Georgia State University
ScholarWorks @ Georgia State University

Computer Science Theses | Department of Computer Science

11-17-2009

STCP: A New Transport Protocol for High-Speed Networks

Ranjitha Shivarudraiah
Georgia State University

Follow this and additional works at: https://scholarworks.gsu.edu/cs_theses

Recommended Citation

Shivarudraiah, Ranjitha, "STCP: A New Transport Protocol for High-Speed Networks." Thesis, Georgia State University, 2009.
doi: https://doi.org/10.57709/1348309

This Thesis is brought to you for free and open access by the Department of Computer Science at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Computer Science Theses by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.

STCP: A NEW TRANSPORT PROTOCOL FOR HIGH-SPEED NETWORKS

by

RANJITHA SHIVARUDRAIAH

Under the Direction of Dr Xiaojun Cao

ABSTRACT

Transmission Control Protocol (TCP) is the dominant transport protocol today and likely to be adopted in future high-speed and optical networks. A number of literature works have been done to modify or tune the Additive Increase Multiplicative Decrease (AIMD) principle in TCP to enhance the network performance. In this work, to efficiently take advantage of the available high bandwidth from the high-speed and optical infrastructures, we propose a Stratified TCP (STCP) employing parallel virtual transmission layers in high-speed networks. In this technique, the AIMD principle of TCP is modified to make more aggressive and efficient probing of the available link bandwidth, which in turn increases the performance. Simulation results show that STCP offers a considerable improvement in performance when compared with other TCP variants such as the conventional TCP protocol and Layered TCP (LTCP).

INDEX WORDS: TCP, Congestion window

STCP: A NEW TRANSPORT PROTOCOL FOR HIGH-SPEED NETWORKS

by

RANJITHA SHIVARUDRAIAH

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2009

Copyright by
Ranjitha Shivarudraiah
2009

STCP: A NEW TRANSPORT PROTOCOL FOR HIGH-SPEED NETWORKS

by

RANJITHA SHIVARUDRAIAH

Committee Chair: Dr. Xiaojun Cao

Committee: Dr Raj Sunderraman

Dr Anu Bourgeois

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

December 2009

ACKNOWLEDGEMENTS

First of all, I would like to express my special and sincere appreciation to my advisor, Dr. Xiaojun Cao, who has been constant in his valuable guidance and encouragement in my research. He has spent considerable time in answering my questions, providing valuable suggestions to my research.

The computer science department has provided an ideal working environment during the years' study under the Chair, Prof. Yi Pan. I specially thank Dr Raj Sunderraman and Dr Anu Bourgeois for being a part of the committee and guiding me to improve and enhance my work. Also I would like to thank the administrative and technical specialists for their great service and help: Tammie, Shaochieh, Celena and Venette.

I would also want to thank my husband, Satish Shivarudrappa for his moral support and encouragement without which my achievement would not be possible.

Last but not the least; I am deeply thankful to my family, especially my parents for their relentless support and care.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION: INTERNET AND TRANSMISSION CONTROL PROTOCOL (TCP)

Internet provides reliable, accurate, and fast exchange of data. The Transmission Control Protocol (TCP) is one of the core protocols of the Internet protocol suite acting at the transport layer and is responsible for reliable data transmission. This chapter describes in detail the importance of the Internet in today's world. More importantly, we describe the role played by TCP in the Internet.

1.1 The Internet in Today's World

The Internet is a network of networks. That is, the Internet is an interconnected set of privately and publicly owned and managed networks. Any network connected to the Internet must run the IP protocol and conform to certain naming and addressing conventions. Internet is indispensible for the following reasons:

1. **Communication**: Exchanging information by overcoming the barriers of distance and time has been made possible by the Internet. Besides, data transfer has become extremely fast and reliable. The world becoming a global village can be largely attributed to the Internet and its services.
2. **Information:** With the Internet, useful information can be obtained with great ease which makes it indispensible for students, researchers, market analysts etc.
3. **Entertainment:** Games, chat room, browsing websites, audio/video streaming are great sources of entertainment.
4. **E-commerce**: Business deals/extensive online shopping (regardless of the product requirement) are among the promising services provided by the Internet.
5. Online banking, ticket reservations, job search are also among the other important services of the Internet.

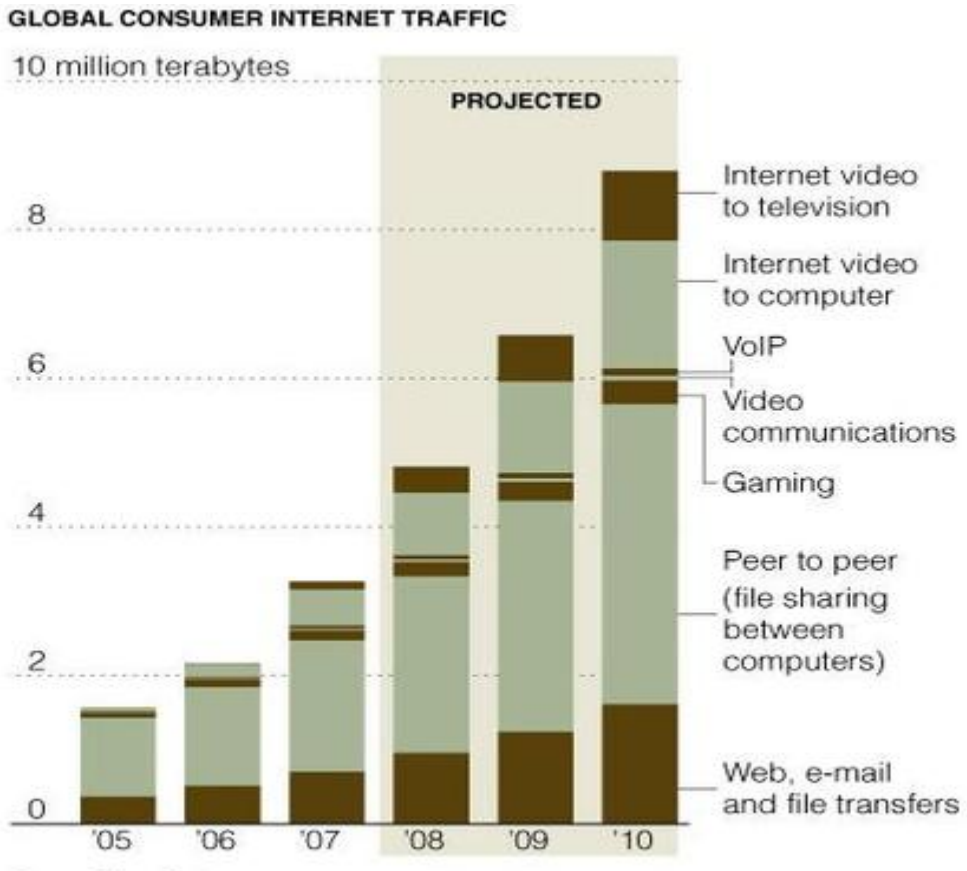


Figure 1: Exponential growth in the Internet traffic (source: CISCO systems survey) [55]

Figure 1 shows the growth of data traffic in the Internet for the past five years and the expected traffic for 2010. As it is observed in the picture the peer to peer file sharing, Internet video to television/computer and web, email applications constitute to a major part of the rapid increase in the global Internet traffic.

1.2 TCP in the Internet

Having described the advantages of the internet we now describe the importance of the TCP protocol in the Internet-protocol suite. Transmission Control Protocol (TCP) is one of the core protocols of the Internet protocol suite. It provides reliable, connection-oriented data transfer, byte-stream service and

is a full-duplex protocol. The TCP functions include flow control [41], congestion control [40]; reliable data transfer along with multiplexing/demultiplexing. TCP is an end-to-end protocol. That is, TCP turns a host-to-host packet delivery service, provided by IP, into a process-to-process communication.

TCP was first introduced in 1981. Figures 2, 3, 4 show the timeline and the development of Internet from early 1940 to 2006. The figures show the milestones of TCP design and development to the development of the Internet and highlight the importance of TCP in the development and growth of the Internet since its inception to the recent years. As shown in Figures 2-4, the Internet has undergone drastic changes. Previously 1440 bps (Bits per Second) of bandwidth was very huge, but now this is hardly enough for the web applications like email, file transfer, IPTV, online games, multimedia applications, large scale science collaborations, VoIP etc. A more recent challenge is to support these applications on the wireless handheld devices without compromising the QoS. To meet these growing data traffic demands, switching infrastructures as well as the transmission algorithms/techniques need to be improved greatly. As the standard TCP/UDP transport protocols are still dominant with considerably good performance in current Internet, the proposals to improve the overall performance are mostly based on the original TCP principles.

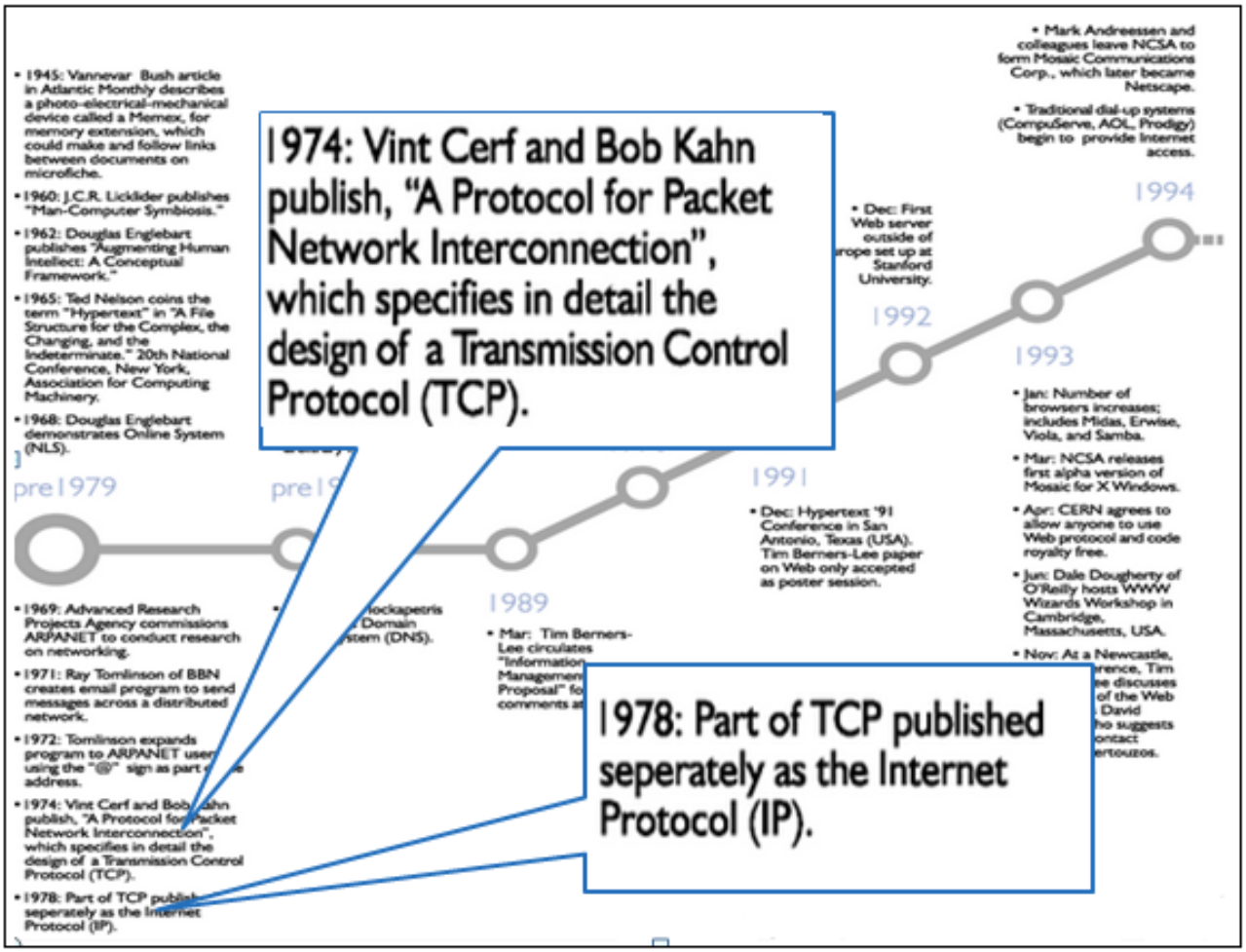


Figure 2: TCP milestones in 1970s [52]

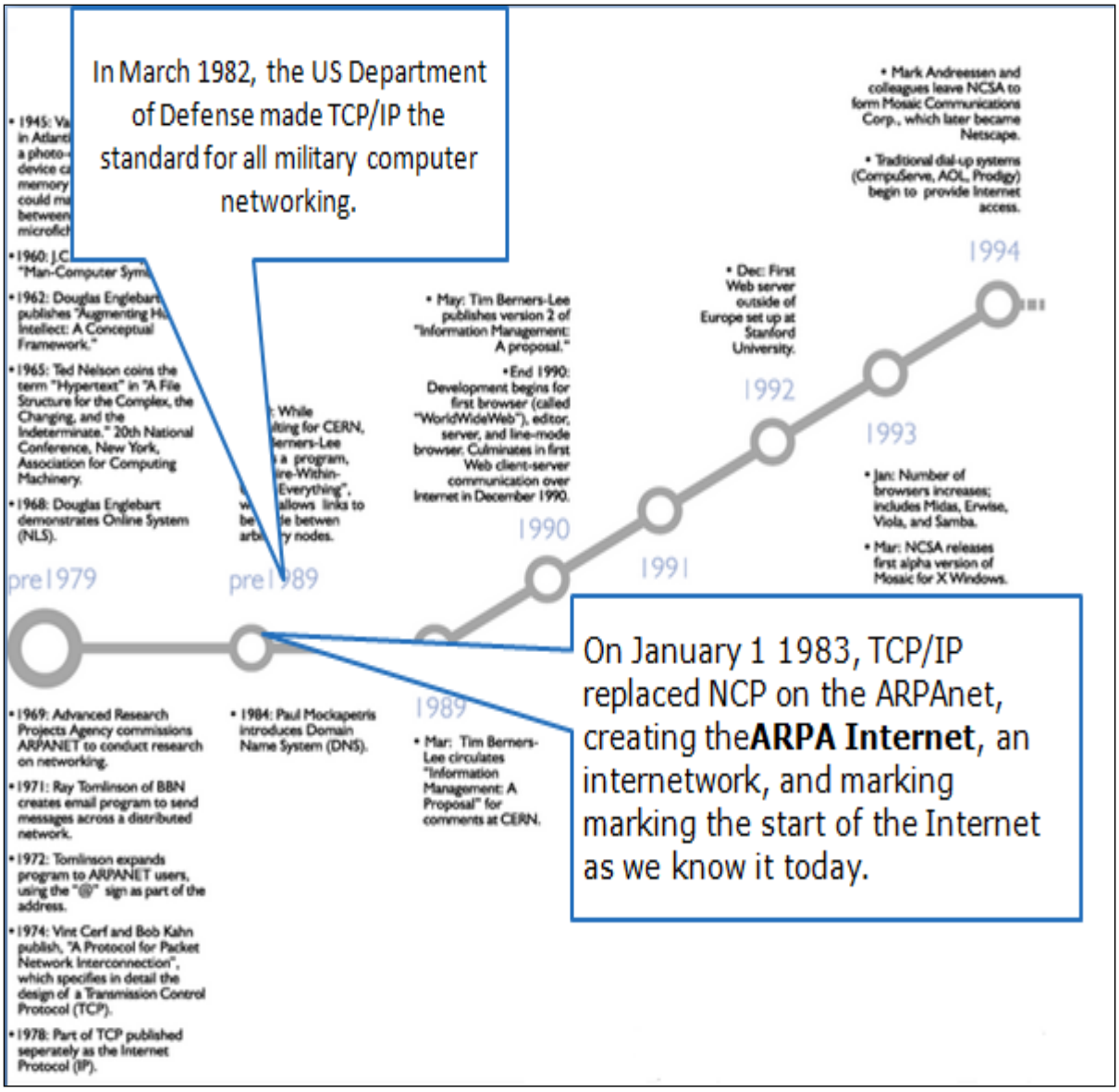


Figure 3: TCP milestones in the 1980s

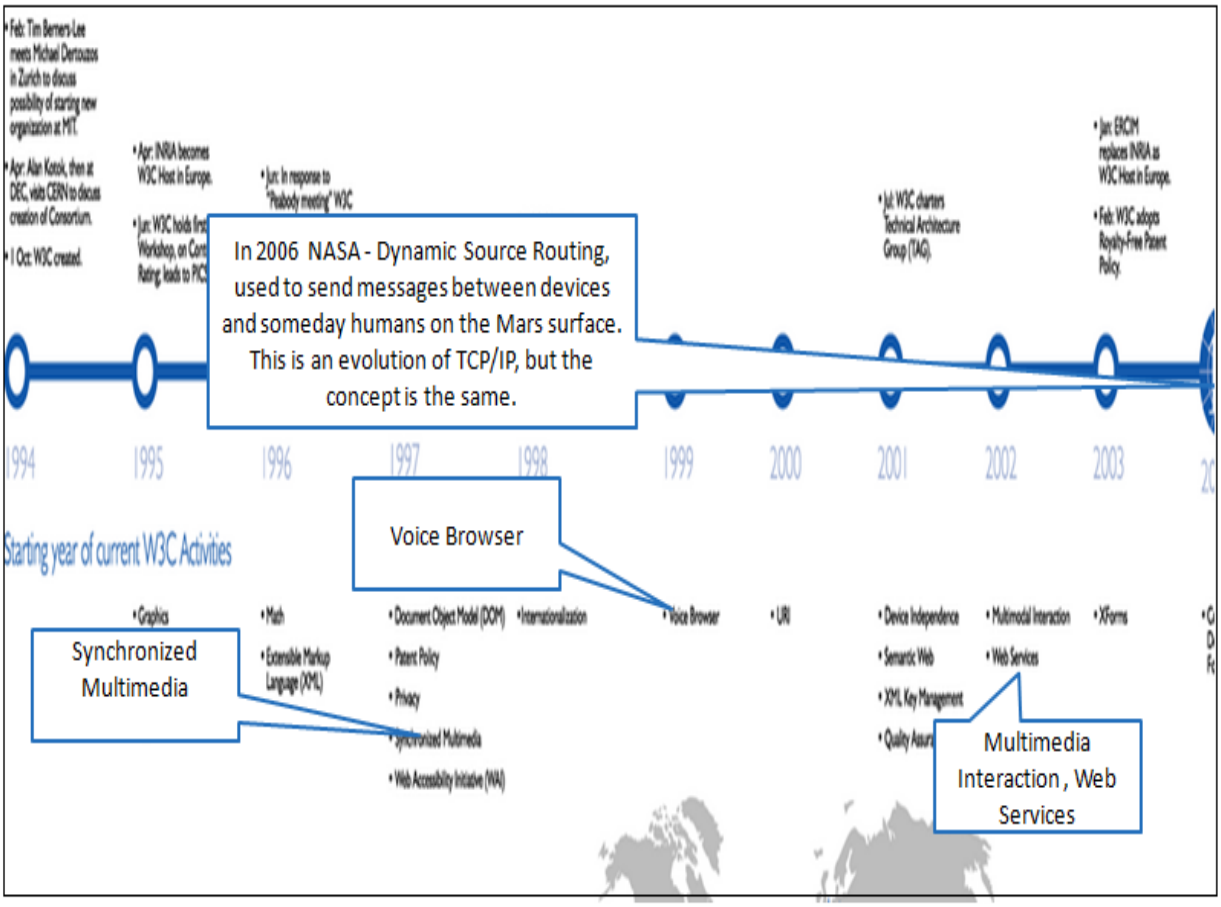


Figure 4: TCP milestones from 1994 to 2006

In particular, recent research has identified the following challenges faced by the conventional transport control protocol.

1. TCP is unable to efficiently utilize the huge bandwidth provided by high-speed and optical infrastructures.

2. TCP cannot satisfy Terabyte/Perabyte data transfer requirements for future data networks.

To further emphasize the above points let us consider Figure 5, which shows the bandwidth utilization of the standard/conventional TCP applications for a 24 hour period. Figure 5 clearly shows that the bandwidth utilization is less than 30% for the TCP based applications.

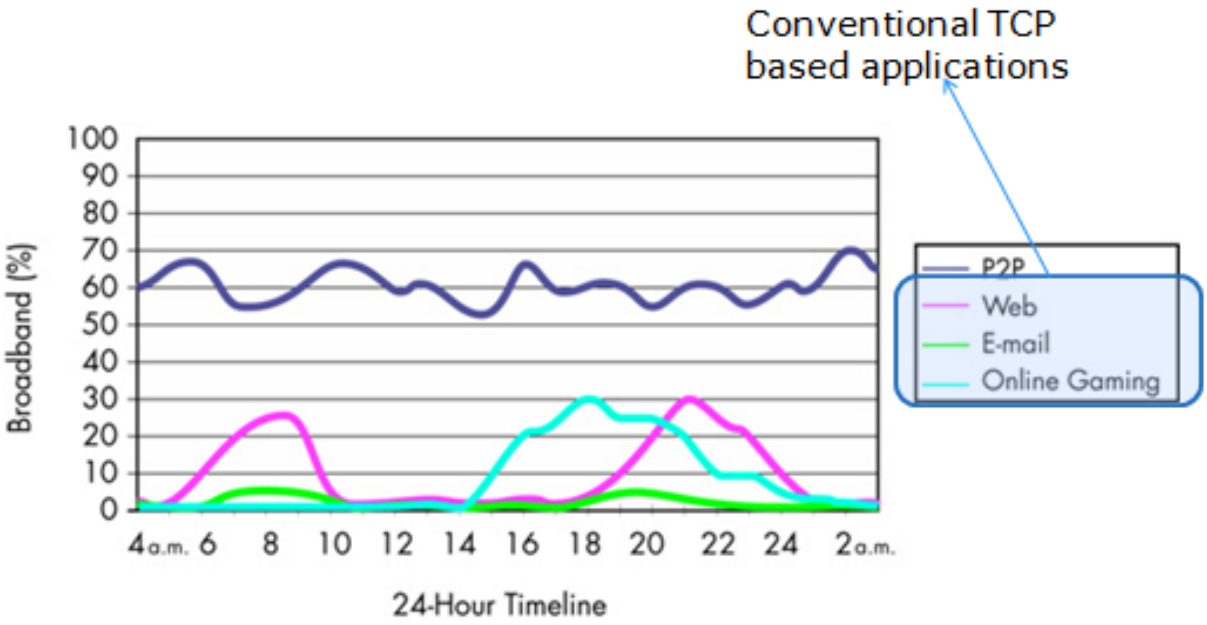


Figure 5: Bandwidth utilization by TCP [58]

Hence there is a need for a study on how to enhance the TCP performance in high-speed optical networks. In this work, we propose a new TCP protocol, namely Stratified TCP or STCP to modify the TCP behavior with more aggressive bandwidth probing and better utilization. A brief introduction to Stratified TCP is given in the next section.

1.3 Overview of the proposed Stratified TCP

Stratified TCP (STCP) is a protocol designed to utilize bandwidth in an aggressive and efficient way. It is based on the principle of virtual layers. In this technique, data transmission starts off with a single layer (as in conventional TCP) and the layers are gradually increased based on certain criteria. To determine the criteria for increasing the number of layers, STCP applies the TCP sending rate equation and modifies

the equation to make it more dynamic to the changing network conditions. Hence STCP owns three major features:

1. Aggressive bandwidth utilization
2. Responds to network changes
3. Keeps the basic principle of TCP congestion control unchanged.

We have implemented the STCP using the well-known Network Simulator - ns-2[18]. The simulation results indicate that the new proposed TCP scheme outperform of the conventional TCP in high-speed networks. STCP's performance is also better than that of Layered TCP [88], which is another protocol based on the principle of "virtual layers" especially for large random networks with varying bandwidths/delay values. Hence, STCP can be considered a promising choice to achieve efficient bandwidth probing in high-speed networks.

The rest of this report is organized as follows. In Chapter 2 and 3 we present the literature survey of the protocols which have been proposed in order to overcome the performance issues of TCP. The general TCP-related proposals for different network environments are described in Chapter 2. In Chapter 3 the parallel transmission techniques for transport control protocols are described. In Chapter 4 we introduce the proposed Stratified TCP (STCP) protocol in detail. The simulation results are then presented and analyzed. Finally, Chapter 5 concludes this work. The related refererences and source codes are also appended at the end of this report.

2 CURRENT PERFORMANCE ISSUES OF TCP AND RELATED INVESTIGATIONS

TCP is an essential protocol of the "Internet Protocol Suite". Although TCP is a hugely successful protocol it continuously faces new challenges. Thus, in this paper, an attempt is made to elaborate major performance issues in TCP and the various solutions proposed to solve these problems. We provide further improvement to increase TCP performance to meet the ever increasing needs of the 21st century. Our problem statement is illustrated as follows:

- Firstly, to find out in detail about the research/work done in improving TCP with respect to various infrastructures like high-speed, optical networks, satellites, wired and wireless networks. Eventually, to recognize the TCP performance issues, find out the proposed solutions/ improvements and suggest other ways of increasing TCP performance
- To propose and implement a protocol that modifies the original TCP behavior to provide increased performance as compared to conventional TCP and hence provides both **reliable** and **fast** data transmission for **high-speed networks.**

We start with a brief history of TCP.TCP was first introduced in 1981.Since its introduction, competitive technologies such as Unix to Unix Copy Program (UUCP) and networks like BITNET were successfully developed based on the TCP. In the 1990s, Asynchronous Transfer Mode (ATM) [33] was introduced as the ultimate unifying network solution. ATM provided a number of features, such as Quality of Service (QoS) that were highly desirable and missing from TCP/IP protocol stack. Though ATM was initially successful it did not manage to play a central role in the success story of the Internet. Today it is viewed as yet another network infrastructure on top of which a TCP/IP protocol runs.

All through these years, TCP evolved to meet the needs of these new technologies. Improvisations have still kept TCP to be robust, scalable, fault-tolerant, and well-performing. But off late, due to the rapid growth of the Internet population, user demand for diversified services has increased. Also, the rise of wireless and mobile computing has further increased the challenges. Some of these applica-

tions require high-quality transport services in terms of, for example, end-to-end throughput, packet loss ratio, and delay.

Thus a study of the present situation and making an effort to provide better performing techniques/ideas is essential. The chapter is divided into multiple modules; each module discusses a problem and the proposed solutions.

2.1 Performance Issue 1: TCP for Wireless Environments

TCP is designed mainly for wired networks, with stationary nodes to provide reliable transport. In wireless networks the TCP performance is not very good. The main reason is the violation of the main assumption in TCP, which says that the packet loss is due to the network only. In mobile networks packet loss can occur due to transmission errors or unavailability of routes. But TCP does not distinguish between these reasons and reduces the congestion window in response, causing unnecessary degradation of throughput. Other problems in "mobile Ad-Hoc networks" [17] [56] are - node mobility, unpredictable radio medium, external interference/jamming, multiple access contention, frequent route changes, high bit rates and network partitions.

2.1.1 Explicit Link Failure Notification (ELFN) and TCP-Feedback (TCP-F) Network Congestion [1]

ELFN and TCP-F are proposals made to **prevent TCP from "wrongly reacting" to packet losses caused by link failures**. In mobile Ad-Hoc networks (MANET), frequent link breakages due to mobility are one of the major factors degrading TCP performance. When link breakage happens, TCP sender will encounter continuous packet losses over an extended period. Its congestion window is reduced and the TCP retransmission timeout (RTO) becomes progressively larger leading to high restart latency and very poor efficiency. Specifically, during the period of link breakage, the more packets sent out by the TCP sender, the more network resource is wasted. A typical solution is to detect the link failures and freeze the TCP

state (including congestion window size and RTO interval) until a new path is re-established. Example schemes include Explicit Link Failure Notification (ELFN) [1] and TCP-Feedback (TCP-F) [1].

Both ELFN (this scheme is similar to Explicit Congestion Notification (ECN)) and TCP-F rely on the intermediate nodes to report the link breakage. In TCP-F (TCP-Feedback), the intermediate node is asked to notify the TCP sender about the network condition. When one intermediate node detects a route failure, it explicitly sends a route failure notification (RFN) to the TCP sender. The difference between TCP-F and ELFN is the response of route failures. TCP-F relies on the intermediate node to send a route reestablishment notification (RRN) to notify that the path is back up. In ELFN, the TCP sender must send probing packets periodically to detect the route recovery.

Solutions to prevent exponential back off of RTO were also proposed, one such scheme is called "Fixed-RTO" [83]. The key idea is the following: when consecutive retransmission timeouts happen for the same data packet, just double the RTO the first timeout, and keep the same value for the subsequent timeouts. Through simulation experiments, the authors show that with fixed-RTO, TCP can achieve throughput comparable to that of the ELFN mechanism [4]. However, ELFN requires support from the intermediate nodes, while fixed-RTO is a pure end-to-end mechanism.

However, there is a complication in the above schemes; if link failures happen frequently, TCP will still suffer significant performance degradation even when the above schemes are applied, since the TCP sender may go to the frozen state repeatedly and simply wait for route reestablishment without sending any new data packets. To overcome this "impasse", another solution is to improve the path availability using multipath routing. Multipath routing maintains several paths to the same destination simultaneously (the conventional Ad-Hoc routing protocols usually only keep a single path to each destination). Thus, the probability that there is no path from the TCP sender to the receiver is effectively reduced. Simulation results show that by careful selection of the multipath routing strategies, we can improve TCP performance by more than 30% even under very high mobility [1].

2.1.2 Ad-Hoc TCP (ATCP)

One of the approaches to improve TCP performance in wireless Ad-Hoc networks is to **modify the network stack slightly**. ATCP [10] is one such approach. Here, a thin layer between Internet protocol and standard TCP greatly increases the end-to-end TCP throughput. ATCP basically uses the feedback from the network layer, in terms of the disconnection and connection signals, and modifies the congestion control mechanisms of TCP. It uses this feedback information to regain the full window after the mobile host gets reconnected. ATCP can be used when the TCP sender is either a mobile host (MH) or a fixed host (FH). Further network stack needs to be modified only at the MH. Experiments show that ATCP is 40% better than TCP Reno in WLAN environments and up to 150% better in WWAN environments [10].

2.1.3 TCP Vegas [48] [53]

TCP Vegas is a **congestion control algorithm**. The main principle of TCP Vegas is that it uses the "packet delay" rather than "packet loss" as a basis to determine the packet-send rate. TCP Vegas detects congestion at an incipient stage based on increasing Round Trip Time (RTT) values of the packets in the connection unlike other flavors like Reno, New Reno etc, which detect congestion only after it has actually happened via packet drops. The algorithm depends heavily on accurate calculation of the Base RTT value. If it is too small then throughput of the connection will be less than the bandwidth available. If the value is too large then it will overrun the connection. However, it is observed that 'TCP-Vegas' seems to be unfair when operated with other protocols. Some solutions are also proposed to overcome this unfairness; two approaches to improve the fairness are mentioned [48]. The first one is to modify the congestion control algorithm of TCP Vegas, and the other is to modify the RED algorithm to detect misbehaved connections and drop more packets from those connections.

TCP Vegas greatly increases performance as compared to TCP Reno in wireless environments. Enhancements to TCP Vegas (modifying the congestion avoidance mechanism of the TCP Vegas) further improve performance. One such solution is named TCP Vegas_M [13], it is dependent on FEDM fuzzy

logic theory to distinguish network congestion states and wireless channel states. Based on Ad-Hoc networks states (congestion, uncertain, and bit error), the TCP sender adopts different congestion avoidance [15] algorithms to control congestion window size. The aim is to optimize Ad-Hoc network throughput and achieve better performance. Studies in [13] show that TCP Vegas-M is able to overcome several of the identified problems and outperform the TCP Reno and TCP Vegas while avoiding the oscillation of the TCP sender congestion window.

2.1.4 Cross-Layer Interaction of TCP [19]

To distinguish between the packet loss due to network and the packet loss due to mobility errors using feedback or information from the lower layers, two policies are proposed as explained below [19].

- The Fractional window Increment policy –The congestion window parameters are optimally chosen while preserving the TCP window mechanism.
- Route failure notification using Bulk-Loss Trigger policy.(ROBUST)

The ROBUST policy is a simple link loss reaction policy for on-demand routing to improve the path robustness against the MAC contention loss driven by congestion. Figure 6 shows the interaction model for the cross layer mechanism.

However, in IEEE 802.11 multihop networks, the link loss has to be treated differently according to its cause such as:

. Node mobility

. Congestion

. Channel noise

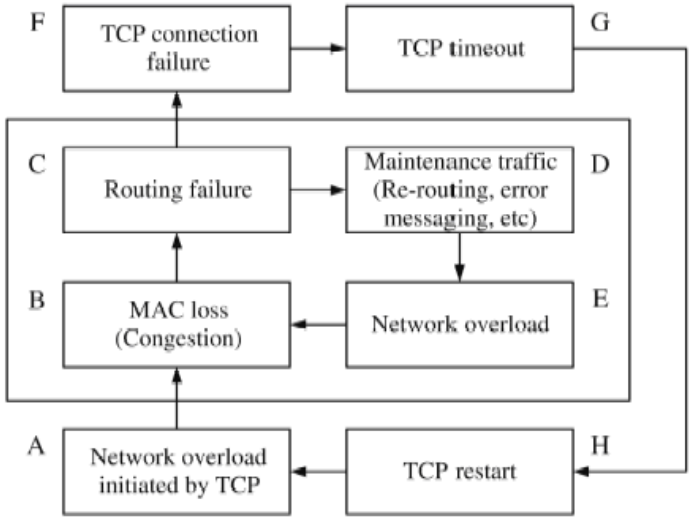


Figure 6: Multi loop cross layer interaction model. [19]

2.1.5 TCP- Veno [5]

The difference between Veno and the conventional TCP is: TCP-Veno monitors the congestion level using an end-to-end estimation algorithm and uses that knowledge to refine the congestion control algorithm of TCP. i.e. Firstly, it refines the multiplicative decrease algorithm of Reno by setting the **congestion threshold** – a key control parameter in TCP - according to the perceived congestion level of the network instead of a fixed drop factor when packet loss is detected. Secondly, it refines the linear increase algorithm by attempting to stay longer in an operating region in which the bandwidth is fully utilized. Simulations show that throughput improvements are as high as 80%, compared to TCP-Reno [5]. Detailed investigation indicates that the improvements are brought about without any harm to concurrently running Reno connections (unlike TCP-Vegas). In addition to wireless networks, Veno can also achieve better improvement relative to Reno in high-latency networks (e.g., networks including satellite links) and asymmetric networks (e.g., networks with ADSL access link).The major advantages of TCP-Veno are:

1) Can achieve better performance than Reno in random-loss, high-latency, and asymmetric environments

2) Can co-exist with concurrent Reno connections without degrading the performance of Reno connections

3) Can be deployed easily with only minor modifications of the sender-side algorithm.

