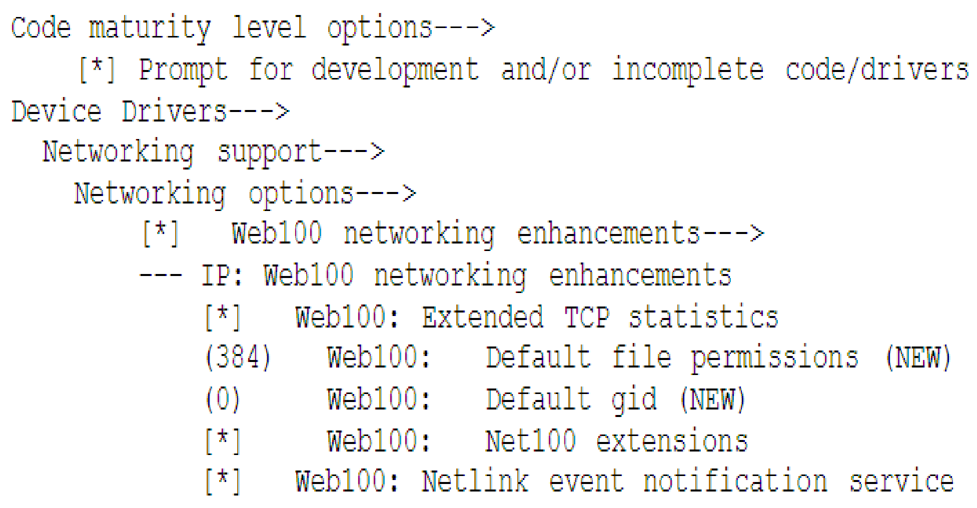
Applying web100 patch:

```
% cd /usr/src/linux
% patch -p1 < [path-to-web100-dir]/web100-[version-name].patch
```

It is a good idea to save a copy of the patch file you will be using for web100, so that you can backout of the patch (to get a clean kernel tree) if necessary. This will also be needed before applying the next version of web100 patch file because the patches are distributed as full patches as opposed to incremental patches. Configuring and building the UML kernel with TCP Extension for high performance. We have found it to be safer to force a full rebuild: `% cd /usr/src/linux`

```
% cp .config config.save
% make mrproper
% cp config.save .config
% make menuconfig
```

To enable web100 features select the following options:



```
Code maturity level options--->
    [*] Prompt for development and/or incomplete code/drivers
Device Drivers--->
  Networking support--->
    Networking options--->
        [*]   Web100 networking enhancements--->
        --- IP: Web100 networking enhancements
            [*]   Web100: Extended TCP statistics
            (384)   Web100:   Default file permissions (NEW)
            (0)     Web100:   Default gid (NEW)
            [*]     Web100:   Net100 extensions
            [*]   Web100: Netlink event notification service
```

## 5.4 Proposed approach

The web100 kernel has added a number of variables or ”Instruments” to the kernel ”sock” structure which tracks a number of critical events on each of the TCP connections on the system. These variables are exposed to the outside world through the /proc file system.

The list of variables that are monitored can be found in the file /proc/web100/header. It should be observed that the variables are grouped in one or more files, which are referred to as ”group”s in the web100 library API. The API provides routines for reading a single variable, or a group of variables. The advantage of reading the variable as a group is that when the values of the variables are read as group, all the values in that group are read atomically. This operation is called a ”snap”, and there are routines provided to get ”snapshot”s of groups.

The kernel API consists of a few essential data structures. A brief overview of these data structures are provided below:

**Agent- web100_agent**: Agent is the top most object of the library, and any web100 based developed using this library starts off by obtaining a pointer to this object. This object contains a list of ”connection” objects and ”group” objects.

**Connection - web100_connection** : This is an object identified by a Connection ID (CID) that is set up by the kernel, one for each TCP connection on the host. For each TCP connection on the system web100 creates a directory where cid is a unique number.

**Group - web100_group**: A group is a collection of web100 "variable" descriptors. A group has a name which shows up as file in /proc/web100/<cid> directory. By having access to a group, one can read all the variables in that group atomically.

**Variable - web100_var**: A variable is descriptor that contains the name of the variable and the necessary information to get its value from the kernel.

**Snapshot - web100_snapshot** : A snapshot contains a connection, a group, and all the values for that combination of connection and group at a particular time.

