A Modification to Swifter Start Algorithm for TCP Congestion Control

Conference Paper · October 2005

CITATION
1

READS
78

1 author:

Ahmed Elbery
National Telecommunication Institute, Egypt
15 PUBLICATIONS **48** CITATIONS

SEE PROFILE

A Modification to Swifter Start Algorithm for TCP Congestion Control

Ehab A. Khalil*, Ahmed A. Elbery**, Bahnasy A. Nosair**, Ibrahim Z. Morsi *
*Minufiya University, Shebin El-Kom, EGYPT, dr.ehab@mailer.menofia.edu.eg ,
** National Institute of Communications, Cairo, EGYPT Ahmed_elbery@yahoo.com

Abstract

The *TCP* congestion control algorithm has become key factor influencing the performance and behavior of the Internet. This paper presents modification to what is called Swifter Start algorithm which has been designed by *BBN* Technologies [1]. The simulation results are promising enough.

Keywords: *TCP*, congestion control, Swift Start algorithm, large bandwidth-delay product, simulation comparison.

1. Introduction

Today as well as tomorrow the Transmission Control Protocol (*TCP*) is dominant transport protocol in the Internet [2], that is due to the rapid growth of the Internet and widespread use of the *TCP/IP* protocol. Despite the fact that *TCP* is a stable and mature protocol and has been well tuned over years, small changes in its congestion control are still in progress to avoid unnecessary retransmit timeouts, to reverse the responses to reordered packets, to allow viable mechanisms for corruption notification, etc., without altering the fundamental underlying dynamics of *TCP* congestion control [3].

Through the last decade the congestion avoidance and control mechanisms in *TCP* have evolved significantly. Previous and current implementations of *TCP* include Jacobson's classical slow start algorithm for congestion avoidance and control [4], as well as more additions as in fast retransmit, fast recovery and fast TCP [5-13].

Using Slow Start algorithm on a common link (low-delay, modest-bandwidth) is suitable because it takes only a few round-trips to correctly estimate and begin transmitting data at the available capacity. However, over long delay links (e.g. satellite links) or a high bandwidth links (several hundred megabits per second), or both, it may take several seconds to complete the first slow start. Indeed, for transactional applications such as *HTTP*, the *TCP* connection may have transferred all the data to be sent before completing the slow start, without ever achieving its potential transmission rate [1].

To improve the *TCP* connection performance, the first problem is to quickly and correctly estimating the capacity so as to enable *TCP* connection to send more data early, but simply estimating the available capacity is not enough. The second problem is to still do not send more data than its relevant share of the available capacity, to avoid overwhelming the router in the connection path.

There are many approaches were proposed to solve the *TCP* problems over high delay-bandwidth networks such Increasing the Initial Window Size [14, 15], Explicit Congestion Notification (*ECN*) [16, 17]. Explicit Control Protocol (*XCP*) Congestion Control for High Bandwidth-Delay Product Networks [18], Quick-Start for *TCP* and *IP* [19], Swift Start *TCP* and others.

Swift Start is a new congestion control algorithm was proposed and designed by *BBN* technologies [1] to increase the performance of *TCP* over high delay-bandwidth product networks by improving its start up, [20]. Swift Start tries to solve the congestion control problems mentioned above by using packet pair and pacing algorithms together [21, 22]. However, the Swift Start has some drawbacks were explicit in [1, 20].

This paper proposes a modification to the Swift Start algorithm to overcome these drawbacks. By using computer's simulation program we've compared the traditional Swift Start and the modified Swift Start, the results show improving in the connection start up by quickly estimating the available bottleneck rate in the connection paths.

2. TCP Congestion Control

To avoid network congestion, when a *TCP* connection is opened and data transmission starts, *TCP* should use congestion control algorithm to probe the network and determine the available capacity over the connection's path. The common used algorithm for *TCP* congestion control is Slow Start and Congestion Avoidance. The slow start has a lot of modifications such as fast retransmit and fast recovery. Also many other algorithms were designed such as Increasing Initial Window, Explicit Congestion Notification (*ECN*), explicit Control Protocol (*XCP*), and Quick-Start for *TCP* and *IP*.