An enhancement of TCP Veno with 'Forward Acknowledgement' was also proposed in [5] known as "TCP Veno with FACK". An additional advantage of the "TCP Veno with FACK" (other than the three advantages of TCP-Veno mentioned above) is - better utilization of link bandwidth.

The main cause of bandwidth under-utilization in TCP-Veno is that it has only refined the Reno AIMD algorithms, leaving traditional Fast retransmit and Fast recovery (FF) algorithm intact. FF will cut congestion window several times if multiple packets are lost within a window of data, and drive the TCP connection into timeout. This causes TCP performance degradation.

TCP Veno-F aims to improve Veno's performance through better recovery scheme

The Veno-F mechanism comprises of two parts, one is **Veno AIMD**, and another is **Forward Ack scheme**.

1. Veno AIMD makes use of the idea of congestion detection scheme in Vegas and intelligently integrates it into Reno's congestion avoidance phase.
2. Forward ACK is evoked when packet loss is detected. The Forward ACK uses SACK (Selective ACK) to report missing segments, and then uses segment parameters and the recovery algorithm. [5]

Network uncertainties prevent the accurate identification of the packet-loss causes. The TCP Veno+ [22] brings in additional accuracy in this regard as compared to the typical TCP-Veno protocol in bursty congestion environments. Another advantage of this protocol is that it maintains better friendlessness to TCP Reno.

2.1.6 TCP "Adaptive Selection" Concept [20]

As mentioned earlier, some of the major challenges posed by mobile networks and satellites to TCP are: long round trip times (RTTs), not negligible packet error rates (PER), and very large bandwidths. However, as most proposals aim to address different impairments, they result in optimizations for specific network environments. Therefore, given the increasing level of heterogeneity of present and future networks, the choice of TCP enhancement seems a quite irresolvable problem, depending on the characteristics of the specific connections. The TCP adaptive-selection concept, aims to circumvent this problem by providing an alternative approach that challenges at the root the idea that only one TCP enhancement must be adopted, not only on the whole network in general, but also on the same server machine. In fact, by extending the concept that underlies adaptive coding and modulation (ACM) to transport layer, TCP adaptive-selection envisages the concurrent adoption of different TCP versions on the same server, the better to match the different impairments present on different connections. Current approaches to avoid performance losses can be categorized into two classes:

(a) Preserving the end-to-end semantics of TCP, and

(b) Violating the end-to-end semantics of the TCP and requiring the intervention of intermediate nodes.

In the former approach, the TCP sender and receiver attempt to differentiate between network congestion and route failures without the intervention of intermediate nodes. In the latter schemes, however, some messages like explicit route failure and route recovery notification or explicit congestion notification are used to allow a TCP sender to be notified and to manage its congestion window size accordingly. Other approaches use the delayed ACK technique efficiently or they restrict the size of the congestion window to a small value so as to avoid excessive channel contention and interferences.

One of the specific proposed solutions includes piggybacking the ACK sequence number onto routing control packets. This allows a TCP sender to be correctly informed of the sequence number that the TCP receiver expects to receive, avoiding spurious retransmissions [21].

An additional field of this ACK sequence number can be easily added into routing control packets by conforming to the IETF MANET packet BB format. Simulation results obtained from both static and dynamic networks show that overall; our approach outperforms general TCP in terms of aggregate throughput and in reducing the number of spurious retransmissions.

TCP trunking [49] is another related method to improve performance dynamically. A TCP trunk is an aggregate traffic stream whose data packets are transported at a rate dynamically determined by TCP's congestion control.

TCP ACK agent and auto-zoom back off algorithm [23]: Schemes especially for 802.11 WLAN with increased TCP performance are mentioned here. A TCP data segment is acknowledged twice, once at the MAC layer, and once at the TCP layer. By a simple cross-layer design, a TCP ACK agent is installed at the WLAN AP. When a MAC acknowledgment is received by the AP, the AP generates a TCP ACK on behalf of the WLAN stations.

Auto-zoom back off algorithm is also proposed to further improve the access performance. In the auto-zoom back off, the contention windows can be progressively reduced to a very low value in case of light or asymmetric traffic, and return to normal quickly when collision occurs. With the TCP ACK agent and auto-zoom back off, we show that significant improvement in TCP throughput performance can be achieved in typical Internet application scenarios.

2.1.7 Schemes to Enhance Performance during Handoffs

2.1.7.1 An Adaptive TCP Algorithm to Support Handoffs in Heterogeneous Overlay Networks [37] [69]

Here we discuss an adaptive TCP algorithm which was proposed to support handoffs in heterogeneous overlay networks. The algorithm is implemented as a collection of units. The main body, namely the TCP implementation unit, considers the optimization of TCP parameters over different wireless networks. This unit is decomposed into three modules: transmission control, congestion control, and error detec-

tion. Based on this module decomposition, several TCP versions are re-constructed, and the different combinations of their congestion control and error detection modules are studied. Units in the adaptive TCP algorithm, such as the real-time report and throughput evaluation units are triggered when a mobile user experiences acute changes in access parameters due to handoffs, after which TCP parameters or TCP module combinations in the TCP implementation unit are re-configured, thus optimizing the TCP performance over heterogeneous wireless networks.

2.1.7.2 Improving TCP performance during soft vertical handoff [45]

TCP performance degradation also occurs due to the change of network bandwidth and propagation delay. During vertical handoff, some undesirable phenomenon may erroneously trigger TCP congestion control operations and thus degrade the TCP performance. When handing over from a fast link to a slow link, one of the problems that TCP may encounter is that the fast link transmission does not tolerate the long delays on the slow link, which could cause TCP timeout.

Proposed in [45] are three schemes to prevent the TCP Timeout: fast response, slow response, and ACK delaying. The simulation results have demonstrated that these schemes can effectively reduce the gap between the TCP delays, experienced on the old fast link and the new slow link so that TCP can gradually adjust **its** RTO to fit into the new network environment.

2.1.8 Contention-based Path Selection (COPAS) [39]

A less researched problem in low-mobility networks is caused as a result of the interplay between the MAC layer and TCP back off policies, which causes nodes to unfairly capture the wireless shared medium, hence preventing neighboring nodes to access the channel. This has been shown to have major negative effects on TCP performance comparable to the impact of mobility.

A proposed solution to this problem is COPAS (COntention-based Path Selection). The key features of this mechanism can be described in two major steps:

1. It uses disjoint forward (sender to receiver for TCP data) and reverse (receiver to sender for TCP ACKs) paths in order to minimize the conflicts of TCP data and ACK packets.
2. COPAS employs a dynamic contention-balancing scheme where it continuously monitors and changes forward and reverse paths according to the level of MAC layer contention, hence minimizing the likelihood of capture.

Through extensive simulation, COPAS is shown to improve TCP throughput by up to 90% while keeping routing overhead low.

2.1.9 Paced TCP [38]

The performance of TCP degrades greatly in multi-hop Ad-Hoc networks due to the fact that the traditional TCP is unable to adapt the unique properties of the IEEE 802.11 wireless technology.

One of the algorithms to overcome this problem is a pure rate-based TCP, Paced TCP to dynamically probe the network's bandwidth which reduces packets dropping from link layer. Paced TCP is able to alleviate the MAC channel contentions and therefore provides better performance. By alleviating packets dropping from link layer, the number of retransmissions at TCP layer can be decrease, and therefore a more efficient communication can be established. With Paced TCP, less retransmissions than TCP Reno is demonstrated. With NS-2 simulator, we show that by probing networks dynamically with pacing, packets dropped by the MAC channel contentions can be reduced, and Paced TCP is able to provide better fairness and throughput stability between competing flows.

2.2 Performance Issue 2: Switching from Wired to Wireless Networks

TCP connection can be established over two completely different classes of sub networks namely, wired and wireless. Therefore, each TCP connection could be split into two connections at the point where the two subnetworks meet. For example, suppose we have a mobile user who is browsing a web site using his laptop. The TCP connection will be split into two: one between the mobile host and the base station

[44] and one between the base station and the web server. The advantage is that we can utilize the best transport protocol for each type of network. By splitting TCP connections we lose the end-to-end semantics of TCP. Consequently, we do not have a process-to-process communication channel. Intermediate nodes act as proxies, inspecting and modifying every single segment. In addition, the performance may also be degraded by splitting a particular connection several times. Handoffs are also not handled as efficiently, and crashes in the base station result in TCP connection termination [70]. We explain some of the solutions proposed to improve TCP performance when it switches from wired to wireless networks in the next section.

2.2.1 Split TCP Connections [66]

Indirect TCP (I-TCP) [67] protocols can isolate mobility and wireless related problems using mobility support routers (MSRs) as intermediaries, which also provide backward compatibility with fixed network protocols. Throughput comparison with regular (BSD) TCP shows that I-TCP performs significantly better in a wide range of conditions related to wireless losses and host mobility. Figure 7 compares the performance of TCP and I-TCP.

WIDE AREA PERFORMANCE COMPARISON WITH HANDOFFS

Protocol	No moves	Overlapped cells	Nonoverlapped cells with 0 sec. b/w cells	Nonoverlapped cells with 1 sec. b/w cells
FH to MH throughput in Kbytes/sec.				
Regular TCP	13.3	13.3	8.9	5.2
I-TCP	26.8	28.0	19.1	16.0
MH to FH throughput in Kbytes/sec.				
Regular TCP	31.0	30.0	16.9	10.6
I-TCP	71.3	61.7	57.4	46.4

Figure 7: Throughput comparison of TCP and I-TCP [66]

2.2.2 Wireless TCP Model for Short-lived Flows [26]

TCP performance affects overall network performance; many studies have been done to model TCP performance in the steady state. However, recent research has shown that most TCP flows are short-lived. Therefore, it is more meaningful to model TCP performance in relation to the initial stage of short-lived flows. In addition, the next generation Internet will be unified all-IP network that includes both wireless and wired networks integrated together. In short, modeling short-lived TCP flows in wireless networks constitutes an important axis of research.

Proposals are made especially for short-lived flows. These wireless TCP schemes are of three types: end-to-end schemes, split connection schemes, and local retransmission schemes. It is difficult to illustrate the models in detail but experiments have shown that the proposed model provides a satisfactory means of modeling the TCP performance of short-lived wireless TCP flows. Also, the above TCP model is for both wired and wireless networks.

2.3 Performance Issue 3: TCP over Satellite [3]

There are three factors that most affect throughput for TCP/IP over a satellite channel. They are

- Long Feedback Delay.
- Large Bandwidth-Delay Product.
- Transmission Errors.

Similar to the mobile networks, TCP in satellite also cannot recognize that corruption and not network congestion caused it to reduce its sending rate. Additional factors that serve to reduce throughput include asymmetric routing and variable RTTs. Before we start with the performance issues of TCP used for satellites let us understand the congestion control mechanisms for satellites. It has two phases:

- **The Slow-Start Algorithm.** Slow-start, as the name implies, causes a TCP sender to gradually increase the amount of data injected into the network following connection establishment, the restart of an idle connection, or a TCP connection time-out.
- **The Fast Retransmit Algorithm.** Fast retransmit enables a TCP sender to rapidly recover from a single lost packet, or one that is delivered out of sequence, without shutting down the CWND. When a TCP receiver detects the loss of a packet, it acknowledges subsequent packets with the ACK number of the last correctly received packet. When the TCP sender receives three duplicate ACKs, it then retransmits the lost packet. The receiver responds with a cumulative ACK for all packets received up to that point.

2.3.1 Performance Enhancement for TCP over Satellite [32]

- ***Large Windows***

Large windows are required for other large bandwidth- delay networks such as ATM, Gigabit Ethernet, and Packet SONET, so that just about all commercially available TCP implementations now support the large window options.

- ***Delayed ACKs***

Instead of generating an ACK for each received segment, a TCP receiver may choose to generate an ACK for every second segment that arrives or, if a second segment does not arrive, wait for a time-out period of up to 500 ms before generating the ACK.

- ***Larger Initial Window***

Slow-start uses an initial window size of one. Starting off with a larger initial window size of three or four segments will allow more segments to flow into the network, generating more ACKs, and will decrease the time it takes to complete the slow-start process.

- ***TCP SACK [25]***

TCP selective acknowledgments enable a TCP receiver to inform the sender of what specific segments were lost so that the TCP sender can retransmit them.

2.3.2 IPSEC over Satellite Links: A New Flow Identification Method [54]

Acknowledgment based transport protocols such as TCP have low performance in satellite links, which are characterized by high latencies and high bit error rates. Low performance of TCP in satellite links is due to the fact that TCP packet losses are assumed to be the cause of congestion in the network, which turns out to be an invalid assumption for satellite links.

Proposed solution: TCP performance enhancing proxies (PEPs) are widely used to overcome the limitations of TCP over satellite links. However, when end-to-end security mechanisms, such as IPSEC, are used, TCP PEP mechanisms can not be used. IPSEC encrypts and/or authenticates the packet header fields that the PEP needs to read or modify. However, mechanisms to integrate IPSEC with TCP PEPs are also proposed.

One such method is described in some detail below. A cryptographic hash of flow identification information is generated and stored in the IP header. The TCP sequence number is also stored in the IP header. Using the hash value and sequence numbers, the PEP is able to match packets and corresponding acknowledgements to regulate the flow. This approach is applicable to PEP mechanisms that need read access to the IP and TCP headers.

2.3.3 TCP-Peach [61]

Current TCP protocols have lower throughput performance in satellite networks mainly due to the effects of long propagation delays and high link error rates.

Solution proposed: TCP-Peach is introduced for satellite networks. TCP-Peach is composed of two new algorithms, namely Sudden Start and Rapid Recovery, as well as the two traditional TCP algorithms, Congestion Avoidance and Fast Retransmit. The new algorithms are based on the novel concept

of using dummy segments to probe the availability of network resources without carrying any new information to the sender. Dummy segments are treated as low-priority segments and accordingly they do not affect the delivery of actual data traffic. Simulation experiments show that TCP-Peach outperforms other TCP schemes for satellite networks in terms of good put. It also provides a fair share of network resources. [61].

2.3.4 Split TCP Connections in Satellites [3]

Satellite systems offer greater challenges to TCP when switching from wired to wireless networks especially due to higher data rates and high altitude satellites with longer delays and for handling broadband Internet applications. In these scenarios, transmission control protocol (**TCP**) plays a critical role. Here, the splitting the TCP connection in two or more segments with one segment connecting terrestrial nodes across the satellite network is implemented.

An evolution of this idea: placing a TCP proxy on board the satellite that further subdivides the end-to-end connection into separate TCP connections between ground and space. In this method we need to focus upon the efficient use of buffer resources on board the satellite, while at the same time enhancing TCP performance. Simulations show that an on-board proxy provides a number of distinct advantages and can enhance throughput up to threefold for both TCP New Reno and TCP Westwood, in some scenarios, with relatively modest on-board buffering requirements. The main points of concern in this method are: the on-board split proxy concept, the buffer management strategy.

2.3.5 Network Striping for Satellites: Split TCP

Several satellite systems currently in operation or under development claim to support broadband Internet applications. In these scenarios, transmission control protocol (TCP) plays a critical role. Unfortunately, when used with satellite links, TCP suffers from a number of well-known performance problems, especially for higher data rates and high altitude satellites with longer delays. In response to these difficulties, the satellite and Internet research communities have developed a large gamut of solutions

ranging from architectural modifications to changes in the TCP protocol. Among these, one approach requiring minimal modifications involves splitting the TCP connection in two or more segments with one segment connecting terrestrial nodes across the satellite network. **Evolution of this basic idea would be to place a TCP proxy on board the satellite that further subdivides the end-to-end connection into separate TCP connections between ground and space**; this is the main principle of the spilt TCP for satellites. The main contributions of this protocol are: the on-board split proxy concept, the buffer management strategy, and enhancement of TCP performance.

Split TCP explanation: The classical "split" TCP concept is extended to a solution where the split occurs on board of satellite. Here, a *forwarding (proxy) agent [11] [14]* on the satellite maintains two separate split TCP connections for each end point of the TCP session. By splitting the TCP connection on board, the benefits are:

1) We increase the speed of error recovery

2) We reduce the propagation delay on each link.

2.4 Performance Issue 4: TCP Fairness

2.4.1 High-Speed TCP Protocols with Pacing for Fairness and TCP Friendliness [79]

Recent studies have pointed out that existing schemes have a severe RTT unfairness problem, where competing flows with different RTTs can consume considerable unfair bandwidth shares. Burstiness is one of the main reasons behind such problems. As the congestion window achieved by a high-speed TCP connection can be quite large, there is a strong possibility that the sender transmits a large burst of packets. As such, the current congestion control mechanisms of high-speed TCP can lead to bursty traffic flows in high-speed networks, with a negative impact on both TCP friendliness and RTT unfairness.

The proposed solution to these problems is to evenly space, or pace packets sent into the network over an entire round-trip time, so that packets are not sent in a burst. Evaluations made for this

approach in a high “bandwidth-delay product” network shows that pacing offers better TCP friendliness and RTT fairness without degrading the bandwidth scalability. [24]

2.4.2 Window Adjustment Method to Enhance TCP efficiency and Fairness [12]

In case of a handover [27] [28] occurrence, a TCP sender may be forced to be sharing a new set of satellites with other users resulting in a change of flows count. Proposals are made saying that TCP rate of each flow should be dynamically adjusted to the available bandwidth when the number of flows that are competing for a single link, changes over time. The above scheme matches the aggregate window size of all active TCP flows to the network pipe. At the same time, it provides all the active connections with feedbacks proportional to their round-trip time values so that the system converges to optimal efficiency and fairness. Feedbacks are signaled to TCP sources through the receiver's advertised window field in the **TCP** header of acknowledgments. Senders should accordingly regulate their sending rates. The proposed scheme is referred to as explicit and fair window adjustment (XFWA) [12]. Extensive simulation results show that the XFWA scheme substantially improves the system fairness, reduces the number of packet drops, and makes better utilization of the bottleneck link.

2.4.3 Utilizing TTL to Enhance TCP Fairness [68]

Among the methods to improve TCP fairness, some queue management schemes, such as FRED, Balanced RED, have been developed, but most of them require maintaining per-flow state in routers. Some methods require modification of the TCP header to achieve the same, thus it is more complex and requires more work.

[68] Proposes to utilize the already existing TTL field in IP header to improve TCP fairness. A three-dimensional two-category classifier is designed by extending our previous work and based-hops [46] [60] [61] fairness enhancement algorithm (BHFE) for AQM is developed. Simulation results show that it is not only effective to enhance TCP fairness in multiple congested routers but also very well to

keep queue length stable and small. Moreover, a very important advantage of BHFE is that it does not require per-flow state in router and any modifications to the standard TCP/IP.

2.4.4 Gentle High Speed TCP (gHSTCP) [84]

Another algorithm to avoid TCP Unfairness, based on HSTCP is the gHSTCP. It can achieve better fairness with competing traditional TCP flows, while extending the advantage of high throughput provided by HSTCP.

2.5 Performance Issue 5: Delay in Congestion Recovery

The slow recovery upon coarse timeout expiration on long, fat pipes, and the reaction to random segment losses are among the problems affecting the current version of TCP. Both problems are known to reduce the throughput of a connection. Proposals made to solve the above problems are discussed in this module.

2.5.1 TCP Net Reno [50]

An important proposed strategy for congestion recovery is the network-sensitive Reno (Net Reno), a set of optimizations that can be added to a traditional Reno TCP sender. Using the TCP's self-clocking property and the packet conservation rule, Net Reno improves Reno and its variants (New-Reno and SACK), in reducing TCP retransmission time-outs (RTOs) and in being conservative in network usage during the fast recovery phase. It is shown that over 85% of RTOs are due to small congestion windows that prevent fast retransmission and recovery algorithms from being effective. This implies that sophisticated recovery schemes such as SACK will have limited benefits for these loads. Net Reno overcomes this problem with a small window optimization. Net Reno can recover any number of packet losses without time-outs as long as the network keeps at least one packet alive for the connection.

2.5.2 Smooth Start and Dynamic Recovery [51]

[51] Proposes a modified congestion control which reduces the network delays and hence increases the performance. The modified algorithm has two parts:

(1) The smooth-start algorithm, which replaces the slow-start algorithm at the start of a TCP connection or after a retransmission timeout.

(2) The dynamic recovery algorithm, which replaces the fast recovery algorithm [47] [50] to recover packet losses when a TCP connection is congested. Both algorithms require modifications only to the sender side of the TCP implementation.

2.5.3 "Robust Recovery" TCP Scheme [57]

It is a robust TCP congestion recovery - called **Robust Recovery** (RR) algorithm to make a TCP flow more robust to bursty packet losses. One of the key features of RR include: The amount of data in flight is accurately measured, since congestion window size **(cwnd)** over-estimates the number of packets in flight during congestion recovery, stalling data transmission.

Results show that the proposed scheme achieves: at least as much performance improvement as TCP SACK and consistently outperforms TCP New-Reno. Furthermore, since it requires neither selective acknowledgments nor receiver modifications, its implementation and deployment is much simpler than that of TCP SACK, and only the servers in the Internet need to be modified slightly, while keeping intact millions of TCP clients scattered in the Internet.

2.5.4 "TCP smart framing ": Algorithm to Reduce Latency [36]

TCP smart framing, or TCP-SF for short, enables the Fast Retransmit/Recovery algorithms even when the congestion window is small. Without modifying the TCP congestion control based on the additive-increase/multiplicative-decrease paradigm, TCP-SF adopts a novel segmentation algorithm: while Classic TCP always tries to send full-sized segments, a TCP-SF source adopts a more flexible segmentation algorithm to try and always have a number of in-flight segments larger than 3 so as to enable fast recovery.

2.6 Performance Issue 6: TCP Variants for High-Speed Networks

The typical TCP algorithm for high-speed networks is Reno TCP and it is described as below:

TCP's congestion management comprises of the slow start and congestion avoidance algorithms that allow TCP to increase the data transmission rate without overwhelming the network. TCP Reno's congestion avoidance mechanism is referred to as AIMD (Additive Increase, Multiplicative Decrease). In the congestion avoidance phase, TCP Reno increases its congestion window (cwnd) by one packet per window of data acknowledged and halves the congestion window for every window of data containing a packet drop. Thus as seen above; AIMD principle of TCP has the following shortcoming:

When the network can afford more traffic, the increase by just one packet per RTT tends to be too conservative, while at the same time the window reduction by a factor of half, when flow increases tends to be too drastic. This results in inefficient link utilization. Therefore, in order to utilize the existing bandwidth more efficiently the AIMD principle needs to be modified for more aggressive probing of the bandwidth. This led to the development of PIPD (Polynomial Increase Polynomial Decrease) family and the MIMD (Multiplicative Increase and Multiplicative Decrease) of algorithms. These new classes of nonlinear congestion control algorithms are especially useful for applications such as Internet audio and video that does not react well to rate reductions, because the rate reduction technique used for these applications will result into the degradation in user-perceived. In this subsection, we discuss the two models namely, MIMD-Poly and PIPD-Poly in detail.

In theses algorithms the window adjustment policy is only one component of the congestion control protocol. Other mechanisms such as congestion detection (loss, ECN etc.), retransmissions (if required), estimation of Round-trip-time etc., remain the same as TCP. The proposed algorithms mainly aim in increasing the window size faster to gain the bandwidth quicker. [2]

These TCP variants are described in brief below:

- **FAST TCP*:*** FAST TCP is a TCP congestion control algorithm designed for high speed, long latency networks. It is based on TCP Vegas instead of Reno TCP. The key difference is that it uses an equation based window control approach rather than the AIMD algorithm of TCP Reno. Also, unlike TCP Reno, it uses queuing delay and packet loss as the congestion measures rather than just packet loss [8] [30].
- **BIC-TCP:** BIC-TCP is another variant of the TCP congestion control algorithm designed for high speed networks with large delays. Like TCP Reno, BIC-TCP uses packet loss as the congestion measure. However, it uses the binary search technique to increase the congestion window in the congestion avoidance phase.
- **Scalable TCP (STCP) [86]:** Scalable TCP involves a simple sender side change to TCP Reno. The legacy window size *lwnd* as the maximum window size that can be achieved by TCP Reno. Associated with this window size is the legacy loss rate *pl* which is the maximum packet loss rate needed to support window larger than *lwnd*. Scalable TCP uses the Reno congestion window update algorithm given as

ACK: $newcwnd = oldcwnd + 1/oldcwnd$

LOSS: $newcwnd = oldcwnd - [0.5 * oldcwnd]$ when $cwndold \leq lwnd$.

When $cwndold > lwnd$ the following Scalable TCP window update algorithm is used:

ACK: $newcwnd = oldcwnd + 0.01$

LOSS: $newcwnd = oldcwnd - [0.125 * oldcwnd]$

- **High Speed–TCP (HSTCP) [6] [85]:** High Speed TCP (HS-TCP) is designed to behave like Reno for small values of the congestion window, but above a chosen value of *cwnd* an aggressive response function is used. When *cwnd* is large (greater than 38 packets), this modification uses a table to determine by how much the congestion window should be increased when an ACK is

received, and it releases less network bandwidth than *cwnd* 2 on packet loss. Hybrid TCP [42] is another important TCP-variant.

- **H-TCP [6]*:*** H-TCP has a similar approach to HSTCP since H-TCP switches to the advanced mode after it has reached a threshold. Instead of using a table like HS-TCP, H-TCP uses a heterogeneous AIMD algorithm.
- **LTCP [88]:** Layered TCP (LTCP for short), a set of simple modifications to the congestion window response of TCP to make it more scalable in highspeed networks. LTCP modifies the TCP flow to behave as a collection of virtual flows to achieve more efficient bandwidth probing. The number of virtual flows emulated is determined based on the dynamic network conditions by using the concept of *virtual layers*, such that the convergence properties and RTT-unfairness behavior is maintained similar to that of TCP.

Yet another algorithm is the TCP Santa Cruz, which is designed to work with path asymmetries, out-of-order packet delivery, and networks with lossy links, limited bandwidth, and dynamic changes in delay. The new congestion-control and error-recovery mechanisms in TCP Santa Cruz are based on: usi ng estimates of delay along the forward path, rather than the round-trip delay; reaching a target operating point for the number of packets in the bottleneck of the connection, without congesting the network; and making resilient use of any acknowledgments received over a window, rather than increasing the

congestion window by counting the number of returned acknowledgments.

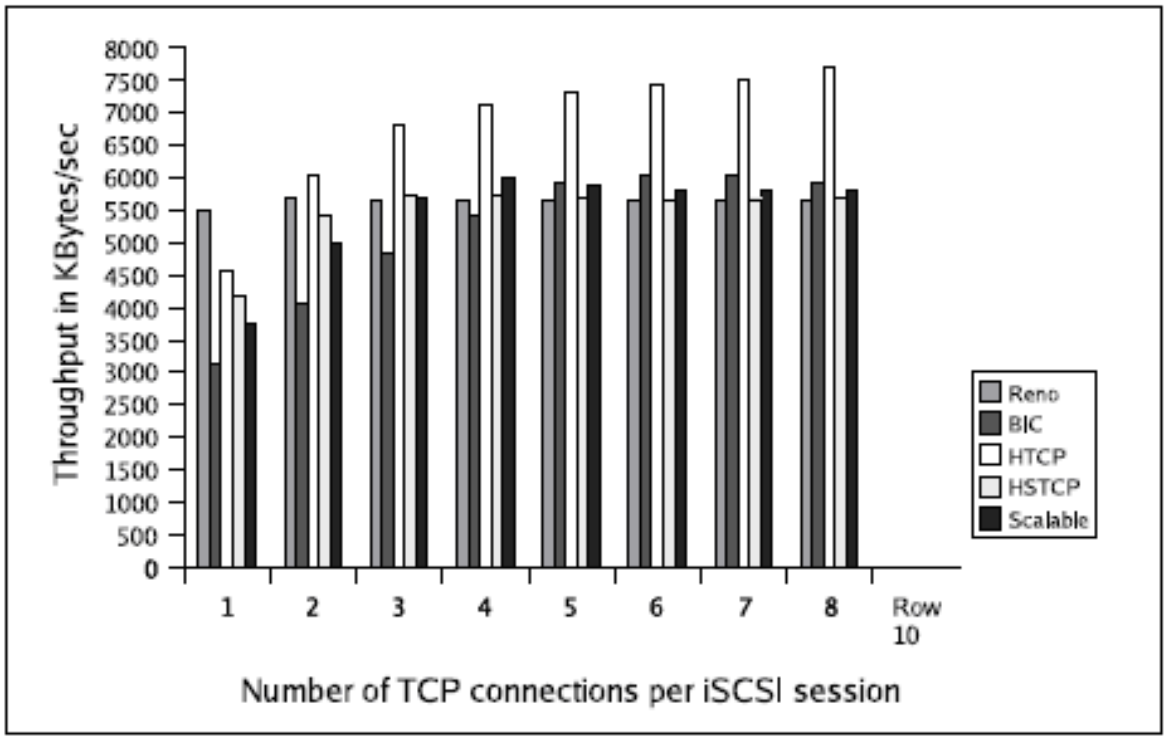


Figure 8: Comparison of the TCP variants in an iSCSI environment using the ns-2 simulator [18] [29]

Figure 8 compares the perfomance of the major TCP variants in an iSCSI environment and figure 9 does a comparison of the TCP variants in general.

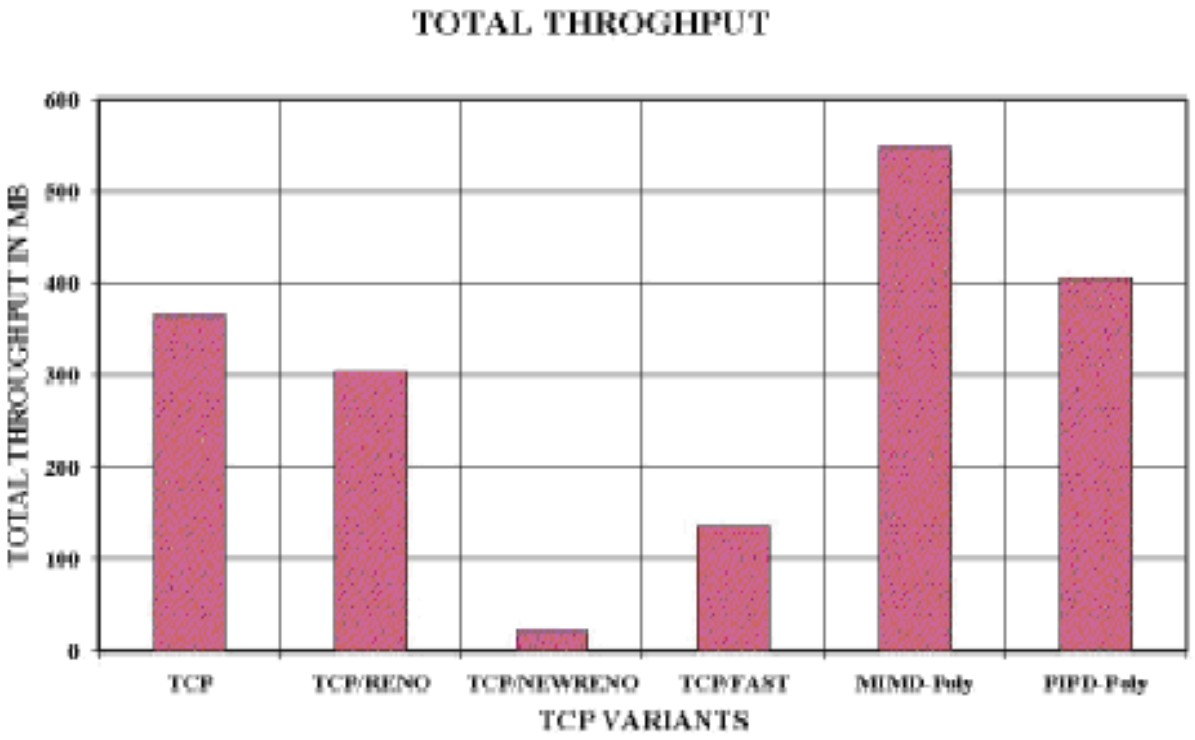


Figure 9 : Total throughput of the TCP variants

As we observe in Figure 10 and 11, HTCP is unfair compare to BIC-TCP which provides lower throughput as compared to HTCP. So an intelligent tradeoff must be made according to the requirement.

