Improving the Performance of HTTP Persistent Connections

Eric Man-Kit Chan

28 December 2001

Abstract of Dissertation Entitled:

Improving the Performance of HTTP Persistent Connections

submitted by Eric Man-Kit Chan for the degree of MSc in Information Technology at The Hong Kong Polytechnic University in December 2001.

Traditionally, TCP was designed for generic purpose, regardless of the nature and characteristics of the applications. However, there are practical needs for improving the performance of TCP to suit particular types of traffic generated by different types of application. Unfortunately, all of these algorithms work well only when the nature of the traffic is consistent. However, in client/server applications, such as HTTP persistent connections that involve multiple request/response in a single TCP connection, these algorithms may not be able to function properly.

We have proposed *Cocktail* approach to improve the overall performance of HTTP persistent connections. The changes only require modifications to a TCP sender, and a TCP with these changes is interoperable with any existing TCP implementation. *Cocktail* is composed of several elements. Each element is assigned to tackle a specific problem caused by the interactions between HTTP persistent connections and existing TCP implementations. A number of simulation experiments have been conducted to evaluate the performance of SSR, RBP and CWV against that of *Cocktail*. The results of the experiments show that *Cocktail* generally performs better than the others in a low RTT environment do. Nevertheless, RBP works best in a high RTT environment.

Acknowledgments

I could not have finished this dissertation without the help and support for many others. I would like to thank Dr. Rocky Chang, my academic supervisor, for his professional advice, enlightenment and patience over the year. Thanks should also be given to Dr. Hong-Va Leung, my professional supervisor, for his technical suggestions in research methods. As usual, I am indebted to my parents for their understanding and assistance. Finally, I would like to express my deepest gratitude to my wife, Eva, for her love, and unconditional support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

1.1.1 From HTTP/1.0 to HTTP/1.1

Since 1990s, the growing popularity of World Wide Web (WWW) has made Hypertext Transfer Protocol (HTTP) dominated much Internet traffic and its role has become more and more important. In fact, HTTP is an application protocol running over Transport Control Protocol (TCP), which provides a reliable end-to-end data transfer channel between the web server and client's web browser. Currently, there are two major releases of HTTP: HTTP/1.0 and HTTP/1.1 [6]. Although both versions of HTTP use TCP for data transport, the way they use is quite different.

HTTP/1.0 opens and closes a new TCP connection for each HTTP request. A typical web page contains a Hypertext Markup Language (HTML) document and a number of embedded images. Since each of these images is an independent object, it has to be retrieved separately by a single HTTP request. As a result, many consecutive short-lived TCP connections are established for retrieving just one web page. Since establishing and closing TCP connections introduce many overheads and latencies, HTTP/1.0 is not efficient in handling web page retrievals.

HTTP/1.1, on the other hand, keeps the TCP connections open and reuse for the transmission of multiple HTTP requests. This technique is called *persistent connections*. Unlike HTTP/1.0, the use of persistent connections in HTTP/1.1 can reduce many latencies and overhead from re-establishing and closing a TCP connection for each HTTP request. This is the key enhancement to HTTP/1.0.

1.1.2 Interaction of TCP and HTTP Persistent Connections

Traditionally, TCP was designed for the use of generic purpose, regardless of the nature and characteristics of the applications [5]. However, there are practical needs for improving the performance of TCP to suit particular types of traffic generated by different type of applications, such as bulk data transfer (e.g. *FTP* program) and interactive data transfer (e.g. *telnet* program). So a number of algorithms have been developed to fulfill their specific requirements.

For instance, the slow-start and congestion avoidance algorithms proposed by Jacobson [12] have been designed to cater for bulk data transfer. The algorithms are used to prevent the sender from injecting excess numbers of segments into the network that may overflow the queue of intermediate router, hence increasing the efficiency of bulk data transfer. Another example is Nagle algorithm [16]. Nagle algorithm has been designed to cater for interactive data transfer. The algorithm is used to minimize the overhead of transmitting short data segments.

Unfortunately, all of these algorithms work well only when the nature of the traffic is consistent, i.e. either pure bulk data or pure interactive data transfer. However, in client/server applications, such as HTTP/1.1 that involves multiple request/response in a single TCP connection, these algorithms may not be able to function properly [15]. Under certain circumstance, using these algorithms can even degrade the overall performance. Therefore, the purpose of this study is to investigate the impact on the performance of HTTP when HTTP persistent connections interact with existing TCP implementation.

1.2 Objectives

There are two main objectives for the study:

- to study the impact of TCP congestion avoidance and control algorithms on HTTP persistent connections; and
- to propose modifications on existing TCP implementation to improve the performance of HTTP persistent connections. The proposed modification should be able to work transparently without modifying any existing applications and be compatible with existing TCP implementations.

1.3 Organization

The study consists of six chapters. Chapter 1 is an introductory part. Chapter 2 contains an overview of TCP and outlines its basic operation, including establishing and terminating of TCP connections. This chapter also outlines the existing TCP avoidance and control algorithms, the operation of HTTP, and the key difference between HTTP/1.0 and HTTP/1.1. Chapter 3 describes the problems of HTTP persistent connections running on existing TCP implementation. Prior works in solving these problems are also compared and evaluated in this chapter. Modifications to existing TCP implementation to improve its performance of HTTP persistent connections are proposed and explained in Chapter 4. Chapter 5 describes the details and the findings of experiments on a simulation platform for evaluating the proposed solutions. The final chapter, Chapter 6, concludes the whole study and suggests areas for further study.

Chapter 2

Overview

Before going into the details of the problems of HTTP persistent connections running on existing TCP implementation, we must understand the basic operation mechanisms of TCP. The chapter is an overview of the basic operation of TCP and HTTP. If you are familiar with the operations of TCP and HTTP, you may skip this chapter and continue the reading from the next chapter.

2.1 Basic Operation of TCP

2.1.1 Connection Establishment and Termination

Since TCP is connection-oriented, data exchange between two parties is not possible unless a connection between two parties is established. TCP adopts three-way handshake to establish a connection [21] and its details are as follows.

The three-way handshake is always initiated on the client side. To establish a connection, as shown in Figure 2.1, client should send a SYN segment, called Segment 1, to the server with an initial sequence number (ISN). When the server receives the SYN Segment 1 from the client, it sends its feedback with its own SYN segment containing the server's initial sequence number.

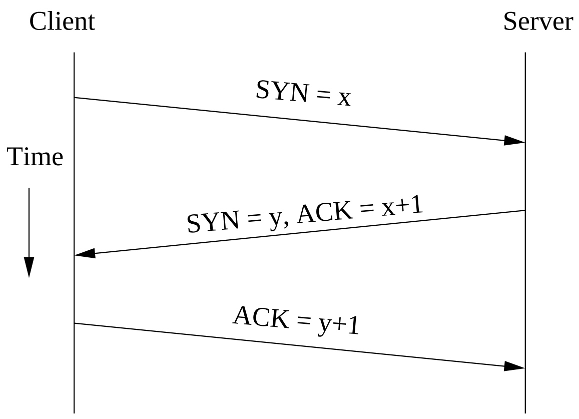

Figure 2.1: TCP Connection Establishment

At the same time, it sends an acknowledgement to the client's SYN by Segment 2, adding one to the client's ISN. Then the client must respond to the server's SYN segment, Segment 2, by acknowledging the server's ISN plus one. This is named as Segment 3. The connection is completed by such three-way handshake at this point.