2.1 Slow Start and Congestion Avoidance:

The slow start uses the congestion window "*CWND*". When a new connection is established with a host, the congestion window is initialized to one segment. Each time an acknowledgement (*ACK*) is received; the *CWND* is increased by one segment. The sender can transmit up to the minimum of the *CWND* and the

advertised window "*RWND*" which is related to the amount of available buffer space at the receiver for this connection.

At some point the capacity of the Internet links can be reached, and an intermediate router will start discarding packets. This informs the sender that its congestion window has gotten too large When the congestion window reaches a certain threshold *SSTHRESH* the congestion avoidance should start to avoid occurrence of congestion. Different additions as in fast retransmit, fast recovery and fast TCP [5-13] were added to slow start and congestion avoidance. Slow start was also modified and proposed for a Higher-Speed *TCP* to improve its performance over high speed links as explained in [23].

2.2 Increasing the Initial Window Size [14, 15]

With the increasing *TCP's* Initial Window (*IW*), an increase in the permitted upper bound for the *TCP's IW* from one segment to between two to four segments. The upper bound for the *IW* is given more precisely as:

IW = min (4 x MSS, max (2 x MSS, 4380 bytes)).…..... (1)

Obviously, the upper bound for the *IW* size is based on the Maximum Segment Size (*MSS*) [14].

This change applies to the *IW* of the connection in the first Round Trip Time (*RTT*) of transmission following the TCP three-way handshake. Neither the *SYN/ACK* nor its *ACK* in the three-way handshake should increase the *IW* size above that outlined in equation (1).

2.3 Explicit Congestion Notification in *IP* [16,17]

Using *ECN* network provides a congestion indication to the hosts by marking packets rather than dropping them. *ECN* uses two flags in the *IP* header with. Bit 6 in the IPv4 Type of Service (*TOS*) is used as *ECN*-Capable Transport (*ECT*) bit. This would be set by the data sender to indicate that the end-points of the transport protocol are ECN-capable. And a bit 7 in the IPv4 *TOS* is designated as the Congestion Experienced (*CE*). The *CE* bit would be set by the router to indicate congestion to the end nodes. When a host receives a *CE* message it will react with it by reducing its congestion window.

2.4 eXplicit Control Protocol (*XCP*) [18]

In *XCP* each packet carries a congestion header which is used to communicate a flow's state to routers and feedback from the routers on to the hosts. The congestion header consists of three fields, the first field ***H_cwnd*** (Header congestion window) is the sender's current congestion window, the second field ***H_rtt*** (Header *RTT*) is the sender's current *RTT* estimate. These are filled in by the sender and never modified in transit. The third field is ***H_feedback*** which takes positive or negative value and is initialized by the sender. Routers along the path modify this field to directly control the congestion windows of the sources. Based on the difference between the link bandwidth and its input traffic rate, the router can decide wither the senders sharing that link should increase or decrease their congestion windows. And informs them by how much they should increase or decrease their congestion windows, by annotating the congestion header of data packets. Feedback is divided between flows based on their *CWND* and *RTT* values so that the system converges to fairness. A more congested router later in the path can further reduce the feedback in the congestion header by overwriting it. Ultimately, the packet will contain the feedback from the bottleneck along the path. When the feedback reaches the receiver, it is returned to the sender in an acknowledgment packet, and the sender updates its *CWND* accordingly.

2.5 Quick-Start for *TCP* and *IP* [19]

By using Quick-Start, the end hosts negotiate the initial sending rate in packets per second in a Quick Start Request option which is add in the IP header of the initial *TCP SYN* or *SYN/ACK* packet. Each router in turn could approve the specified initial rate, reduce the specified initial rate, or indicate that nothing above the default initial rate for that protocol would be allowed. The Quick-Start mechanism also can determine if there are routers along the path that do not understand the Quick Start Request option, or have not agreed to the initial rate described in the option.

Quick-Start is designed to allow *TCP* connections to use high initial windows in circumstances when there is significant unused bandwidth along the path, and all of the routers along the path support the Quick-Start Request.