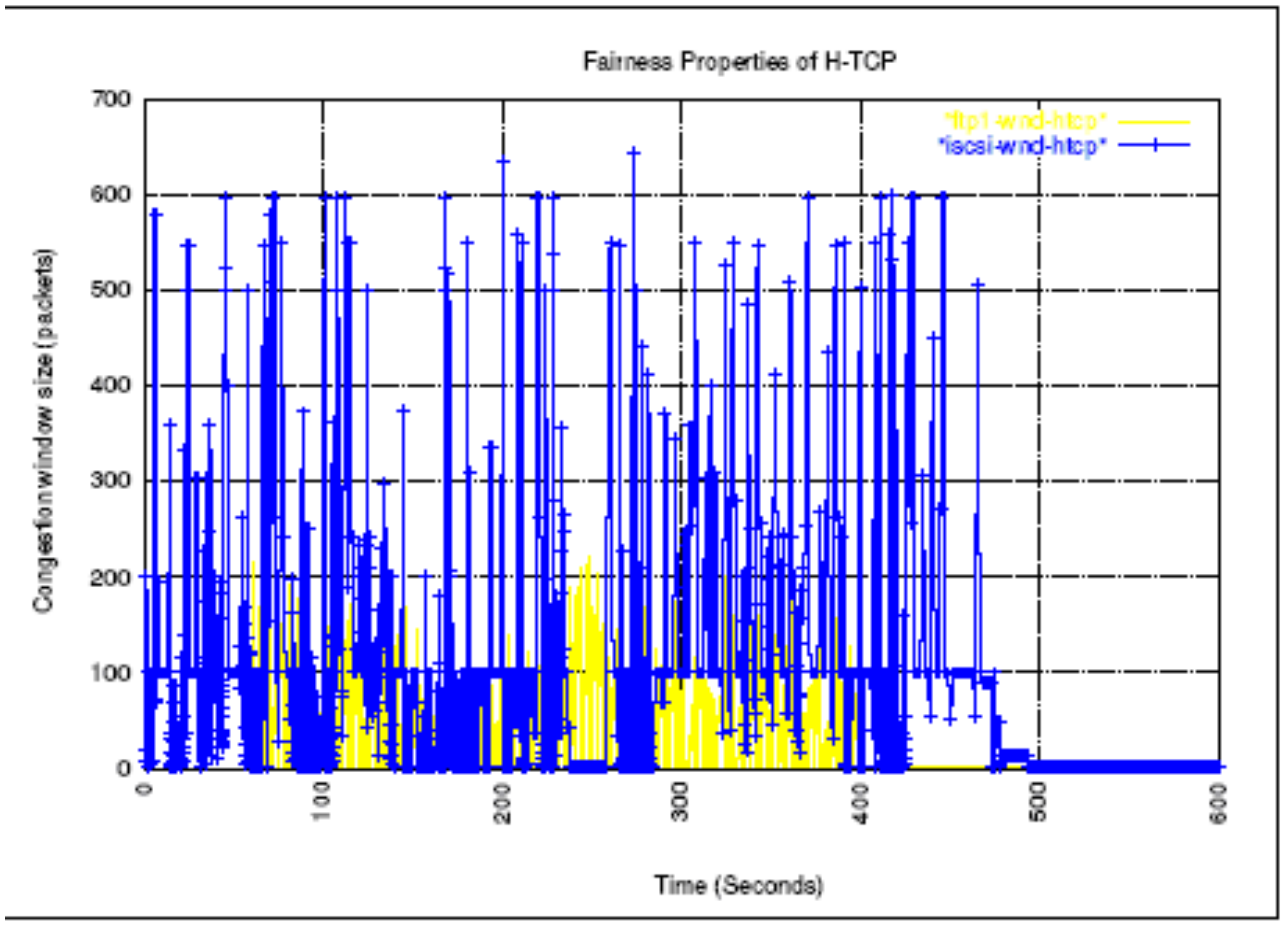


Figure 10: HTCP Fairness using ns-2[18]

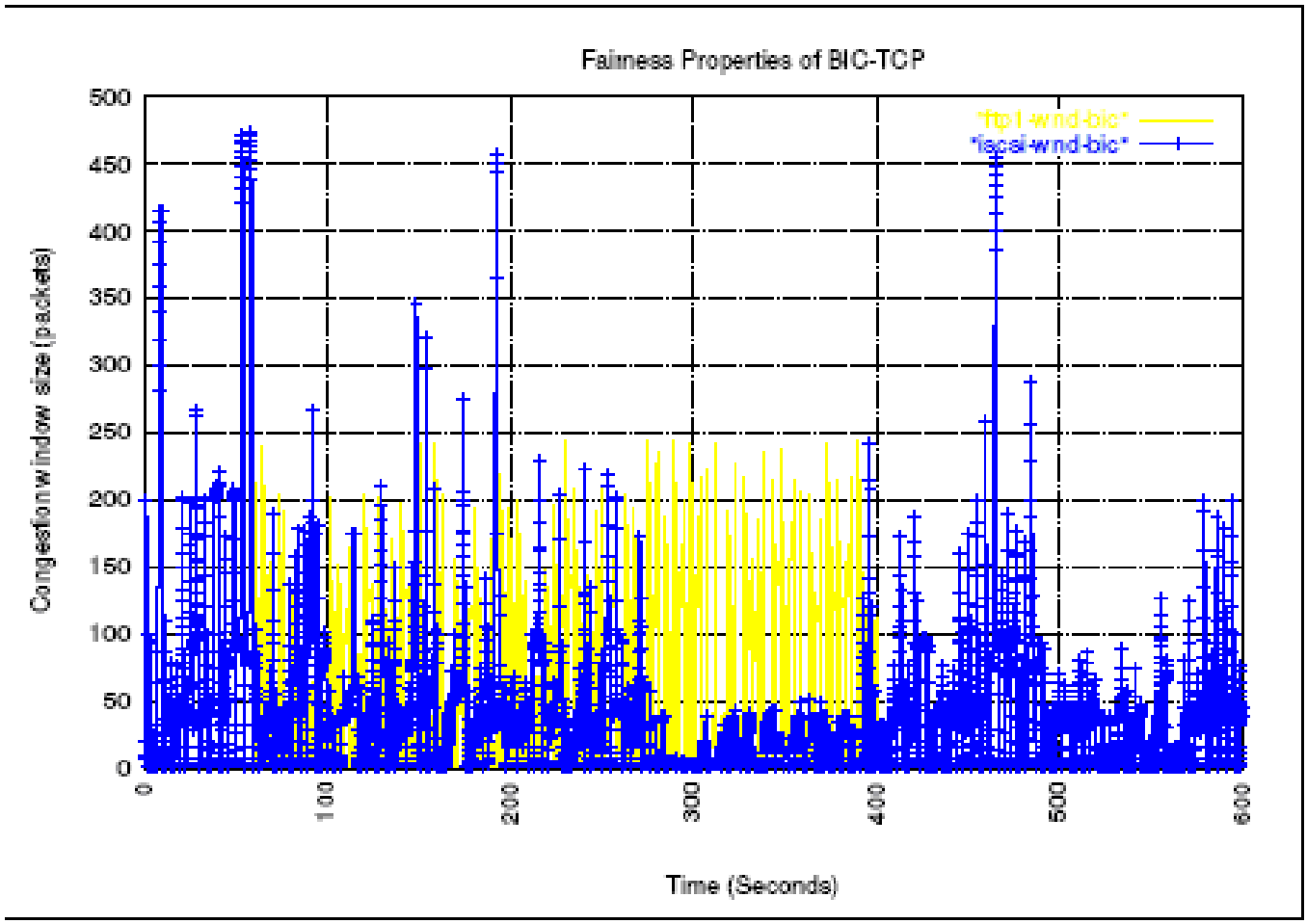


Figure 11: BIC-TCP Fairness using ns-2[18].

2.6.1 End-To-End Protocol Solutions for Infrastructured Wireless High-Speed Networks

TCP Westwood [5] [37] [43]

TCP Westwood (TCPW) [37] does not rely on the traditional additive increase multiplicative decrease (AIMD) algorithm but instead on a more aggressive estimation of the available bandwidth after a loss event has occurred. Thus, Westwood relies on a dynamic algorithm that infers the network state from the received ACKs. This information is used in an optimistic statistical estimation of the available bandwidth. Since the bandwidth changes with each packet sent, Westwood performs bandwidth estimation [9] upon the reception of each ACK.

2.6.1.1 TCP-Jersey [99]

TCP-Jersey not only addresses the problem of noncongestion random loss, but also attempts to deal with the congestion loss more efficiently. To explicitly differentiate between the congestion and non-congestion loss, TCPJersey uses two fundamentally different and separate mechanisms:

One for the aggressive modification of the congestion window in case of congestion related losses and the other for dealing with non-congestion related packet losses.To deal with congestion loss, TCP-Jersey uses a dynamic algorithm for changing the size of the congestion window. Much like West-wood, it attempts to aggressively estimate the congestion window after the loss has occurred using the *available bandwidth estimator* (ABE) algorithm.

The second mechanism detects the type of packet loss via a modified version of the explicit con-gestion notification scheme (ECN) [7]. ECN works in cooperation with random early detection (RED) to probabilistically mark the packets with the congestion bit when the router queue exceeds the minimum threshold and drop every packet when the queue exceeds the maximum threshold.

2.6.1.2 JTCP [99]

JTCP assumes that the network congestion can be inferred from the difference in the interarrival times of successive packet ACKs. This is the same paradigm used in TCP Veno. JTCP tackles only random wire-less loss. One basic concept used in JTCP is the *interarrival jitter*. It is defined as the time difference of two packets on the sender side and the time difference of the same two packets on the receiver side. If the interarrival jitter is greater than zero that means that the second packet traveled through the net-work longer than the first one. Thus, some time was lost in the queues of the network routers. A second important concept is the *jitter ratio* (*Jr*). Note that if the arrival rate of packets at the router is greater than its service rate, a queue is going to form at that router. Jitter ratio can be defined as the variance of the queue length and provides for the ability to detect whether the packets are being queued at the router or not. JTCP uses the jitter ratio in combination with the traditional loss events to determine the

type of loss in the network. Figure 12 shows the performance of JTCP in comparison to other TCP-variants when random losses occur.

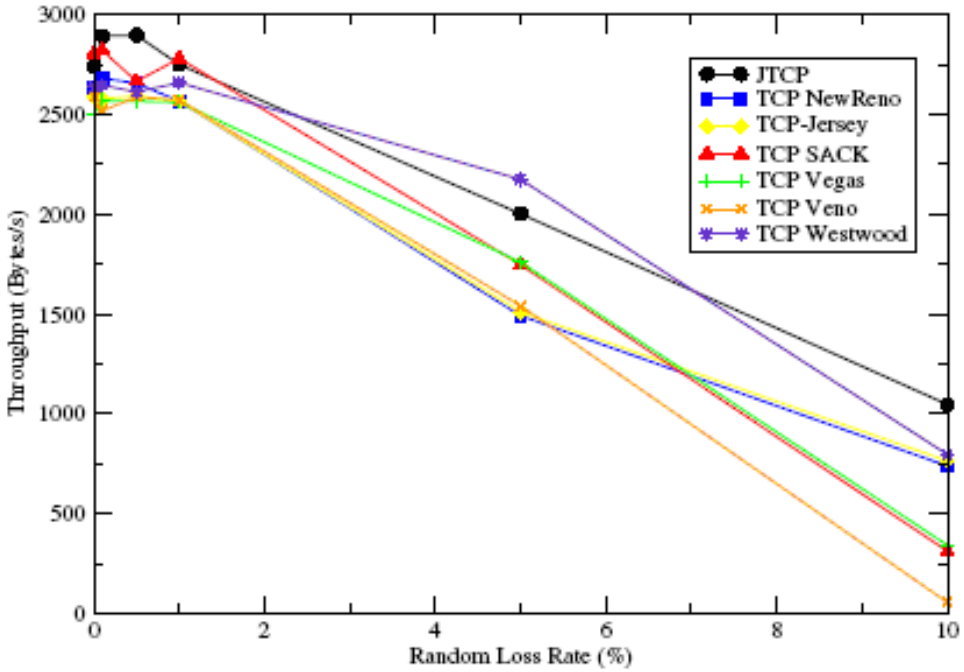


Figure 12: Throughput under random losses in a typical congested network

2.6.2 TCP Symbiosis [64]

TCP Symbiosis has a robust, self-adaptive, and scalable congestion control mechanism for TCP. It differs from the other algorithms in the following way:

The window size of a TCP connection is decided in response to information of the physical and available bandwidths of the end-to-end network path. The bandwidth information is obtained by an inline network measurement technique we have previously developed. Using the bandwidth information we can resolve the inherent problems in existing AIMD/MIMD-based algorithms such as periodic packet loss and unfairness caused by the difference in RTT.

Algorithms from biophysics are used to update the window size: the logistic growth model and the Lotka-Volterra competition model.The greatest advantage of using these models is that we can refer to

previous discussions and results for various characteristics of the mathematical models, including scalability, convergence, fairness, and stability in these models. Through mathematical analysis and extensive simulation experiments, comparison of the proposed mechanism with traditional TCP Reno, High-Speed TCP, Scalable TCP and FAST TCP [87], exhibits its effectiveness in terms of scalability to the network bandwidth and delay, convergence time, fairness among competing connections, and stability.

2.6.3 TCP Tuning Daemons for Efficient Link Utilization [62]

Many high performance distributed applications require high network throughput but are able to achieve only a small fraction of the available bandwidth. A common cause of this problem is improperly tuned network settings. Tuning techniques, such as setting the correct TCP buffers and using parallel streams, are well known in the networking community, but outside the networking community they are infrequently applied. Proposals are made for a tuning daemon that uses TCP instrumentation data from the UNIX kernel to transparently tune TCP parameters for specified individual flows over designated paths. No modifications are required to the application, and the user does not need to understand network or TCP characteristics.

2.6.4 Performance Issues and TCP Improvement Techniques for Optical Networks

In 2004, [59] conducted experiments on the three most common TCP implementations (at that time) namely:

1. Reno
2. New-Reno
3. Selective Acknowledgement (SACK)

A performance comparison was made in both the conventional networks and in OBS networks. It was found that the performance of SACK was the best in OBS networks. All three TCP implementations react to a Time Out (TO) loss in the same way (i.e. using Slow Start). But in OBS networks, burst loss is largely due to contention and may not be indicative of congestion. Therefore, the TO event may be a

false TO.It has been observed that such FTOs can significantly degrade the performance of all existing TCP implementations. Hence a new TCP implementation called Burst TCP (BTCP) which can detect FTOs and react properly, was proposed.

In 2006 [63] suggested that the bursty traffic pattern of TCP was another reason for TCP's low-performance. Hence, packet pacing was introduced for burst loss avoidance. Packet pacing is a technique to evenly space inter-transmission time of packets to avoid bursty traffic pattern. A detailed analysis was made to study the effect of packet pacing on High speed TCP variants on OBS networks.
The results can be outlined as follows:

- BIC-TCP, SACK showed a performance improvement was pacing was used with TCP.
- HS-TCP shows a decrease in throughput.
- Multi-flows of TCP showed an increase in throughput too with packet pacing.

The paper [65] proposed B-Reno, a new TCP implementation designed for TCP over"Optical Burst Switch" networks in 2007. B-Reno could overcome the inefficiencies of Reno and New-Reno in dealing with consecutive multiple packet losses and thus improve their throughputs over OBS networks. Moreover, B-Reno could also achieve performance similar with that of SACK over OBS networks while avoiding SACK's complex mechanisms at both the sender's and the receiver's protocol stack.

A novel congestion control scheme for TCP over OBS networks, called Statistical Additive Increase Multiplicative Decrease (SAIMD) was proposed in 2007 [77]. SAIMD aims to improve the throughput performance for high-bandwidth TCP flows in OBS networks by solving the false congestion detection problem. It also significantly outperforms the conventional TCP counterparts without losing fairness. An analytical model and extensive simulations proved the improvement in throughput.

An alternate approach was used in [80]. Instead of coming up with new protocols there were modifications made to the network architecture. The TCP protocol implementation was done over a novel OBS network architecture implementing a specific contention resolution scheme based on wave-

length conversion, burst segmentation, and optical buffering. For better performances, segment retransmission scheme in which segments lost in case of contention resolution failure are retransmitted at OBS layer was proposed. Also, an analytic model is developed for a mathematical analysis of the proposed scheme. Simulation experiments are conducted to validate the proposed approach and evaluate its performances. Novel Burst drop policies were also introduced [81].Here, the ”Drop Policy” was based on Hop Number Factor (HNF) in the core nodes of TCP over OBS networks, which is combined with Multiple Thresholds in the edge nodes. The proposed drop policy, HNF, takes the retransmission cost into account, and gives the burst with the larger Hop Number Factor the higher priority in the contention. On the other hand, for the retransmission is not always effective, multiple Edge Thresholds, Retransmission Number Threshold (RNT), Traffic load Threshold (TlT) and Total hop Threshold (ThT), are proposed to limit the unnecessary retransmission in the OBS layer. If any of the thresholds is exceeded, the retransmission will be handed over to the upper layer, the TCP layer, which will be more cost-effective.

The split technique was proposed for hybrid networks (IP and OBS core) in 2008 [82] which proved (by simulation results) to considerably increase the TCP-performance in hybrid networks.

2.7 Summary

In this chapter we have described the general proposals made to improve the performance of TCP in the literature. The proposals are basically targeting the specific challenges faced by various infrastructures like the wireless, wired, high-speed, optical, and satellite insfrastructures. The aim of this literature survey is to recognize the issues of TCP and explore the existing solutions proposed for those problems, which will provide the groundwork to design and implement a new and more efficient protocol.

3 PARALLEL TCP TRANSMISSION SCHEMES

In this chapter we concentrate on how the parallel transmission techniques are adopted in the Internet and TCP studies. The basic idea of the parallel transmission technique is to send data simultaneously over multiple virtual channels for faster transmission. This technique is also related to the "virtual layer" concept on which our proposed TCP is designed. However, as it will be described in next chapter, there is an important distinction between the two techniques: in **parallel transmission data is sent simultaneously over a fixed number of links**, but in the **"virtual layer**" concept the **layer-increase is gradual** and based on definite criteria from monitoring the network real-time status.

3.1 Parallel Connections

To efficiently utilize the large link capacity from high-speed networks, the concept of network striping is also proposed to optimize the network performance by opening parallel TCP connections. Some protocols using this approach are: XFTP, GridFTP, Storage resource broker.

In the MulTCP scheme [71] the authors present a mechanism where a single TCP flow behaves as a collection of several virtual flows. In [72], the authors describe a scheme for using virtual round trip time for choosing a tradeoff between fairness and the effectiveness of network usage. In [73] the authors describe pTCP, a scheme for managing the striped TCP connections that could take different network paths. However, all the above mentioned schemes use a *fixed* number of parallel connections and choosing the *optimal* number of flows to maximize the performance without affecting the fairness properties is a significant challenge.

Among the sections that will follow, the first section will talk about Grid computing and the GridFTP protocol which is built over the conventional TCP to improve the data transmission speed. After which, the concept of parallel transmission using the concept of virtual layers is discussed. MulTCP, LTCP

are the protocols based on this principle. Next, the proposals related mainly to network striping are discussed. The BitTorrent protocol, which is another widely accepted solution for file sharing in distributed networks, is discussed next. Lastly, the utilities related to the parallel transmission are mentioned.

3.2 GridFTP [16]

There are various types of storage systems in grid computing environments:

- Distributed Parallel Storage System (DPSS)
- High Performance Storage System (HPSS)
- Storage Resource Broker (SRB).

Both the storage providers and users would benefit from a common level of interoperability between all of these disparate systems: a common—but extensible—underlying data transfer protocol. A common data transfer protocol for all of these customized storage systems would confer benefits to both the keepers of large datasets and the users of these datasets. Dataset storage providers would gain a broader user base, because their data would be available to any client. Dataset storage users would gain access to a broader range of storage systems and data. In addition, these benefits can be gained without the performance and complexity problems of the layered client or gateway approach. This was the main motivation for proposing the GridFTP protocol.

GridFTP is a Data Grid services which complements and builds on the Globus toolkit [75] (glossary). For example, the GridFTP transfer service and the replica management service use the Grid Security Infrastructure (GSI), which is a part of the Globus toolkit middleware. GSI in turn, provides public-key-based authentication and authorization services.

Some of the GridFTP Features are as follows:

- **Grid security infrastructure and Kerberos support:** Robust and flexible authentication, integrity, and confidentiality features are critical when transferring or accessing files.

- **Third-party control of data transfer:** To manage large datasets for distributed communities, an authenticated *third-party* control of data transfers between storage servers is provided.
- **Parallel data transfer:** On wide-area links, using multiple TCP streams in parallel (even between the same source and destination) can improve aggregate bandwidth over using a single TCP stream. GridFTP supports parallel data transfer through FTP command extensions and data channel extensions.
- **Striped data transfer:** Data may be striped or interleaved across multiple servers, as in a DPSS network disk cache. GridFTP includes extensions that initiate striped transfers, which use multiple TCP streams to transfer data that is partitioned among multiple servers. Striped transfers provide further bandwidth improvements over those achieved with parallel transfers.
- **Partial file transfer:** GridFTP provides commands to support transfers of arbitrary subsets or regions of a file.
- **Automatic negotiation of TCP buffer/window sizes:** GridFTP extends the standard FTP command set and data channel protocol to support both manual setting and automatic negotiation of TCP buffer sizes for large files and for large sets of small files.
- **Support for reliable and restartable data transfer:** Reliable transfer is important for many applications that manage data. Fault recovery methods are needed to handle failures such as transient network and server outages. The FTP standard includes basic features for restarting failed transfers that are not widely implemented. GridFTP exploits these features and extends them to cover the new data channel protocol.

An enhancement over GridFTP was proposed in 2006, known as GridFTP-APT [75]. Here, an automatic parallelism tuning mechanism called GridFTP-APT (GridFTP with Automatic Parallelism Tuning) that adjusts the number of parallel TCP connections only using information measurable in the Grid

middleware. Through simulation experiments, it is demonstrated that GridFTP-APT significantly improves the performance of GridFTP in various network environments.

GridFTP-APT principle: GridFTP-APT starts from a small number of parallel TCP connections, and multiplicatively increases the number of parallel TCP connections at every chunk transfer until GridFTP goodput decreases. GridFTP-APT determines the bracket - the range of the number of parallel TCP connections covering the optimal value that maximizes the GridFTP goodput. In what follows, N is the number of parallel TCP connections used for a chunk transfer, G (N) the GridFTP goodput measured at the chunk transfer, and N-k the number of parallel TCP connections used for the k-th previous chunk transfer. GridFTP-APT starts from a small number of parallel TCP connections, and multiplicatively increases the number of parallel TCP connections at every chunk transfer until GridFTP goodput decreases.

3.3 MulTCP [72]

When different services are offered on Internet with fairness and cost consideration, the concept of "**weighted proportional fairness**" comes into picture. One such implementation of this concept is MulTCP. MulTCP is a TCP that behaves as if it was a collection of multiple virtual TCPs. To prevent the network from collapsing when congestion occurs, TCP has been provided with mechanisms that will reduce its throughput when losses are detected. Throughput of a single TCP connection is inversely proportional to both the square root of its loss rate p and to its round trip time R:

$$T = \frac{C}{\mathrm{R}\sqrt{p}}$$

Where the exact value of C depends on the approximations made. When multiple TCP streams go through a congested gateway, they experience approximately the same loss rate and thus get about the same fair share of the gateway's bandwidth.

MulTCP is a TCP control algorithm which takes a factor N as parameter and results in a TCP connection getting the same share of congested gateways bandwidth as N standard TCPs would get.

A TCP goes through different phases when it starts up, experiences loss or gets into some sort of steady state. In any of these phases, the MulTCP has to behave like N concurrent TCP connections would:

Slow start: During slow start a TCP opens its congestion window exponentially by sending two packets for every acknowledgement received. Interestingly, N TCPs doing slow start still send only two packets per acknowledgement received. However, N TCPs would start by sending N single packets, resulting in N acknowledgements being received and 2N packets being sent out after one RTT. The same behavior could be achieved by MuITCP if it sent out N packets at startup and then two packets for every acknowledgement received. This, however, leads to very bursty patterns if N is large. Burst may result in bursts of losses which in turn prevent the connection of rapidly reaching steady state. MulTCP thus uses a smoother option. It starts like a normal TCP by sending a single packet. After that, it sends three packets for each acknowledgement received until it has opened its congestion window as far as N TCPs would have.

After k round trip times N TCPs have a congestion window of $N2^k$. One MulTCP sending three packets for each acknowledgement would have a window of 3^k. Thus they have the same window after kg round trip times where,

$$k_N = \frac{\log N}{\log 3 - \log 2}$$

this happens when the window has a size of Wn: 3 kn. The resulting pseudo code looks like this:

```
if (cwnd < ssthresh) {/*slow start*/

        if (cwnd <= pow (3.0, log (N)/log (3)-log (2))))
                cwnd += 2;
else
        cwnd +=1;
}
```

Linear increase*:* When the congestion window reaches ssthresh a TCP increases its window by one packet per RTT. N TCPs increase their window by N packets per RTT.

Multiplicative decrease: When a TCP notices congestion through the loss of a packet it halves its congestion window, sets ssthresh to the new value of the congestion window and goes back to linear increase. When N TCPs are sending data and one packet is lost, only one TCP will halve its window. Thus MulTCP, when it experiences loss, only halves one Nth of its congestion window by setting cwnd and ssthresh to (**N-0.5) / N** of cwnd. This assumes that at the time of loss all N virtual TCPs had the same values for these variables. This is macroscopically true since the fairness properties also hold between the virtual TCPs. Moreover, looking at this in more detail, we can easily see that N TCPs experiencing a total of k losses randomly distributed amongst them end up with a sum of congestion windows which has a statistical mean of $\mathbf{[(N-0.5) / N]}^{K}$. This is equal to the congestion window of a single TCP which reduces its window by **(N-0.5) / N** for each loss N.

```
If (cwnd < ssthresh)
        cwnd = cwnd/2;
else
        cwnd = cwnd * (N-0.5) / N;
ssthresh = int (cwnd);
```

3.4 LTCP [88]

LTCP is based on the very simple concept of “virtual layers” or virtual flows. To start out with, every LTCP flow has only one layer. If the sending rate of the flow increases, without observing any losses, then based on some criteria, it increases the number of layers and continues to do so until a loss event is observed. When operating at any given layer K, the flow behaves as if it were a collection of K virtual flows, increasing the aggressiveness of probing for bandwidth. Just like the standard implementations of TCP, the LTCP protocol is ack-clocked and the congestion window of an LTCP flow changes with each incoming ack. However, since the LTCP flow operating at layer K emulates K virtual flows, it increases the congestion window by K packets per RTT. For determining the number of layers that a flow should operate at, the following scheme is used. Let’s suppose that each layer K is associated with a step-size δK. When the current congestion window exceeds the window corresponding to the last addition of a layer (WK) by the step-size δK, a new layer is added. Thus, $W1 = 0$, $W2 = W1 + \delta 1$, $WK = WK-1 + \delta K-1$ (1) and the number of layers = K, when $WK \leq W < WK+1$ (Figure 13).

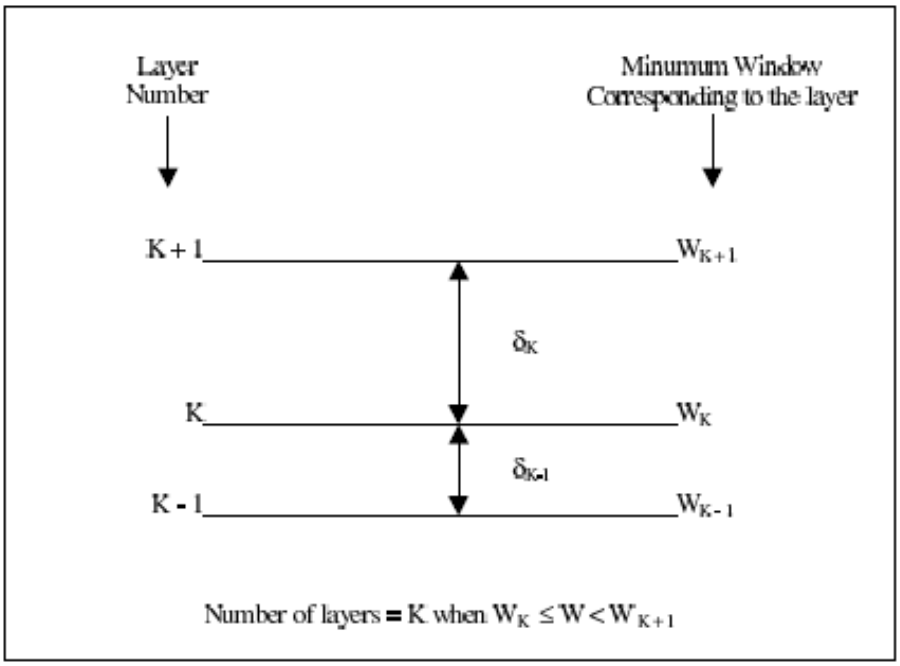


Figure 13: LTCP principle

The step size δK associated with the layer K should be chosen such that convergence is possible when several flows share the bandwidth. Consider the simple case when the link is to be shared by two LTCP flows. Say, the flow that started earlier operates at a higher layer K1 (with a larger window) compared to the later starting flow operating at a smaller layer K2 (with the smaller window). In the absence of network congestion, the first flow increases the congestion window by K1 packets per RTT, whereas the second flow increases by K2 packets per RTT. **In order to ensure that the first flow does not continue to increase at a rate faster than the second flow, it is essential that the first flow adds layers at a rate slower than the second flow**. Thus, if δ (K1) is the stepsize associated with layer K1 and δ (K2) is the stepsize associated with layer K2, then

$$\frac{\delta_{K1}}{K1} > \frac{\delta_{K2}}{K2}$$

When K1 > K2, for all values of K1, K2 >= 2.

This equation is called the **convergence principle**. The decrease behavior is guided in a similar way.

3.5 Network Striping: pTCP [74]

pTCP is an end-to-end protocol for striped connections. It provides mechanisms to use striped connections with aggregate bandwidth along multiple paths.

The key obstacles to achieving the aggregate bandwidth for striped connections are the following:

- Each of the individual paths can have vastly differing characteristics in terms of bandwidth and delay (round-trip time). If data-striping is done without taking into account these differences, the bandwidth achieved by the striped connection can be significantly lower than the maximum possible [34].
- Fluctuations of individual path characteristics may occur.

The problems that arise due to bandwidth differences can be solved by making sure that the data-striping ratio is the same as the ratio of the bandwidths of the different paths. Solutions to the above mentioned problems are provided by pTCP which are explained as below:

i) **Decoupling of Functionalities**: pTCP decouples functionalities associated with per-path characteristics from those that pertain to the aggregate connection. The component in pTCP that handles per-path functionalities is called TCP-v (TCP-virtual). TCP-v is a modified version of TCP that deals only with virtual packets and virtual buffers. Each micro flow of a striped connection is controlled by an independent TCP-v (as shown in figure 14).

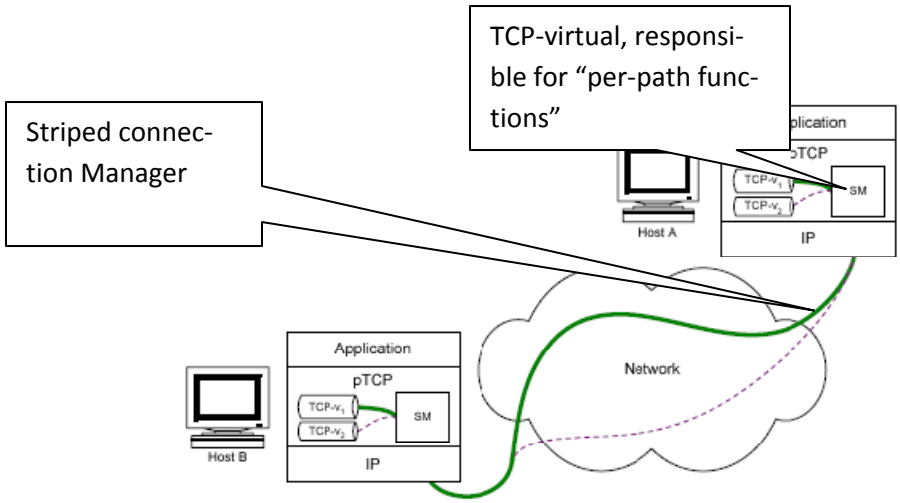


Figure 14: pTCP for striped connections

ii*)* **Congestion control** is handled by TCP-v. SM is responsible for striping data across the different TCP-vs, it handles buffer management (including sequencing at the receiver), and consequently flow control.

iii**) Delayed binding**: pTCP uses a delayed binding strategy that satisfies the requirement of striping based on ratios of cwnd, rtt, and at the same time dynamically adapts to instantaneous fluctuations in bandwidth and delay.

iv) **Packet re-striping**: In steady state, pTCP will ensure that the number of outstanding packets in a micro-flow is proportional to the bandwidth along the corresponding path. Moreover, the delayed binding strategy further ensures that all bound packets are already in transit in the network. However, during congestion when packets are lost in the network, the reduction of the congestion window by a TCP-v can result in bound packets falling outside the congestion window. If such packets are lost during the congestion, then they will remain un-transmitted till the congestion window of that TCP-v expands beyond their sequence numbers. This can potentially result in an overflow of the receive buffer if the other micro-flows are active in the meantime, finally resulting in a connection stall.

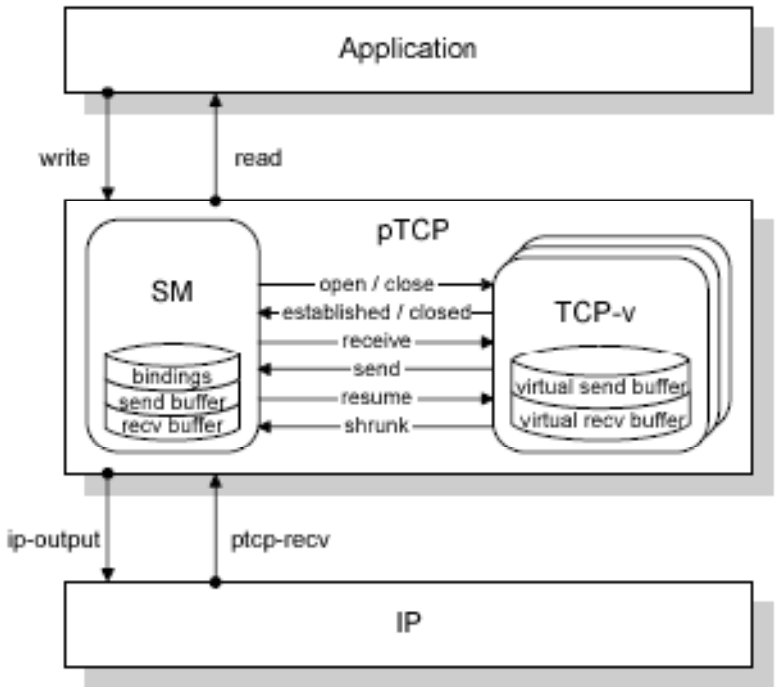


Figure 15: pTCP architecture

pTCP handles the problem by unbinding packets that fall outside of the congestion window of TCP-v they were assigned to. Such unbinding results in making those packets available to the next TCP-v that can send more data (Figure 15). pTCP also uses the selective acknowledgements to further reduce packets.

3.6 BitTorrent Protocol [78]

BitTorrent is a peer-to-peer file sharing protocol used to distribute large amounts of data. The initial distributor of the complete file or collection acts as the first seed. Each peer who downloads the data also uploads them to other peers. Relative to standard internet hosting, this provides a significant reduction in the original distributor's hardware and bandwidth resource costs. It also provides redundancy against system problems and reduces dependence on the original distributor.

- **BitTorrent client** is any program that implements the BitTorrent protocol. Each client is capable of preparing, requesting, and transmitting any type of computer file over a network, using the protocol.
- **A peer** is any computer running an instance of a client.

Basic operation principle: To share a file or group of files, a peer first creates a small file called a "torrent" (e.g. MyFile.torrent). This file contains metadata about the files to be shared and about the tracker, the computer that coordinates the file distribution. Peers that want to download the file must first obtain a torrent file for it, and connect to the specified tracker, which tells them from which other peers to download the pieces of the file.