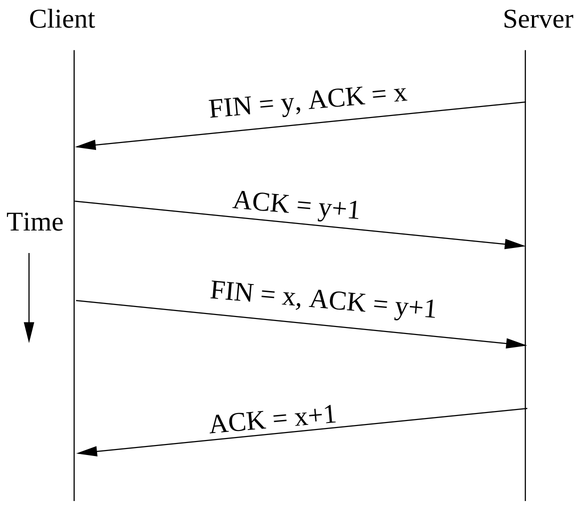

Figure 2.2: TCP Connection Termination

While establishment of TCP requires three segments, termination needs four. Individual connection has to be closed independently due to the full-duplex nature of TCP. Either party can send a FIN segment when the data transfer is completed. For instance, if the server wants to terminate the connection, it should send a FIN segment to the client, as shown in Figure 2.2. The latter should then respond by sending an ACK segment of the received sequence number plus one. Meanwhile, a FIN segment is sent by the client. Finally, the server returns an ACK segment with the received sequence number plus one and the TCP connection is closed.

2.1.2 TCP Transmission Policy

TCP transmission policy of TCP comprises error control procedures and flow control mechanism. The former uses go-back-N and the latter uses sliding window.

Go-Back-N

The idea of go-back-N is that if the segments are out-of-order, the client will send acknowledgements for the same highest in-order sequence number. In the other words, the client acknowledges the server the highest in-order sequence number segment it receives.

In case of congestion between the network linking the client and the server, there will be packet loss which will be signaled by a timeout and receipt of duplicate ACKs. The server will assume the segment with the sequence number immediately after the highest acknowledged by the duplicate ACKs received is loss. It will then retransmit those missing segments it assumes.

Sliding Window

The purpose of sliding window is to allow the server to send a number of data segments simultaneously without having to wait for ACKs. The size of the *window* constrains the maximum amount of data that can be sent at one time. Basically, the sliding window mechanism works like this:

1. The server transmits all new segments in the window and wait for acknowledgements.

2. When the acknowledgment arrives, the window slides and its size is set to the value advertised in the acknowledgement.

3. The server retransmits the segments that have not been acknowledged for some time. When the acknowledgment arrives, it causes the window slides and the transmission continues from the segment following the one transmitted last.

Figure 2.3 is a diagrammatic illustration of the mechanism. We assume that the web server gives response by transmitting six full-size segments upon receiving a short HTTP request messages from a web browser. Suppose the buffers for both client and server is 4096-byte and the size of the window four segments. The server is therefore limited to send at most four segments at the same time. The following will occur at different time:

1. At T_0, the web server writes 4096-byte of data into the sender buffer.

2. At T_1, TCP transmits the first four segments.

3. At T_2, the segments are correctly received on the client side. The client acknowledges the first two segments. Since the receiver buffer is full, the advertised window is zero. The server must stop sending data until the application process on the client side removes some data from the buffer.

4. At T_3, the client acknowledges the third and fourth segments. Since the application process on client side read 1024-byte of data, it will advertise a window of 1024 bytes starting at the next byte expected.

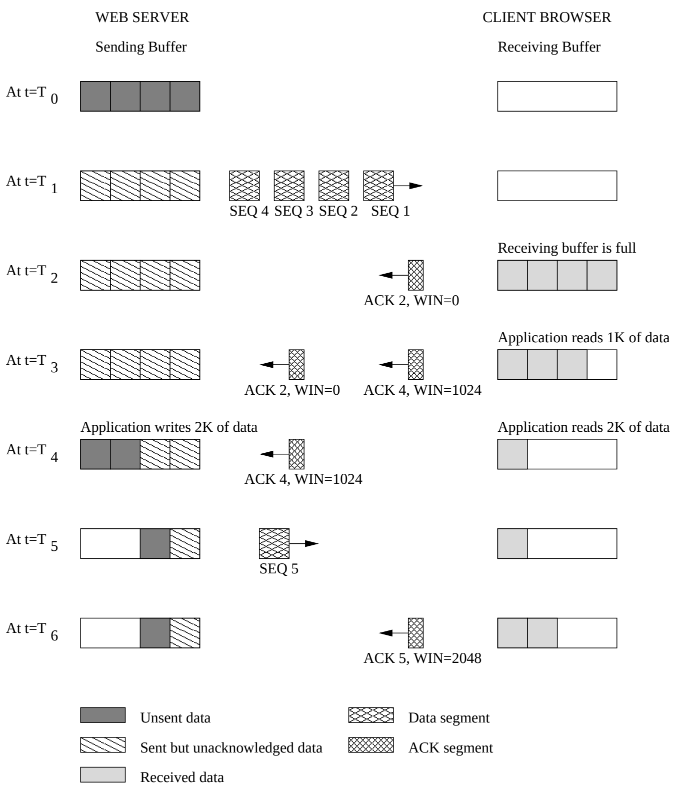

Figure 2.3: TCP Transmission Policy

5. At T_4, the first ACK segment is received by the server, then frees up 2048 bytes of space in sender buffer. The web server writes 2048-byte of data into the sender buffer. However, the ACK segment just received specifies that the advertised window is zero. So the sender TCP cannot transmit any data. Meanwhile, the application on client side reads 2048-byte of data.

6. At T_5, the server receives the second ACK segment. Then 2048 bytes of space are free in the sender's buffer. The ACK segment just received specifies that the advertised window is 1024. Thus, the sender TCP is able to transmit another 1024-byte of data.

7. At T_6, upon receiving the data segment, the sender TCP acknowledges

the data segment and advertise a window of 2048.

2.1.3 Slow-Start and Congestion Avoidance Algorithms

To understand more about the transmission policy, two important algorithms should be discussed: *Slow-Start* and *Congestion Avoidance* [20]. The purpose of these algorithms is to enable the TCP server to govern the amount of outstanding data going into the network. It operates by three parameters: congestion window (*cwnd*), receiver's advertised window (*rwnd*) and slow-start threshold (*ssthresh*).

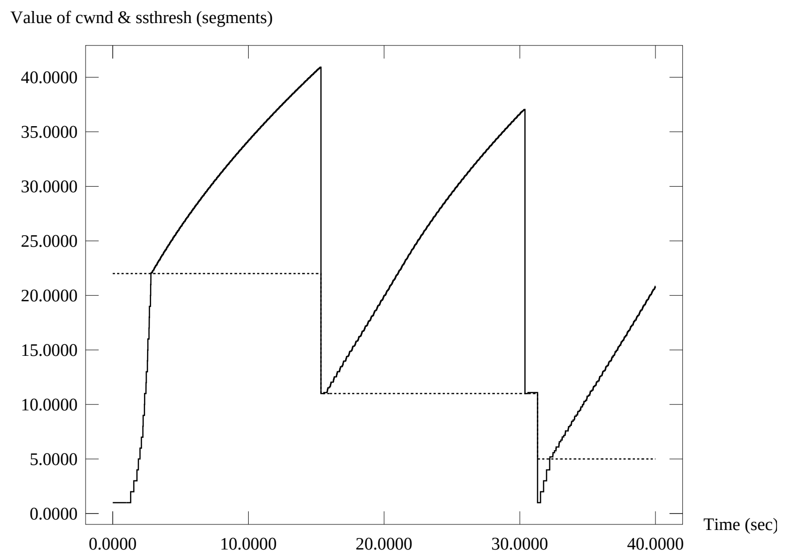

Figure 2.4: Slow-Start and Congestion Avoidance Algorithms

The minimum value of *cwnd* and *rwnd* limits the size of data transfer. *cwnd* is the maximum amount of data the server can inject into the network prior to receiving ACKs, while *rwnd* is the maximum amount of outstanding data on the client side. On the other hand, *ssthresh* is a critical value of *cwnd* to determine which algorithm, the slow-start or congestion avoidance algorithm should be used as the data transmission controller. Slow-start is used when *cwnd* < *ssthresh*, while congestion avoidance is used when *cwnd* > *ssthresh*. When *cwnd* and *ssthresh* are equal, either slow start or congestion avoidance is adopted.