3. Swift Start Algorithm [1]

The goal of this algorithm is to quickly determine the path bottleneck capacity and so the congestion window by employing packet pair algorithm [21], and using pacing [20] to spread out the congestion window over *RTT*. As mentioned in [1], with this approach, the *TCP* connection starts with four-segment which are sent in burst. However, when the acknowledgements of the segments are received, the sending TCP performs the packet pair algorithm to calculate the bottleneck capacity as follow:

$$BW = \Delta t \times SegSize$$
$$Capacity = BW \times RTT$$

Where Δt is the time delay between the arrival time of the acknowledgment for the first and second segment (*t_ack2 - t_ack1*). Also the sending *TCP* uses pacing to spread the packets over the *RTT*. Figure 1 shows the network model with *RTT* = 0.47sec, bottleneck

bandwidth of 49.5 Mbps (*OC1*), and the hosts have received buffer of 65535 bytes. Figure 2 shows the sent sequence number versus time for transferring 10 Mbytes file using slow start *TCP* and Swift Start *TCP*. Figure 3 focuses on the start up of Figure 2.

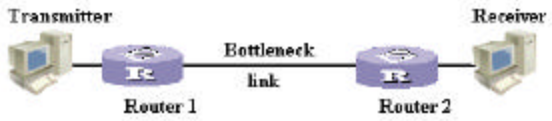

Figure 1 Network Model

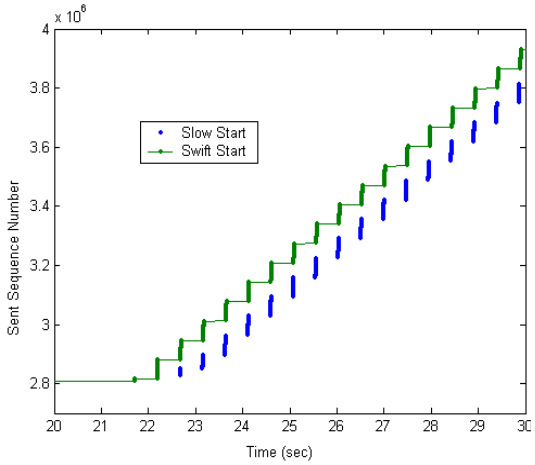

Figure 2 Sent Sequence Number for RTT = 0.47 sec

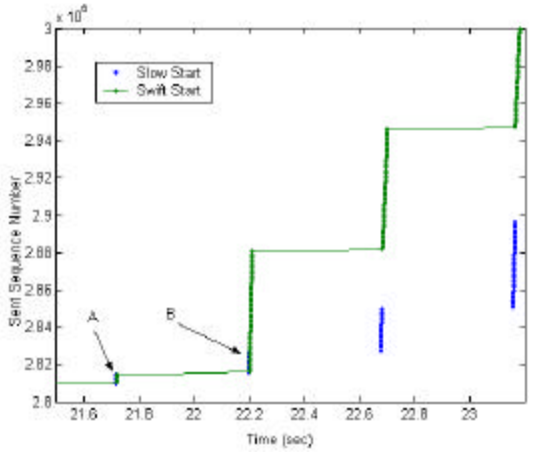

Figure 3 Sent Sequence Number for RTT = 0.47 sec

Figure 3 indicates that both algorithms are started the connection and data transmission at the same time, with *IW* of 4 segments at point "A", and at the beginning of the next *RTT* both are starting transmission of a new window, but at this time the window size for slow start is very small than that of Swift Start, therefore, Swift Start can not send more than the Flow Control Received Window (*RWND*) –Which is equal 64 Kbytes. However, the difference will continue increasing until the slow start *CWND* reaches its *RWND*, at this point both of them will send *RWND* bytes every *RTT* (i.e. the sending rate = *RWND / RTT*).
Figures 2, 3 confirm that the Swift Start algorithm improves the start up of the *TCP* connection.