Though both ultimately transfer files over a network, a BitTorrent download differs from a classic full-file HTTP request in several fundamental ways:

- BitTorrent makes many small data requests over different TCP sockets, while web-browsers typically make a single HTTP GET request over a single TCP socket.
- BitTorrent downloads in a random or in a "rarest-first" approach that ensures high availability, while HTTP downloads in a sequential manner.

Taken together, these differences allow BitTorrent to achieve much lower cost to the content provider, much higher redundancy, and much greater resistance to abuse or to "flash crowds" than a regular HTTP server. However, this protection comes at a cost: downloads can take time to rise to full speed because it may take time for enough peer connections to be established, and it takes time for a node to receive sufficient data to become an effective uploader. As such, a typical BitTorrent download will gradually rise to very high speeds, and then slowly fall back down toward the end of the download. This contrasts with an HTTP server that, while more vulnerable to overload and abuse, rises to full speed very quickly and maintains this speed throughout.

In general, BitTorrent's non-contiguous download methods have prevented it from supporting "progressive downloads" or "streaming playback". Various improvements are suggested for the bit-torrent clients for better performance (“BitCod Client”). Also, network topology improvements are also suggested. For instance, use of a mesh instead of a tree topology is proposed to improve the performance [76].

3.7 Utilities/ Libraries Based on Parallel Transmission Principle [16]

- *PSockets* (Parallel Sockets) is a library which is used to make it easier to develop applications that use network striping.
- SEMPLAR, a library for remote, parallel I/O that combines the standard programming interface of MPI-IO with the remote storage functionality of the SDSC Storage Resource Broker (SRB). SEMPLAR relies on parallel TCP streams to maximize the remote data throughput in a design that preserves the parallelism of the access all the way from the storage to the application.
- Using TPF: TCP Plugged File System for Efficient Data Delivery over TCP [31].

3.8 Summary

In this chapter we presented a survey of the protocols/techniques based on parallel transmission/connection. The important proposals discussed are GridFTP, Bittorrent, pTCP and MulTCP. These protocols/applications increase the network performance to a great extent when compared to the conventional TCP based applications, which also motivate the proposal and implementation of stratified TCP (to be detailed in next chapter).

4 STRATIFIED TCP (STCP)

In the light of the drawbacks of TCP mentioned in the last two chapters we realize that a novel and optimal protocol which makes efficient use of the bandwidth is necessary. In this chapter we propose and describe the **Stratified TCP** protocol.

The Stratified TCP (STCP) employs parallel virtual transmission layers in high-speed networks. In this technique, the AIMD principle of TCP is modified to make the link-bandwidth probing more aggressive and efficient, which in turn increases the performance. Simulation results show that STCP offers a considerable improvement in performance when compared with other TCP variants such as the conventional TCP protocol and Layered TCP (LTCP).

This chapter is organized in the following manner. We first describe the motivation for coming up with Stratified TCP. The working principle, algorithm, and the implementation are described in the following sections. We then analyze the simulation results. The conclusions and future work are described in the next chapter.

4.1 Motivation

In the previous chapters, we describe various proposals and TCP variants for high-speed networks. For instance, Layered TCP (LTCP) which is based on the “virtual layers” principle has shown an improvement in performance. However, it has certain shortcomings:

1. **The convergence principle followed to implement the “increase behavior” is vague**.

The increase in the number of flows follows the following equation:

δ (K1) /K1 > δ (K2) /K2, where

- K1 - It is flow that starts before the layer K2 begins
- δ (K1) - step-size for layer K1
- δ (K2) - step-size for layer K2

This ratio should be satisfied in order to maintain the convergence property, so that the layers which start off later do not increase at a larger rate than the initial flows. The major concern here is described as follows:

The increase behavior is just a ratio and is very vague. For instance, if δ (K1) /K1 >> δ (K2) /K2, we do not get a clear idea as to what is the exact difference between the two ratios.

2. **The response of LTCP is not dynamic since the RTT is not a part of the increase function.**

Ideally, the RTT values decide the network conditions to a great extent. Therefore, they should be continuously monitored to predict the network conditions more accurately.

3. **LTCP performance degrades for multiple flows if the RTT value for each flow varies to a large extent (when compared to RTT of the other flows).**

The LTCP protocol works well in a multi-flow environment only if the flows have ”similar RTTs”, otherwise the performance may degrade and be comparable to conventional TCP.

As we see above, these three problems are of considerable concern. Therefore, the development of a protocol that solves these problems to a satisfiable degree is of great value and hence the need for STCP.

4.2 Principle of Proposed Stratified TCP (STCP)

Stratified TCP is based on the concept of “virtual layers” as in Layered TCP (LTCP), but it follows a different approach for congestion control. Figure 16 explains the principle of STCP graphically.

In STCP, each virtual layer or flow is referred to as the”stratum”. Each virtual flow is seperated by a certain value known as the ”stratum-interval” , δ(SI) which in turn decides when a new stratum should be added. The most important point is that the increase-factor or α is calculated dynamically.

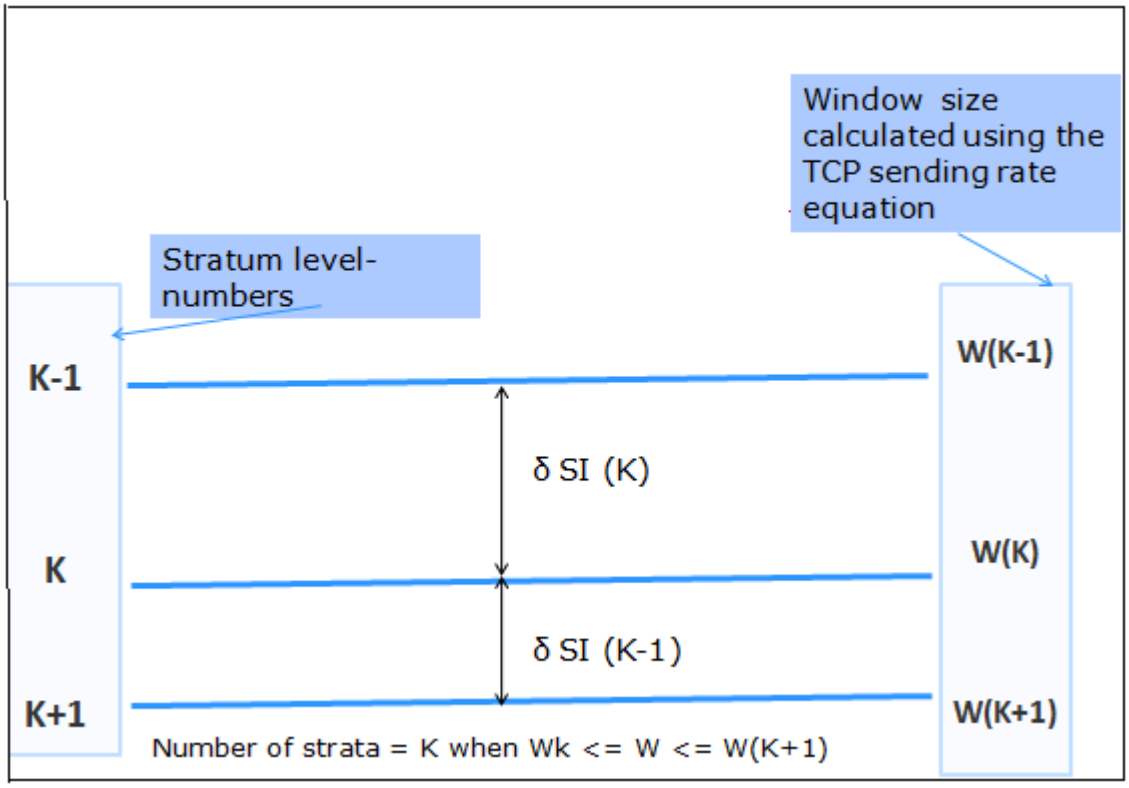


Figure 16 : Principle of STCP

The new value of congestion-window is obtained using the “α” value and the previous window size. After this, the strata are increased until the congestion window reaches the current congestion window size in a progression whose interval is given by δ (SI) i.e.value corresponding to each stratum. Since the

window size equation involves the current value of RTT, the response behavior is more adaptive to the dynamic network environments. Let us consider the fundamental equation for the sending rate for TCP.

Equation 1

TCP sending rate= $\sqrt{\frac{\alpha \times 2-\beta}{2 \times \beta}} \times \sqrt{\frac{1}{p}} \times \frac{1}{RTT}$ here,

- α is the increase factor
- β is the decrease factor
- p is the rate of packet loss
- RTT is the Round Trip Time

There are two scenarios namely the “high-window” (83000 MSS) and the “low window” (38 MSS) scenarios. The threshold values for the increase factor, α is calculated separately for these two scenarios. Once these values are calculated, the new window size is **W (k) = α * W (k-1),** where the symbols W (k) and W (k-1) represent the window sizes for layer k and layer k-1 respectively.

4.3 STCP Algorithm

This subsection outlines the high-level sequence of events as implemented in STCP code.There are four essential steps to implement the algorithm as explained below:

1. **Initialize the required STCP variables**

   The Stratified TCP variables related to the layers and the congestion window are initialised in this step:

   - The number of initial layers to 1
   - The number of fractional layers to 0
   - Initial congestion window threshold to LOW WINDOW value 38

2. "Increase parameter (α)" calculation for the slow start phase

The congestion control is implemented in two phases namely, the "slow start" and the congestion avoidance phase. The value for the slow-start phase is calculated by doubling the window size for every RTT, which makes the increase behavior aggressive.

3. "Increase parameter (α)" calculation for the "congestion avoidance" phase

In the congestion avoidance stage, the "α" is calculated dynamically using the"SENDINGRATE" equation (equation 1). By mathematical simplification of equation 1, the "α" value can be calculated in the following way:

Equation 2

$$\alpha = \frac{RTT_{fact}{}^2 \times W_{thresh}{}^2}{6166}$$

- RTTfact is the RTT compensation factor.This variable takes into account the current value of RTT and hence makes the increase behavior adaptive.
- Wthresh may be the low/high window value depending on the "cwnd" parameter value.

4. Congestion window calculation for the newly calculated α value

This 'α' value is used to calculate the window increase- parameter for the "congestion avoidance phase". The decrease behavior is similarly implemented using the same equation 1.

4.4 Implementation

In this section the implementation of LTCP is described. Eventually the difference in the implementation of STCP and LTCP is described. The "increase-behavior" **LTCP congestion control** can be divided into two phases:

1. **Slow Start**: In this stage the congestion window is doubled for every RTT tunil the first packet loss
2. **Congestion avoidance:** The congestion window is increased additively in TCP, and a similar algorithm is followed by LTCP to increase the number of layers.

Stratified TCP on the other hand, starts off from the first principle.The algorithm is explained below.

Starting off from equation 1 (TCP sending rate equation), we can make the rate more adaptive in the following way:

Equation 3

$$\text{Window Size} = \sqrt{\frac{\alpha(w)\times 2-\beta(w)}{2\times\beta}} \times \sqrt{\frac{1}{p}} \times \frac{1}{RTTcurr}$$

- α (w) is a function of the window size (directly proportional).
- RTTcurr is the current value of the RTT.
- β (w) is fixed for the time being to a value of 0.10. The reason for choosing 0.10 is that recent research has proved that bandwidth recovery is better and faster [88].

There are three scenarios that can be considered here:

- **RTT increases at a high rate indicating congestion is possible soon:** In this case, as the value of the current window contains the RTTcurr, the size of the window is adaptive and is proportionately reduced as the RTT increases.Also; future values of RTT are continuously monitored. Eventually, the decrease is not drastic but gradual.
- **RTT is constant indicating that the network conditions are stable:** In this case, the window size remains constant. We could include an “increase factor” when RTT is stable, but as of now, the window size will remain the same.
- **RTT is reducing indicating that the network is capable of handling more traffic:** Again, here there should be increase based on the latest monitored values of RTT.

The modified window response draws some points from the HS-TCP. HS-TCP has a set of tabulated values. These values are calculated based on continuous research and testing. Each set of network parameters are optimal and give maximum network performance. Two such sets of values are the "**High window response**" and the "**Low window response**". Each of these scenarios has a value for:

- Packet loss rate
- Congestion window threshold ($W_{th\text{-}h}$ and $W_{th\text{-}l}$ for the high window and low-window scenarios respectively).

The values of the network paramters for the two scenarios are as follows:

1. **Low window response**

   The parameters are Wth = 38 MSS, p= 1/1000 we can find the value for α (W) th-l using the above values in equation 3.

2. **High window response**

   The parameters are Wth = 83* 1000 MSS, p=1/10000000 we can find the value for α (W) th-h using the data in equation 3. Once these values are calculated for the two scenarios, the "current window size" can be calculated using:

$$1)\ \text{Wcurr} = \sqrt{\frac{\alpha(w)_{th-l} \times 2 - \beta(w)}{2 \times \beta(w)}} \times \sqrt{\frac{1}{p}} \times \frac{1}{RTT_{curr}} \quad \text{Low window}$$

$$2)\ \text{Wcurr} = \sqrt{\frac{\alpha(w)_{th-h} \times 2 - \beta(w)}{2 \times \beta(w)}} \times \sqrt{\frac{1}{p}} \times \frac{1}{RTT_{curr}} \quad \text{High Window}$$

Now, to calculate the number of layers, when Wcurr > Wth consider,

- Low Window Response:

Wcurr or Wk = Wth + α (w) th-l * A. Here, Wth = 38 MSS, p=1/1000

- High Window Response:

Wcurr or Wk = Wth + α (w) th-h * A. Here, Wth = 83000 MSS, p= 1/10000000

'A' is a constant that will decide the number of flows to be increased. A= W (k-1) (where "K" is the stratum-number and Wth the threshold window value), this is similar to the way in which step size is calculated in LTCP.

4.5 Analysis

A brief analysis is made of the above algorithm and its advantages over other similar protocols are highlighted.

4.5.1 Parameter Considerations

Unlike, other protocols based on the virtual layer technique, the Stratified TCP starts off from a fundamental sending rate principle and tries to modify it to suit the dynamic network conditions. The congestion control algorithm is more dynamic and more responsive to the network changes because α value is calculated dynamically. The RTT factor is included in the calculations for the current window size and hence, the RTT rightly influences the window size making the protocol more adaptive to changing conditions.

4.5.2 Tradeoffs

Some of the tradeoffs made are outlined as below:

1. For increased network adaptability there is a neglible overhead of calculating certain network parameters frequently. But these calculations make the protocol more dynamic.
2. The concept of multi-flows with varying RTT values is not considered at this stage to keep things simple.The idea is to first investigate if the basic principle would improve the performance as compared to other parallel transmission protocols and then look into improving the multi-flow scenario. However, stratified TCP uses the same principle as in LTCP to find the "RTT compensation factor" to neutralize the effect of flows with differing RTT value.

4.6 Performance Evaluation

4.6.1 Dumbbell simulation topology

The implementation of the algorithm was done using the C++ language. The results are verified using the ns-2 simulator (2.26 version). Figure 17 shows the network topology used in the simulations.The topology is a simple dumbbell network. The bottleneck link bandwidth is set to 1Gbps unless otherwise specified. The links that connect the senders and the receivers to the router have a bandwidth of 2.4Gbps. The routers have the default queue size set to 5000 packets which is one third the delay-bandwidth product of the bottleneck link. DropTail queue management is used at the routers. The parameter β was set to 0.10. The traffic consists of FTP transfer between the senders and receivers.

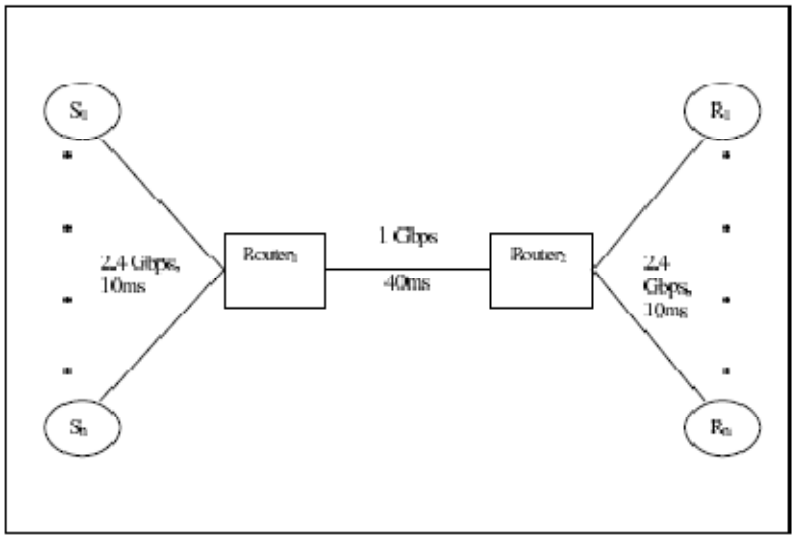


Figure 17: Simulation topology [88]

The results (plotted from the trace files generated) for the throughput, delay and congestion window size are shown as below.

Analysis of the congestion window-variation: As seen in Figure 18, the "slope" of the congestion window functions for STCP is steeper when compared to LTCP and TCP. Therefore, the window is increased at a greater rate and hence the bandwidth utilization is better. It is also observed that when a packet

loss occurs the decrease is also less than LTCP, which indicates that STCP takes lesser time to overcome congestion and regain the lost bandwidth.

Analysis of throughput: Figure 19 shows the variation of throughput when the number of sources is gradually increased. It is seen that STCP has greater throughput than LTCP and TCP when the number of sources are less than three. For greater number of sources LTCP and STCP exhibit similar behavior.

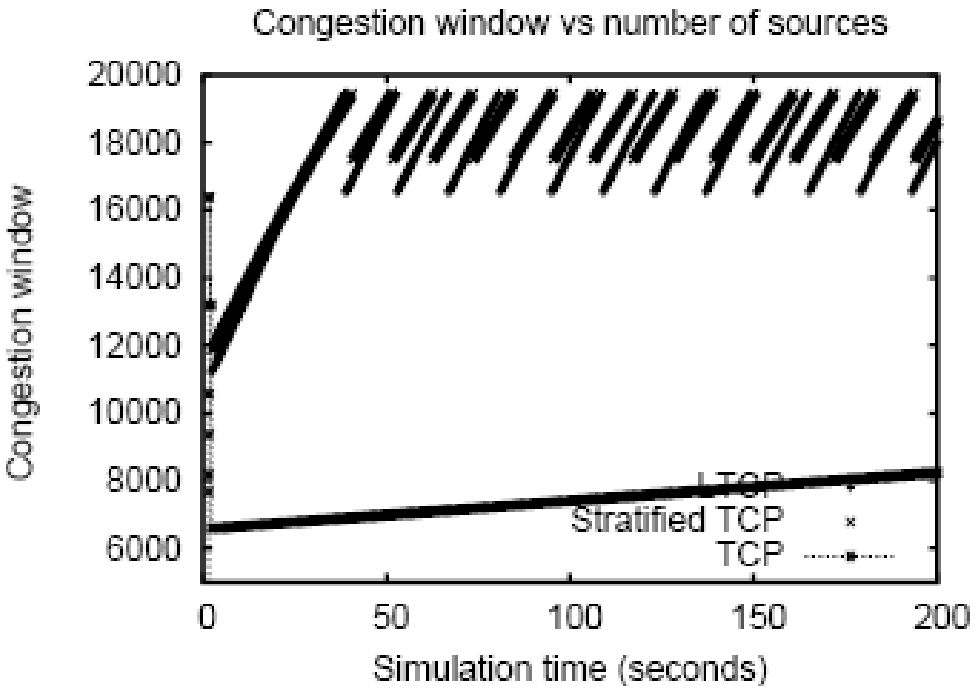


Figure 18: Comparison of congestion window variation in TCP/LTCP/STCP

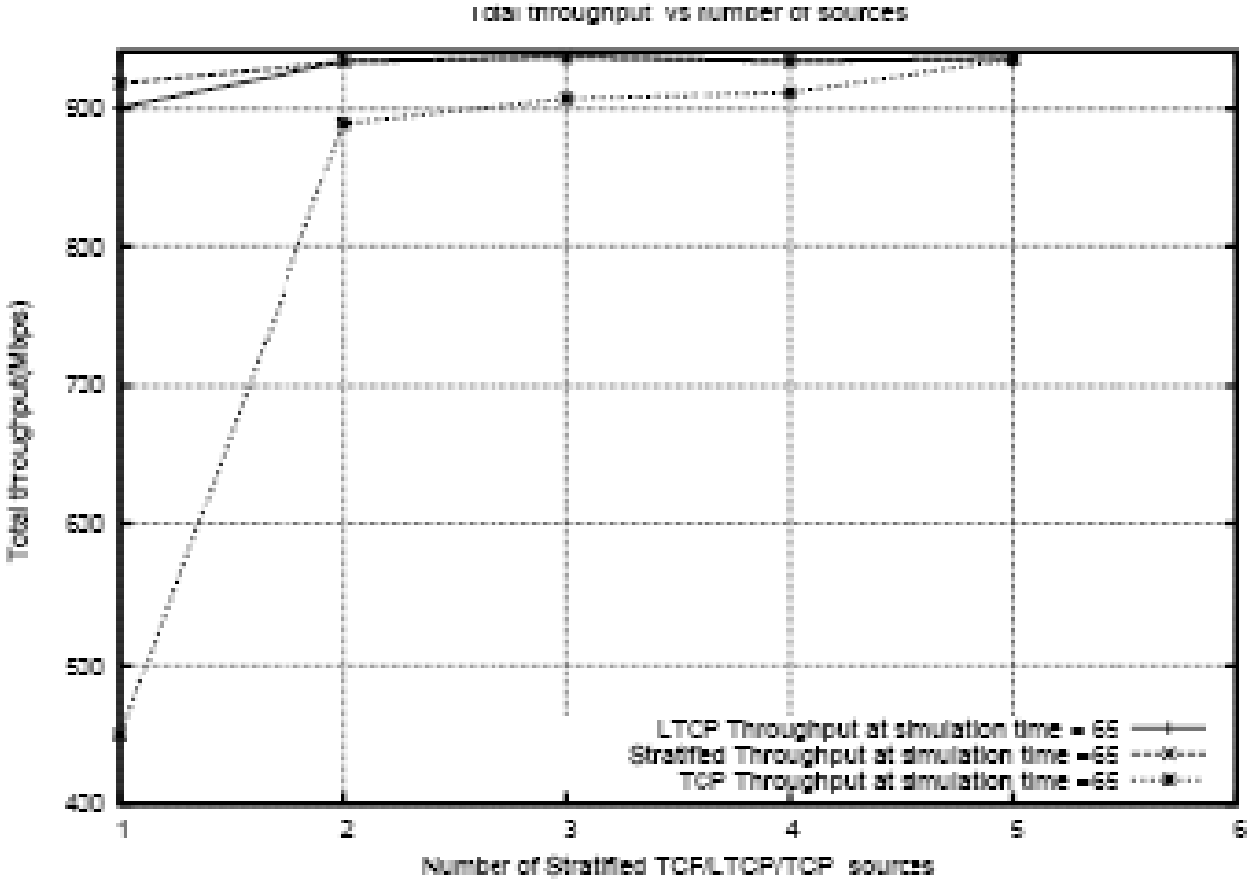


Figure 19: Throughput comparison of TCP/LTCP/STCP

Analysis of the delay: In Figure 20, the variation of the average delay with respect to the number of sources is shown. STCP and LTCP show similar behavior but they exhibit greater delays when compared to conventional TCP. The increase in delay in STCP can be attributed to the aggressive usage of bandwidth which may increase the queue size to a considerable extent causing queuing delay in the network. The congestion window and the throughout graphs show that STCP is an improvement over the conventional TCP protocol and LTCP. It is far better than the existing TCP protocol but provides the same amount of reliability.

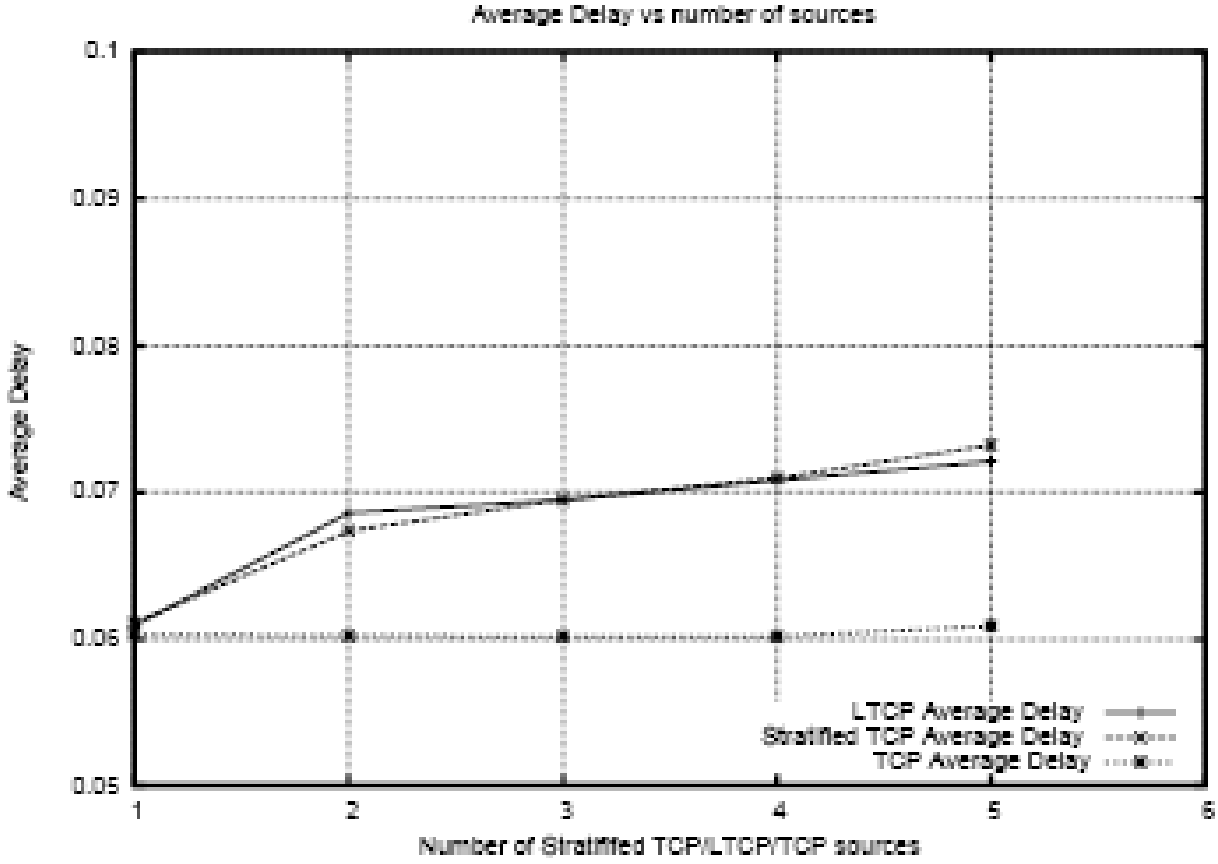


Figure 20: Delay comparison of TCP/LTCP/STCP

Intra-STCP fairness or Intra-protocol fairness: The intra-protocol fairness is calculated by starting multiple STCP flows at the same time. The average throughput per-flow bandwidth is noted. Similarly, the maximum and the minimum throughputs are also noted. The difference between the maximum and the minimum throughput values is not large indicating that the variation of throughput is considerably small in a multi-flow scenario. To prove this numerically, we calculate the Jain's Fairness index which is close to 1 in all the cases indicating that STCP is fair towards the other STCP flows.

Table 1 : Fairness among STCP flows (starting at the same time)

Number of Flows	Avg per flow through-put(Mbps)	Min per flow through-put(Mbps)	Max per flow through-put(Mbps)	Jain Fairness Index
2	466.48	445.53	487.43	0.9979
4	233.27	218.21	259.77	0.9954
6	156.29	119.86	184.76	0.9686

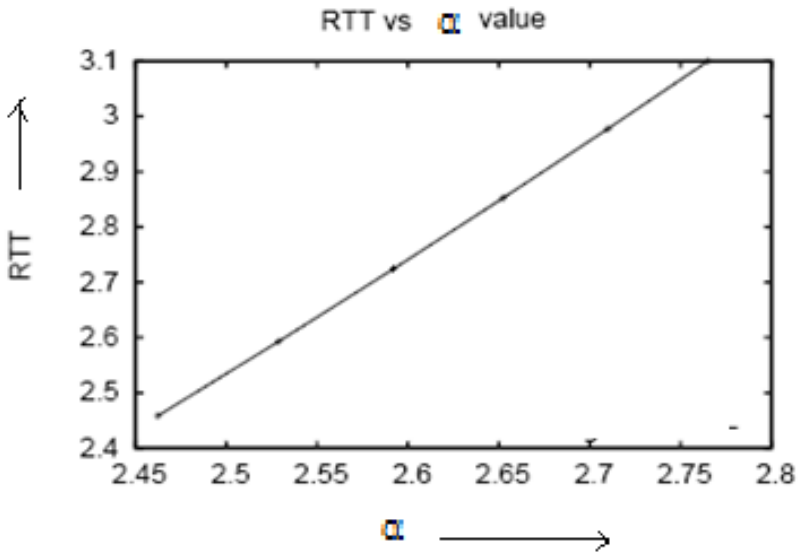


Figure 21: RTT variation with respect to α value

In Table 1 the intra-protocol fairness for flows ranging from 2 to 6 are tabulated and the Jain's fairness index calculated. A study of the variation of RTT with respect to the 'α' value is shown in Figure 21. The results are captured in a low window region where the packet loss-rates are low. As mentioned in the previous sections the STCP has the congestion control implemented for the low and high window scenarios. Each scenario has a value of packet loss associated with it. The increase factor can be increased for a particular scenario if the packet loss is within the tolerable rates for that particular scenario. Hence,

in the low-window scenario, when the packet-loss rate is within limits there is a scope for increasing the 'α' value with the increase in RTT resulting in aggressive bandwidth utilization.

Interaction with UDP Traffic: The UDP does not respond to congestion and hence is called "non-interactive" traffic. The effect of the CBR/UDP traffic on STCP was also observed in our experiments. The CBR agent at about 500 Mbps (half of the bottleneck link capacity) was started and stopped at regular as well as irregular intervals for both LTCP and STCP. It is observed that the response of LTCP and STCP is very similar to the UDP traffic. When the CBRagent is active the sending rate is greatly reduced but as soon as the CBR agent is stopped both LTCP and STCP quickly increase their sending rate.

4.6.2 Random topology

Figure 22 shows the topology used to test STCP with Layered TCP and conventional TCP. The TCP source agents and the receivers are denoted by Sx and Rx where 'x' is the node identifier. The STCP source agents and receiver agents are denoted by Ssx and Srx respectively. At first the network has only TCP and STCP agents. The values for throughput, average delay and the congestion window rate are calculated. Then the STCP agents are replaced by LTCP agents and the same performance parameters are measured again.

Analysis of the congestion window-variation: Figure 23 shows the congestion window variation of LTCP vs STCP. It is observed that the rate of variation for STCP is slightly higher as compared to LTCP indicating that the number of packets/segments sent by STCP is greater than LTCP which in turn indicates better performance in from STCP.

Analysis of the delay: The average delay results were calculated after as simulation time of 65. The results are tabulated in Table 2. It is clear that the delay of TCP is comparable to that of STCP. Hence, STCP can be considered to be a promising choice for real-time networks with random topologies.