Slow-Start

To avoid network congestion, when the TCP starts to transmit the data into the network, it should explore the uncertain network condition slowly and find out the capacity available. Slow-start is then used at such initial data transfer stage or after repairing loss detected by the retransmission timer for this purpose.

When TCP connection is just established or packet loss is detected by the retransmission timer, the initial value of *cwnd* is initialized to one, i.e. the maximum of data segment that can be injected into network is limited to one. When an ACK segment is received, TCP doubles the value of *cwnd* to two. Now TCP can send at most two data segments into the network. During slow-start, TCP increases the value of *cwnd* exponentially for each incoming ACK until *ssthresh* is hit. When *cwnd* exceeds *ssthresh*, slow-start ends and congestion avoidance algorithm takes over.

The minimum initial value of *cwnd* commonly adopted is one MSS. However, TCP can use a larger initial window in a non-standard and experimental TCP extension as defined in RFC 2414 [2]. In that case, as long as the total size of the segments does not exceed 4380 bytes, a server may use a three or four MSS as initial value of *cwnd*. Regarding the *ssthresh*, its initial value can be arbitrarily high, probably as high as the advertised window. In fact, a higher initial value of *cwnd* can affect the start up performance of TCP. This issue will be further discussed in the later chapter.

Congestion Avoidance

During congestion avoidance phase, *cwnd* is incremented by one for each incoming ACK [3]. One formula commonly used to update *cwnd* during congestion avoidance is in Equation 2.1:

$$cwnd_{new} = cwnd_{current} + \frac{1}{cwnd_{current}} \tag{2.1}$$

When segment loss is detected by receipt of duplicate ACK, *ssthresh* is set to:

$$ssthresh = \max(\frac{cwnd}{2}, 2MSS) \tag{2.2}$$

Upon a timeout, *cwnd* will be set to one since it is well below *ssthresh*, and slow-start will be clicked started again.

2.1.4 Fast Retransmit and Fast Recovery Algorithms

As discussed before, the client has to send duplicate ACK when an out-of-order segment is arrives. However, the duplicate ACK does not tell whether it comes from a lost segment, simply a reordering of segments, or replication of ACK or data segments by the network. However, if three or more consecutive duplicate ACK are received, it is a piece of strong evidence that the segment is lost. In this case, the server will immediately retransmit the segment under fast retransmit, without having to wait for timeout.

If the server goes back to slow-start to retransmit the missing segment, it will suspend the data flow (or the ACK-clock). To get rid of this, fast recovery algorithms can help by turning to use congestion avoidance algorithm to take over the retransmission. The operation of the fast retransmit and fast recovery algorithms is as follows:

1. When the third duplicate ACK is received, the *ssthresh* is set to not more than the value given in Equation 2.2.

2. The server then retransmits the lost segment and sets the *cwnd* to *ssthresh* plus three MSS. The congestion window is intentionally en-

larged by the number of segments that have left in the network and which the receiver has buffered.

3. The server increases *cwnd* by one MSS in each additional duplicate ACK. This can inflate the congestion window in order to reflect the additional segment that has left in the network.

4. Then the server transmits a segment, if permitted by the new value of *cwnd* and the receiver's advertised window, *rwnd*.

5. Upon receiving the next new data ACK, the server sets *cwnd* to *ssthresh* as calculated in the first step. This is known as *deflating* the window.

The new data ACK should be the induced by the retransmission in the first step, one RTT after the retransmission. This ACK should also acknowledge all the segments sent between the lost segment and the third duplicate ACK

2.1.5 Modifications on TCP Fast Recovery

Unfortunately, in the same window, fast recovery is not efficient in recovering from multiple segment losses [10]. To make TCP's recovery from such multiple losses more rapid, it needs some further modification algorithms known as TCP NewReno [7]. The trick is that it operates by updating the fast recovery algorithm to use information given by *Partial ACK*, which only covers new data instead of all outstanding if there is segment loss, to trigger retransmission of segments. It does not have to rely on the availability of Selective Acknowledgments (SACK). The NewReno modification is very similar to RFC 2581 [3] except that

- Another variable called *recover* is added; and

- Responses to partial or new ACK are different from that of RFC 2581.

TCP NewReno defines a *fast recovery procedure*. It starts when three consecutive ACKs are received. It ends under two circumstances: (1) when there is a retransmission timeout; or (2) when an ACK that acknowledges all the data up to and including the outstanding data when the fast recovery procedure starts. The procedure is basically the same as in Section 2.1.4 except Step 1 and Step 5:

- Step 1: Upon receiving the third duplicate ACK but the sender is not in the fast recovery procedure, the server sets *ssthresh* to no more than the value computed by Equation 2.2. On the other hand, the highest sequence number transmitted in the variable *recover* is then recorded.
- Step 5: When an ACK that acknowledges new data arrives, this ACK could be the acknowledgement that the retransmission from Step 2, or a later retransmission induces. Two cases will occur:
    1. When the ACK acknowledges all data up to and including *recover* (see Step 1), all the intermediate segments transmitted between the original transmission of the lost segment and the receipt of the third duplicate ACK are acknowledged. Then the TCP sets the *cwnd* to $min(ssthresh, FlightSize{+}MSS)$ where the $FlightSize$ refers to the amount of data outstanding in this step when the fast recovery is existed. Finally it exits the Fast Recovery procedure; or
    2. When the ACK DOES NOT acknowledge all the data up to and including *recover*, the ACK is only partial and *partial window deflation* has to be done. The server retransmits the first segment that is not acknowledged and deflates the congestion window by the amount of new data acknowledged. One MSS is then added back. If the new *cwnd* permits, the server sends a new segment. This ensures approximately *ssthresh* amount of data remains out-

standing when the fast recovery ends. Instead of exiting the fast recovery procedure, the server will repeat Step 3 and Step 4 as mentioned in Section 2.1.4.

2.1.6 Bandwidth-Delay Product

Bandwidth-Delay product is used to measure the capacity of a link [21]. A link with bandwidth B (measured in bits/second) and delay D (measured in seconds) has a bandwidth product of $B \times D$. Comparing it with the number of segments on a link, we can see how efficient the link is used.

For instance, a number of segments are acknowledged and transmitted immediately without delay, it takes about $2D$ seconds for a round trip, i.e. from putting a segment on the link and receiving the ACK. Thus approximately $2BD$ segments can be injected into the link in that period. These approximately $2D$ seconds is known as the Round Trip Time (RTT). Suppose the send window is smaller, the server is sending less than the full capacity, resulting in inefficient use of the pipe. In practice, the bandwidth is limited to window-size (in bits) and RTT (in bits/second).

2.1.7 Delayed Acknowledgments

Delayed acknowledgments are very useful in frequent data exchange environment [4]. To minimize the number of segments injected into in the network, the client can defer sending out ACKs at intervals. The delayed ACKs can be piggy-backed on a data segment in the opposite direction. They can also carry updated information, tell the server that the client has read the previous data and the server can send larger bulk of data. The server utilizes the ACK from the client to control the number of segment injected into the network. Unfortunately, it is contrary to the desire to minimize the number of segments in the network. So the client should send a minimum one ACK per two full-sized segments it receives.