3.1. Motivation

Swift Start faces many problems when combining with other techniques such Delayed Acknowledgment and acknowledgment compression.

a) Effect of Delayed Acknowledgment (*DACK*)

The majority of *TCP* receivers implement the delayed acknowledgment algorithm [24,25] for reducing the number of pure data less acknowledgment packets sent. A *TCP* receiver, which is using that algorithm, will only send acknowledgments for every other received segment. If no segment is received within a specific time, an *ACK* will send, (this time typically is 200 ms.). The algorithm will directly influence packet pair estimation, because the *ACK* is not sent promptly, but it may be delayed some time (200 ms). If the second segment of the packet pair arrives within 200 ms the receiver will send single *ACK* for the two segments instead of sending an *ACK* for each segment. Hence the sender can not make that estimation.

b) Effect of Acknowledgment Compression [26, 27]

Router that supports acknowledgment compression will send only one *ACK* if it receives two consecutive *ACK* messages of a connection within small interval. This will also affect the Swift Start and cause it to falsely estimate the path capacity.

c) Effect of ACK path:

It is well known that the traditional packet pair algorithm has a problem in which it assumes that the *ACK* path does not affect the delay between *ACKs*. But both *ACKs* may be subjected to different queuing delays in their path, which may causes an over or under estimate of the bottleneck capacity. However, to avoid congestion due to over estimation the Swift Start uses only a fraction of the calculated bandwidth; this fraction is determined by a variable a which indicates the rates between 1 and 8.

The following section presents the propose modification in packet pair algorithm to avoid the mentioned defects in traditional packet pair, and use the modification packet pair algorithm in the Swift Start.

3.2. Modification Packet Pair

The objective of this modification is to avoid error sources in the traditional packet pair algorithm. However the idea behind that is instead of time depending on the interval between the acknowledgments that may cause errors, the time

between the original messages will be calculated by the receiver when they arrive it, and then the receiver sends this information to the source when acknowledging it.

The sender sends its data in form of packet pairs, and identifies them by First/Second (*F/S*) flag. When the receiver receives the first message, it will record its sequence number and its arrival time, and it will send the acknowledgment on this message normally according its setting. When it receives the second one, it will check whether it is the second for the recorded one or not, if it is the second for the recorded one, the receiver will calculate the interval Δt between the arrival time of the second one and that of the first one:-

$$\Delta t = t\_seg1 - t\_seg2 \qquad \mu sec \ \ldots\ldots\ldots (2)$$

Where *t_seg1* and *t_seg2* are the arrival time of the first and second segments respectively. However, when the receiver sends the acknowledgment for the second segment, it will insert the value of Δt into the transport header option field. The sender's *TCP* will extract Δt from the header and calculate the available bit rate *BW*:

$$BW = \Delta t \times SegSize \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots (3)$$

Where *SegSize* is the length of the second segment.
If the receiver uses the *DACK* technique, it will record the first segment arrival time and wait for 200 ms, when it receives the second one it will calculate Δt and wait for new 200 ms, if it receives another packet pair it will calculate another Δt, whenever it sends an acknowledgment, it will send Δt in it.

By this way the error sources are avoided and the estimated capacity is the actual capacity without neither over estimation nor under estimation. So it is not needed to use only a fraction of the capacity like the traditional Swift Start.

4. Simulation and Results.

The simulation program has been done using the network model shown in Figure 1 which consists of transmitter and receiver hosts connected with two *CISCO* routers model 3640 with 5000 packet/sec forwarding rate and memory size 265 Mbytes through a bottleneck link. So, the *OPNET* modeler has been applied to implement the traditional and the modification Swift Start algorithms. By considering that the transmitter host sends 10 Mbytes file to the receiver host; both hosts have a huge *TCP* receiving buffer, to make the *RWND* as large as possible. However, using this model a comparison has been done between the transmission efficiency regular *TCP*, traditional Swift Start and modified Swift Start using different scenarios, with different *RTTs* and different link bandwidth, also with and without delayed acknowledgment, to see the effect of each of them.

4.1. Swift start versus slow start

Figure 4 shows the relation between the sent sequence numbers versus the time in case of low bandwidth connection which is typically 1.544 Mbps. The Figure indicates that the start up for Swift Start is better than Slow start although the small bandwidth. But at the second *RTT* the Swift Start starts sending with a high rate at point **A**, and then decreases this rate, that's because at the beginning of the second *RTT TCP* receives the acknowledgements by which it calculates the new sending rate, then subtracts the previous sending rate from the new calculated one, and the difference is sent in burst, then the reset of the window will be paced. This action also happened at the beginning of the third *RTT* at point B. The Slow Start curve in Figure 4 also shows at point C which corresponding to the fifth *RTT* that the sending rate is high and then it is decreases suddenly, that is because in the fourth *RTT* the amount of data sent was large in such away that the first group of the segments were sent immediately and the last group were queued and separated by some time, so their acknowledgments which trigger the *TCP* to send new message, arrived with time separation, so the segments are sent also with time separation.