Table 2: Average delay results for the random topology

Protocol	**Value of delay in miliseconds**
TCP	0.060013
LCTP	0.062589
STCP	0.063424

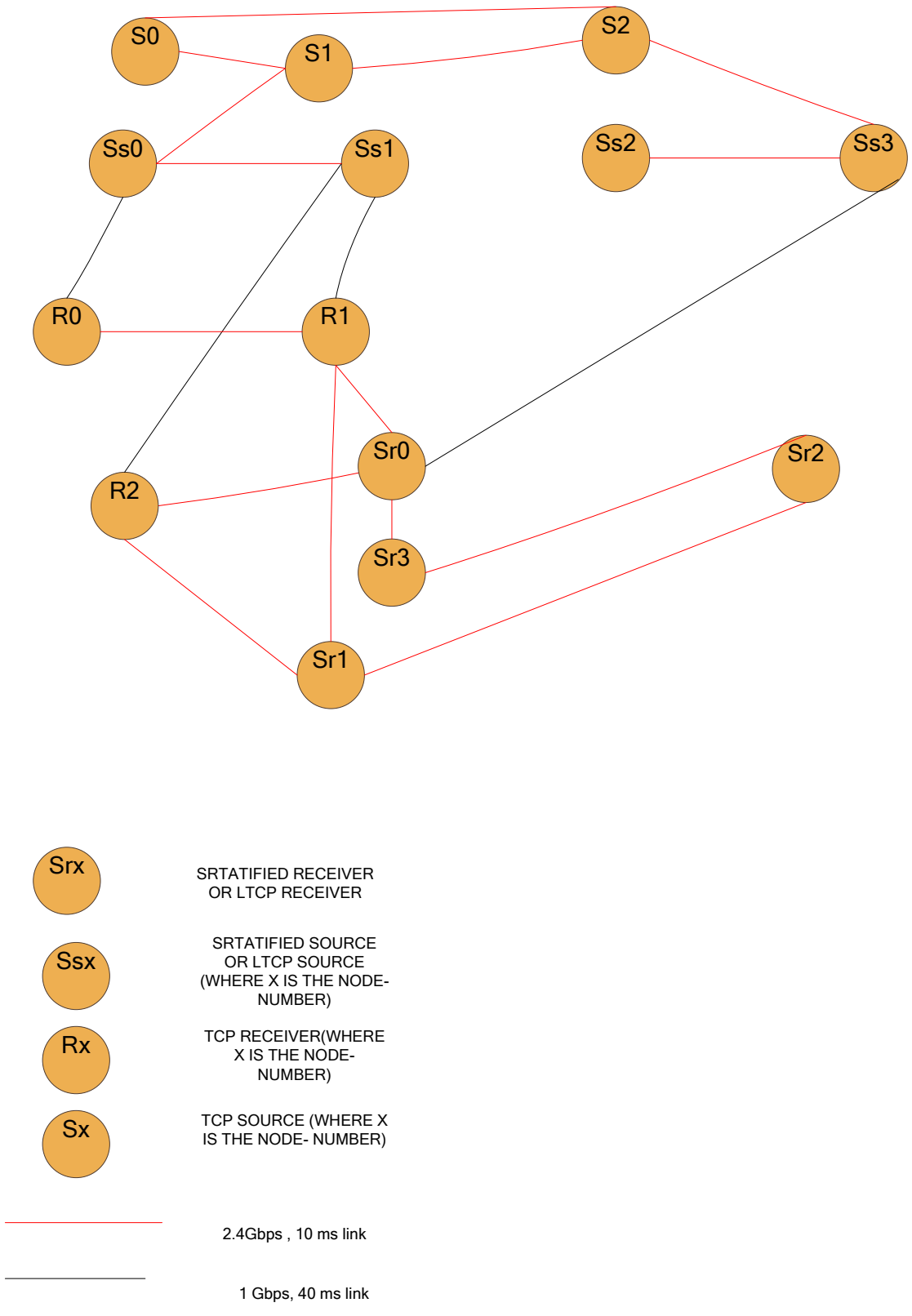


Figure 22: A random topology

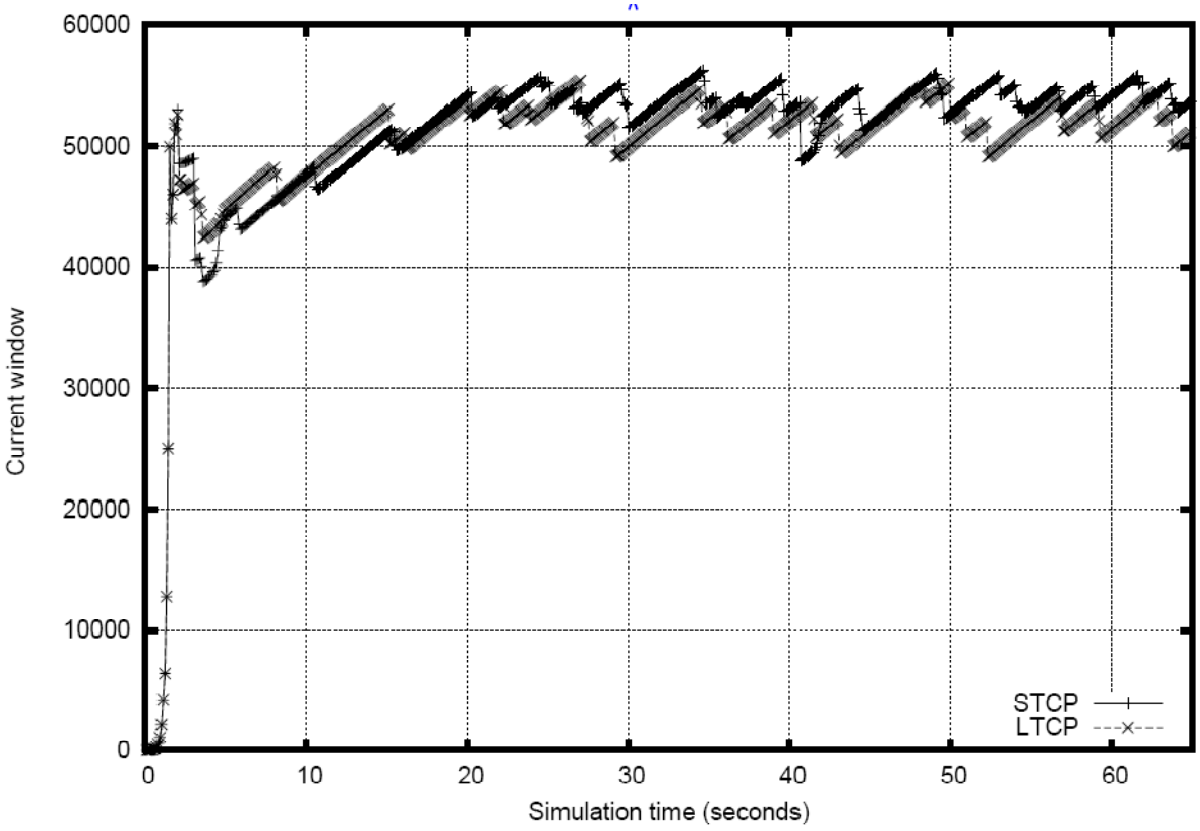


Figure 23: Congestion window variation (LCTP vs STCP)

Analysis of throughput: When the network containing TCP and STCP agents was tested the following results were obtained (Tables 3 and 4)

TCP results:

Table 3: TCP throughput results when the network contains TCP and STCP agents

Source agent identifier for TCP	Throughput (Mbps)	Average throughput(Mbps)
Source 0	183.52246153846153	**188.8149**
Source 1	247.6039384615384	
Source 2	135.3184	

STCP results:

Table 4: STCP throughput results when the network contains TCP and STCP agents

Source agent identifier for STCP	Throughput (Mbps)	Average throughput(Mbps)
Source 0	504.68824615384614	**555.8052**
Source 1	938.46043076923081	
Source 2	508.2107076923076	
Source 3	271.86141538461538	

When the network containing TCP and Layered TCP agents was tested the following results were obtained (Tables 5 and 6).

TCP results:

Table 5: TCP throughput results when the network contains TCP and LTCP agents.

Source agent identifier for TCP	Throughput (Mbps)	Average throughput(Mbps)
Source 0	190.25772307692307	**187.8735**
Source 1	255.83273846153847	
Source 2	117.53033846153846	

LTCP results:

Table 6: LTCP throughput results when the network contains TCP and LTCP agents

Source agent identifier for LTCP	Throughput (Mbps)	Average throughput(Mbps)
Source 0	489.65390769230771	**561.007**
Source 1	936.88996923076922	
Source 2	455.41932307692309	
Source 3	362.06498461538462	

As shown in Tables 3, 4, 5 and 6, STCP is better than the TCP and is slightly better than LTCP in terms of throughput when they are tested on networks with random topologies.

4.7 Summary

In this chapter, we have explained the principle, algorithm, and the implementation of the proposed protocol, namely Stratified TCP. The protocol is implemented in the C++ language and tested using the Network simulator (2.26 version). The results are evaluated with respect to throughput, delay, fairness etc for a simple dumbbell topology and also for a random topology. On the basis of these results it can be concluded that STCP is a promising choice for general and/or random networks with varying high-speed link bandwidths and delay values.

5 CONCLUSIONS

The literature survey of the schemes related to TCP indicates that TCP is a highly effective protocol for reliable data transfer. However, due to the growing demand for increased data-transmission-speeds, there is a need for better techniques which transfer huge amounts of data at great speeds under different network scenarios. In this work, a number of these schemes have been described in detail, carefully weighing their pros and cons. A new algorithm, known as **Stratified TCP (STCP)** is also suggested and implemented, which upholds the basic-working-principle of TCP and slightly modifies its congestion control algorithm to perform considerably better than TCP in high speed networks in terms of throughput. The STCP is based on the "virtual layer" principle which has already been proved to be successful in high-speed networks.

STCP uses the TCP sending rate equation to calculate the current congestion window. It is highly responsive to the changes in the network because it adapts dynamically to the changing RTT values, which is an indicator of network-congestion. To further prove this point, α (increase) parameter-variation with respect to RTT values is also studied. The "α" value also depends on the "packet-loss-rate". If the packet-loss rate is well under the threshold- value, as described in the previous chapter (in the result analysis section) α value is increased otherwise it is decreased when the packet-loss rate is above our threshold value. The graph of RTT vs 'α' further proves the point made above because there is a linear variation between the RTT and alpha values when the loss rate is negligible.

Simulations are done on NS 2.26 version of the network simulator using standard TCL scripts. The results are analyzed against the standard performance-measurement parameters like throughput, fairness, delay, and variation of the congestion window. Tests were conducted on two different topologies namely, the dumbbell topology and the random topology. For the dumbbell topology it is observed from the congestion-window variation results that the slope for the variation of the congestion window

in STCP is steeper than LTCP or conventional TCP. As a result we conclude that STCP makes better usage of bandwidth and increases the congestion window more aggressively and adaptively to take **maximum advantage of the available bandwidth in high-speed networks**. The fairness results of STCP in a network of many STCP flows show that STCP exhibits **high intra-protocol fairness properties**. The STCP delay introduces very limited dynamic calculations which are a very small overhead, considering the performance improvement as a result of these calculations. The throughout results of STCP emphasize the fact that STCP-performance is better than that of conventional TCP and LTCP.

For the random topology, which reflects the real-time network conditions in a better way, the throughput, congestion window rate and delay results indicate that STCP is far better than conventional TCP and STCP has a delay comparable to that of TCP. Hence, it is suitable for high-speed network scenarios.

Therefore, the simulation results clearly show that there is a considerable improvement in the performance of STCP when compared to both conventional TCP protocol and LTCP. This makes STCP an efficient alternative for reliable and fast data-transmission for huge volumes of data.

To conclude, STCP is a novel technique which uses the basic principle of TCP and makes suitable changes to respond to dynamic conditions of the network. Protocols like LTCP lack the capability to adapt to the changing network conditions and STCP is a clear improvement over such protocols. STCP also emphasizes the concept of “strata or layers” which is a proven method to meet the increasing data-transmission demands for today’s data-centric applications. STCP needs no changes in the infrastructure and easily adapts to the high-speed network infrastructure, which is another major advantage.

There are still a number of scopes for improvement. For example, when there are flows with varying RTT values, the RTT compensation factor used in STCP is not effective in handling these varying flows. In real-time, networks usually have flows of varying RTT values. Hence, it is important for the transmission protocol to provide the same performance irrespective of the difference between the RTT-values for each

of these flows. The second possible improvement is to further enhance the fairness among multiple TCP flows with different end-to-end delays. Particularly, in high-speed networks, the impact of such unfairness can be significant and may degrade the quality of service in the system by a nontrivial margin.

6 REFERENCES

[1] Haejung Lim, K.X., M. Gerla, *TCP performance over multipath routing in mobile ad hoc networks* 2003. **Vol 2** p. 1064--1068.

[2]. Chandrasekaran, M.K., M. Wahida Banu, R.S.D.Chandrasekaran, M. Kalpana, M. Wahida Banu, R.S.D., *Interaction between polynomial congestion control algorithms MIMD-poly and PIPD-poly and other TCP variants in TCP/IP networks.*

[3]. Luglio, M.S., M.Y. Gerla, M. Stepanek, J. , *On-board satellite "split TCP" proxy.* Feb 2004. **22**(2): p. 362- 370.

[4]. Hasegawa, T.K., T. Yoshiizumi, K. Miki, T. Hokamura, K. Sawada, KS, KDD R&D LabsInc., *Protocol architecture of high speed TCP/IP service overinternational ATM network.*,2002-08-06 p. 159-168.

[5]. Ke Zhang, C.P.F., *An enhancement of TCP Veno with forward acknowledgement,* ACM, 2008. **31**(15): p. 3683-3690.

[6]. Haining Wang, Kang G. Shin, *Robust TCP Congestion Recovery*, icdcs, pp.0199, 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01), 2001.

[7]. Chang, C.L.C.F.C.L.S., *Improvements achieved by SACK employing TCP Veno equilibrium-oriented mechanism over lossy networks,* IEEE CNF, 2001. **1**: p. 202-209

[8]. Tomoya Hatano, Hiroshi Shigeno, Ken-ichi Okada, *TCP-friendly Congestion Control for HighSpeed Network*, saint, pp.10, 2007 International Symposium on Applications and the Internet (SAINT'07), 2007

[9]. Parvez, N.H., L and W. Manitoba Univ., Man., Canada, *Improving TCP performance in wired-wireless networks by using a novel adaptive bandwidth estimation mechanism.* IEEE CNF, 2004. **5**: p. 2760-2764.

[10]. Singh, A.K.I., S. and I. Dept of Comput. Sci. & Eng., Bombay, India; *ATCP: Improving TCP per formance over mobile wireless environments, IEEE* CNF, 2002.

[11]. Spatscheck, O.H., J.S. Hartman, J.H. Peterson, L.L, *Optimizing TCP forwarder performance.* IEEE CNF, 2000. **8**(2): p. 146-157.

[12]. Taleb, T.K., N. Nemoto, Y, *An explicit and fair window adjustment method to enhance TCP efficiency and fairness over multihops Satellite networks.* IEEE CNF, 2004. **22**(2): p. 371-387.

[13]. Huang, H.-f.L.L.-j.L.Z.-y.Y.X.-y., *TCP Vegas_M: A Novel TCP Algorithm in Mobile Ad hoc Network.* IEEE CNF, 2006: p. 629-633.

[14]. Yamanegi, K.H., T. Hasegawa, G. Murata, M. Shimonishi, H. Murase, T, *Implementation Experiments of the TCP Proxy Mechanism.* IEEE CNF, 2005: p. 17-22.

[15]. Chen, M.Z., J. Murthi, M.N. Kamal Premaratne *TCP congestion avoidance: a network calculus interpretation and performance improvements.* IEEE CNF, 2005. **2**: p. 914-925.

[16]. Hiroyuki Ohsaki, Makoto Imase, "On Modeling GridFTP Using Fluid-Flow Approximation for High Speed Grid Networking," saint-w, pp.638, 2004 Symposium on Applications and the Internet-Workshops (SAINT 2004 Workshops), 2004.

[17]. Jain, A.P., A. Thakur, R.C. Bhatia, M.P.S, *TCP analysis over wireless mobile ad hoc networks.* IEEE CNF, 2002: p. 95-99.

[18]. Hasegawa, G.M., M. Miyahara, H, *Fairness and stability of congestion control mechanisms of TCP Hasegawa.* IEEE CNF, 1999. **3**: p. 1329-1336

[19]. Kitae Nahm, Ahmed Helmy, C.-C. Jay Kuo, *Cross-Layer Interaction of TCP and Ad Hoc Routing Protocols in Multihop IEEE 802.11 Networks*,*IEEE Transactions on Mobile Computing*, vol. 7, no. 4, pp. 458-469, Apr. 2008, doi:10.1109/TMC.2007.70779.

[20]. Caini, C.F., R. Lacamera, D, *The TCP "Adaptive-Selection" Concept.* IEEE CNF, 2008. **2**(1): p. 83-89.

[21]. Dongkyun Kim Toh, C.K.H.Y., *The Impact of Spurious Retransmissions on TCP Performance in AD HOC Mobile Wireless Networks.* IEEE CNF, 2007: p. 1-5.

[22]. Lee, Z.Z.C.F.B.S., *A refinement to improve TCP Veno performance under bursty congestion* IEEE CNF, 2005. **2**: p. 5pp-834.

[23]. Qixiang Pang Liew, S.C.L., V.C.M, *Performance improvement of 802.11 wireless network with TCP ACK agent and auto-zoom back off algorithm.* IEEE CNF. **3**: p. 2046-2050.

[24]. Cho, Y.-S.C.K.-W.L.T.-M.H.Y.-Z., *High-speed TCP protocols with pacing for fairness and TCP friend-liness.* IEEE, 2004. **4**: p. 21-24.

[25] Vangala, S.L., M.A, *TCP SACK-aware snoop protocol for TCP over wireless networks.* IEEE, 2003. **4**: p. 2624-2628.

[26]. Choe, S.P.S.A.Y.C.S., *Wireless TCP model for short-lived flows.* IEEE, 2003. **3**(1090-3038): p. 1725-1729.

[27]. Lili Qiu Yin Zhang Keshav, S., *On individual and aggregate TCP performance.* IEEE, 1999: p. 203-212.

[28]. Hyunyong Choi, B.G.L., *A TCP agent scheme based on active buffer control to support lossless handover in broadband wireless networks.* IEEE, 2002. **3**: p. 2493-2497.

[29]. Girish Motwani, G., K, *Evaluation of advanced TCP stacks in the iSCSI environment using simula-tion model.* IEEE, 2005: p. 210-217.

[30]. Jie Feng, L.X., *On the Time Scale of TCP-Friendly Admission Control Protocols.* IEEE, 2008: p. 45-51.

[31]. Sang Seok Lim, Kyu Ho Park, "TPF: TCP Plugged File System for Efficient Data Delivery over TCP," *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 459-473, Apr. 2007, doi:10.1109/TC.2007.1009

[32]. Christopher Metz, "TCP over Satellite... The Final Frontier," *IEEE Internet Computing*, vol. 3, no. 1, pp. 76-80, Jan./Feb. 1999, doi:10.1109/4236.747326.

[33]. Yan-Jun Li, E., S, *TCP/IP performance and behavior over an ATM network.* IEEE, 1996: p. 1-9.

[34]. Parsa, C., Garcia-Luna-Aceves, J.J, *Improving TCP congestion control over Internets with heterogeneous transmission media.* IEEE, 1999: p. 213-221.

[35]. Kobayashi, M.M., T, *Asymmetric TCP splicing for content-based switches.* IEEE. **2**: p. 1321-1326.

[36]. Marco Mellia, M.M., and Claudio Casetti, TCP *smart framing: a segmentation algorithm to reduce TCP latency.* IEEE/ACM, 2005. **13**(2): p. 316-329.

[37]. Grieco, L.A.M., S, *Mathematical analysis of Westwood+TCP congestion control.* IEEE, 2005. **152**(1): p. 35-42.

[38]. Chang-Yi Luo Komuro, N.T., K. Tsuboi, T, *Paced TCP: A Dynamic Bandwidth Probe TCP with Pacing in AD HOC Networks.* IEEE, 2007: p. 1-5.

[39]. Cordeiro, C.D.A.D., S.R. Agrawal, D.P., *COPAS: dynamic contention-balancing to enhance the performance of TCP over multi-hop wireless networks.* IEEE, 2002.

[40]. B.P. Lee, R.K. Balan, L. Jacob, W.K.G. Seah, A.L. Ananda, "TCP tunnels: avoiding congestion collapse," lcn, pp.408, 25th Annual IEEE International Conference on Local Computer Networks (LCN'00), 2000

[41]. Hee-Jin Jang, Y.-J.S., *A flow control scheme for improving TCP throughput and fairness for wireless networks.* IEEE, 2003. **2**: p. 999-1003.

[42]. Yoonjoo Kwon, W.S., Giljae Lee, Jaeseung Kwak Hybridtcp *for High Speed Data Transfer.* IEEE, 2007. **3**: p. 2133-2136.

[43]. Todorovic, M.L.-B., N, *Efficiency Study of TCP Protocols in InfrastructuredWireless Networks.* IEEE: p. 103-103.

[44]. Katsuhiro Naito, Kazuo Mori, Hideo Kobayashi, "Proposal of a Base Station Diversity Technique Based on TCP Performance," wicon, pp.104-111, First International Conference on Wireless Internet (WICON'05), 2005.

[45]. Haijie Huang, Jianfei Cai, "Improving TCP Performance during Soft Vertical Handoff," aina, vol. 2, pp.329-332, 19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 2 (INA, USW, WAMIS, and IPv6 papers), 2005.

[46]. Shugong Xu, T.S., Myung Lee, *TCP over wireless multi-hop networks.* IEEE, 2001. **1**: p. 282-288.

[47]. Claudio Casetti Mario, M.G., Scott Seongwook Lee, Saverio Mascolo, Medy Sanadidi, *Tcp with Faster Recovery.* 2000.

[48]. Go Hasegawa, K.K., Masayuki Murata, *Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet*. 2000. p. 177-186.

[49]. H.T. Kung, S.Y. Wang, "TCP Trunking: Design, Implementation and Performance," icnp, pp.222, Seventh International Conference on Network Protocols (ICNP'99), 1999.

[50]. Lin, D.K., H.T, *TCP fast recovery strategies: analysis and improvements.* IEEE, 1998. **1**: p. 263-271.

[51]. H. Wang, C. Williamson, "A New Scheme for TCP Congestion Control: Smooth-Start and Dynamic Recovery," mascots, pp.69, Sixth IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'98), 1998.

[52]. Ji, M.X.H.L.Y., *RWER TCP Throughput Enhancement-Based on a GE-PON System.* IEEE, 2007: p. 448-451.

[53]. Shihada, B.Q.Z.P.-H.H., *Threshold-based TCP Vegas over Optical Burst Switched Networks.* IEEE, 2006: p. 119-124.

[54]. Isci, D.D.A., F. Caglayan, M.U., *IPSEC over satellite links: a new flow identification method.* IEEE, 2006: p. 140-145.

[55]. Xin Yu Kedem, Z.M., *Reducing the effect of mobility on TCP by making route caches quickly adapt to topology changes.* IEEE, 2004. **7**: p. 4103-4110.

[56]. Zhou J, Shi B, Zou L*. Improve TCP performance in ad-hoc network by TCP-RC*. In Proc. 14th IEEE Int. Symposium on Personal, Indoor and Mobile Radio Communications , Beijing, China, Sept. 2003, pp.216—220.

[57]. D.S. Lee and C C. Lin*, Window adaptive TCP for EGPRS networks.* Journal of Information Science and Engineering, vol. 20, no. 5, pp. 805–820, September 2004.

[58]. www.ixiacom.com/.../white_papers/p2p/img3.jpg

[59]. Xians, Y., Q. Chunming, and L. Yona. *TCP implementations and false time out detection in OBS networks*. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. 2004.

[60]. Sang Bae , K.X., Sungwook Lee, M Gerla, *Measured analysis of TCP behavior across multihop wireless and wired networks.* IEEE, 2002. **1**: p. 153-157.

[61]. Fu, Z.Z., P. Luo, H. Lu, S. Zhang, L. Gerla, M., *The impact of multihop wireless channel on TCP throughput and loss.* IEEE, 2003. **3**: p. 1744-1753.

[62]. Palazzo, I.F.A.a.G.M.a.S., *TCP-Peach: a new congestion control scheme for satellite IP networks* IEEE/ACM Transactions on Networking, 2001. **9**: p. 307-321.

[63]. Moonsoo, K. and M. Jeonghoon. *On the Pacing Technique for High Speed TCP Over Optical Burst Switching Networks*. *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*. 2006.

[64]. Go Hasegawa, M.M., *TCP symbiosis: congestion control mechanisms of TCP based on Lotka-Volterra competition model.* ACM International Conference Proceeding Series, 2006. **200.**

[65]. Yan, Z., W. Sheng, and L. Lemin. *B-Reno: A New TCP Implementation Designed for TCP over OBS Networks*. Future Generation Communication and Networking (FGCN 2007). 2007.

[66]. Wang, B., et al., *Multimedia streaming via TCP: an analytic performance study*, in *Proceedings of the 12th annual ACM international conference on Multimedia*. 2004, ACM: New York, NY, USA.

[67]. Bakre, A.V. and B.R. Badrinath, *Implementation and Performance Evaluation of Indirect TCP.* IEEE Trans. Comput., 1997. **46**(3): p. 260-278.

[68]. Pan, H.W.F.R.D.M.W., *Utilizing TTL to Enhance TCP Fairness.* Communications and Networking in China, 2007. CHINACOM '07. , 2007: p. 208-212.

[69]. Fanglei, S. and V.O.K. Li. *An Adaptive TCP Algorithm to Support Handoffs in Heterogeneous Overlay Networks*. in *TENCON 2006. 2006 IEEE Region 10 Conference*. 2006.

[70]. Maki, I., et al. *Performance analysis and improvement of TCP proxy mechanism in TCP overlay networks*. in *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*. 2005.

[71]. Pentikousis, K., *Can TCP be the transport protocol of the 21st century?* Crossroads, 2000. **7**(2): p. 25-29.

[72]. Crowcroft, J. and P. Oechslin, *Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP.* SIGCOMM Comput. Commun. Rev., 1998. **28**(3): p. 53-69.

[73]. Hacker, T.J., B.D. Noble, and B.D. Athey. *Improving throughput and maintaining fairness using parallel TCP*. in *INFOCOM 2004.* Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies. 2004.

[74]. Hung-Yun Hsieh, Raghupathy Sivakumar, *pTCP: An End-to-End Transport Layer Protocol for Striped Connections,* icnp, pp.24, 10th IEEE International Conference on Network Protocols (ICNP'02), 2002.

[75]. Foster, I. and C. Kesselman, *The Globus toolkit*, in *The grid: blueprint for a new computing infrastructure*. 1999, Morgan Kaufmann Publishers Inc. p. 259-278.

[76]. Takeshi Ito, Hiroyuki Ohsaki, Makoto Imase, *GridFTP-APT: Automatic Parallelism Tuning Mechanism for Data Transfer Protocol GridFTP*,ccgrid, pp.454-461, Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06), 2006.

[77]. Shihada, B., H. Pin-Han, and Z. Qiong. *A Novel False Congestion Detection Scheme for TCP over OBS Networks*. Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE. 2007.

[78]. http://en.wikipedia.org/wiki/BitTorrent_ (protocol) (Accessed on Dec 25th).

[79]. Hacker, T.J., B.D. Noble, and B.D. Athey. *Improving throughput and maintaining fairness using parallel TCP*. in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. 2004.

[80]. Lazzez, A., N. Boudriga, and M.S. Obaidat. *Improving TCP QoS over OBS networks: A scheme based on optical segment retransmission.* Performance Evaluation of Computer and Telecommunication Systems, 2008. SPECTS 2008. International Symposium on. 2008.

[81]. Shuping, P., et al. *Drop Policy Combined with Multiple Edge Thresholds in TCP over OBS Networks*. PhotonicsGlobal@Singapore, 2008. IPGC 2008. IEEE. 2008.

[82]. Padmanabhan, D., R. Bikram, and V.M. Vokkarane. *TCP Over Optical Burst Switching (OBS): To Split or Not To Split?* Computer Communications and Networks, 2008. ICCCN '08. Proceedings of 17th International Conference on. 2008.

[83]. T. D. Dyer and R. V. Boppana, *A comparison of TCP performance over three routing protocols for mobile ad hoc networks*, Proceedings of ACM MobiHoc'01, Oct. 2001.

[84]. Zongsheng Zhang, Go Hasegawa, Masayuki Murata, *Performance Analysis and Improvement of HighSpeed TCP with TailDrop/RED Routers*, mascots, pp.505-512, 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04), 2004.

[85]. Sally Floyd, “HighSpeed TCP for Large Congestion Windows”, RFC 3649 December 2003.

[86]. Tom Kelly, *Scalable TCP: Improving Performance in HighSpeed Wide Area Networks*, ACM Computer Communications Review, April 2003.

[87]. Cheng, J., D.X. Wei, and S.H. Low. *FAST TCP: motivation, architecture, algorithms, performance*. in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. 2004.

[88]. Bhandarkar, S., S. Jain, and A.L.N. Reddy, *LTCP: improving the performance of TCP in highspeed networks*. SIGCOMM Comput. Commun. Rev., 2006. 36(1): p. 41-50.

[89]. Todorovic, M. and N. Lopez-Benitez. *Efficiency Study of TCP Protocols in InfrastructuredWireless Networks*. in *Networking and Services, 2006. ICNS '06. International conference on*. 2006.