2.1.8 Nagle Algorithm

Nagle algorithm [16] attempts to minimize the number of segments in the network, hence ease the traffic load and prevent overloading the routers and switches. It is a very effective tool that most TCP implementations contain this algorithm.

Basically, the problem is: in the early days of Internet, the links, routers and servers were relatively slow. Much of the network traffic is terminal traffic comprising short segments which carried only a single keystroke. The network was congested by these small but frequent segments. So Nagle has proposed the algorithm, which states that when a TCP connection has outstanding data that has not yet been acknowledged, segment size less than 1 MSS cannot be sent until the outstanding data is acknowledged.

2.2 Basic Operation of HTTP

HTTP has been used in the World Wide Web since 1990. The first version of HTTP, known as HTTP/0.9, was a simple protocol for Web documents and raw data transfer. HTTP/1.0 as defined by RFC1945, improved the protocol by allowing messages to be in the format of MIME-like messages, containing meta-information about the data transferred and modifiers on the request/response semantics. HTTP/1.0 is an application level protocol and uses TCP for data transport. The client, typically a Web browser, establishes a connection to the server and issues a request. The server then processes the request, returns a response, and closes the connection.

Typical Web pages contain a Hypertext Markup Language (HTML) document and a number of embedded images. Each of these images is an independent object. The common behavior of a client is fetching the base HTML document, and then immediately fetching the embedded objects referenced in the document. As HTTP/1.0 uses a new TCP connection for each HTTP request, the transmission of a page with HTML content and embedded objects involves many short-lived TCP connections. For example, a request of a typical HTML document that contains eight images will involve nine TCP connections – one for the base HTML document, and eight for the images. Clearly, this is not an efficient mechanism to retrieve Web documents.

As mentioned earlier, HTTP/1.0 opens and closes a new TCP connection for each operation. A TCP connection is established by a three-way handshake. Since most Web objects are small, it induces significant overhead in establishing and closing a TCP connection. Furthermore, when a TCP connection is first opened, the server employs slow start as discussed before. Slow-start uses the first several data segment to probe the network condition to determine the optimal transmission rate. However, because Web objects are usually small, most objects are transferred before their TCP connections complete the slow start phase and therefore fail to exploit the available bandwidth. In order to reduce the number of connections established which result in latencies and processing overheads, HTTP/1.1 introduces the use of persistent connections.

2.3 HTTP Persistent Connections

HTTP/1.0 establishes a new TCP connection for each HTTP request, resulting in many consecutive short-lived TCP connections. To resolve the problem, HTTP/1.1 introduces the persistent connection mechanism. HTTP/1.1 specifies that TCP connections should remain open until explicitly it is closed by either party. It allows multiple request/response interactions to take place before the connection is closed. Therefore, the latencies and overhead from closing and re-establishing TCP connections can be reduced.

Chapter 3

Problems with HTTP Persistent Connections

In the previous chapter, we have explained the basic operation of TCP and HTTP/1.0. We have also discussed the differences between HTTP/1.0 and HTTP/1.1, as well as some of the rationale behind these changes. In this chapter, we will discuss the problems with HTTP persistent connection, the default feature of HTTP/1.1 and evaluate the previous attempts by other studies to solve the problems.

3.1 Restart of Idle TCP Connection

As mentioned earlier, HTTP/1.1 uses persistent connections to reduce the overhead from closing and re-establishing TCP connections by re-using a single TCP connection across multiple request/response transactions. Since HTTP/1.1 is an application protocol which makes use of TCP for end-to-end data transmission, its performance is affected by the implementation of TCP. In fact, TCP congestion avoidance mechanism works well for a single burst of data. However, it does not match large but intermittent bursts of traffic like HTTP/1.1.

In a TCP connection, the congestion control scheme uses ACKs from the client to dynamically calculate reasonable operating values for TCP parameters, which in turn determine when and how much the server can pump into the network. In particular, when a TCP connection has outstanding data that has not yet been acknowledged, another data segment cannot be sent until the outstanding data is acknowledged. The beauty of this algorithm is that it is self-clocking: the faster the ACKs return, the faster the data is sent.

The problem is that: typically, there is a short idle period between two consecutive HTTP requests. When the connection is idle, the flow of data segments as well as ACKs suspends. Since TCP depends on ACK-clocking for flow control, the idle periods also disturb the flow control mechanism. When a burst of data is sent after an idle period, depending on the implementation, TCP may or may not enforce slow-start algorithm by re-initializing the congestion parameters (*cwnd* and *ssthresh*).

If slow-start is enforced after the idle period, the server is conservative towards the network but the optimal transmission rate cannot be maintained since a new slow-start is performed. On the other hand, if the slow-start is not enforced, the network condition throughout the idle period is unchanged. The optimal transmission rate can be maintained for the coming HTTP requests. However, if the network during that period unpredictably turns very congested, the situation will be totally different. As the value of *cwnd* originally maintained becomes too large rather than optimal, TCP will pump excessive number segments into the network that the intermediate routers cannot handle, network congestion arises. Therefore, the server will suffer from an excessive number of retransmissions. Eventually the performance will be substantially degraded.

3.2 Related Work

The problem of restarting idle TCP connection was addressed by a number of studies and a number of solutions have also been proposed.

3.2.1 Slow-Start Restart

Currently, slow-start restart (SSR) is the most popular method to handle idle TCP connections. It was proposed by Jacobson [12] who stated that a TCP should use slow-start to restart transmission after a relatively long idle period. To be more specific, if a TCP connection is idle for more than one RTT, *cwnd* is reduced to one before the next transmission commences. The ultimate purpose of this mechanism is to restart the ACK clock. However, problems arise when this interacts with HTTP/1.1.

As described, when the idle TCP connection is restarted, slow-start restart cannot maintain the optimal transmission rate which established before the connection became idle. In addition, very often, the pattern of HTTP requests is unpredictable. Using timer to detect an idle TCP connection and decide whether to decrease *cwnd* fails to deflate *cwnd* in HTTP persistent connections since prior to the timeout, the user may have already made another HTTP request [11]. This will reset the timer and it fails to test for idle connections and leads to very large *cwnd*.

3.2.2 Rate-Based Pacing

When the TCP has been idle for a relatively long period, instead of sending a large burst of back-to-back segments or restart the congestion control, the rate-based pacing (RBP) tries to transmit a controlled number of evenly spaced segments at a conservation rate [22]. Immediately after the first ACK is received, RBP discontinues because the ACK-clock has been re-established.

3.2.3 Congestion Window Validation

With HTTP persistent connections, the TCP is sometimes idle or application-limited. Under these circumstances, the invalidation of the congestion window will occur and the congestion window cannot truly reflect the status of the network. Congestion Window Validation (CWV) [8] is a simple modification to TCP congestion control algorithm to tackle this problem. By using CWV, if the connection is idle, the TCP decays a portion of the congestion window for every RTO.

In addition, during TCP's slow-start or congestion avoidance phases, the growth of the congestion window will make the congestion window inaccurate in reflecting the network condition in application-limited period. This is because the amount of data injected into the network is less than amount allowed by the *cwnd*. CWV also caters for the problem: When the *cwnd* is fully utilized, i.e. the congestion window is valid, *cwnd* will grow as usual. When the *cwnd* is still sending data but the amount is less that allowed by *cwnd*, its growth is prohibited.

3.3 Evaluation of Different Approaches

In the previous section, a number of proposed solutions to solve the problem of restarting idle TCP connection have been explained. In this section, we are going to evaluate and compare their performance based on the results of the simulation experiments.

3.3.1 Methodology

The purpose of the experiment is to test how different proposed solutions mentioned in the previous section affect the TCP's performance when the TCP connection is idle and then restarted.