The core of the project is the per-connection TCP instrument set. It is defined in standards language in an IETF MIB document [10], and implemented in what we call the Kernel Instrument Set (KIS). The architecture of our Linux implementation is depicted in Figure 2.1 Attached to each socket is a structure containing KIS variables and meta-data. Its fields are updated from key points in the protocol stack. An abstraction of this structure is exposed through a "/proc" filesystem interface. To allow for future interfaces and other operating systems, we defined a portable API for accessing the instruments, implemented as a library. Since the instruments were designed to support a MIB, it is fairly natural to implement an SNMP agent on top of this library for exporting the instruments. It is a natural extension of this API to implement per-connection TCP controls. A "read" accesses an instrument, while a "write" accesses a control.

The Figure 5.2 shows per network connection based working scenario. Here we are reading per connection, per agent, per group, per snap variables and set accordingly the tune groups. The data transaction occur through tune connection will be with optimized QoS parameter.

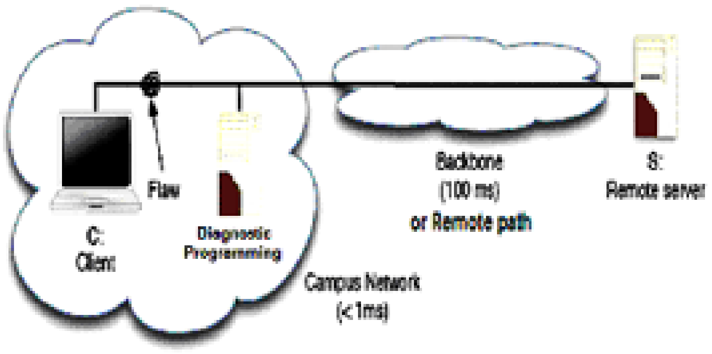


Figure 5.2: Implementation Scenario.

### 5.4.1 Proposed Algorithm

The proposed algorithm show belelow.

**Algorithm 5.1** The Proposed Algorithm

```
1   Create localType agent 'agnt'
2   Create group 'grp'
3   Identity address type 'addType'
4   While 'grp'
5       Disable kernel default buffer setting facility
6       Check address type ipv4/v6
7       Read local address and port'ladd', 'lport'
8       Read remote address and'radd', 'rport'
9       Define snap 'snp'
10          If rport is not in reserved list
11          While Instruments in snap
12              If Instrument_con in "Established" state
13              Enable window scalling
14              Set rclWnd = 2 * curr_cWnd
15                  If rclWnd $\geq$ 12000000 then
16                  Disble SACK
```

```
17          End_while_instruments
18      Disable spoofin, sy-flood attack if require
19      Disable ICMP redirect if require
20  End while_grp
21  Release memory if con closed
```

The project module starts up, to read /proc/web100 file and read per connection kernel instruments, convert IP and port address from kernel space to user space. It also converts the instruments value in normal data type for the display and modification. The rclWnd (Receiver Limiting Window) is set to $\geq 2$ * curr_cwnd (current congestion window) to fully open up the TCP. We enable the scaling window size option for extension for the high-performance. We require to check the size of rclWnd, which should not exceed 12MB. If it is greater than 12MB and selective retransmission event occurs, then searching for interested packet takes a longer time than RTO (retransmission time out), which forces the slow start [11]. Next, save the rcvWnd to /proc/weblOO/cid/tune file. If the local/remote port is on a Stop or not Selected Connection, list them and leave the connection un-tuned. At the end release the memory of closed connection.

Slow start threshold (shThrsh) is used when switching from exponential increase to linear increase. The value for ssthresh for a given path is cached in the routine table If there is a retransmission on a connection to a given host, then all connections to that host for the next 10 minute will use a reduced ssthresh. Or, if previous connection to that host is practically good, then you might stay slow start in long time. We are using following sysctl (Configure Linux kernel parameters at runtime) parameter to disable the same.

`net.ipv4.web100_no_metrics_save`

We disable the following parameters to prevent a hacker from using a spoofing attack against the IP address of the server.

`net.ipv4.conf.eth0.accept_source_route,`

`net.ipv4.conf.lo.accept_source_route,`

`net.ipv4.conf.default.accept_source_route,`

`net.ipv4.conf.all.accept_source_route`

The following parameter enables TCP-SYN cookies, which protects the server from syn-flood attacks ie both denial-of-service (DoS) or distributed denial-of-service (DDoS):

`net.ipv4.tcp_syncookies`

The following parameters were used to configure the server to ignore redirects from machines that are listed as gateways. Redirect can be used to perform attacks, so we only want to allow them from trusted sources:

`net.ipv4.conf.eth0.secure_redirects,`

`net.ipv4.conf.lo.secure_redirects,`

`net.ipv4.conf.default.secure_redirects,`

`net.ipv4.conf.all.secure_redirects`

In addition, we could allow the interface to accept or not accept any ICMP redirects. The ICMP redirect is a mechanism for routers to convey routing information to hosts. We disable these redirects using the following parameter:

`net.ipv4.conf.eth0.accept_redirects,`

`net.ipv4.conf.lo.accept_redirects,`

`net.ipv4.conf.default.accept_redirects,`

`net.ipv4.conf.all.accept_redirects`

For ignoring all kinds of icmp packets or pings we used

`net.ipv4.icmp_echo_ignore_all`

Some routers send invalid responses to broadcast frames, and each one generates a warning that is logged by the kernel. These responses can be ignored using this parameter:

`net.ipv4.icmp_ignore_bogus_error_responses`

One of the issues found in servers with many simultaneous TCP connections is the large number of connections that are open but unused. TCP has a keepalive function that probes these connections and, by default, drops them after 7200 seconds (2 hours). This length of time might be too large for your server and can result in excess memory usage and a decrease performance. Setting keepalive to 1800 seconds (30 minutes), for example, might be more appropriate:

```
net.ipv4.tcp_keepalive_time
```

# Chapter 6

# Testing and Analysis

The network measurement tools available to application developers and system administrators, are used to measure physical data-link bandwidth, round trip time, loss rate, router buffer sizes at each hop in the network, and measure end-to-end network bandwidth. Various traffic generation and traffic monitoring tools were used for verification per connection throughput measurement and generation of results. This chapter covers the brief explanation of that testing tools and test-bed.

## 6.1 Testing and Analysis tools

The network measurement tools available to application developers and system administrators are used to measure physical data-link bandwidth, round trip time, loss rate, router buffer sizes at each hop in the network, and measure end-to-end network bandwidth.

The UNIX ping utility is used to transmit and receive ICMP Echo packets to a destination host to determine if the host is reachable, to measure round trip time (RTT), and to measure packet loss on the network path to the host. The RTT measurements made by ping can be used to estimate the ”pipe” capacity (capacity = BW * RTT) of the network between two hosts. Since the test load put on the network by ping consists of small periodic ICMP packets, the packet loss rate measured by ping is not very useful for estimating available TCP bandwidth using Equation 2.2. The RTT measurement, however, is useful for deriving the maximum packet loss rate necessary to support a desired TCP bandwidth in Equation 2.3.

Traceroute [12] is used to discover the IP network route between two hosts and the RTT to each hop in the network route. Traceroute is used to diagnose routing problems between hosts.

Iperf [13] is a tool that measures TCP and UDP transfer rates between host pairs. Iperf is used to estimate the maximum network bandwidth available to an application and to investigate the relationship between UDP packet injection rate and packet loss on a network between two hosts.

Tcpdump [14] is used for per connection packet capturing and displaying all packets on a network segment connected to a network adapter that is configured in "promiscuous mode".

Tcptrace [15] can be run on a network dumpfile trivially as in tcptrace dumpfile where dumpfile is a file containing traffic captured from the network.

Xplot [16] is a Mathematics function plotting program based on OpenGL, we can use it with tcpdump and tcptrace.

## 6.2 Test-bed setup and Experiments

Figure 6.1 shows there are two senders and two receivers connected with high speed routers. All nodes having Realtek RTL8168/8111 PCI-E Gigabit Ethernet NIC and are connected with cat-6 cable.

The host to host testing with iperf, resulted in a difference of around 305Mbps of speed. For 16KB of window size we are having throughput of 132Mbps maximum while for 2MB of window size it is 438Mbps. This is shown in Figure 6.2.

We are transferring large avi file from computer-2 to computer-4 using FTP. We takes all the result for near about 2 minutes of transaction. We have use the combination of tcpDump, tcpTrace and xplots for per-connection throughput measurement

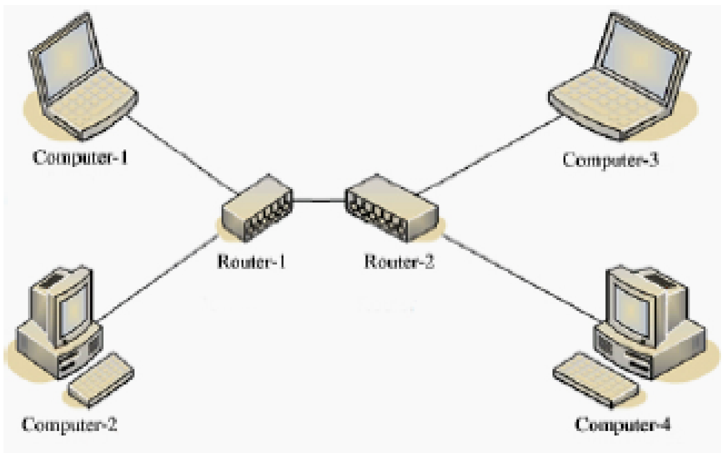


Figure 6.1: Test-bed setup.

commands are as follow.