7 APPENDICES

7.1 APPENDIX I: THE C++ Code Used to Implement the Stratified TCP

7.1.1 The Header File Containing the Declarations for the STCP Variables.

```
/* -*- Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1991-1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *     This product includes software developed by the Computer Systems
 *     Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * @(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/tcp/tcp.h,v 1.107 2003/02/12 04:16:10
sfloyd Exp $ (LBL)
 */


/**
CODE MODIFIFED BY: RANJITHA SHIVARUDRAIAH
OBJECTIVE:          TO IMPLEMENT STRATIFIED TCP( STCP)
COURSE:             MS THESIS
                   (A STUDY OF PARALLEL TRANSPORT TECHNIQUES
                     FOR HIGH-SPEED NETWORKS)

CODE DESCRIPTION: "tcp.h" contains the declarations for all the TCP variants that have
been tested/implemented by sfloyd and group. In this file, the variables needed for
the implementation of the Stratified TCP protocol are added. The modifications are
recognized by the tag "STCP_ADDITIONS".
```

```
*/


#ifndef ns_tcp_h
#define ns_tcp_h

#include "agent.h"
#include "packet.h"

//class EventTrace;

struct hdr_tcp {
#define NSA 3
       double ts_;             /* time packet generated (at source) */
       double ts_echo_;        /* the echoed timestamp (originally sent by
                                  the peer) */
       int seqno_;             /* sequence number */
       int reason_;            /* reason for a retransmit */
       int sack_area_[NSA+1][2];  /* sack blocks: start, end of block */
       int sa_length_;         /* Indicate the number of SACKs in this  *
                                * packet.  Adds 2+sack_length*8 bytes   */
       int ackno_;             /* ACK number for FullTcp */
       int hlen_;              /* header len (bytes) for FullTcp */
       int tcp_flags_;         /* TCP flags for FullTcp */
       int last_rtt_;             /* more recent RTT measurement in ms, */
                           /*   for statistics only */

       static int offset_; // offset for this header
       inline static int& offset() { return offset_; }
       inline static hdr_tcp* access(Packet* p) {
              return (hdr_tcp*) p->access(offset_);
       }

       /* per-field member functions */
       double& ts() { return (ts_); }
       double& ts_echo() { return (ts_echo_); }
       int& seqno() { return (seqno_); }
       int& reason() { return (reason_); }
       int& sa_left(int n) { return (sack_area_[n][0]); }
       int& sa_right(int n) { return (sack_area_[n][1]); }
       int& sa_length() { return (sa_length_); }
       int& hlen() { return (hlen_); }
       int& ackno() { return (ackno_); }
       int& flags() { return (tcp_flags_); }
       int& last_rtt() { return (last_rtt_); }
};

/* these are used to mark packets as to why we xmitted them */
#define TCP_REASON_TIMEOUT 0x01
#define      TCP_REASON_DUPACK   0x02
#define      TCP_REASON_RBP            0x03   // used only in tcp-rbp.cc
#define TCP_REASON_PARTIALACK   0x04

/* these are reasons we adjusted our congestion window */

#define      CWND_ACTION_DUPACK  1      // dup acks/fast retransmit
#define      CWND_ACTION_TIMEOUT 2      // retransmission timeout
#define      CWND_ACTION_ECN           3      // ECN bit [src quench if supported]

/* these are bits for how to change the cwnd and ssthresh values */

#define      CLOSE_SSTHRESH_HALF 0x00000001
#define      CLOSE_CWND_HALF           0x00000002
```

```
#define      CLOSE_CWND_RESTART  0x00000004
#define      CLOSE_CWND_INIT            0x00000008
#define      CLOSE_CWND_ONE             0x00000010
#define CLOSE_SSTHRESH_HALVE     0x00000020
#define CLOSE_CWND_HALVE   0x00000040
#define THREE_QUARTER_SSTHRESH  0x00000080
#define CLOSE_CWND_HALF_WAY      0x00000100
#define CWND_HALF_WITH_MIN 0x00000200

/*
 * tcp_tick_:
 * default 0.1,
 * 0.3 for 4.3 BSD,
 * 0.01 for new window algorithms,
 */

#define NUMDUPACKS 3             /* This is no longer used.  The variable */
                           /* numdupacks_ is used instead. */
#define TCP_MAXSEQ 1073741824   /* Number that curseq_ is set to for */
                           /* "infinite send" (2^30)            */

#define TCP_TIMER_RTX            0
#define TCP_TIMER_DELSND   1
#define TCP_TIMER_BURSTSND 2
#define TCP_TIMER_DELACK   3
#define TCP_TIMER_Q         4
#define TCP_TIMER_RESET        5

class TcpAgent;

class RtxTimer : public TimerHandler {
public:
       RtxTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
       virtual void expire(Event *e);
       TcpAgent *a_;
};

class DelSndTimer : public TimerHandler {
public:
       DelSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
       virtual void expire(Event *e);
       TcpAgent *a_;
};

class BurstSndTimer : public TimerHandler {
public:
       BurstSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
       virtual void expire(Event *e);
       TcpAgent *a_;
};


class TcpAgent : public Agent {
public:
       TcpAgent();
         virtual void recv(Packet*, Handler*);
       virtual void timeout(int tno);
       virtual void timeout_nonrtx(int tno);
       int command(int argc, const char*const* argv);
       virtual void sendmsg(int nbytes, const char *flags = 0);
```

```
        void trace(TracedVar* v);
        virtual void advanceby(int delta);
protected:
        virtual int window();
        virtual double windowd();
        void print_if_needed(double memb_time);
        void traceAll();
        virtual void traceVar(TracedVar* v);
        virtual int headersize();   // a tcp header

        virtual void delay_bind_init_all();
        virtual int delay_bind_dispatch(const char *varName, const char *localName,
TclObject *tracer);

        /*
         * State encompassing the round-trip-time estimate.
         * srtt and rttvar are stored as fixed point;
         * srtt has 3 bits to the right of the binary point, rttvar has 2.
         */
        TracedInt t_seqno_; /* sequence number */
#define T_RTT_BITS 0
        TracedInt t_rtt_;          /* round trip time */
        int T_SRTT_BITS;        /* exponent of weight for updating t_srtt_ */
        TracedInt t_srtt_;         /* smoothed round-trip time */
        int srtt_init_;            /* initial value for computing t_srtt_ */
        int T_RTTVAR_BITS;      /* exponent of weight for updating t_rttvar_ */
        int rttvar_exp_;        /* exponent of multiple for t_rtxcur_ */
        TracedInt t_rttvar_;       /* variance in round-trip time */
        int rttvar_init_;       /* initial value for computing t_rttvar_ */
        double t_rtxcur_;   /* current retransmit value */
        double rtxcur_init_;    /* initial value for t_rtxcur_ */
        TracedInt t_backoff_;      /* current multiplier, 1 if not backed off */
        virtual void rtt_init();
        virtual double rtt_timeout();    /* provide RTO based on RTT estimates */
        virtual void rtt_update(double tao);    /* update RTT estimate */
        virtual void rtt_backoff();             /* double multiplier */

        double ts_peer_;        /* the most recent timestamp the peer sent */

        /* connection and packet dynamics */
        virtual void output(int seqno, int reason = 0);
        virtual void send_much(int force, int reason, int maxburst = 0);
        virtual void newtimer(Packet*);
        virtual void dupack_action();           /* do this on dupacks */
        virtual void send_one();         /* do this on 1-2 dupacks */
        double linear(double x, double x_1, double y_1, double x_2, double y_2);
        /* the "linear" function is for experimental highspeed TCP */
        void opencwnd();

        void slowdown(int how);                 /* reduce cwnd/ssthresh */
        void ecn(int seqno);             /* react to quench */
        virtual void set_initial_window();      /* set IW */
        double initial_window();         /* what is IW? */
        void reset();
        void newack(Packet*);
        void tcp_eln(Packet *pkt); /* reaction to ELN (usually wireless) */
        void finish(); /* called when the connection is terminated */
        void reset_qoption();      /* for QOption with EnblRTTCtr_ */
        void rtt_counting();       /* for QOption with EnblRTTCtr_ */
        int network_limited();     /* Sending limited by network? */
        double limited_slow_start(double cwnd, double max_ssthresh, double increment);
                        /* Limited slow-start for high windows */
```

```
virtual int numdupacks(double cwnd);    /* for getting numdupacks_ */
virtual void processQuickStart(Packet *pkt);
virtual void endQuickStart();

/* Helper functions. Currently used by tcp-asym */
virtual void output_helper(Packet*) { return; }
virtual void send_helper(int) { return; }
virtual void send_idle_helper() { return; }
virtual void recv_helper(Packet*) { return; }
virtual void recv_newack_helper(Packet*);
virtual void partialnewack_helper(Packet*) {};

/* Timers */
RtxTimer rtx_timer_;
DelSndTimer delsnd_timer_;
BurstSndTimer burstsnd_timer_;
virtual void cancel_timers() {
       rtx_timer_.force_cancel();
       burstsnd_timer_.force_cancel();
       delsnd_timer_.force_cancel();
}
virtual void cancel_rtx_timer() {
       rtx_timer_.force_cancel();
}
virtual void set_rtx_timer();
void reset_rtx_timer(int mild, int backoff = 1);
int timerfix_;             /* set to true to update timer *after* */
                    /* update the RTT, instead of before   */
int rfc2988_;       /* Use updated RFC 2988 timers */
double boot_time_;  /* where between 'ticks' this sytem came up */
double overhead_;
double wnd_;
double wnd_const_;
double wnd_th_;            /* window "threshold" */
double wnd_init_;
double wnd_restart_;
double tcp_tick_;   /* clock granularity */
int wnd_option_;
int wnd_init_option_;   /* 1 for using wnd_init_ */
                    /* 2 for using large initial windows */
double decrease_num_;   /* factor for multiplicative decrease */
double increase_num_;   /* factor for additive increase */
double k_parameter_;     /* k parameter in binomial controls */
double l_parameter_;     /* l parameter in binomial controls */
int precision_reduce_;  /* non-integer reduction of cwnd */
int syn_;           /* 1 for modeling SYN/ACK exchange */
int delay_growth_;         /* delay opening cwnd until 1st data recv'd */
int tcpip_base_hdr_size_;  /* size of base TCP/IP header */
int ts_option_size_;    // header bytes in a ts option
int bug_fix_;       /* 1 for multiple-fast-retransmit fix */
int less_careful_;  /* 1 for Less Careful variant of bug_fix_, */
                    /*  for illustration only  */
int ts_option_;            /* use RFC1323-like timestamps? */
int maxburst_;             /* max # packets can send back-2-back */
int maxcwnd_;       /* max # cwnd can ever be */
 int numdupacks_;   /* dup ACKs before fast retransmit */
int numdupacksFrac_;       /* for a larger numdupacks_ with large */
                    /* windows */
double maxrto_;            /* max value of an RTO */
double minrto_;         /* min value of an RTO */
int old_ecn_;       /* For backwards compatibility with the
                     * old ECN implementation, which never
                     * reduced the congestion window below
```

```
                     * one packet. */
FILE *plotfile_;
/*
 * Dynamic state.
 */
TracedInt dupacks_; /* number of duplicate acks */
TracedInt curseq_;  /* highest seqno "produced by app" */
int last_ack_;          /* largest consecutive ACK, frozen during
                         *          Fast Recovery */
TracedInt highest_ack_;    /* not frozen during Fast Recovery */
int recover_;       /* highest pkt sent before dup acks, */
                    /*   timeout, or source quench/ecn */
int last_cwnd_action_;     /* CWND_ACTION_{TIMEOUT,DUPACK,ECN} */
TracedDouble cwnd_; /* current window */
double base_cwnd_;  /* base window (for experimental purposes) */
double awnd_;       /* averaged window */
TracedInt ssthresh_;       /* slow start threshold */
int count_;         /* used in window increment algorithms */
double fcnt_;       /* used in window increment algorithms */
int rtt_active_;    /* 1 if a rtt sample is pending */
int rtt_seq_;       /* seq # of timed seg if rtt_active_ is 1 */
double rtt_ts_;           /* time at which rtt_seq_ was sent */
TracedInt maxseq_;  /* used for Karn algorithm */
                    /* highest seqno sent so far */
int ecn_;           /* Explicit Congestion Notification */
int cong_action_;   /* Congestion Action.  True to indicate
                       that the sender responded to congestion. */
 int ecn_burst_;          /* True when the previous ACK packet
                         *  carried ECN-Echo. */
int ecn_backoff_;   /* True when retransmit timer should begin
                        to be backed off.  */
int ect_;           /* turn on ect bit now? */
 int eln_;               /* Explicit Loss Notification (wireless) */
 int eln_rxmit_thresh_;  /* Threshold for ELN-triggered rxmissions */
 int eln_last_rxmit_;    /* Last packet rxmitted due to ELN info */
double firstsent_;  /* When first packet was sent  --Allman */
double lastreset_;  /* W.N. Last time connection was reset - for */
                    /* detecting pkts from previous incarnations */
int slow_start_restart_; /* boolean: re-init cwnd after connection
                         goes idle.  On by default. */
int restart_bugfix_;    /* ssthresh is cut down because of
                       timeouts during a connection's idle period.
                       Setting this boolean fixes this problem.
                       For now, it is off by default. */
int closed_;            /* whether this connection has closed */
 TracedInt ndatapack_;   /* number of data packets sent */
 TracedInt ndatabytes_;  /* number of data bytes sent */
 TracedInt nackpack_;    /* number of ack packets received */
 TracedInt nrexmit_;     /* number of retransmit timeouts
                       when there was data outstanding */
 TracedInt nrexmitpack_; /* number of retransmited packets */
 TracedInt nrexmitbytes_; /* number of retransmited bytes */
 TracedInt necnresponses_; /* number of times cwnd was reduced
                       in response to an ecn packet -- sylvia */
 TracedInt ncwndcuts_;     /* number of times cwnd was reduced
                       for any reason -- sylvia */
int trace_all_oneline_;    /* TCP tracing vars all in one line or not? */
int nam_tracevar_;      /* Output nam's variable trace or just plain
                       text variable trace? */
int first_decrease_;       /* First decrease of congestion window.  */
                    /* Used for decrease_num_ != 0.5. */
 TracedInt singledup_;   /* Send on a single dup ack.  */
int noFastRetrans_; /* No Fast Retransmit option.  */
```

```
        int oldCode_;       /* Use old code. */
        int useHeaders_;    /* boolean: Add TCP/IP header sizes */

        int timeout_count_ ;       /* count of num timeouts */

        /* for experimental high-speed TCP */
        /* These four parameters define the HighSpeed response function. */
        int low_window_;    /* window for turning on high-speed TCP */
        int high_window_;   /* target window for new response function */
        double high_p_;           /* target drop rate for new response function */
        double high_decrease_;    /* decrease rate at target window */
        /* The next parameter is for Limited Slow-Start. */
        int max_ssthresh_;  /* max value for ssthresh_ */

        // LTCP Variables
        int ltcp_num_layer_;  // layer number
        double ltcp_frac_layer_; // fractional layer
        double ltcp_min_rtt_;      // minimum RTT seen so far
        double ltcp_est_rtt_;      // current estimate of RTT
        double ltcp_rtt_fact_;     // RTT compensation Factor for ltcp
        double ltcp_rtt_comp_fact_const_; // value of constant for RTT compensation
factor

        int ltcp_win_thresh_ ; // Thresh at which layer 2 is added
         double ltcp_alpha_ ; // stepsize_k = ltcp_alpha_ * stepsize_k-1
         double ltcp_beta_ ; // design param for loss in util
        double ltcp_win_[100];   // win for 100 layers - enough for upto 10Gbps
        void init_ltcp_vars(); // initialisation

        /*********************
        STCP_ADDITIONS BEGINS
        **********************/

        //Stratified  TCP Variables
        int stcp_num_layer_;  // STCP layer number
        double stcp_frac_layer_; //STCP fractional layer
        double stcp_min_rtt_;      // STCPminimum RTT seen so far
        double stcp_est_rtt_;      //STCP current estimate of RTT
        double stcp_rtt_fact_;     //STCP RTT compensation Factor for stcp
        double stcp_rtt_comp_fact_const_; //STCP value of constant for RTT compensation
factor
        int stcp_win_thresh_l ; // Thresh at which layer 2 is added
     double stcp_alpha_ ; // stepsize_k = stcp_alpha changed and is different from LTCP
     int stcp_win_thresh_h ;
        double stcp_beta_ ; // design param for loss in util
        double stcp_win_[100];   // win for 100 layers - enough for upto 10Gbps
        void init_stcp_vars(); // initialisation
        double getThresholdAlpha(int currWin);

        /*********************
        STCP_ADDITIONS ENDS
        **********************/



        /* These three functions are just an easy structuring of the code. */
        double increase_param();  /* get increase parameter for current cwnd */
        double decrease_param();  /* get decrease parameter for current cwnd */
        double compute_p(); /* compute p for calculating parameters */

        /* The next three parameters are for CPU overhead, for computing */
        /*   the HighSpeed parameters less frequently.  A better solution */
        /*   might be just to have a look-up array.  */
```

```
      double cwnd_last_;  /* last cwnd for computed parameters */
    double increase_last_; /* increase param for cwnd_last_ */
      double cwnd_frac_;  /* for determining when to recompute params. */
    /* end of section for experimental high-speed TCP */

      /* for Quick-Start, experimental. */
      int rate_request_;  /* Rate request in pps, for QuickStart. */
      int qs_enabled_; /* to enable QuickStart. */
      int qs_requested_;
      int qs_approved_;
      int ttl_diff_;
       /* end of section for Quick-Start. */

       /* support for event-tracing */
       //EventTrace *et_;
       void trace_event(char *eventtype);

      /* these function are now obsolete, see other above */
      void closecwnd(int how);
      void quench(int how);

      void process_qoption_after_send() ;
      void process_qoption_after_ack(int seqno) ;

      int QOption_ ; /* TCP quiescence option */
      int EnblRTTCtr_ ; /* are we using a corase grained timer? */
      int T_full ; /* last time the window was full */
      int T_last ;
      int T_prev ;
      int T_start ;
      int RTT_count ;
      int RTT_prev ;
      int RTT_goodcount ;
      int F_counting ;
      int W_used ;
      int W_timed ;
      int F_full ;
      int Backoffs ;

      int control_increase_ ; /* If true, don't increase cwnd if sender */
                          /*  is not window-limited.  */
      int prev_highest_ack_ ; /* Used to determine if sender is */
                          /*  window-limited.  */
};

/* TCP Reno */
class RenoTcpAgent : public virtual TcpAgent {
 public:
      RenoTcpAgent();
      virtual int window();
      virtual double windowd();
      virtual void recv(Packet *pkt, Handler*);
      virtual void timeout(int tno);
      virtual void dupack_action();
 protected:
      int allow_fast_retransmit(int last_cwnd_action_);
      unsigned int dupwnd_;
};

/* TCP New Reno */
class NewRenoTcpAgent : public virtual RenoTcpAgent {
 public:
      NewRenoTcpAgent();
```

```
	virtual void recv(Packet *pkt, Handler*);
	virtual void partialnewack_helper(Packet* pkt);
	virtual void dupack_action();
 protected:
	int newreno_changes_;	/* 0 for fixing unnecessary fast retransmits */
				/* 1 for additional code from Allman, */
				/* to implement other algorithms from */
				/* Hoe's paper, including sending a new */
				/* packet for every two duplicate ACKs. */
				/* The default is set to 0. */
	int newreno_changes1_;  /* Newreno_changes1_ set to 0 gives the */
				/* Slow-but-Steady variant of NewReno from */
				/* RFC 2582, with the retransmit timer reset */
				/* after each partial new ack. */
				/* Newreno_changes1_ set to 1 gives the */
				/* Impatient variant of NewReno from */
				/* RFC 2582, with the retransmit timer reset */
				/* only for the first partial new ack. */
				/* The default is set to 0 */
	void partialnewack(Packet *pkt);
	int allow_fast_retransmit(int last_cwnd_action_);
	int acked_, new_ssthresh_;  /* used if newreno_changes_ == 1 */
	double ack2_, ack3_, basertt_; /* used if newreno_changes_ == 1 */
	int firstpartial_;  /* For the first partial ACK. */
	int partial_window_deflation_; /* 0 if set cwnd to ssthresh upon */
				/* partial new ack (default) */
				/* 1 if deflate (cwnd + dupwnd) by */
				/* amount of data acked */
				/* "Partial window deflation" is */
				/* discussed in RFC 2582. */
	int exit_recovery_fix_;	 /* 0 for setting cwnd to ssthresh upon */
				/* leaving fast recovery (default) */
				/* 1 for setting cwnd to min(ssthresh, */
				/* amnt. of data in network) when leaving */
};

/* TCP vegas (VegasTcpAgent) */
class VegasTcpAgent : public virtual TcpAgent {
 public:
	VegasTcpAgent();
	~VegasTcpAgent();
	virtual void recv(Packet *pkt, Handler *);
	virtual void timeout(int tno);
protected:
	double vegastime() {
		return(Scheduler::instance().clock() - firstsent_);
	}
	virtual void output(int seqno, int reason = 0);
	virtual void recv_newack_helper(Packet*);
	int vegas_expire(Packet*);
	void reset();
	void vegas_inflate_cwnd(int win, double current_time);

	virtual void delay_bind_init_all();
	virtual int delay_bind_dispatch(const char *varName, const char *localName,
TclObject *tracer);

	double t_cwnd_changed_; // last time cwnd changed
	double firstrecv_;  // time recv the 1st ack

	int    v_alpha_;          // vegas thruput thresholds in pkts
	int    v_beta_;
```

```
        int    v_gamma_;           // threshold to change from slow-start to
                              // congestion avoidance, in pkts

        int    v_slowstart_;    // # of pkts to send after slow-start, deflt(2)
        int    v_worried_;      // # of pkts to chk after dup ack (1 or 2)

        double v_timeout_;      // based on fine-grained timer
        double v_rtt_;
        double v_sa_;
        double v_sd_;

        int    v_cntRTT_;       // # of rtt measured within one rtt
        double v_sumRTT_;       // sum of rtt measured within one rtt

        double v_begtime_;  // tagged pkt sent
        int    v_begseq_;   // tagged pkt seqno

        double* v_sendtime_;       // each unacked pkt's sendtime is recorded.
        int*   v_transmits_;       // # of retx for an unacked pkt

        int    v_maxwnd_;   // maxwnd size for v_sendtime_[]
        double v_newcwnd_;  // record un-inflated cwnd

        double v_baseRTT_;  // min of all rtt

        double v_incr_;           // amount cwnd is increased in the next rtt
        int    v_inc_flag_; // if cwnd is allowed to incr for this rtt

        double v_actual_;   // actual send rate (pkt/s; needed for tcp-rbp)

        int ns_vegas_fix_level_;   // see comment at end of tcp-vegas.cc for details of
fixes
};

// Local Variables:
// mode:c++
// End:

#endif
```