The evaluation of the proposed solutions is conducted by a simulation tool. *ns* [1] has been chosen for this purpose as the experiment can be done in a more controlled environment. The network topology is shown in Figure 3.1.

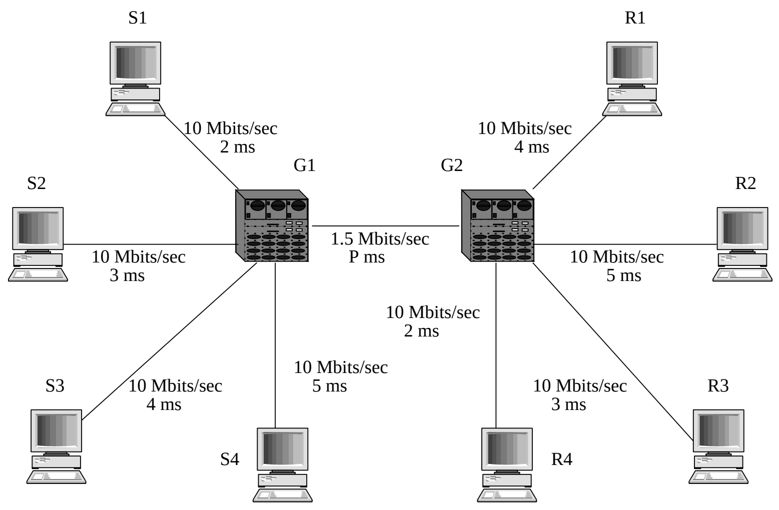

Figure 3.1: Network configuration for the simulation

It is an attempt to simplify a complicated network topology for the sake of analysis. It is a simulation of a typical Internet/Intranet environment. It comprises two LAN environments. The first is composed of S_1, S_2, S_3, S_4 and G_1 while the other is composed of R_1, R_2, R_3, R_4 and G_2. G_1 and G_2 are the routers which link the LANs together. S_1, S_2, S_3 and S_4 are the sending hosts with Nagle algorithm disabled. R_1, R_2, R_3 and R_4 are the receiving hosts with delayed acknowledgment enabled. The buffer size of each router is 15 packets. The segment size for transfer is 1460 bytes, and the maximum window size of the connection is 22 segments.

Rather than simulating the entire HTTP protocol, we have only considered the HTTP responses which consisted of a burst of data sent from the server to the client. In the initial setting, S_1 sends 1000 data segments to R_1 in an environment free of other traffic. After finished sending the data, the TCP connection is idled for a period which is equivalent to around 1.5 RTO.

During the idle period, S_2, S_3 and S_4 begin to generate traffic with the rate of 512KB to R_2, R_3 and R_4 which simulate the real world environment with other network traffic. Then S_1 and R_1 restart sending a certain amount of data again. The network condition after the restart of S_1 and R_1 has changed. Such kind of change in network environment is intentionally created to test how the proposed solutions handle this kind of unpredictable change.

To test how S_1 and R_1 handle the situation when the traffic is relatively busy, the time required for completing all transmission using different proposed solutions mentioned before is recorded.

In this experiment, the testing will be conducted by studying the effect of three factors on the performance of different proposed solutions in handling restarted idle TCP. The three factor are (1) the propagation delay between R_1 and R_2; (2) burst size; and (3) the queue management approaches.

To start experiment, the propagation delay is fixed at 10 ms. Burst size is set to 100 segments and the size of each data segment is fixed. The variable is the queue management approaches (i.e. drop-tail or RED). The time required for completing all transmission when using different queue management is recorded. Then the experiment is repeated for a propagation delay from 10 ms to 300 ms and burst size from 100 to 1000 segments.

Similarly, for a given propagation delay and queue management approach, the burst size is varied and transmission time is recorded. Finally, for a given burst size and queue management approach, the propagation delay is varied and the transmission time is again recorded. Hence the experiment results are a set of matrices when (1) propagation delay ranges from 10 ms to 300 ms; (2) burst size ranges from 100 to 1000 segments; and (3) using either drop-tail or RED queue management approach. The results of the simulation experiments are as shown in Table 3.1 to Table 3.4.

Delay	*SSR*		*RBP*		*CWV*	
10	2.498	(2.991)	3.895	(2.891)	4.112	(2.995)
20	4.238	(2.901)	4.390	(2.884)	4.045	(3.141)
30	2.699	(2.773)	3.932	(3.917)	3.322	(3.710)
50	3.534	(3.130)	4.301	(3.521)	3.678	(3.667)
100	3.330	(4.052)	4.398	(4.303)	5.123	(5.025)
150	5.998	(5.167)	5.480	(5.254)	6.590	(6.580)
200	7.219	(5.890)	5.912	(5.851)	8.262	(6.109)
300	10.596	(7.123)	2.888	(2.809)	12.187	(8.327)

Table 3.1: Time required for SSR, RBP, and CWV to transfer a burst of 100 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

Delay	*SSR*		*RBP*		*CWV*	
10	4.843	(6.445)	7.430	(5.731)	7.587	(6.567)
20	6.987	(5.364)	8.430	(6.019)	7.937	(6.020)
30	5.435	(5.488)	7.023	(6.594)	6.548	(6.871)
50	6.186	(5.756)	7.765	(7.511)	7.891	(6.395)
100	5.762	(7.234)	6.768	(6.902)	7.653	(8.246)
150	10.542	(9.299)	8.765	(8.625)	10.301	(10.195)
200	12.987	(10.408)	11.399	(10.162)	13.114	(11.989)
300	18.436	(12.789)	5.801	(5.7111)	18.781	(15.041)

Table 3.2: Time required for SSR, RBP, and CWV to transfer a burst of 200 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

Delay	*SSR*		*RBP*		*CWV*	
10	13.844	(15.778)	19.219	(15.975)	18.999	(16.755)
20	18.240	(14.667)	19.581	(15.102)	19.267	(15.803)
30	13.802	(13.223)	16.693	(15.021)	16.023	(17.624)
50	15.167	(14.810)	19.431	(18.435)	19.087	(17.019)
100	14.627	(15.010)	17.223	(18.511)	19.355	(18.395)
150	19.994	(18.129)	17.151	(17.511)	20.792	(19.925)
200	21.414	(20.186)	19.944	(20.278)	21.101	(22.129)
300	30.851	(27.398)	14.551	(14.556)	30.295	(27.872)

Table 3.3: Time required for SSR, RBP, and CWV to transfer a burst of 500 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

Delay	*SSR*		*RBP*		*CWV*	
10	23.548	(29.894)	38.215	(31.925)	39.614	(32.779)
20	39.104	(28.556)	40.248	(30.364)	41.112	(30.779)
30	27.530	(28.645)	33.275	(30.667)	32.987	(33.600)
50	28.521	(29.461)	37.997	(34.793)	37.669	(33.432)
100	29.995	(29.801)	32.651	(30.812)	30.889	(31.793)
150	33.925	(35.147)	32.831	(34.112)	33.127	(36.530)
200	41.173	(37.781)	38.912	(35.911)	42.194	(38.455)
300	55.916	(39.959)	30.123	(28.717)	55.319	(42.781)

Table 3.4: Time required for SSR, RBP, and CWV to transfer a burst of 1000 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

3.3.2 Experiment Results

At various propagation delay

Referring to Figure 3.2 to Figure 3.9, it is discovered that when SSR is used, the transmission time is the shortest among the three proposed solutions when the propagation delay between R_1 and R_2 is within 10 ms to 100 ms. In other words, SSR works best in low RTT environment, irrespective of the burst size and queue management approach adopted.