`% tcpdump -ni eth0 -w`

Tracing the output.tarace file we are using

`% tcptrace -G output.trace,` and for plotting different graph we are using following commands

```
% xplot *tput.xpl,
% xplot *sqg.xpl
% xplot *tline.xpl
```

Figure 6.3 shows the throughput using tcpDump, tcpTrace and xPlot per network connection port. Y-axis contain the window size in Kilo Bytes and the x-axis contain the time. The instantaneous throughput value by yellow dots are at around 150MB.

The Figure 6.4, shows the window buffer after tuning Qos parameter. Y-axis contain the window size in Kilo Bytes and the x-axis contain the time. we can see maximum dots at 550MB. We can conclude that the throughput increase from 150-580Mbps.

The Figure 6.5 show the instantaneous window size of tuned and un-tuned connection, the announce window is maximum 2896B for un-tuned connection, showing fixed window size for all the transaction because of the limited window buffer size

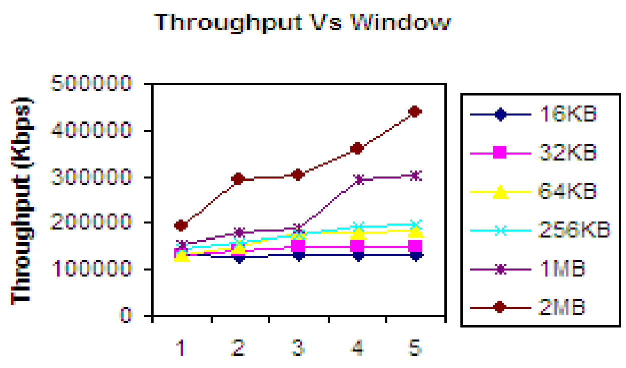


Figure 6.2: Throughput Vs Window size.

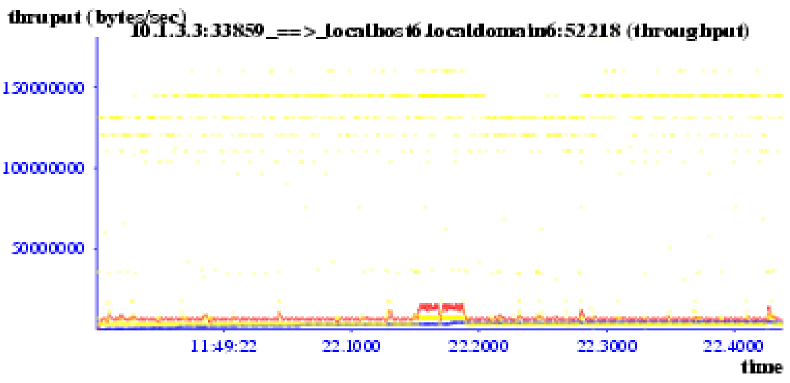


Figure 6.3: Throughput before window tuning.

available at receiving end.

The Figure 6.5 also shows the announce window size which is different at every transaction. This is because at the receiving end the window size is allow to tune according to BDP. It conclude that for Tuned connection there is scope for TCP to fully open-up.

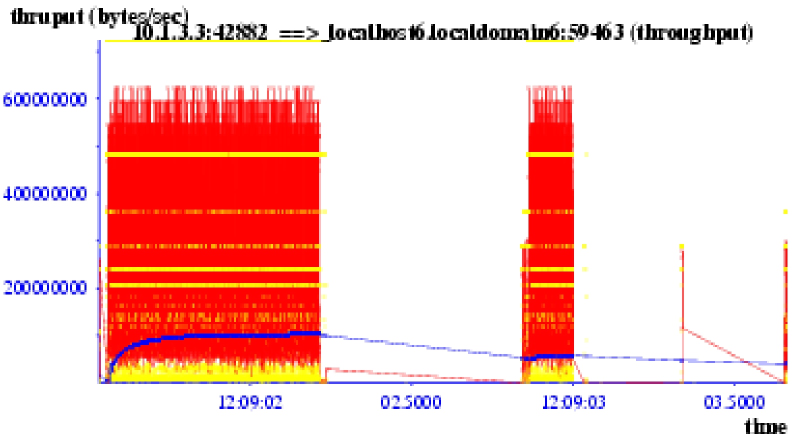


Figure 6.4: Throughput after window tuning.