7.1.2 The C++ File which Implements The STCP Algorithm

```
/* -*- Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t -*- */
/*
 * Copyright (c) 1991-1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *     This product includes software developed by the Computer Systems
 *     Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be used
 *    to endorse or promote products derived from this software without
```

```
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */



/**
CODE MODIFIFED BY: RANJITHA SHIVARUDRAIAH
OBJECTIVE:          TO IMPLEMENT STRATIFIED TCP( STCP)
COURSE:             MS THESIS
                     (A STUDY OF PARALLEL TRANSPORT TECHNIQUES
                        FOR HIGH-SPEED NETWORKS)
CODE DESCRIPTION: "tcp.cc" contains the implementation for all the TCP variants that
have been tested/implemented by sfloyd and group. We have made additions/modifications
to this file  in order to implement the algorithm for Stratified TCP( STCP).Each TCP
variant is implemented in a "switch" case statement. The case number used for STCP is
"100". The case numbers are also known as the window option variables which can be
added in the TCL script in order to execute and test a particular
protocol in this code.
The modifications are recognized by the tag "STCP_ADDITIONS".
*/

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/tcp/tcp.cc,v 1.144 2003/02/12 04:16:09
sfloyd Exp $ (LBL)";
#endif

#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include "ip.h"
#include "tcp.h"
#include "flags.h"
#include "random.h"
#include "basetrace.h"
#include "hdr_qs.h"

int hdr_tcp::offset_;
FILE *RTT_file;
FILE *ALPHA_file;
static class TCPHeaderClass : public PacketHeaderClass {
public:
        TCPHeaderClass() : PacketHeaderClass("PacketHeader/TCP",
                                     sizeof(hdr_tcp)) {
             bind_offset(&hdr_tcp::offset_);
       }
} class_tcphdr;

static class TcpClass : public TclClass {
public:
       TcpClass() : TclClass("Agent/TCP") {}
```

```
	TclObject* create(int , const char*const*) {
		return (new TcpAgent());
	}
} class_tcp;

TcpAgent::TcpAgent() : Agent(PT_TCP),
	t_seqno_(0), t_rtt_(0), t_srtt_(0), t_rttvar_(0),
	t_backoff_(0), ts_peer_(0),
	rtx_timer_(this), delsnd_timer_(this),
	burstsnd_timer_(this),
	dupacks_(0), curseq_(0), highest_ack_(0), cwnd_(0), ssthresh_(0),
	count_(0), fcnt_(0), rtt_active_(0), rtt_seq_(-1), rtt_ts_(0.0),
	maxseq_(0), cong_action_(0), ecn_burst_(0), ecn_backoff_(0),
	  ect_(0), lastreset_(0.0),
	  restart_bugfix_(1), closed_(0), nrexmit_(0),
	first_decrease_(1), qs_requested_(0), qs_approved_(0),
	ltcp_win_thresh_(50), ltcp_beta_(0.15), ltcp_rtt_comp_fact_const_(0.5),
	/*********************
	STCP_ADDITIONS BEGINS
	**********************/
	stcp_win_thresh_l(50),stcp_win_thresh_h(83000), stcp_beta_(0.10),
stcp_rtt_comp_fact_const_(0.5)
	/*********************
	STCP_ADDITIONS ENDS
	**********************/
{
#ifdef TCP_DELAY_BIND_ALL
#else /* ! TCP_DELAY_BIND_ALL */
	// not delay-bound because delay-bound tracevars aren't yet supported
	bind("t_seqno_", &t_seqno_);
	bind("rtt_", &t_rtt_);
	bind("srtt_", &t_srtt_);
	bind("rttvar_", &t_rttvar_);
	bind("backoff_", &t_backoff_);
	bind("dupacks_", &dupacks_);
	bind("seqno_", &curseq_);
	bind("ack_", &highest_ack_);
	bind("cwnd_", &cwnd_);
	bind("ssthresh_", &ssthresh_);
	bind("maxseq_", &maxseq_);
	  bind("ndatapack_", &ndatapack_);
	  bind("ndatabytes_", &ndatabytes_);
	  bind("nackpack_", &nackpack_);
	  bind("nrexmit_", &nrexmit_);
	  bind("nrexmitpack_", &nrexmitpack_);
	  bind("nrexmitbytes_", &nrexmitbytes_);
	  bind("necnresponses_", &necnresponses_);
	  bind("ncwndcuts_", &ncwndcuts_);
#endif /* TCP_DELAY_BIND_ALL */

}

void TcpAgent::init_ltcp_vars()
{
	// Initialization of LTCP variables
	ltcp_num_layer_ = 1;
	ltcp_frac_layer_ = 0;
	ltcp_win_[0] = ltcp_win_[1] = 0;
	ltcp_win_[2] = ltcp_win_thresh_;
	for(int i=3; i < 100; i++)
	{
		ltcp_alpha_ = (double)(i+1)/(double)(i-2);
		ltcp_win_[i] = (double) ltcp_alpha_ * ltcp_win_[i-1];
```

```
        }

       ltcp_min_rtt_ = (int(t_srtt_) >> T_SRTT_BITS)*tcp_tick_*1000;
       ltcp_rtt_fact_ = (double)(ltcp_rtt_comp_fact_const_*(pow(ltcp_min_rtt_,
0.333)));
       ltcp_rtt_fact_ = (ltcp_rtt_fact_ > 1) ? ltcp_rtt_fact_ : 1;
       //printf("LTCP variables: winthresh: %d, beta: %f, num_layer: %d, frac_layer:
%f, rtt_fact_:%f\n",
       //     ltcp_win_thresh_, ltcp_beta_, ltcp_num_layer_, ltcp_frac_layer_,
ltcp_rtt_fact_);
}
/*********************
STCP_ADDITIONS BEGINS
*********************/

/*Code description : This is the initialization function which will be executed
*                                  before the congestion control algorithm starts.
*
*
*/
void TcpAgent::init_stcp_vars()
    {
          // Initialization of STCP variables
          stcp_num_layer_ = 1;
          stcp_frac_layer_ = 0;
          stcp_win_[0] = stcp_win_[1] = 0;
          stcp_win_[2] = stcp_win_thresh_l;
               for(int i=3; i < 100; i++){
              stcp_alpha_ = (double)(i+1)/(double)(i-2);
              stcp_win_[i] = (double) stcp_alpha_ * stcp_win_[i-1];
          }
          stcp_min_rtt_ = (int(t_srtt_) >> T_SRTT_BITS)*tcp_tick_*1000;
          stcp_rtt_fact_ = (double)(stcp_rtt_comp_fact_const_*(pow(stcp_min_rtt_,
0.333)));
          stcp_rtt_fact_ = (stcp_rtt_fact_ > 1) ? stcp_rtt_fact_ : 1;
          printf("Stratified LTCP variables: winthresh: %d, beta: %f, num_layer: %d,
frac_layer: %f, rtt_fact_:%f\n",
          stcp_win_thresh_l, stcp_beta_, stcp_num_layer_, stcp_frac_layer_,
stcp_rtt_fact_);
}

/*********************
STCP_ADDITIONS ENDS
*********************/
void
TcpAgent::delay_bind_init_all()
{

        // Defaults for bound variables should be set in ns-default.tcl.
        delay_bind_init_one("window_");
        delay_bind_init_one("windowInit_");
        delay_bind_init_one("windowInitOption_");

        delay_bind_init_one("syn_");
        delay_bind_init_one("windowOption_");
        delay_bind_init_one("windowConstant_");
        delay_bind_init_one("windowThresh_");
        delay_bind_init_one("delay_growth_");
        delay_bind_init_one("overhead_");
        delay_bind_init_one("tcpTick_");
        delay_bind_init_one("ecn_");
        delay_bind_init_one("old_ecn_");
        delay_bind_init_one("eln_");
```

```
        delay_bind_init_one("eln_rxmit_thresh_");
        delay_bind_init_one("packetSize_");
        delay_bind_init_one("tcpip_base_hdr_size_");
      delay_bind_init_one("ts_option_size_");
        delay_bind_init_one("bugFix_");
      delay_bind_init_one("lessCareful_");
        delay_bind_init_one("slow_start_restart_");
        delay_bind_init_one("restart_bugfix_");
        delay_bind_init_one("timestamps_");
        delay_bind_init_one("maxburst_");
        delay_bind_init_one("maxcwnd_");
      delay_bind_init_one("numdupacks_");
      delay_bind_init_one("numdupacksFrac_");
        delay_bind_init_one("maxrto_");
      delay_bind_init_one("minrto_");
        delay_bind_init_one("srtt_init_");
        delay_bind_init_one("rttvar_init_");
        delay_bind_init_one("rtxcur_init_");
        delay_bind_init_one("T_SRTT_BITS");
        delay_bind_init_one("T_RTTVAR_BITS");
        delay_bind_init_one("rttvar_exp_");
        delay_bind_init_one("awnd_");
        delay_bind_init_one("decrease_num_");
        delay_bind_init_one("increase_num_");
      delay_bind_init_one("k_parameter_");
      delay_bind_init_one("l_parameter_");
        delay_bind_init_one("trace_all_oneline_");
        delay_bind_init_one("nam_tracevar_");

        delay_bind_init_one("QOption_");
        delay_bind_init_one("EnblRTTCtr_");
        delay_bind_init_one("control_increase_");
      delay_bind_init_one("noFastRetrans_");
      delay_bind_init_one("precisionReduce_");
      delay_bind_init_one("oldCode_");
      delay_bind_init_one("useHeaders_");
      delay_bind_init_one("low_window_");
      delay_bind_init_one("high_window_");
      delay_bind_init_one("high_p_");
      delay_bind_init_one("high_decrease_");
      delay_bind_init_one("max_ssthresh_");
      delay_bind_init_one("cwnd_frac_");
      delay_bind_init_one("timerfix_");
      delay_bind_init_one("rfc2988_");
      delay_bind_init_one("singledup_");
      delay_bind_init_one("rate_request_");
      delay_bind_init_one("qs_enabled_");

#ifdef TCP_DELAY_BIND_ALL
      // out because delay-bound tracevars aren't yet supported
        delay_bind_init_one("t_seqno_");
        delay_bind_init_one("rtt_");
        delay_bind_init_one("srtt_");
        delay_bind_init_one("rttvar_");
        delay_bind_init_one("backoff_");
        delay_bind_init_one("dupacks_");
        delay_bind_init_one("seqno_");
        delay_bind_init_one("ack_");
        delay_bind_init_one("cwnd_");
        delay_bind_init_one("ssthresh_");
        delay_bind_init_one("maxseq_");
        delay_bind_init_one("ndatapack_");
        delay_bind_init_one("ndatabytes_");
```

```
        delay_bind_init_one("nackpack_");
        delay_bind_init_one("nrexmit_");
        delay_bind_init_one("nrexmitpack_");
        delay_bind_init_one("nrexmitbytes_");
        delay_bind_init_one("necnresponses_");
        delay_bind_init_one("ncwndcuts_");
#endif /* TCP_DELAY_BIND_ALL */

       Agent::delay_bind_init_all();

        reset();
}

int
TcpAgent::delay_bind_dispatch(const char *varName, const char *localName, TclObject
*tracer)
{
        if (delay_bind(varName, localName, "window_", &wnd_, tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "windowInit_", &wnd_init_, tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "windowInitOption_", &wnd_init_option_,
tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "syn_", &syn_, tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "windowOption_", &wnd_option_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "windowConstant_",  &wnd_const_, tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "windowThresh_", &wnd_th_ , tracer)) return
TCL_OK;
        if (delay_bind_bool(varName, localName, "delay_growth_", &delay_growth_ ,
tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "overhead_", &overhead_, tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "tcpTick_", &tcp_tick_, tracer)) return
TCL_OK;
        if (delay_bind_bool(varName, localName, "ecn_", &ecn_, tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "old_ecn_", &old_ecn_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "eln_", &eln_ , tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "eln_rxmit_thresh_", &eln_rxmit_thresh_ ,
tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "packetSize_", &size_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "tcpip_base_hdr_size_",
&tcpip_base_hdr_size_, tracer)) return TCL_OK;
       if (delay_bind(varName, localName, "ts_option_size_", &ts_option_size_,
tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "bugFix_", &bug_fix_ , tracer)) return
TCL_OK;
        if (delay_bind_bool(varName, localName, "lessCareful_", &less_careful_ ,
tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "timestamps_", &ts_option_ , tracer))
return TCL_OK;
        if (delay_bind_bool(varName, localName, "slow_start_restart_",
&slow_start_restart_ , tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "restart_bugfix_", &restart_bugfix_ ,
tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "maxburst_", &maxburst_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "maxcwnd_", &maxcwnd_ , tracer)) return
TCL_OK;
       if (delay_bind(varName, localName, "numdupacks_", &numdupacks_, tracer)) return
TCL_OK;
```

```
       if (delay_bind(varName, localName, "numdupacksFrac_", &numdupacksFrac_,
tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "maxrto_", &maxrto_ , tracer)) return
TCL_OK;
       if (delay_bind(varName, localName, "minrto_", &minrto_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "srtt_init_", &srtt_init_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "rttvar_init_", &rttvar_init_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "rtxcur_init_", &rtxcur_init_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "T_SRTT_BITS", &T_SRTT_BITS , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "T_RTTVAR_BITS", &T_RTTVAR_BITS , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "rttvar_exp_", &rttvar_exp_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "awnd_", &awnd_ , tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "decrease_num_", &decrease_num_, tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "increase_num_", &increase_num_, tracer))
return TCL_OK;
       if (delay_bind(varName, localName, "k_parameter_", &k_parameter_, tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "l_parameter_", &l_parameter_, tracer))
return TCL_OK;


        if (delay_bind_bool(varName, localName, "trace_all_oneline_",
&trace_all_oneline_ , tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "nam_tracevar_", &nam_tracevar_ ,
tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "QOption_", &QOption_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "EnblRTTCtr_", &EnblRTTCtr_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "control_increase_", &control_increase_ ,
tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "noFastRetrans_", &noFastRetrans_,
tracer)) return TCL_OK;
        if (delay_bind_bool(varName, localName, "precisionReduce_",
&precision_reduce_, tracer)) return TCL_OK;
       if (delay_bind_bool(varName, localName, "oldCode_", &oldCode_, tracer)) return
TCL_OK;
       if (delay_bind_bool(varName, localName, "useHeaders_", &useHeaders_, tracer))
return TCL_OK;
       if (delay_bind(varName, localName, "low_window_", &low_window_, tracer)) return
TCL_OK;
       if (delay_bind(varName, localName, "high_window_", &high_window_, tracer))
return TCL_OK;
       if (delay_bind(varName, localName, "high_p_", &high_p_, tracer)) return TCL_OK;
       if (delay_bind(varName, localName, "high_decrease_", &high_decrease_, tracer))
return TCL_OK;
       if (delay_bind(varName, localName, "max_ssthresh_", &max_ssthresh_, tracer))
return TCL_OK;
       if (delay_bind(varName, localName, "cwnd_frac_", &cwnd_frac_, tracer)) return
TCL_OK;
       if (delay_bind_bool(varName, localName, "timerfix_", &timerfix_, tracer))
return TCL_OK;
       if (delay_bind_bool(varName, localName, "rfc2988_", &rfc2988_, tracer)) return
TCL_OK;
```

```
        if (delay_bind(varName, localName, "singledup_", &singledup_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "rate_request_", &rate_request_ , tracer))
return TCL_OK;
        if (delay_bind_bool(varName, localName, "qs_enabled_", &qs_enabled_, tracer))
return TCL_OK;

#ifdef TCP_DELAY_BIND_ALL
       // not if (delay-bound delay-bound tracevars aren't yet supported
        if (delay_bind(varName, localName, "t_seqno_", &t_seqno_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "rtt_", &t_rtt_ , tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "srtt_", &t_srtt_ , tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "rttvar_", &t_rttvar_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "backoff_", &t_backoff_ , tracer)) return
TCL_OK;

        if (delay_bind(varName, localName, "dupacks_", &dupacks_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "seqno_", &curseq_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "ack_", &highest_ack_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "cwnd_", &cwnd_ , tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "ssthresh_", &ssthresh_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "maxseq_", &maxseq_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "ndatapack_", &ndatapack_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "ndatabytes_", &ndatabytes_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "nackpack_", &nackpack_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "nrexmit_", &nrexmit_ , tracer)) return
TCL_OK;
        if (delay_bind(varName, localName, "nrexmitpack_", &nrexmitpack_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "nrexmitbytes_", &nrexmitbytes_ , tracer))
return TCL_OK;
        if (delay_bind(varName, localName, "necnresponses_", &necnresponses_ ,
tracer)) return TCL_OK;
        if (delay_bind(varName, localName, "ncwndcuts_", &ncwndcuts_ , tracer)) return
TCL_OK;
#endif

        return Agent::delay_bind_dispatch(varName, localName, tracer);
}

/* Print out all the traced variables whenever any one is changed */
void
TcpAgent::traceAll() {
       double curtime;
       Scheduler& s = Scheduler::instance();
       char wrk[500];
       int n;

       curtime = &s ? s.clock() : 0;
       sprintf(wrk,"time: %-8.5f saddr: %-2d sport: %-2d daddr: %-2d dport:"
              " %-2d maxseq: %-4d hiack: %-4d seqno: %-4d cwnd: %-6.3f"
              " ssthresh: %-3d dupacks: %-2d rtt: %-6.3f srtt: %-6.3f"
              " rttvar: %-6.3f bkoff: %-d", curtime, addr(), port(),
```

```
                daddr(), dport(), int(maxseq_), int(highest_ack_),
                int(t_seqno_), double(cwnd_), int(ssthresh_),
                int(dupacks_), int(t_rtt_)*tcp_tick_,
                (int(t_srtt_) >> T_SRTT_BITS)*tcp_tick_,
                int(t_rttvar_)*tcp_tick_/4.0, int(t_backoff_));
        n = strlen(wrk);
        wrk[n] = '\n';
        wrk[n+1] = 0;
        if (channel_)
                (void)Tcl_Write(channel_, wrk, n+1);
        wrk[n] = 0;
        return;
}

/* Print out just the variable that is modified */
void
TcpAgent::traceVar(TracedVar* v)
{
        double curtime;
        Scheduler& s = Scheduler::instance();
        char wrk[500];
        int n;

        curtime = &s ? s.clock() : 0;
        if (!strcmp(v->name(), "cwnd_") || !strcmp(v->name(), "maxrto_"))
                sprintf(wrk,"%-8.5f %-2d %-2d %-2d %-2d %s %-6.3f",
                        curtime, addr(), port(), daddr(), dport(),
                        v->name(), double(*((TracedDouble*) v)));
        else if (!strcmp(v->name(), "minrto_"))
                sprintf(wrk,"%-8.5f %-2d %-2d %-2d %-2d %s %-6.3f",
                        curtime, addr(), port(), daddr(), dport(),
                        v->name(), double(*((TracedDouble*) v)));
        else if (!strcmp(v->name(), "rtt_"))
                sprintf(wrk,"%-8.5f %-2d %-2d %-2d %-2d %s %-6.3f",
                        curtime, addr(), port(), daddr(), dport(),
                        v->name(), int(*((TracedInt*) v))*tcp_tick_);
        else if (!strcmp(v->name(), "srtt_"))
                sprintf(wrk,"%-8.5f %-2d %-2d %-2d %-2d %s %-6.3f",
                        curtime, addr(), port(), daddr(), dport(),
                        v->name(),
                        (int(*((TracedInt*) v)) >> T_SRTT_BITS)*tcp_tick_);
        else if (!strcmp(v->name(), "rttvar_"))
                sprintf(wrk,"%-8.5f %-2d %-2d %-2d %-2d %s %-6.3f",
                        curtime, addr(), port(), daddr(), dport(),
                        v->name(),
                        int(*((TracedInt*) v))*tcp_tick_/4.0);
        else
                sprintf(wrk,"%-8.5f %-2d %-2d %-2d %-2d %s %d",
                        curtime, addr(), port(), daddr(), dport(),
                        v->name(), int(*((TracedInt*) v)));
        n = strlen(wrk);
        wrk[n] = '\n';
        wrk[n+1] = 0;
        if (channel_)
                (void)Tcl_Write(channel_, wrk, n+1);
        wrk[n] = 0;
        return;
}

void
TcpAgent::trace(TracedVar* v)
{
        if (nam_tracevar_) {
```

```
		Agent::trace(v);
	} else if (trace_all_oneline_)
		traceAll();
	else
		traceVar(v);
}

//
// in 1-way TCP, syn_ indicates we are modeling
// a SYN exchange at the beginning.  If this is true
// and we are delaying growth, then use an initial
// window of one.  If not, we do whatever initial_window()
// says to do.
//

void
TcpAgent::set_initial_window()
{
	if (syn_ && delay_growth_)
		cwnd_ = 1.0;
	else
		cwnd_ = initial_window();
}

void
TcpAgent::reset_qoption()
{
	int now = (int)(Scheduler::instance().clock()/tcp_tick_ + 0.5);

	T_start = now ;
	RTT_count = 0 ;
	RTT_prev = 0 ;
	RTT_goodcount = 1 ;
	F_counting = 0 ;
	W_timed = -1 ;
	F_full = 0 ;
	Backoffs = 0 ;
}

void
TcpAgent::reset()
{
	rtt_init();
	rtt_seq_ = -1;
	/*XXX lookup variables */
	dupacks_ = 0;
	curseq_ = 0;
	set_initial_window();

	t_seqno_ = 0;
	maxseq_ = -1;
	last_ack_ = -1;
	highest_ack_ = -1;

	// Initialize STCP variables
	if (wnd_option_ == 100)
		init_stcp_vars();

	ssthresh_ = int(wnd_);

	if (max_ssthresh_ > 0 && max_ssthresh_ < ssthresh_)
		ssthresh_ = max_ssthresh_;
	wnd_restart_ = 1.;
```

```
	awnd_ = wnd_init_ / 2.0;
	recover_ = 0;
	closed_ = 0;
	last_cwnd_action_ = 0;
	boot_time_ = Random::uniform(tcp_tick_);
	first_decrease_ = 1;
	/* W.N.: for removing packets from previous incarnations */
	lastreset_ = Scheduler::instance().clock();

	/* Now these variables will be reset
	   - Debojyoti Dutta 12th Oct'2000 */

	ndatapack_ = 0;
	ndatabytes_ = 0;
	nackpack_ = 0;
	nrexmitbytes_ = 0;
	nrexmit_ = 0;
	nrexmitpack_ = 0;
	necnresponses_ = 0;
	ncwndcuts_ = 0;

	 cwnd_last_ = 0.0;

	if (control_increase_) {
		prev_highest_ack_ = highest_ack_ ;
	}

	if (QOption_) {
		int now = (int)(Scheduler::instance().clock()/tcp_tick_ + 0.5);
		T_last = now ;
		T_prev = now ;
		W_used = 0 ;
		if (EnblRTTCtr_) {
			reset_qoption();
		}
	}
}

/*
 * Initialize variables for the retransmit timer.
 */
void TcpAgent::rtt_init()
{
	t_rtt_ = 0;
	t_srtt_ = int(srtt_init_ / tcp_tick_) << T_SRTT_BITS;
	t_rttvar_ = int(rttvar_init_ / tcp_tick_) << T_RTTVAR_BITS;
	t_rtxcur_ = rtxcur_init_;
	t_backoff_ = 1;
}

double TcpAgent::rtt_timeout()
{
	double timeout;
	if (rfc2988_) {
	// Correction from Tom Kelly to be RFC2988-compliant, by
	// clamping minrto_ before applying t_backoff_.
		if (t_rtxcur_ < minrto_)
			timeout = minrto_ * t_backoff_;
		else
			timeout = t_rtxcur_ * t_backoff_;
	} else {
		timeout = t_rtxcur_ * t_backoff_;
		if (timeout < minrto_)
```

```
                timeout = minrto_;
        }

        if (timeout > maxrto_)
                timeout = maxrto_;

         if (timeout < 2.0 * tcp_tick_) {
                if (timeout < 0) {
                        fprintf(stderr, "TcpAgent: negative RTO!  (%f)\n",
                                timeout);
                        exit(1);
                }
                timeout = 2.0 * tcp_tick_;
        }
        return (timeout);
}


/* This has been modified to use the tahoe code. */
void TcpAgent::rtt_update(double tao)
{
        double now = Scheduler::instance().clock();
        if (ts_option_)
                t_rtt_ = int(tao /tcp_tick_ + 0.5);
        else {
                double sendtime = now - tao;
                sendtime += boot_time_;
                double tickoff = fmod(sendtime, tcp_tick_);
                t_rtt_ = int((tao + tickoff) / tcp_tick_);
        }
        if (t_rtt_ < 1)
                t_rtt_ = 1;

        //
        // srtt has 3 bits to the right of the binary point
        // rttvar has 2
        //
         if (t_srtt_ != 0) {
                register short delta;
                delta = t_rtt_ - (t_srtt_ >> T_SRTT_BITS);     // d = (m - a0)
                if ((t_srtt_ += delta) <= 0)      // a1 = 7/8 a0 + 1/8 m
                        t_srtt_ = 1;
                if (delta < 0)
                        delta = -delta;
                delta -= (t_rttvar_ >> T_RTTVAR_BITS);
                if ((t_rttvar_ += delta) <= 0)    // var1 = 3/4 var0 + 1/4 |d|
                        t_rttvar_ = 1;
        } else {
                t_srtt_ = t_rtt_ << T_SRTT_BITS;         // srtt = rtt
                t_rttvar_ = t_rtt_ << (T_RTTVAR_BITS-1);       // rttvar = rtt / 2
        }
        //
        // Current retransmit value is
        //    (unscaled) smoothed round trip estimate
        //    plus 2^rttvar_exp_ times (unscaled) rttvar.
        //
        t_rtxcur_ = (((t_rttvar_ << (rttvar_exp_ + (T_SRTT_BITS - T_RTTVAR_BITS))) +
                t_srtt_)  >> T_SRTT_BITS ) * tcp_tick_;

        return;
}

void TcpAgent::rtt_backoff()
```

```
{
	if (t_backoff_ < 64)
		t_backoff_ <<= 1;

	if (t_backoff_ > 8) {
		/*
		 * If backed off this far, clobber the srtt
		 * value, storing it in the mean deviation
		 * instead.
		 */
		t_rttvar_ += (t_srtt_ >> T_SRTT_BITS);
		t_srtt_ = 0;
	}
}

/*
 * headersize:
 *      how big is an IP+TCP header in bytes; include options such as ts
 * this function should be virtual so others (e.g. SACK) can override
 */
int TcpAgent::headersize()
{
	 int total = tcpip_base_hdr_size_;
	if (total < 1) {
		fprintf(stderr,
		  "TcpAgent(%s): warning: tcpip hdr size is only %d bytes\n",
		  name(), tcpip_base_hdr_size_);
	}
	if (ts_option_)
		total += ts_option_size_;
	 return (total);
}

void TcpAgent::output(int seqno, int reason)
{
	int force_set_rtx_timer = 0;
	Packet* p = allocpkt();
	hdr_tcp *tcph = hdr_tcp::access(p);
	hdr_flags* hf = hdr_flags::access(p);
	hdr_ip *iph = hdr_ip::access(p);
	int databytes = hdr_cmn::access(p)->size();
	tcph->seqno() = seqno;
	tcph->ts() = Scheduler::instance().clock();
	tcph->ts_echo() = ts_peer_;
	tcph->reason() = reason;
	tcph->last_rtt() = int(int(t_rtt_)*tcp_tick_*1000);

	if (ecn_) {
		hf->ect() = 1;      // ECN-capable transport
	}
	if (cong_action_) {
		hf->cong_action() = TRUE;  // Congestion action.
		cong_action_ = FALSE;
	 }
	/* Check if this is the initial SYN packet. */
	if (seqno == 0) {
		if (syn_) {
			databytes = 0;
			curseq_ += 1;
			hdr_cmn::access(p)->size() = tcpip_base_hdr_size_;
		}
		if (ecn_) {
			hf->ecnecho() = 1;
```

```
//                  hf->cong_action() = 1;
                    hf->ect() = 0;
                }
                if (qs_enabled_) {
                        hdr_qs *qsh = hdr_qs::access(p);
                        if (rate_request_ > 0) {
                                // QuickStart code from Srikanth Sundarrajan.
                                qsh->flag() = QS_REQUEST;
                                Random::seed_heuristically();
                                qsh->ttl() = Random::integer(256);
                                ttl_diff_ = (iph->ttl() - qsh->ttl()) % 256;
                                qsh->rate() = rate_request_;
                                qs_requested_ = 1;
                        } else {
                                qsh->flag() = QS_DISABLE;
                        }
                }
        }
        else if (useHeaders_ == true) {
                hdr_cmn::access(p)->size() += headersize();
        }
         hdr_cmn::access(p)->size();

        /* if no outstanding data, be sure to set rtx timer again */
        if (highest_ack_ == maxseq_)
                force_set_rtx_timer = 1;
        /* call helper function to fill in additional fields */
        output_helper(p);

         ++ndatapack_;
         ndatabytes_ += databytes;
        send(p, 0);
        if (seqno == curseq_ && seqno > maxseq_)
                idle();  // Tell application I have sent everything so far
        if (seqno > maxseq_) {
                maxseq_ = seqno;
                if (!rtt_active_) {
                        rtt_active_ = 1;
                        if (seqno > rtt_seq_) {
                                rtt_seq_ = seqno;
                                rtt_ts_ = Scheduler::instance().clock();
                        }

                }
        } else {
                ++nrexmitpack_;
                nrexmitbytes_ += databytes;
        }
        if (!(rtx_timer_.status() == TIMER_PENDING) || force_set_rtx_timer)
                /* No timer pending.  Schedule one. */
                set_rtx_timer();
}

/*
 * Must convert bytes into packets for one-way TCPs.
 * If nbytes == -1, this corresponds to infinite send.  We approximate
 * infinite by a very large number (TCP_MAXSEQ).
 */
void TcpAgent::sendmsg(int nbytes, const char* /*flags*/)
{
        if (nbytes == -1 && curseq_ <= TCP_MAXSEQ)
                curseq_ = TCP_MAXSEQ;
        else
```

```
                curseq_ += (nbytes/size_ + (nbytes%size_ ? 1 : 0));
        send_much(0, 0, maxburst_);
}

void TcpAgent::advanceby(int delta)
{
  curseq_ += delta;
        if (delta > 0)
                closed_ = 0;
        send_much(0, 0, maxburst_);
}


int TcpAgent::command(int argc, const char*const* argv)
{
        if (argc == 3) {
                if (strcmp(argv[1], "advance") == 0) {
                        int newseq = atoi(argv[2]);
                        if (newseq > maxseq_)
                                advanceby(newseq - curseq_);
                        else
                                advanceby(maxseq_ - curseq_);
                        return (TCL_OK);
                }
                if (strcmp(argv[1], "advanceby") == 0) {
                        advanceby(atoi(argv[2]));
                        return (TCL_OK);
                }
                if (strcmp(argv[1], "eventtrace") == 0) {
                        et_ = (EventTrace *)TclObject::lookup(argv[2]);
                        return (TCL_OK);
                }
                /*
                 * Curtis Villamizar's trick to transfer tcp connection
                 * parameters to emulate http persistent connections.
                 *
                 * Another way to do the same thing is to open one tcp
                 * object and use start/stop/maxpkts_ or advanceby to control
                 * how much data is sent in each burst.
                 * With a single connection, slow_start_restart_
                 * should be configured as desired.
                 *
                 * This implementation (persist) may not correctly
                 * emulate pure-BSD-based systems which close cwnd
                 * after the connection goes idle (slow-start
                 * restart).  See appendix C in
                 * Jacobson and Karels ``Congestion
                 * Avoidance and Control'' at
                 * <ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z>
                 * (*not* the original
                 * '88 paper) for why BSD does this.  See
                 * ``Performance Interactions Between P-HTTP and TCP
                 * Implementations'' in CCR 27(2) for descriptions of
                 * what other systems do the same.
                 *
                 */
                if (strcmp(argv[1], "persist") == 0) {
                        TcpAgent *other
                          = (TcpAgent*)TclObject::lookup(argv[2]);
                        cwnd_ = other->cwnd_;
                        awnd_ = other->awnd_;
                        ssthresh_ = other->ssthresh_;
                        t_rtt_ = other->t_rtt_;
```

```
                        t_srtt_ = other->t_srtt_;
                        t_rttvar_ = other->t_rttvar_;
                        t_backoff_ = other->t_backoff_;
                        return (TCL_OK);
                }
        }
        return (Agent::command(argc, argv));
}

int TcpAgent::window()
{
        return (cwnd_ < wnd_ ? (int)cwnd_ : (int)wnd_);
}

double TcpAgent::windowd()
{
        return (cwnd_ < wnd_ ? (double)cwnd_ : (double)wnd_);
}

/*
 * Try to send as much data as the window will allow.  The link layer will
 * do the buffering; we ask the application layer for the size of the packets.
 */
void TcpAgent::send_much(int force, int reason, int maxburst)
{
        send_idle_helper();
        int win = window();
        int npackets = 0;

        if (!force && delsnd_timer_.status() == TIMER_PENDING)
                return;
        /* Save time when first packet was sent, for newreno  --Allman */
        if (t_seqno_ == 0)
                firstsent_ = Scheduler::instance().clock();

        if (burstsnd_timer_.status() == TIMER_PENDING)
                return;
        while (t_seqno_ <= highest_ack_ + win && t_seqno_ < curseq_) {
                if (overhead_ == 0 || force) {
                        output(t_seqno_, reason);
                        npackets++;
                        if (QOption_)
                                process_qoption_after_send () ;
                        t_seqno_ ++ ;
                        if (qs_approved_ == 1) {
                                // delay = effective RTT / window
                                double delay = (double) t_rtt_ * tcp_tick_ / win;
                                delsnd_timer_.resched(delay);
                                return;
                        }
                } else if (!(delsnd_timer_.status() == TIMER_PENDING)) {
                        /*
                         * Set a delayed send timeout.
                         */
                        delsnd_timer_.resched(Random::uniform(overhead_));
                        return;
                }
                win = window();
                if (maxburst && npackets == maxburst)
                        break;
        }
        /* call helper function */
        send_helper(maxburst);
```

```
}

/*
 * We got a timeout or too many duplicate acks.  Clear the retransmit timer.
 * Resume the sequence one past the last packet acked.
 * "mild" is 0 for timeouts and Tahoe dup acks, 1 for Reno dup acks.
 * "backoff" is 1 if the timer should be backed off, 0 otherwise.
 */
void TcpAgent::reset_rtx_timer(int mild, int backoff)
{
	if (backoff)
		rtt_backoff();
	set_rtx_timer();
	if (!mild)
		t_seqno_ = highest_ack_ + 1;
	rtt_active_ = 0;
}

/*
 * Set retransmit timer using current rtt estimate.  By calling resched(),
 * it does not matter whether the timer was already running.
 */
void TcpAgent::set_rtx_timer()
{
	rtx_timer_.resched(rtt_timeout());
}

/*
 * Set new retransmission timer if not all outstanding
 * or available data acked, or if we are unable to send because
 * cwnd is less than one (as when the ECN bit is set when cwnd was 1).
 * Otherwise, if a timer is still outstanding, cancel it.
 */
void TcpAgent::newtimer(Packet* pkt)
{
	hdr_tcp *tcph = hdr_tcp::access(pkt);
	/*
	 * t_seqno_, the next packet to send, is reset (decreased)
	 *   to highest_ack_ + 1 after a timeout,
	 *   so we also have to check maxseq_, the highest seqno sent.
	 * In addition, if the packet sent after the timeout has
	 *   the ECN bit set, then the returning ACK caused cwnd_ to
	 *   be decreased to less than one, and we can't send another
	 *   packet until the retransmit timer again expires.
	 *   So we have to check for "cwnd_ < 1" as well.
	 */
	if (t_seqno_ > tcph->seqno() || tcph->seqno() < maxseq_ || cwnd_ < 1)
		set_rtx_timer();
	else
		cancel_rtx_timer();
}

/*
 * for experimental, high-speed TCP
 */
double TcpAgent::linear(double x, double x_1, double y_1, double x_2, double y_2)
{
	// The y coordinate factor ranges from y_1 to y_2
	//  as the x coordinate ranges from x_1 to x_2.
	double y = y_1 + ((y_2 - y_1) * ((x - x_1)/(x_2-x_1)));
	return y;
}
```

```
/*
 * Limited Slow-Start for large congestion windows.
 * This is only used when max_ssthresh_ is non-zero.
 */
double TcpAgent::limited_slow_start(double cwnd, double max_ssthresh, double
increment)
{
       int round = int(cwnd / (double(max_ssthresh)/2.0));
       double increment1 = 1.0/(double(round));
       if (increment < increment1)
              increment = increment1;
       return increment;
}

/*
 * For retrieving numdupacks_.
 */
int TcpAgent::numdupacks(double cwnd)
{
        int cwndfraction = (int) cwnd/numdupacksFrac_;
       if (numdupacks_ > cwndfraction) {
              return numdupacks_;
        } else {
              return cwndfraction;
       }
}

/*
 * Calculating the packet drop rate p for highspeed TCP.
 */
double TcpAgent::compute_p()
{
       double p;
       double low_p = 1.5/(low_window_*low_window_);
       p = exp(linear(log(cwnd_), log(low_window_), log(low_p), log(high_window_),
log(high_p_)));
       return p;
}

/*
 * Calculating the decrease parameter for highspeed TCP.
 */
double TcpAgent::decrease_param()
{
       double decrease;
       decrease = linear(log(cwnd_), log(low_window_), 0.5, log(high_window_),
high_decrease_);
       return decrease;
}

/*
 * Calculating the increase parameter for highspeed TCP.
 */
double TcpAgent::increase_param()
{
       double increase, decrease, p, answer;
       /* extending the slow-start for high-speed TCP */

       /* for highspeed TCP -- from Sylvia Ratnasamy, */
       /* modifications by Sally Floyd and Evandro de Souza */
       // p ranges from 1.5/W^2 at congestion window low_window_, to
       //    high_p_ at congestion window high_window_, on a log-log scale.
        // The decrease factor ranges from 0.5 to high_decrease
```

```
        //  as the window ranges from low_window to high_window,
        //  as the log of the window.
        // For an efficient implementation, this would just be looked up
        //   in a table, with the increase and decrease being a function of the
        //   congestion window.

        if (cwnd_ <= low_window_) {
                answer = 1 / cwnd_;
                        return answer;
        } else if (cwnd_ >= cwnd_last_ && cwnd_ < cwnd_frac_ * cwnd_last_ ) {
                answer = increase_last_ / cwnd_;
                   return answer;
        } else {
                p = compute_p();
                decrease = decrease_param();
                increase = (cwnd_ * cwnd_ *2.0* decrease * p)/(2.0 - decrease);
                //      double max_increase = 157.8;
                //     if (increase > max_increase) {
                //             increase = max_increase;
                //     }
                answer = increase / cwnd_;
                cwnd_last_ = cwnd_;
                increase_last_ = increase;
                        return answer;
        }
}

/*
 * open up the congestion window
 */
void TcpAgent::opencwnd()
{
        double increment;
        int lim_slowstart_flag = 0; // used by LTCP
        /*********************
        STCP_ADDITIONS BEGINS
        *********************/
        int stcp_lim_slowstart_flag = 0; // used by STCP
        /*********************
        STCP_ADDITIONS ENDS
        *********************/

        if (cwnd_ < ssthresh_) {
                /* slow-start (exponential) */
                cwnd_ += 1;
        } else {
                /* linear */
                double f;
                switch (wnd_option_) {
                case 0:
                        if (++count_ >= cwnd_) {
                                count_ = 0;
                                ++cwnd_;
                        }
                        break;

                case 1:
                        /* This is the standard algorithm. */
                        increment = increase_num_ / cwnd_;
                        if ((last_cwnd_action_ == 0 ||
                          last_cwnd_action_ == CWND_ACTION_TIMEOUT)
                          && max_ssthresh_ > 0) {
                                increment = limited_slow_start(cwnd_,
```

```
                max_ssthresh_, increment);
        }
        cwnd_ += increment;
        break;

case 2:
        /* These are window increase algorithms
         * for experimental purposes only. */
        f = (t_srtt_ >> T_SRTT_BITS) * tcp_tick_;
        f *= f;
        f *= wnd_const_;
        f += fcnt_;
        if (f > cwnd_) {
                fcnt_ = 0;
                ++cwnd_;
        } else
                fcnt_ = f;
        break;

case 3:
        f = awnd_;
        f *= f;
        f *= wnd_const_;
        f += fcnt_;
        if (f > cwnd_) {
                fcnt_ = 0;
                ++cwnd_;
        } else
                fcnt_ = f;
        break;

   case 4:
            f = awnd_;
            f *= wnd_const_;
            f += fcnt_;
            if (f > cwnd_) {
                    fcnt_ = 0;
                    ++cwnd_;
            } else
                    fcnt_ = f;
            break;
case 5:
            f = (t_srtt_ >> T_SRTT_BITS) * tcp_tick_;
            f *= wnd_const_;
            f += fcnt_;
            if (f > cwnd_) {
                    fcnt_ = 0;
                    ++cwnd_;
            } else
                    fcnt_ = f;
            break;
   case 6:
            /* binomial controls */
            cwnd_ += increase_num_ / (cwnd_*pow(cwnd_,k_parameter_));
            break;
case 8:
        /* high-speed TCP */
        increment = increase_param();
        if ((last_cwnd_action_ == 0 ||
          last_cwnd_action_ == CWND_ACTION_TIMEOUT)
          && max_ssthresh_ > 0) {
                increment = limited_slow_start(cwnd_,
                  max_ssthresh_, increment);
```

```
                    }
                    cwnd_ += increment;
                        break;
                case 100:
                /*********************
                STCP_ADDITIONS BEGINS
                **********************/
                      stcp_est_rtt_ = (int(t_srtt_) >> T_SRTT_BITS)*tcp_tick_*1000;
                      if ((stcp_min_rtt_ > stcp_est_rtt_) || (stcp_min_rtt_ == 0)) {
                            stcp_min_rtt_ = stcp_est_rtt_;
                            stcp_rtt_fact_ =
(double)(stcp_rtt_comp_fact_const_*(pow(stcp_min_rtt_, 0.333)));
                            printf("IN CASE 100: the rtt_comp_fact calculated is
%lf\n",stcp_rtt_fact_);
                            stcp_rtt_fact_ = (stcp_rtt_fact_ > 1) ? stcp_rtt_fact_ : 1;
                            printf("IN CASE 100: the FINAL FINAL rtt_comp_fact
calculated is %lf\n",stcp_rtt_fact_);

                      }
                      printf("IN CASE 100: the stcp_min_rtt_ calculated is
%lf\n",stcp_min_rtt_);
                      printf("IN CASE 100: the stcp_est_rtt_ calculated is
%lf\n",stcp_est_rtt_);
                      increment = (stcp_rtt_fact_ * (stcp_num_layer_ +
stcp_frac_layer_))/cwnd_;
                      if ((last_cwnd_action_ == 0 ||
                            last_cwnd_action_ == CWND_ACTION_TIMEOUT)
                            && max_ssthresh_ > 0) {
                            increment = limited_slow_start(cwnd_,
                            max_ssthresh_, increment);
                            stcp_lim_slowstart_flag = 1;
                      }
                      cwnd_ += increment;
                    if (!stcp_lim_slowstart_flag){
                      int numLayers =0;
                        stcp_min_rtt_ = (int(t_srtt_) >> T_SRTT_BITS)*tcp_tick_*1000;
                   stcp_rtt_fact_ = (double)(stcp_rtt_comp_fact_const_*(pow(stcp_min_rtt_,
0.333)));
                        stcp_rtt_fact_ = (stcp_rtt_fact_ > 1 ) ? stcp_rtt_fact_ : 1;
                        //additions made to write the RTT and the alpha values into a
file and plot the resulting graph.
                       RTT_file  = fopen("RTT_file.txt","a+");
                       fprintf(RTT_file,"",stcp_rtt_fact_);
                       fprintf(RTT_file,"","SOMETHIN ");
                       fclose(RTT_file);
                       printf("IN CASE 100: value of RTT_fact after recalculating :
%lf\n", stcp_rtt_fact_);
                  if(cwnd_ > 83000){
                        stcp_alpha_  =(double) (stcp_win_thresh_h*stcp_win_thresh_h *
stcp_rtt_fact_*stcp_rtt_fact_ ) /6166;
                        }
                  else if(cwnd_ >=  38){
                             stcp_alpha_  =(double)
(stcp_win_thresh_l*stcp_win_thresh_l * stcp_rtt_fact_*stcp_rtt_fact_ )/6166;
                        }
                        ALPHA_file  = fopen("ALPHA_file.txt","a+");
                  fprintf(ALPHA_file,"%lf\t%lf\n",stcp_rtt_fact_,stcp_alpha_);
                  fclose(ALPHA_file);
                        printf("IN CASE 100: the value of ALPHA after recalculating the
number of flows is ===== %lf\n",stcp_alpha_);
                        for(int i=3; i < 100; i++){
                              stcp_win_[i] = (double) stcp_alpha_ * stcp_win_[i-1];
                  }
```

```
                        while (cwnd_ > stcp_win_[stcp_num_layer_ + 1 ]){
                               stcp_num_layer_++;
                        }
                        stcp_frac_layer_= (cwnd_ - stcp_win_[stcp_num_layer_])/
                                    (stcp_win_[stcp_num_layer_+1] -
stcp_win_[stcp_num_layer_]);

                         }
                        break;
        /*********************
        STCP_ADDITIONS ENDS
        **********************/
               default:
#ifdef notdef
                        /*XXX*/
                        error("illegal window option %d", wnd_option_);
#endif
                        abort();
               }
        }
        // if maxcwnd_ is set (nonzero), make it the cwnd limit
        if (maxcwnd_ && (int(cwnd_) > maxcwnd_))
               cwnd_ = maxcwnd_;

        return;
}

void
TcpAgent::slowdown(int how)
{
        double decrease;  /* added for highspeed - sylvia */
        double win, halfwin, decreasewin;
        int slowstart = 0;
        ++ncwndcuts_;
        // we are in slowstart for sure if cwnd < ssthresh
        if (cwnd_ < ssthresh_)
               slowstart = 1;
         if (precision_reduce_) {
               halfwin = windowd() / 2;
                  if (wnd_option_ == 6) {
                          /* binomial controls */
                          decreasewin = windowd() - (1.0-
decrease_num_)*pow(windowd(),l_parameter_);
                  } else if (wnd_option_ == 8 && (cwnd_ > low_window_)) {
                          /* experimental highspeed TCP */
                      decrease = decrease_param();
                      //if (decrease < 0.1)
                      //     decrease = 0.1;
                      decrease_num_ = decrease;
                          decreasewin = windowd() - (decrease * windowd());
                  } else if (wnd_option_ == 100)
               {
                                /*********************
                                            STCP_ADDITIONS BEGINS
                                            **********************/
                         decrease_num_ = (1 - stcp_beta_);
                         double stcp_decrease_ = stcp_beta_ * (double)windowd() ;
                         decreasewin  = windowd() - stcp_decrease_ ;
                         // Recalculate num_of_flows_
                          while (decreasewin <stcp_win_[stcp_num_layer_])
                          {
                                 stcp_num_layer_ -= 1;
                          }
```

```
                        stcp_frac_layer_ = (decreasewin -
stcp_win_[stcp_num_layer_])/
                                                (stcp_win_[stcp_num_layer_+1]
- stcp_win_[stcp_num_layer_]);
                                        /*********************
                                           STCP_ADDITIONS ENDS
                                           **********************/

            } else {
                  decreasewin = decrease_num_ * windowd();
            }
            win = windowd();
      } else  {
            int temp;
            temp = (int)(window() / 2);
            halfwin = (double) temp;
               if (wnd_option_ == 6) {
                     /* binomial controls */
                     temp = (int)(window() - (1.0-
decrease_num_)*pow(window(),l_parameter_));
               } else if ((wnd_option_ == 8) && (cwnd_ > low_window_)) {
                     /* experimental highspeed TCP */
                  decrease = decrease_param();
                  //if (decrease < 0.1)
                      //       decrease = 0.1;
                  decrease_num_ = decrease;
                      temp = (int)(windowd() - (decrease * windowd()));
               } else if (wnd_option_ == 100) {
                  /*********************
                  STCP_ADDITIONS BEGINS
                  **********************/
                  decrease_num_ = (1 - stcp_beta_);
                  double stcp_decrease_ = stcp_beta_ * windowd() ;
                         temp = (int) (windowd() - stcp_decrease_) ;

                  // Recalculate num_of_flows_
                  while (temp < stcp_win_[stcp_num_layer_])
                  {
                         stcp_num_layer_ -= 1;
                  }

                  stcp_frac_layer_ = (temp - stcp_win_[stcp_num_layer_])/
                                           (stcp_win_[stcp_num_layer_+1] -
stcp_win_[stcp_num_layer_]);

                  /*********************
                  STCP_ADDITIONS ENDS
                  **********************/
            } else {
                  temp = (int)(decrease_num_ * window());
            }
            decreasewin = (double) temp;
            win = (double) window();
      }
      if (how & CLOSE_SSTHRESH_HALF)
            // For the first decrease, decrease by half
            // even for non-standard values of decrease_num_.
            if (first_decrease_ == 1 || slowstart ||
                  last_cwnd_action_ == CWND_ACTION_TIMEOUT) {
                  // Do we really want halfwin instead of decreasewin
            // after a timeout?
                  ssthresh_ = (int) halfwin;
            } else {
```

```
                ssthresh_ = (int) decreasewin;
        }
         else if (how & THREE_QUARTER_SSTHRESH)
        if (ssthresh_ < 3*cwnd_/4)
                ssthresh_  = (int)(3*cwnd_/4);
      if (how & CLOSE_CWND_HALF)
        // For the first decrease, decrease by half
        // even for non-standard values of decrease_num_.
        if (first_decrease_ == 1 || slowstart || decrease_num_ == 0.5){
                cwnd_ = halfwin;
       /*********************
        STCP_ADDITIONS BEGINS
           **********************/
                                        if(wnd_option_ == 100){
                                                // STCP When getting out of slowstart
for the first time set window to halfwin
                                                // calculate stcp_num_layer_
                            stcp_num_layer_ = 1;
                                                while (cwnd_ >
stcp_win_[stcp_num_layer_ + 1]){
                                                stcp_num_layer_++;
                                              }
                                                stcp_frac_layer_ = (cwnd_ -
stcp_win_[stcp_num_layer_])/
                                                (stcp_win_[stcp_num_layer_+1] -
stcp_win_[stcp_num_layer_]);
                        }

        }
        /*********************
        STCP_ADDITIONS ENDS
        **********************/

        else   cwnd_ = decreasewin;
         else if (how & CWND_HALF_WITH_MIN) {
        // We have not thought about how non-standard TCPs, with
        // non-standard values of decrease_num_, should respond
        // after quiescent periods.
           cwnd_ = decreasewin;
           if (cwnd_ < 1)
                   cwnd_ = 1;
      }
      else if (how & CLOSE_CWND_RESTART)
        cwnd_ = int(wnd_restart_);
      else if (how & CLOSE_CWND_INIT)
        cwnd_ = int(wnd_init_);
      else if (how & CLOSE_CWND_ONE)
        cwnd_ = 1;
      else if (how & CLOSE_CWND_HALF_WAY) {
        // cwnd_ = win - (win - W_used)/2 ;
        cwnd_ = W_used + decrease_num_ * (win - W_used);
           if (cwnd_ < 1)
                   cwnd_ = 1;
      }
      if (ssthresh_ < 2)
        ssthresh_ = 2;
      if (how & (CLOSE_CWND_HALF|CLOSE_CWND_RESTART|CLOSE_CWND_INIT|CLOSE_CWND_ONE))
        cong_action_ = TRUE;

      fcnt_ = count_ = 0;
      if (first_decrease_ == 1)
        first_decrease_ = 0;
      // for event tracing slow start
```

```
	if (cwnd_ == 1 || slowstart)
		// Not sure if this is best way to capture slow_start
		// This is probably tracing a superset of slowdowns of
		// which all may not be slow_start's --Padma, 07/'01.
		trace_event("SLOW_START");



}



/*
 * Process a packet that acks previously unacknowleged data.
 */
void TcpAgent::newack(Packet* pkt)
{
	double now = Scheduler::instance().clock();
	hdr_tcp *tcph = hdr_tcp::access(pkt);
	/*
	 * Wouldn't it be better to set the timer *after*
	 * updating the RTT, instead of *before*?
	 */
	if (!timerfix_) newtimer(pkt);
	dupacks_ = 0;
	last_ack_ = tcph->seqno();
	prev_highest_ack_ = highest_ack_ ;
	highest_ack_ = last_ack_;

	if (t_seqno_ < last_ack_ + 1)
		t_seqno_ = last_ack_ + 1;
	/*
	 * Update RTT only if it's OK to do so from info in the flags header.
	 * This is needed for protocols in which intermediate agents
	 * in the network intersperse acks (e.g., ack-reconstructors) for
	 * various reasons (without violating e2e semantics).
	 */
	hdr_flags *fh = hdr_flags::access(pkt);
	if (!fh->no_ts_) {
		if (ts_option_)
			rtt_update(now - tcph->ts_echo());

		if (rtt_active_ && tcph->seqno() >= rtt_seq_) {
			if (!ect_ || !ecn_backoff_ ||
			    !hdr_flags::access(pkt)->ecnecho()) {
				/*
				 * Don't end backoff if still in ECN-Echo with
				 * a congestion window of 1 packet.
				 */
				t_backoff_ = 1;
				ecn_backoff_ = 0;
			}
			rtt_active_ = 0;
			if (!ts_option_)
				rtt_update(now - rtt_ts_);
		}
	}
	if (timerfix_) newtimer(pkt);
	/* update average window */
	awnd_ *= 1.0 - wnd_th_;
	awnd_ += wnd_th_ * cwnd_;
}
```

```
/*
 * Respond either to a source quench or to a congestion indication bit.
 * This is done at most once a roundtrip time;  after a source quench,
 * another one will not be done until the last packet transmitted before
 * the previous source quench has been ACKed.
 *
 * Note that this procedure is called before "highest_ack_" is
 * updated to reflect the current ACK packet.
 */
void TcpAgent::ecn(int seqno)
{
	if (seqno > recover_ ||
	      last_cwnd_action_ == CWND_ACTION_TIMEOUT) {
		recover_ =  maxseq_;
		last_cwnd_action_ = CWND_ACTION_ECN;
		if (cwnd_ <= 1.0) {
			if (ecn_backoff_)
				rtt_backoff();
			else ecn_backoff_ = 1;
		} else ecn_backoff_ = 0;
		slowdown(CLOSE_CWND_HALF|CLOSE_SSTHRESH_HALF);
		++necnresponses_ ;
		// added by sylvia to count number of ecn responses
	}
}

/*
 *  Is the connection limited by the network (instead of by a lack
 *    of data from the application?
 */
int TcpAgent::network_limited() {
	int win = window () ;
	if (t_seqno_ > (prev_highest_ack_ + win))
		return 1;
	else
		return 0;
}

void TcpAgent::recv_newack_helper(Packet *pkt) {
	//hdr_tcp *tcph = hdr_tcp::access(pkt);
	newack(pkt);
	if (!ect_ || !hdr_flags::access(pkt)->ecnecho() ||
		(old_ecn_ && ecn_burst_)) {
		/* If "old_ecn", this is not the first ACK carrying ECN-Echo
		 * after a period of ACKs without ECN-Echo.
		 * Therefore, open the congestion window. */
		/* if control option is set, and the sender is not
			 window limited, then do not increase the window size */

		if (!control_increase_ ||
		   (control_increase_ && (network_limited() == 1)))
		       opencwnd();
	}
	if (ect_) {
		if (!hdr_flags::access(pkt)->ecnecho())
			ecn_backoff_ = 0;
		if (!ecn_burst_ && hdr_flags::access(pkt)->ecnecho())
			ecn_burst_ = TRUE;
		else if (ecn_burst_ && ! hdr_flags::access(pkt)->ecnecho())
			ecn_burst_ = FALSE;
	}
```

```
	if (!ect_ && hdr_flags::access(pkt)->ecnecho() &&
		!hdr_flags::access(pkt)->cong_action()){
		ect_ = 1;
	}
	/* if the connection is done, call finish() */
	if ((highest_ack_ >= curseq_-1) && !closed_) {
		closed_ = 1;
		finish();
	}
	if (QOption_ && curseq_ == highest_ack_ +1) {
		cancel_rtx_timer();
	}
}

/*
 * Set the initial window.
 */
double
TcpAgent::initial_window()
{
	//
	// init_option = 1: static iw of wnd_init_
	//
	if (wnd_init_option_ == 1) {
		return (wnd_init_);
	}
	 else if (wnd_init_option_ == 2) {
		// do iw according to Internet draft
		if (size_ <= 1095) {
			return (4.0);
		} else if (size_ < 2190) {
			return (3.0);
		} else {
			return (2.0);
		}
	}
	// XXX what should we return here???
	fprintf(stderr, "Wrong number of wnd_init_option_ %d\n",
		wnd_init_option_);
	abort();
	return (2.0); // XXX make msvc happy.
}

/*
 * Dupack-action: what to do on a DUP ACK.  After the initial check
 * of 'recover' below, this function implements the following truth
 * table:
 *
 *	bugfix ecn	last-cwnd == ecn	action
 *
 *	0	0	0			tahoe_action
 *	0	0	1			tahoe_action [impossible]
 *	0	1	0			tahoe_action
 *	0	1	1			slow-start, return
 *	1	0	0			nothing
 *	1	0	1			nothing		[impossible]
 *	1	1	0			nothing
 *	1	1	1			slow-start, return
 */

/*
 * A first or second duplicate acknowledgement has arrived, and
 * singledup_ is enabled.
```

```
 * If the receiver's advertised window permits, and we are exceeding our
 * congestion window by less than numdupacks_, then send a new packet.
 */
void
TcpAgent::send_one()
{
	if (t_seqno_ <= highest_ack_ + wnd_ && t_seqno_ < curseq_ &&
	    t_seqno_ <= highest_ack_ + cwnd_ + dupacks_ ) {
		output(t_seqno_, 0);
		if (QOption_)
			process_qoption_after_send () ;
		t_seqno_ ++ ;
		// send_helper(); ??
	}
	return;
}

void
TcpAgent::dupack_action()
{
	int recovered = (highest_ack_ > recover_);
	if (recovered || (!bug_fix_ && !ecn_)) {
		goto tahoe_action;
	}

	if (ecn_ && last_cwnd_action_ == CWND_ACTION_ECN) {
		last_cwnd_action_ = CWND_ACTION_DUPACK;
		slowdown(CLOSE_CWND_ONE);
		reset_rtx_timer(0,0);
		return;
	}

	if (bug_fix_) {
		/*
		 * The line below, for "bug_fix_" true, avoids
		 * problems with multiple fast retransmits in one
		 * window of data.
		 */
		return;
	}

tahoe_action:
	// we are now going to fast-retransmit and willtrace that event
	trace_event("FAST_RETX");

	recover_ = maxseq_;
	last_cwnd_action_ = CWND_ACTION_DUPACK;
	slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_ONE);
	reset_rtx_timer(0,0);
	return;
}

/*
 * When exiting QuickStart, reduce the congestion window to the
 *   size that was actually used.
 */
void TcpAgent::endQuickStart()
{
	qs_approved_ = 0;
	int new_cwnd = maxseq_ - last_ack_;
	if (new_cwnd > 1 && new_cwnd < cwnd_) {
		cwnd_ = new_cwnd;
		if (cwnd_ < initial_window())
```

```
                cwnd_ = initial_window();
        }
}

void TcpAgent::processQuickStart(Packet *pkt)
{
        // QuickStart code from Srikanth Sundarrajan.
        hdr_tcp *tcph = hdr_tcp::access(pkt);
        hdr_qs *qsh = hdr_qs::access(pkt);
        double now = Scheduler::instance().clock();
        int app_rate;

         // printf("flag: %d ttl: %d ttl_diff: %d rate: %d\n", qsh->flag(),
        //     qsh->ttl(), ttl_diff_, qsh->rate());
        qs_requested_ = 0;
        qs_approved_ = 0;
        if (qsh->flag() == QS_RESPONSE && qsh->ttl() == ttl_diff_ &&
             qsh->rate() > 0) {
                app_rate = (int) (qsh->rate() * (now - tcph->ts_echo())) ;
               printf("Quick Start approved, rate %d, window %d\n",
                                qsh->rate(), app_rate);
                  if (app_rate > initial_window()) {
                          wnd_init_option_ = 1;
                          wnd_init_ = app_rate;
                          qs_approved_ = 1;
                  }
         } else { // Quick Start rejected
                  printf("Quick Start rejected\n");
         }

}

/*
 * main reception path - should only see acks, otherwise the
 * network connections are misconfigured
 */
void TcpAgent::recv(Packet *pkt, Handler*)
{
        hdr_tcp *tcph = hdr_tcp::access(pkt);
        if (qs_approved_ == 1 && tcph->seqno() > last_ack_)
                endQuickStart();
        if (qs_requested_ == 1)
                processQuickStart(pkt);
#ifdef notdef
        if (pkt->type_ != PT_ACK) {
                Tcl::instance().evalf("%s error \"received non-ack\"",
                                      name());
                Packet::free(pkt);
                return;
        }
#endif
        /* W.N.: check if this is from a previous incarnation */
        if (tcph->ts() < lastreset_) {
                // Remove packet and do nothing
                Packet::free(pkt);
                return;
        }
        ++nackpack_;
        ts_peer_ = tcph->ts();
        int ecnecho = hdr_flags::access(pkt)->ecnecho();
        if (ecnecho && ecn_)
                ecn(tcph->seqno());
        recv_helper(pkt);
```

```
	/* grow cwnd and check if the connection is done */
	if (tcph->seqno() > last_ack_) {
		recv_newack_helper(pkt);
		if (last_ack_ == 0 && delay_growth_) {
			cwnd_ = initial_window();
		}
	} else if (tcph->seqno() == last_ack_) {
		  if (hdr_flags::access(pkt)->eln_ && eln_) {
				tcp_eln(pkt);
				return;
		  }
		if (++dupacks_ == numdupacks_ && !noFastRetrans_) {
			dupack_action();
		} else if (dupacks_ < numdupacks_ && singledup_ ) {
			send_one();
		}
	}

	if (QOption_ && EnblRTTCtr_)
		process_qoption_after_ack (tcph->seqno());

	Packet::free(pkt);
	/*
	 * Try to send more data.
	 */
	send_much(0, 0, maxburst_);
}

/*
 * Process timeout events other than rtx timeout. Having this as a separate
 * function allows derived classes to make alterations/enhancements (e.g.,
 * response to new types of timeout events).
 */
void TcpAgent::timeout_nonrtx(int tno)
{
	if (tno == TCP_TIMER_DELSND)  {
	 /*
		* delayed-send timer, with random overhead
		* to avoid phase effects
		*/
		send_much(1, TCP_REASON_TIMEOUT, maxburst_);
	}
}

void TcpAgent::timeout(int tno)
{
	/* retransmit timer */
	if (tno == TCP_TIMER_RTX) {

		// There has been a timeout - will trace this event
		trace_event("TIMEOUT");

		 if (cwnd_ < 1) cwnd_ = 1;
		if (qs_approved_ == 1) qs_approved_ = 0;
		if (highest_ack_ == maxseq_ && !slow_start_restart_) {
			/*
			 * TCP option:
			 * If no outstanding data, then don't do anything.
			 */
			 // Should this return be here?
			 // What if CWND_ACTION_ECN and cwnd < 1?
			 // return;
		} else {
```

```
                        recover_ = maxseq_;
                        if (highest_ack_ == -1 && wnd_init_option_ == 2)
                                /*
                                 * First packet dropped, so don't use larger
                                 * initial windows.
                                 */
                                wnd_init_option_ = 1;
                        if (highest_ack_ == maxseq_ && restart_bugfix_)
                                /*
                                * if there is no outstanding data, don't cut
                                * down ssthresh_.
                                */
                                slowdown(CLOSE_CWND_ONE);
                        else if (highest_ack_ < recover_ &&
                          last_cwnd_action_ == CWND_ACTION_ECN) {
                                /*
                                * if we are in recovery from a recent ECN,
                                * don't cut down ssthresh_.
                                */
                                slowdown(CLOSE_CWND_ONE);
                        }
                        else {
                                ++nrexmit_;
                                last_cwnd_action_ = CWND_ACTION_TIMEOUT;
                                slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_RESTART);
                        }
                }
                /* if there is no outstanding data, don't back off rtx timer */
                if (highest_ack_ == maxseq_ && restart_bugfix_) {
                        reset_rtx_timer(0,0);
                }
                else {
                        reset_rtx_timer(0,1);
                }
                last_cwnd_action_ = CWND_ACTION_TIMEOUT;
                send_much(0, TCP_REASON_TIMEOUT, maxburst_);
        }
        else {
                timeout_nonrtx(tno);
        }
}

/*
 * Check if the packet (ack) has the ELN bit set, and if it does, and if the
 * last ELN-rxmitted packet is smaller than this one, then retransmit the
 * packet.  Do not adjust the cwnd when this happens.
 */
void TcpAgent::tcp_eln(Packet *pkt)
{
        //int eln_rxmit;
        hdr_tcp *tcph = hdr_tcp::access(pkt);
        int ack = tcph->seqno();

        if (++dupacks_ == eln_rxmit_thresh_ && ack > eln_last_rxmit_) {
                /* Retransmit this packet */
                output(last_ack_ + 1, TCP_REASON_DUPACK);
                eln_last_rxmit_ = last_ack_+1;
        } else
                send_much(0, 0, maxburst_);

        Packet::free(pkt);
        return;
}
```

```
/*
 * This function is invoked when the connection is done. It in turn
 * invokes the Tcl finish procedure that was registered with TCP.
 */
void TcpAgent::finish()
{
	Tcl::instance().evalf("%s done", this->name());
}

void RtxTimer::expire(Event*)
{
	a_->timeout(TCP_TIMER_RTX);
}

void DelSndTimer::expire(Event*)
{
	a_->timeout(TCP_TIMER_DELSND);
}

void BurstSndTimer::expire(Event*)
{
	a_->timeout(TCP_TIMER_BURSTSND);
}

/*
 * THE FOLLOWING FUNCTIONS ARE OBSOLETE, but REMAIN HERE
 * DUE TO OTHER PEOPLE's TCPs THAT MIGHT USE THEM
 *
 * These functions are now replaced by ecn() and slowdown(),
 * respectively.
 */

/*
 * Respond either to a source quench or to a congestion indication bit.
 * This is done at most once a roundtrip time;  after a source quench,
 * another one will not be done until the last packet transmitted before
 * the previous source quench has been ACKed.
 */
void TcpAgent::quench(int how)
{
	if (highest_ack_ >= recover_) {
		recover_ =  maxseq_;
		last_cwnd_action_ = CWND_ACTION_ECN;
		closecwnd(how);
	}
}

/*
 * close down the congestion window
 */
void TcpAgent::closecwnd(int how)
{
	static int first_time = 1;
	if (first_time == 1) {
		fprintf(stderr, "the TcpAgent::closecwnd() function is now deprecated,
please use the function slowdown() instead\n");
	}
	switch (how) {
	case 0:
		/* timeouts */
		ssthresh_ = int( window() / 2 );
		if (ssthresh_ < 2)
```

```
                ssthresh_ = 2;
            cwnd_ = int(wnd_restart_);
            break;

        case 1:
            /* Reno dup acks, or after a recent congestion indication. */
            // cwnd_ = window()/2;
            cwnd_ = decrease_num_ * window();
            ssthresh_ = int(cwnd_);
            if (ssthresh_ < 2)
                ssthresh_ = 2;
            break;

        case 2:
            /* Tahoe dup acks
             * after a recent congestion indication */
            cwnd_ = wnd_init_;
            break;

        case 3:
            /* Retransmit timeout, but no outstanding data. */
            cwnd_ = int(wnd_init_);
            break;
        case 4:
            /* Tahoe dup acks */
            ssthresh_ = int( window() / 2 );
            if (ssthresh_ < 2)
                ssthresh_ = 2;
            cwnd_ = 1;
            break;

        default:
            abort();
        }
        fcnt_ = 0.;
        count_ = 0;
}

/*
 * Check if the sender has been idle or application-limited for more
 * than an RTO, and if so, reduce the congestion window.
 */
void TcpAgent::process_qoption_after_send ()
{
        int tcp_now = (int)(Scheduler::instance().clock()/tcp_tick_ + 0.5);
        int rto = (int)(t_rtxcur_/tcp_tick_) ;
        /*double ct = Scheduler::instance().clock();*/

        if (!EnblRTTCtr_) {
            if (tcp_now - T_last >= rto) {
                // The sender has been idle.
                slowdown(THREE_QUARTER_SSTHRESH) ;
                for (int i = 0 ; i < (tcp_now - T_last)/rto; i ++) {
                    slowdown(CWND_HALF_WITH_MIN);
                }
                T_prev = tcp_now ;
                W_used = 0 ;
            }
            T_last = tcp_now ;
            if (t_seqno_ == highest_ack_+ window()) {
                T_prev = tcp_now ;
                W_used = 0 ;
            }
```

```
                else if (t_seqno_ == curseq_-1) {
                        // The sender has no more data to send.
                        int tmp = t_seqno_ - highest_ack_ ;
                        if (tmp > W_used)
                                W_used = tmp ;
                        if (tcp_now - T_prev >= rto) {
                                // The sender has been application-limited.
                                slowdown(THREE_QUARTER_SSTHRESH);
                                slowdown(CLOSE_CWND_HALF_WAY);
                                T_prev = tcp_now ;
                                W_used = 0 ;
                        }
                }
        } else {
                rtt_counting();
        }
}

/*
 * Check if the sender has been idle or application-limited for more
 * than an RTO, and if so, reduce the congestion window, for a TCP sender
 * that "counts RTTs" by estimating the number of RTTs that fit into
 * a single clock tick.
 */
void
TcpAgent::rtt_counting()
{
         int tcp_now = (int)(Scheduler::instance().clock()/tcp_tick_ + 0.5);
        int rtt = (int(t_srtt_) >> T_SRTT_BITS) ;

        if (rtt < 1)
                rtt = 1 ;
        if (tcp_now - T_last >= 2*rtt) {
                // The sender has been idle.
                int RTTs ;
                RTTs = (tcp_now -T_last)*RTT_goodcount/(rtt*2) ;
                RTTs = RTTs - Backoffs ;
                Backoffs = 0 ;
                if (RTTs > 0) {
                        slowdown(THREE_QUARTER_SSTHRESH) ;
                        for (int i = 0 ; i < RTTs ; i ++) {
                                slowdown(CWND_HALF_WITH_MIN);
                                RTT_prev = RTT_count ;
                                W_used = 0 ;
                        }
                }
        }
        T_last = tcp_now ;
        if (tcp_now - T_start >= 2*rtt) {
                if ((RTT_count > RTT_goodcount) || (F_full == 1)) {
                        RTT_goodcount = RTT_count ;
                        if (RTT_goodcount < 1) RTT_goodcount = 1 ;
                }
                RTT_prev = RTT_prev - RTT_count ;
                RTT_count = 0 ;
                T_start  = tcp_now ;
                F_full = 0;
        }
        if (t_seqno_ == highest_ack_ + window()) {
                W_used = 0 ;
                F_full = 1 ;
                RTT_prev = RTT_count ;
        }
```

```
	else if (t_seqno_ == curseq_-1) {
		// The sender has no more data to send.
		int tmp = t_seqno_ - highest_ack_ ;
		if (tmp > W_used)
			W_used = tmp ;
		if (RTT_count - RTT_prev >= 2) {
			// The sender has been application-limited.
			slowdown(THREE_QUARTER_SSTHRESH) ;
			slowdown(CLOSE_CWND_HALF_WAY);
			RTT_prev = RTT_count ;
			Backoffs ++ ;
			W_used = 0;
		}
	}
	if (F_counting == 0) {
		W_timed = t_seqno_  ;
		F_counting = 1 ;
	}
}

void TcpAgent::process_qoption_after_ack (int seqno)
{
	if (F_counting == 1) {
		if (seqno >= W_timed) {
			RTT_count ++ ;
			F_counting = 0 ;
		}
		else {
			if (dupacks_ == numdupacks_)
				RTT_count ++ ;
		}
	}
}

void TcpAgent::trace_event(char *eventtype)
{
	if (et_ == NULL) return;
	int seqno = t_seqno_;
	char *wrk = et_->buffer();
	char *nwrk = et_->nbuffer();
	if (wrk != 0)
		sprintf(wrk,
			"E "TIME_FORMAT" %d %d TCP %s %d %d %d",
			et_->round(Scheduler::instance().clock()),   // time
			addr(),                      // owner (src) node id
			daddr(),                     // dst node id
			eventtype,                   // event type
			fid_,                        // flow-id
			seqno,                       // current seqno
			int(cwnd_)                        //cong. window
			);

	if (nwrk != 0)
		sprintf(nwrk,
			"E -t "TIME_FORMAT" -o TCP -e %s -s %d.%d -d %d.%d",
			et_->round(Scheduler::instance().clock()),   // time
			eventtype,                   // event type
			addr(),                      // owner (src) node id
			port(),                      // owner (src) port id
			daddr(),                     // dst node id
			dport()                      // dst port id
			);
	et_->trace();
```

```
}
```