On the other hand, when the propagation delay is high (between 150 ms and 300 ms) with any burst size and either queue management approach, the transmission time is the shortest when RBP is used.

At various burst size

In the testing, the best performers (i.e. SSR or RBP depending on the propagation delay) still work best even at different burst size. However, this time, we should focus on the *absolute difference* in performance of the proposed solutions. It is found that when the burst size is small, say at around 100 to 200 segments, the difference in performance is insignificant. In contrast, when the burst size is about 500 segments or above, the difference is quite substantial.

At various queue management approaches

When RED is used, overall transmission time is reduced irrespective of which proposed solution is used. So their relative performance remains the same: SSR and RBP work best in low and high RTT delay environments respectively.

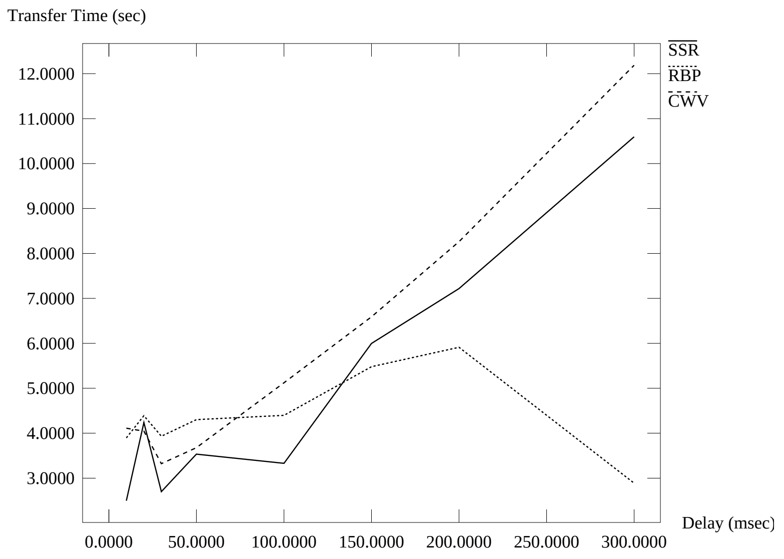

Figure 3.2: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 100 segments and the drop-tail queue management approach is used.

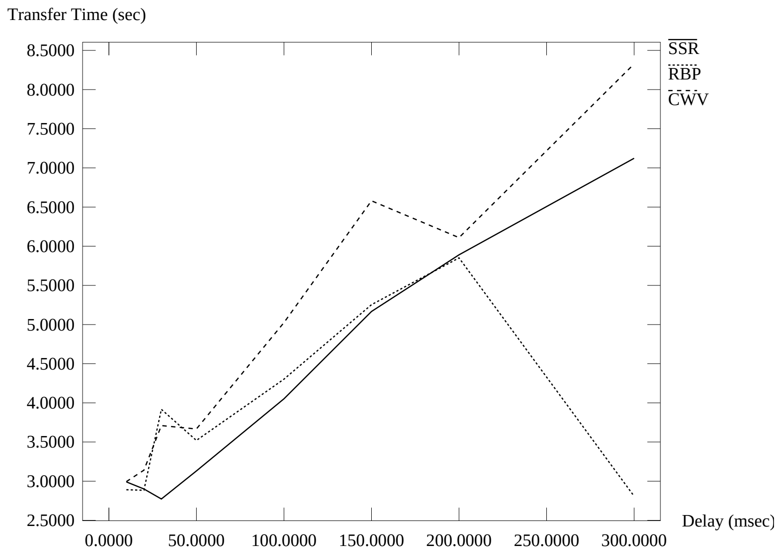

Figure 3.3: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 100 segments and the RED queue management approach is used.

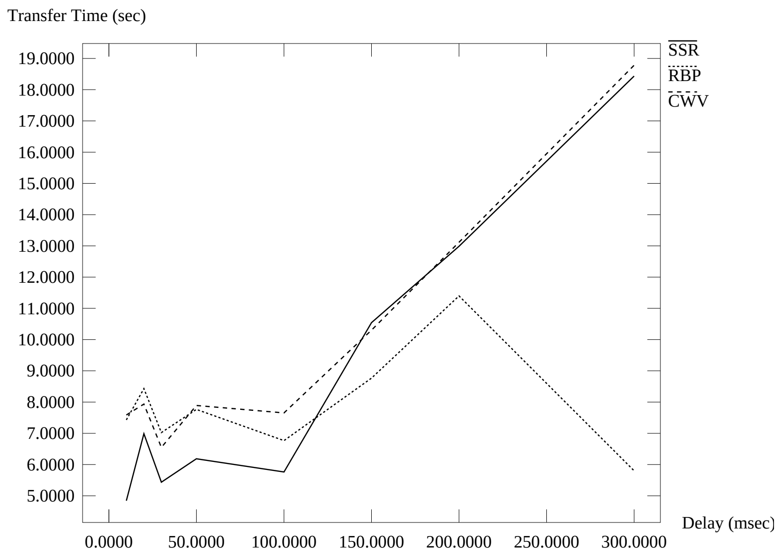

Figure 3.4: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 200 segments and the drop-tail queue management approach is used.

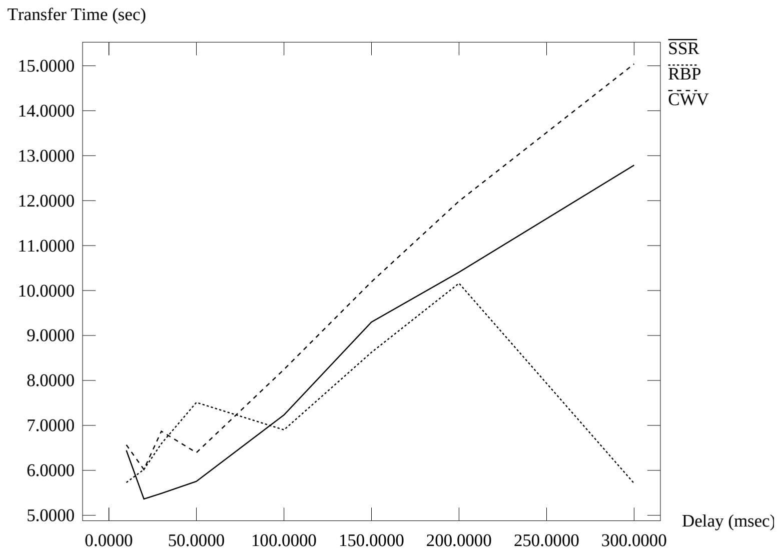

Figure 3.5: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 200 segments and the RED queue management approach is used.

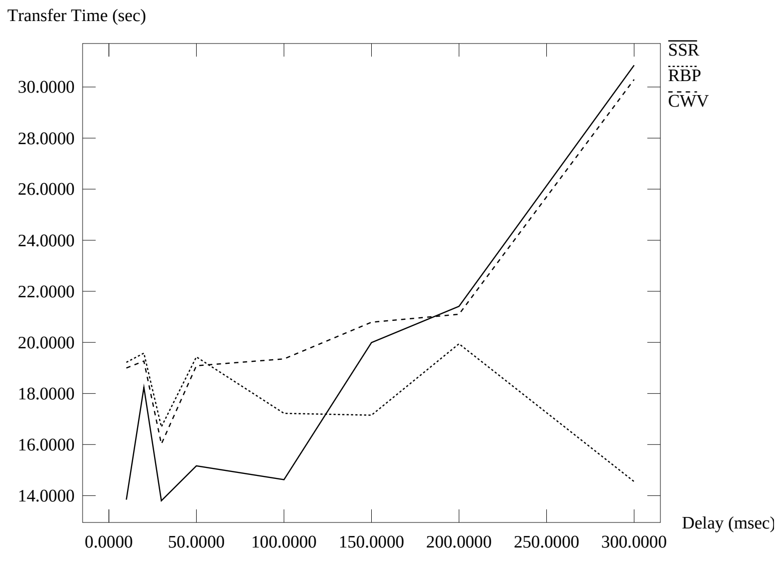

Figure 3.6: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 500 segments and the drop-tail queue management approach is used.