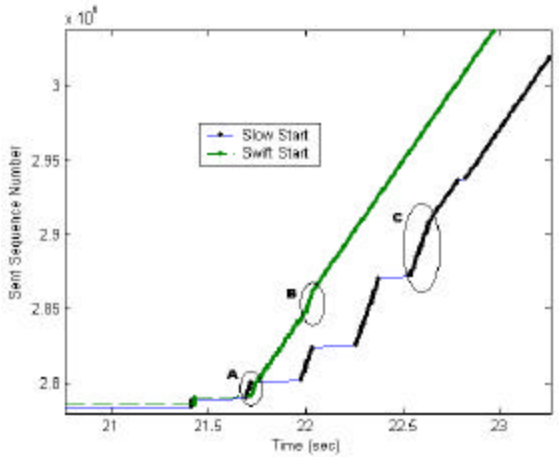

Figure 4 Sent Sequence Number for RTT = 348 ms and bottleneck rate is 1.5 Mbps.

4.2. Swift Start with Delayed Acknowledgment.

As mentioned above, that Delayed Acknowledgment technique will affect the operation of Swift Start, because the sender will receive a single acknowledgment for both the packet pair segments, hence it can not find the interval between these segments and so the congestion window will not be increased than the *IW* which is set to 4 segments.

Figure 5 shows the sent sequence number versus time for slow start and Swift Start, with delayed acknowledgment enabled, for *RTT*=274 ms and bottleneck bandwidth =49.5 Mbps. Figures 5, 6 show the Swift Start sends only 4 segments each *RTT*, so its sending rate is very low.

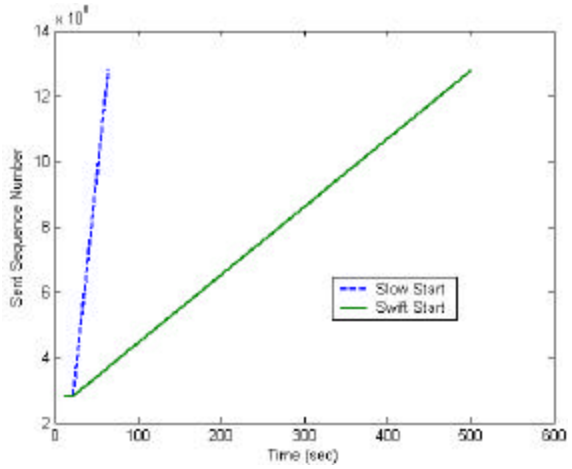

Figure 5 Sent sequence number versus Time, with Delayed Acknowledgment Enabled, for RTT = 274 ms and Bottleneck Bandwidth =49.5 Mbps

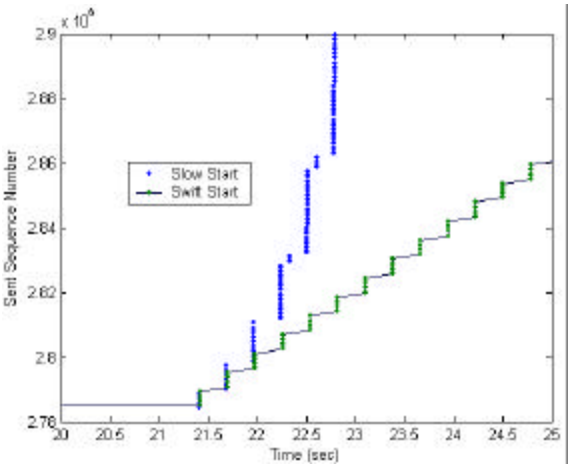

Figure 6 The First Few Seconds of the Connection

4.3. The Modification Swift Start

This subsection presents the modification Swift Start and uses it with the same connection to compare between it and the traditional Swift Start. Figure 7 shows the flowchart of the receiver *TCP*.