7.2 APPENDIX II: Scripts

7.2.1 TCL Script to Simulate and Test STCP

```
#
#SCRIPT WRITTEN BY: RANJITHA SHIVARUDRAIAH
#OBJECTIVE:         TO SIMULATE A DUMBELL TOPOLOGY AND TEST STRATIFIED TCP( STCP)
#COURSE:            MS THESIS
#                    (A STUDY OF PARALLEL TRANSPORT TECHNIQUES
#                      FOR HIGH-SPEED NETWORKS)
#CODE DESCRIPTION:  This TCL script is used to simulate the dumbell topology and test
#                   the stcp implementation.
#                   In order to execute the STCP the following line should be added in
#                   the beginning:
#                   "Agent/TCP set windowOption_ 100". 100 indicates the case number
#                   in tcp.cc
#                   where Stratified TCP is implemented.

# Create a simulator object

set ns [new Simulator]

set numFlows 0
set snumFlows 6
set bw 1000
set bdelay 40
#congestion avoidance algorithm in case 100 of tcp.cc
Agent/TCP set windowOption_ 100
set f [open out.tr w]
$ns trace-all $f

set nf [open out.nam w]
$ns namtrace-all $nf

set tcpStart 0
set txEnd 65

set buffer [expr ($bw*$bdelay*1000)/([Agent/TCP set packetSize_]*8)]
Queue set limit_ $buffer
set util [open result.tr w]

# Source Nodes
for {set i 0} {$i < $numFlows } {incr i} {
       set s($i) [$ns node]
}

# Receiver nodes
for {set i 0} {$i < $numFlows } {incr i} {
       set r($i) [$ns node]
}

# Stratified LTCP Source Nodes
for {set i 0} {$i < $snumFlows } {incr i} {
       set ls($i) [$ns node]
```

```
}

#Stratified Receiver nodes
for {set i 0} {$i < $snumFlows } {incr i} {
       set lr($i) [$ns node]
}


# Bottle neck nodes
set R1 [$ns node]
set R2 [$ns node]

# connect source/reciever to Router
for {set i 0} {$i < $numFlows } {incr i} {
       $ns duplex-link $s($i) $R1 2.4Gb 10ms DropTail
       $ns duplex-link $R2 $r($i) 2.4Gb 10ms DropTail
       set rtt [expr 2*(10+10+$bdelay)]
       puts "RTT of TCP source$i: $rtt (ms)"
       puts $util "RTT of TCP source$i: $rtt (ms)"
}

# connect Layered source/receiver to Router
for {set i 0} {$i < $snumFlows } {incr i} {
       $ns duplex-link $ls($i) $R1 2.4Gb 10ms DropTail
       $ns duplex-link $R2 $lr($i) 2.4Gb 10ms DropTail
       set rtt [expr 2*(10+10+$bdelay)]
       puts "RTT of Stratified source$i: $rtt (ms)"
       puts $util "RTT of Stratified source$i: $rtt (ms)"

}

#Setup a  UDP connection and add the UDP agent
set udp_s [new Agent/UDP]
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp_s
$ns attach-agent $R1 $udp_s
set null0 [new Agent/Null]
$ns attach-agent $R2 $null0
$ns connect $udp_s $null0
$ns at 0.0 "$cbr0 start"
$ns at 50.0 "$cbr0 stop"
$ns at 100.0 "$cbr0 start"
$ns at 150.0 "$cbr0 stop"
$udp_s set packetSize_ 500
$cbr0 set interval_ 5
$cbr0 set rate_ 500Mb
$udp_s set fid_ 3

# Bottleneck
$ns duplex-link $R1 $R2 [expr $bw]Mb [expr $bdelay]ms DropTail

# Source/Receiver Agent
for {set i 0} {$i < $numFlows } {incr i} {

    set tcp($i) [new Agent/TCP/Sack1]
    $ns attach-agent $s($i) $tcp($i)
    $tcp($i) set window_ 1000000000
    set ftp($i) [new Application/FTP]
    $ftp($i) attach-agent $tcp($i)
    set rcvr($i) [new Agent/TCPSink/Sack1]
    $ns attach-agent $r($i) $rcvr($i)

    $ns connect $tcp($i) $rcvr($i)
```

```
}

#Stratified TCP Source Agent
for {set i 0} {$i < $snumFlows } {incr i} {

    set stcp($i) [new Agent/TCP/Sack1]
    $ns attach-agent $ls($i) $stcp($i)
    $stcp($i) set window_ 1000000000
    $stcp($i) set windowOption_ 100
    set sftp($i) [new Application/FTP]
    $sftp($i) attach-agent $stcp($i)
    $sftp($i) set fid 0
    set srcvr($i) [new Agent/TCPSink/Sack1]
    $ns attach-agent $lr($i) $srcvr($i)

    $ns connect $stcp($i) $srcvr($i)
}

# Start the agents
for {set i 0} {$i < $numFlows } {incr i} {
  $ns at $tcpStart "$ftp($i) start"
  $ns at $txEnd "$ftp($i) stop"
}
for {set i 0} {$i < $snumFlows } {incr i} {
  $ns at 0.01 "$sftp($i) start"
  $ns at $txEnd "$sftp($i) stop"
}

for {set i 0} {$i < $numFlows } {incr i} {
  set initLPktCnt($i) 0
}

for {set i 0} {$i < $snumFlows } {incr i} {
       set initLLPktCnt($i) 0
}


# flow mon object
set slink [$ns link $R1 $R2]
set fmon [$ns makeflowmon Fid]
$ns attach-fmon $slink $fmon
set fm [open flow.tr w]
$fmon attach $fm
$ns at 100 "$fmon dump"
$ns at 300 "$fmon dump"
$ns at 800 "$fmon dump"
$ns at $txEnd "$fmon dump"


set qmon [$ns monitor-queue $R1 $R2 ""]
set initLPkt 0
set initLDrop 0

set ftotw [open aggwin.tr w ]
set ftotr [open aggrate.tr w ]

for {set i 0} {$i < $snumFlows } {incr i} {
 set flw($i) [open layerwin$i.tr w]
 set flr($i) [open layerrate$i.tr w]
}


for {set i 0} {$i < $numFlows } {incr i} {
```

```
 set ftw($i) [open tcpwin$i.tr w]
 set ftr($i) [open tcprate$i.tr w]
}

puts "Bottleneck Bandwidth: $bw (Mb); Bottleneck delay: $bdelay (ms); Buffer Size:
$buffer"
puts $util "Bottleneck Bandwidth: $bw (Mb); Bottleneck delay: $bdelay (ms); Buffer
Size: $buffer"

$ns at [expr ($tcpStart+100)] "getInitVal"
$ns at $txEnd "calculate"
$ns at $txEnd "finish"

set initTime 0
set calculateTime 0

proc getInitVal {} {
       global ns initLPktCnt initLLPktCnt initLPkt initLDrop tcp qmon numFlows
snumFlows initTime stcp

       set initTime [$ns now]

        for {set i 0} {$i < $numFlows } {incr i} {
            set initLPktCnt($i) [$tcp($i) set ack_]
       }

       for {set i 0} {$i < $snumFlows } {incr i} {
            set initLLPktCnt($i) [$stcp($i) set ack_]
       }

       set initLDrop [$qmon set pdrops_]
       set initLPkt [$qmon set parrivals_]

}

proc calculate {} {
       global ns initLPktCnt initLLPktCnt initLDrop initLPkt util qmon tcp stcp
numFlows snumFlows
       global initTime calculateTime

       set calculateTime [$ns now]
       set curTp 0
        set scurTp 0
       set throughput 0
       set sthroughput 0
       set interval [expr ($calculateTime - $initTime)]
    for {set i 0} {$i < $numFlows } {incr i} {
         set curPktCnt [expr [$tcp($i) set ack_] - $initLPktCnt($i)]
         set throughput [expr (double($curPktCnt) * [$tcp(0) set packetSize_] * 8 /
($interval * 1000000))]
       puts $util "TCP Throughput of source$i : $throughput ($curPktCnt)"
         set curTp [expr $curTp + $curPktCnt]
         set throughput [expr (double($curTp) * [$tcp(0) set packetSize_] * 8 /
($interval * 1000000))]
    }
    for {set i 0} {$i < $snumFlows } {incr i} {
         set curLPktCnt [expr [$stcp($i) set ack_] - $initLLPktCnt($i)]
         set sthroughput [expr (double($curLPktCnt) * [$stcp(0) set packetSize_] * 8 /
($interval * 1000000))]
         puts $util " STCP Throughput of source$i : $sthroughput ($curLPktCnt)"
              set scurTp [expr $scurTp + $curLPktCnt]
         set sthroughput [expr (double($scurTp) * [$stcp(0) set packetSize_] * 8 /
($interval * 1000000))]
```

```
    }
    set congDrop [expr (double ([$qmon set pdrops_] - $initLDrop)/([$qmon set
parrivals_] - $initLPkt))*100]
}

proc finish {} {
       global ns flw flr ftotw ftotr ftw ftr util tcp stcp numFlows snumFlows *f* *nf*
*nq*

       $ns flush-trace

       for {set i 0} {$i < $snumFlows } {incr i} {
          close $flw($i)
          close $flr($i)
       }
       close $ftotw
       close $ftotr

       for {set i 0} {$i < $numFlows } {incr i} {
          close $ftw($i)
          close $ftr($i)
       }
        close $util
             puts "running nam..."
        exec nam out.nam &

             exit 0
}

for {set i 0} {$i < $numFlows } {incr i} {
  set lastTput($i) 0
}

for {set i 0} {$i < $snumFlows } {incr i} {
  set slastTput($i) 0
}

proc recordThruput {} {
       global ns
       global tcp stcp flr ftotr ftr numFlows snumFlows
       global lastTput slastTput

       set thrutime 0.1
       set now [$ns now]
       set totrate 0

       for {set i 0} {$i < $snumFlows } {incr i} {
          set curLPktCnt [expr ([$stcp($i) set ack_])]
          set scurTput [expr (double($curLPktCnt) * [$stcp($i) set packetSize_] * 8 /
($thrutime * 1000000))]
          set sthroughput [expr ($scurTput - $slastTput($i))]
          set slastTput($i) $scurTput
          puts $flr($i) "$now $sthroughput"
          set totrate [expr $totrate + $sthroughput]
        }

       for {set i 0} {$i < $numFlows } {incr i} {
          set curPktCnt [expr ([$tcp($i) set ack_])]
          set curTput [expr (double($curPktCnt) * [$tcp($i) set packetSize_] * 8 /
($thrutime * 1000000))]
          set throughput [expr ($curTput - $lastTput($i))]
          set lastTput($i) $curTput
          puts $ftr($i) "$now $throughput"
```

```
          set totrate [expr $totrate + $throughput]
       }

       puts $ftotr "$now $totrate"
       $ns at [expr $now+$thrutime] "recordThruput"
}

proc record {} {
       global ns
       global tcp stcp flw ftotw ftw util numFlows snumFlows

       set time 0.1
       set now [$ns now]

       set totwin 0
puts "Simulation time: [format %.2f $now] "
       puts $util "Simulation time: [format %.2f $now] "
       for {set i 0} {$i < $snumFlows } {incr i} {
            set swin [$stcp($i) set cwnd_]
            set totwin [expr $totwin + $swin]
               puts $flw($i) "$now $swin"
       }

       for {set i 0} {$i < $numFlows } {incr i} {
            set win [$tcp($i) set cwnd_]
            set totwin [expr $totwin + $win]
            puts $ftw($i) "$now $win"
       }

       puts $ftotw "$now $totwin"

       $ns at [expr $now+$time] "record"
}

$ns at 0.2 "recordThruput"
$ns at 0.1 "record"
$ns run
```

7.2.2 The AWK Script which Calculates the Delay of STCP/TCP/LTCP.

```
#
#SCRIPT WRITTEN BY:    RANJITHA SHIVARUDRAIAH
#LANGUAGE USED:        awk
#OBJECTIVE:            TO READ FROM A TRACE FILE AND MEASURE THE DELAY OF STCP/TCP
#COURSE:               MS THESIS
#                           (A STUDY OF PARALLEL TRANSPORT TECHNIQUES
#                            FOR HIGH-SPEED NETWORKS)
#CODE DESCRIPTION:     This awk script reads from the trace file generated by the
#                      simulations.
#                      The trace file is generated by the "trace-all()" function and
#                      follows a certain
#                      order in which the parameters are displayed. This principle is
#                      used to find the individual
#                      values of the arguments and calculate the delay from these
#                      values.


BEGIN {
       highest_packet_id = 0;
}

{
```

```
action = $1;

time = $2;

from = $3;

to = $4;

type = $5;

pktsize = $6;

flow_id = $8;

src = $9;

dst = $10;

seq_no = $11;

packet_id = $12;

total =0 ;

avg =0;

count =0;

if ( packet_id > highest_packet_id )
highest_packet_id = packet_id;

if ( start_time[packet_id] == 0 )
start_time[packet_id] = time;

if ( flow_id == 0  && action != "d" ) {

       if ( action == "r" ) {

              end_time[packet_id] = time;
       }

} else {

       end_time[packet_id] = -1;

  }

}
END {
for ( packet_id = 0; packet_id <= highest_packet_id; packet_id++ ) {

       start = start_time[packet_id];

       end = end_time[packet_id];

       packet_duration = end - start;

       if ( start < end ) {
              total = total + packet_duration;count++;
       }
}
avg =total/count;
```

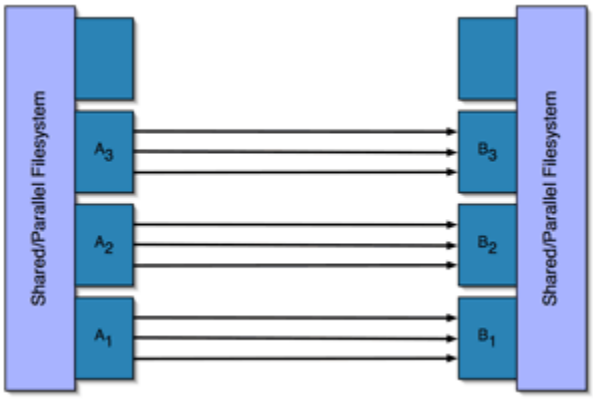
```
printf("AVERAGE delay is %f\n", avg);}
```

7.3 APPENDIX III: Glossary

Goodput: the application level throughput, i.e. the number of useful bits per unit of time forwarded by the network from a certain source address to a certain destination, excluding protocol overhead, and excluding retransmitted data packets.

iSCSI: In computing, the iSCSI (for "Internet SCSI") protocol allows clients (called initiators) to send SCSI commands (CDBs) to SCSI storage devices (targets) on remote servers. It is a popular Storage Area Network (SAN) protocol, allowing organizations to consolidate storage into data center storage arrays while providing hosts (such as database and web servers) with the illusion of locally-attached disks.

Network striping (inverse multiplexing): To provide multiple delivery paths between the source and the destination. Aggregation of multiple, parallel network connections - stripes - may provide a way to increase bandwidth and lower latency [35]. The potential benefits obtainable by network striping are (1) multiple low cost network interfaces may provide a cost-effective alternative to a single expensive high-speed network interface, (2) multiple interfaces may provide higher performance than can be achieved by using the current single interface technology, and (3) the reliability of network subsystems may be improved by using multiple stripes.



(The GridFTP server in striped configuration uses multiple nodes to fully utilize the network for individual transfers)

Grid computing*:* This refers to the computational and networking infrastructure that is designed to provide pervasive, uniform and reliable access to data, computational, and human resources distributed over wide area environments [4].

The Globus Toolkit: It was developed within the Globus project provides middleware services for Grid computing environments.

BitTorrent client: It is any program that implements the BitTorrent protocol. Each client is capable of preparing, requesting, and transmitting any type of computer file over a network, using the protocol.