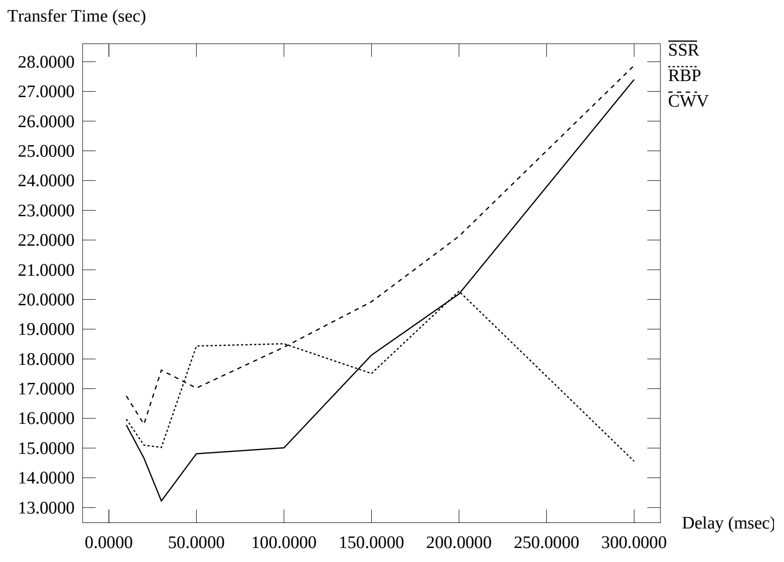

Figure 3.7: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 500 segments and the RED queue management approach is used.

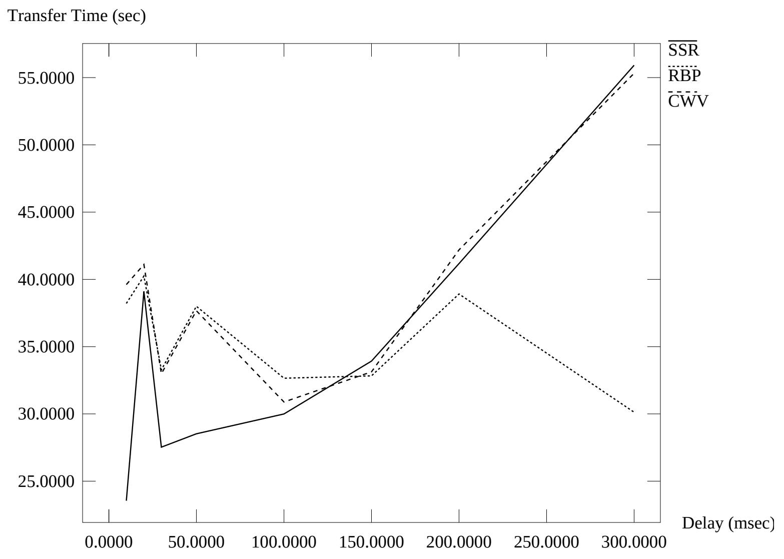

Figure 3.8: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 1000 segments and the drop-tail queue management approach is used.

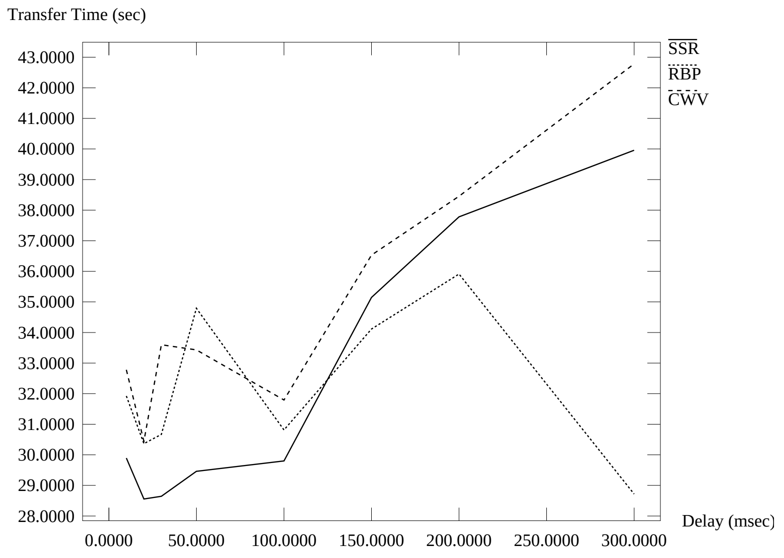

Figure 3.9: Performance comparison of SSR, RBP, and CWV at various propagation delays when the burst size is 1000 segments and the RED queue management approach is used.

3.3.3 Implication of the Findings

TCP with Slow-Start Restart

It is observed that SSR works best at low RTT environment. At high RTT environment, however, SSR performs similarly as CWV. Although, SSR is the most conservative method among the three proposed solutions, it is the best solution to handle the problem of restarting idle TCP connection.

TCP with Rate-Based Pacing

From the results of the simulation experiments, we can conclude that RBP has superb performance at high RTT environment. At low RTT environment, its performance is just close to that of CWV but worse than that of SSR. However, as the RTT increases, it begins to outperform other proposed solutions.

TCP with Congestion Window Validation

It is observed that the performance of CWV is the worst at both high and low RTT environments. At low RTT environment, the performance is about the same as that of RBP but worse than SSR. At high RTT environment, its performance is about the same as that of CWV, irrespective of the burst size and queue management approach is used.

Chapter Conclusion

In short, according to the findings above, SSR is the most effective solution in handling restarting an idle TCP when the network turns extremely busy during the idle period when RTT is at medium to low level, which is close to the real Internet/Intranet environment. At a high RTT environment like satellite links, RBP is better.

3.4 Other Problems

As mentioned, restarting an idle TCP connection in HTTP persistent connections without considering the fluctuation of network condition during the idle period may result in TCP injecting too many data segments into the network and eventually downgrading the performance. In the previous section, we have evaluated the performance of different approaches to solve this problem.

Apart from the problems of restarting an idle TCP connection, there are a number of generic TCP problems that may also degrade the overall performance of HTTP persistent connections. They are (1) initial TCP window size; (2) interaction of Nagle algorithm and delayed acknowledgment; (3) initial value of *ssthresh*; and (4) validation of congestion window. We will discuss these issues one by one.

3.4.1 Initial TCP Window Size

A simplistic implementation of delayed ACK can cause unnecessary idle time during the initial data transfer phase in a client-server network environment. Figure 3.10 illustrates the scenario as follows.

Upon receiving a HTTP request from a client's browser, since the response message cannot fit in one TCP segment, TCP will break it up into multiple segments. During the initial slow-start phase, the server TCP is allowed to send only one segment. Therefore, only a partial server response is sent. The browser, upon receiving the first segment, is not able to respond because the data is incomplete.