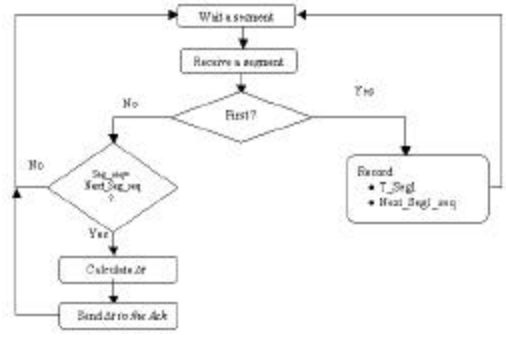

Figure 7 Flowchart for Modification Swift Start receiver

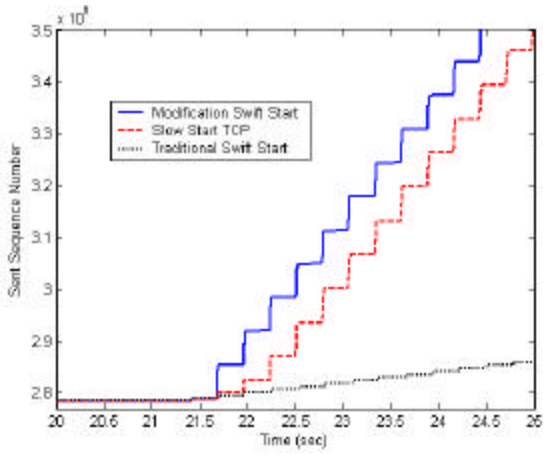

Figure 8 Sent Sequence Number versus Time for RTT=274 ms and Bottleneck Rate =49.5 Mbps.

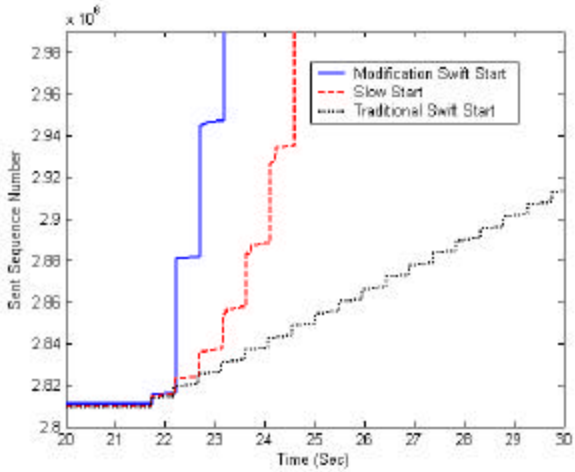

Figure 9 Sent Sequence Number versus Time for RTT=486 ms and Bottleneck Rate =49.5 Mbps.

Figures 8 and 9 show the sent sequence number versus time for *RTT*= 274 ms and *RTT*= 386 ms respectively, with *DACK* is enabled, and compares between the traditional and modification Swift Start. It is to be noted that in both of the connection, the performance of modification Swift Start better than the traditional slow start, that is because it quickly estimates the path capacity. Also, noted that the traditional Swift Start performance is very low because it does not receive *ACK* for each segment so it cannot estimate the available capacity and its *CWND* remain 4 segments.

5. Conclusion

It is well known that the Swift Start algorithm has a better performance in high delay-bandwidth product network and also in low delay moderate networks, but its estimation is not accurate. Also it has a problem when using Delayed Acknowledgment or acknowledges compression,

This paper presents a modification to the traditional Swift Start algorithm to overcome its drawbacks.

However, the modification Swift Start algorithm results have confirmed its succeed in improving the start up connection by quickly estimating to the available bottleneck rate in the connection path, and its performance does not affected when using Delayed Acknowledgment or acknowledges compression.

Currently the performance evaluation of the modification Swift Start algorithm in high delay bandwidth product networks is in progress.