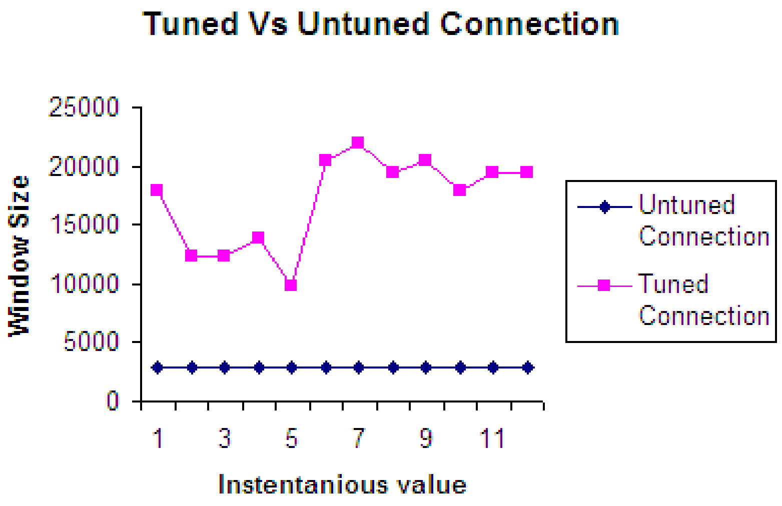


Figure 6.5: Tuned Vs Un-tuned connection window size.

# Chapter 7

# Conclusion and Future Scope

## 7.1 Conclusion

This Thesis demonstrates the kernel space and user space can be effectively used in combination with network tuning. The suite of network performance tools currently available to identify structural and host tuning problems that can adversely affect end-to-end TCP performance. The TCP tuning using the /proc file system is a novel approach. We have identified requirements for QoS parameter tuning TCP to achieve maximum throughput across all connections simultaneously within the resource limits of the sender. The technique to modify the TCP implementation of Linux kernel 2.6.26.5 to optimize the memory and bandwidth requirements of network connections results in greatly improved performance, a decrease in packet loss under bottleneck conditions, and greater control of buffer utilization by the end hosts. The auto tuning feature incorporated in the TCP architecture implements with a per flow or destination base approach.

## 7.2 Future Scope

To improve the performance of the application beyond the results presented, several approaches can be made. First, a thorough examination of the characteristics of the application should be performed to discover any more tuning opportunities. Second, attempts should be made in concert with Network Administrators to determine if the MTU of the network path between the server and client can be increased. Finally, an investigation of the sources of packet loss for reasons other than congestion will be undertaken. Potential sources of packet loss include operating system implementation errors, improperly configured network equipment, and all of the other sources.

In future studies we plan to investigate the application of our mechanism for space application networks. We further wish to write the same algorithms without support of web100. Finally, we are very much interested in developing methods to optimize QoS parameter threshold settings based upon specific traffic patterns and to extend these capabilities to create adaptive threshold settings.

# References

[1] W. Stevens, “Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms,” *RFC 2001*, January 1997.

[2] “Web100 project, www.web100.org,”

[3] V. Jecobson, “Congestion avoidance and control,” *Proceedings of ACM SIGCOMM '88*, May 1992.

[4] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, “The macroscopic behavior of the tcp congestion avoidance algorithm,” *Computer Communication Review*, vol. 27, July 1997.

[5] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2001.

[6] H. Sivakumar, S.Bailey, and R. L. Grossman, “Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks,” in *Proceedings of Supercomputing*, IEEE, 2000.

[7] B. Mah, “pchar: A tool for measuring internet path characteristics,”

[8] D. Comer, *Internetworking with TCP/IP Principles Protocols and Architectures*, vol. 1. Prentice Hall New Jersey.

[9] D. L. Kleinrock, *Queueing Systems*, vol. 1. Wiley New York, 1975.

[10] J. W. Heffner, “High bandwidth tcp queuing,” *advisor*, 2002.

[11] “Tcp tuning, www-didc.lbl.gov/tcp-tuning/linux.html,”

[12] V. Jacobson, “Traceroute: A tool for printing the route packets take to a network host,”

[13] “Iperf tool, www.noc.ucf.edu/tools/iperf/,”

[14] V. Jacobson and C. Leres, “Tcpdump project,”

[15] “Tcptrace, ohio university, http://irg.cs.ohiou.edu/software/tcptrace/download.html,”

[16] “Xplot project, http://www.xplot.org,”