6. References

1. C. Partridge, D. Rockwell, M. Allman, R. Krishnan, J. Sterbenz **"A Swifter Start For TCP"** *BBN Technical Report No. 8339, 2002*
2. J. Postl, **" Transmission Control Protocol,"** *RFC793, September 1981.*
3. Y. J. Zhu, and L. Jacob**, " On Making TCP Robust Against Spurious Retransmissions,"** *Computer Communications, Vol.28, Issue 1, pp.25-36, Jan. 2005.*
4. V. Jacobson, **"Congestion Avoidance and Control,"** *Proceedings of the ACM SIGCOMM '88 Conference, pp. 314–329, August 1988.*
5. C.Jin, D.X.Wei, and S. Low, **"Fast TCP Modification, Architecture, Algorithm, Performance,"** *Proc. of IEEE INFOCOM'04, 2004.*
6. C.Jin, D.Wei, and S.Low, **"Fast TCP Modification, Architecture, Algorithm, Performance,"** *Caltech CS Report Caltech CSTR:2003:010, 2003*
7. S. Floyd, and T. Henderson **"The new Reno Modification to TCP Fast Recovery Algorithm,"** *RFC 2582, April 1999*
8. M. Allman, W. Richard Stevens **"TCP Congestion Control,"** *RFC 2581, NASA Glenn Research Center, April 1999.*
9. V. Padmanabhan and R. Katz, **"TCP Fast Start: A Technique for Speeding up Web Transfers,"** *Globecom Sydney Australia, Nov. 1998.*
10. W. Stevens **"TCP Slow Start, Congestion Avoidance Fast Retransmit, and Fast Recovery Algorithms,"** *RFC 2001 Jan. 1997*
11. S. Floyd, **"TCP and successive fast retransmits,"**ftp://ftp.ee.lbl.gov/papers/fastretransmit.pps. *Feb 1995*
12. V. Jacobson, **"Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno,"** *Proceedings of the British Columbia Internet Engineering Task Force, July 1990.*
13. V. Jacobson **"Fast Retransmit, Message to the End2End,"** *IETF Mailing List , April 1990*
14. M. Allman, S. Floyd, C. Partridge **"Increasing TCP's Initial Window,"** *RFC 2414, September 1998.*
15. M. Allman, C. Hayes, and S. Ostermann. **"An Evaluation of TCP with Larger Initial Windows,"** *ACM Computer Communication Review, 8(3), July 1998.*
16. K. Ramakrishnan, S. Floyd, and D. Black, **"The addition of explicit congestion notification (ECN) to IP,"** *IETF, RFC3168, September 2001.*
17. K. Ramakrishnan, S. Floyd **"A Proposal to add Explicit Congestion Notification (ECN) to IP,"** *RFC 2481, January 1999*
18. D. Katabi, M. Handley, and C. Rohrs, **"Internet Congestion Control for High Bandwidth-Delay Product Networks,"** *ACM SIGCOMM' 2002, August 2002.*
19. A. Jain M. Allman S. Floyd **"Quick-Start for TCP and IP,"** *draft-ietf-tsvwg-quickstart-00, 31 May 2005*
20. Frances J. Lawas-Grodek and Diepchi T. Tran **"Evaluation of Swift Start TCP in Long-Delay Environment,"** *Glenn Research Center, Cleveland, Ohio October 2004.*
21. Ningning Hu, Peter Steenkiste **"Estimating Available Bandwidth Using Packet Pair Probing,"** *CMU-CS-02-166 School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213 September 9, 2002.*
22. Aggarwal, A.; Savage, S.; and Anderson, T., **"Understanding the Performance of TCP Pacing,"** *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies, pp. 1157-1165, vol. 3, 2000.*
23. S. Floyd, S. Ratnasamy, and S. Shenker, **"Modifying TCP's Congestion Control for High Speeds,"** 2002, Preliminary Draft. URL http://www.icir.org/floyd/papers/hstcp.pdf.
24. Afifi, H., Elloumi, O., Rubino, G **"A Dynamic Delayed Acknowledgment Mechanism to Improve TCP Performance for Asymmetric Links,"** *Computers and Communications, 1998. ISCC '98. Proceedings. Third IEEE Symposium pp.188 – 192, on 30 June-2 July 1998.*
25. D. D.Clark, **"Window and Acknowledgement Strategy in TCP,"** *RFC 813, July 1982*
26. Mogul, J.C., **"Observing TCP Dynamics in Real Networks,"** Proc. *ACM SIGCOMM '92, pp. 305-317, Baltimore, MD, August 1992.*
27. Zhang, L., S. Shenker, and DD. Clark, **"Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic,"** *Proc. ACM SIGCOMM '91, pp. 133-148, Zurich, Switzerland, August 1991.*