In the mean time, the client TCP holds back the acknowledgment of the first segment while the browser is waiting for the rest of the response data from the server before it can issue the next HTTP request. But server is waiting for the browser to acknowledge before it can send the rest of the response data. Then a temporary deadlock is formed. Eventually, the client

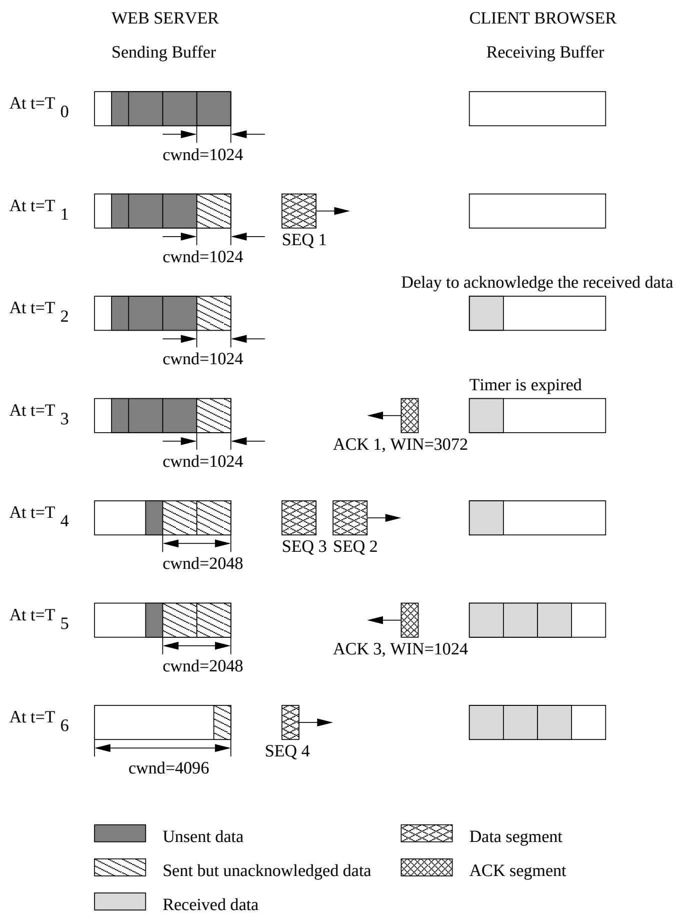

Figure 3.10: Delayed acknowledgments.

TCP will give up the waiting after a delayed acknowledgment interval, and send back an ACK. Note that the effect of delayed acknowledgment is to delay segment transmission. As the delay is much shorter than the initial retransmission delay, this does not cause retransmissions by the server or client.

Figure 3.11 shows the simulation performance of TCP with and without delayed acknowledgment enabled on the client side. The dotted line and solid line represent the sequence number of the data segments against the send time when the delayed acknowledgment is enabled and disabled respectively.

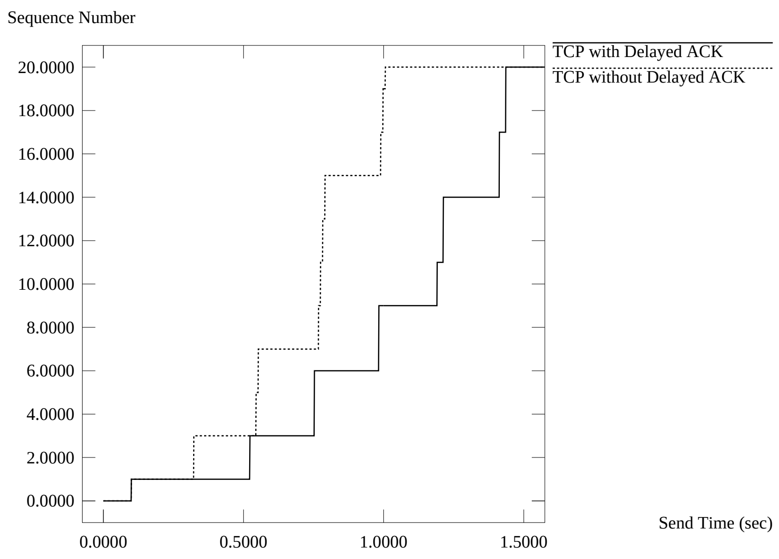

Figure 3.11: Performance of TCP with and without delayed acknowledgments.

The simulation is conducted by sending 20 data segments from the server to the client and check the time required for the transmission.

In the situation where the delayed acknowledgment is disabled, we can see from Figure 3.11 that the server sends the first segment at 0.1 second. Upon the client's receipt of the first segment sent by the server, it would immediately send the acknowledgment back to the server so that the server can continue with sending other segment at 0.32 second. In contrast, when the delayed acknowledgment is enabled, the client will wait for about 200 ms before sending the acknowledgment to the server, instead of doing it immediately after receiving the first segment. As a result, the server has to wait until 0.52 second before it can send out the outstanding segments.

We can see that the total time required by TCP with and without delayed acknowledgment for sending 20 segments is 1.44 seconds and 1.01 seconds respectively. Thus when the delayed acknowledgment is enabled, the transmission is 42.57% slower than when it is disabled.

3.4.2 Interaction of Nagle Algorithm and Delayed Acknowledgment

As discussed in Section 2.1.8, Nagle algorithm was designed for terminal I/O traffic to avoid small amount of data, instead of full-size segments, being exchanged across the connection. By default, Nagle algorithm is enabled. However, it is not suitable for the traffic of HTTP persistent connection when delayed acknowledgment is enabled on the client's browser [9].

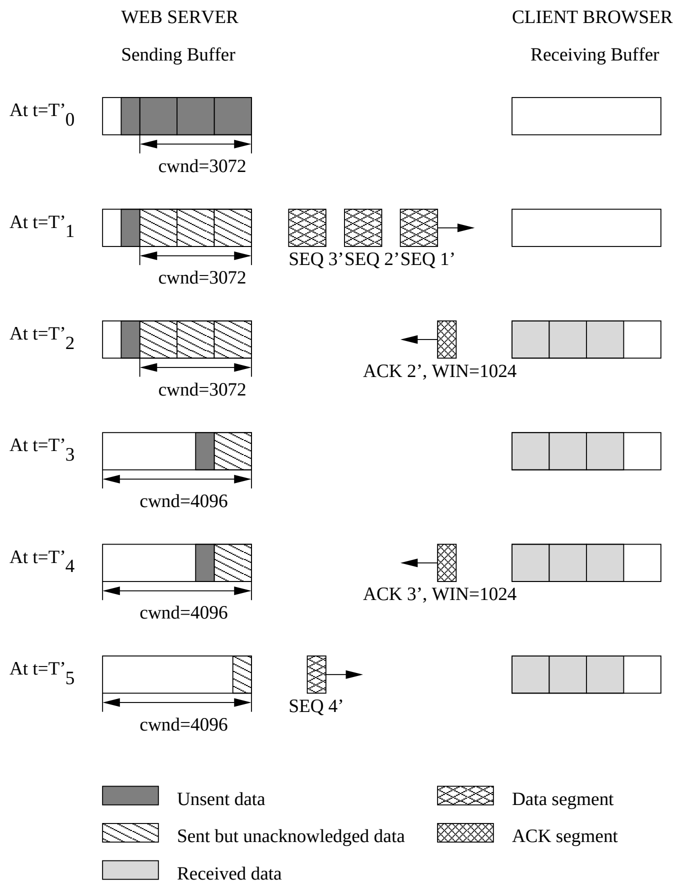

Figure 3.12: Interaction of Nagle algorithm and delayed acknowledgment.

Figure 3.12 illustrates the problem. Assume the web server writes 5KB of data at the application-layer. TCP breaks the data into three full-sized segments of 1460 bytes each plus one segment of 740 bytes. At the beginning, Nagle algorithm is not invoked because the sending buffer does not have any outstanding unacknowledged data. TCP will send the first three full-sized segments. However, since the delayed acknowledgment is enabled on client's browser, the acknowledgment of the third segment will be delayed as it is not a pair. On the other hand, the outstanding unacknowledged data and the absence of a full-sized segment triggers the Nagle algorithm. As a result, the client will delay sending the final non full-sized segment and eventually a temporary deadlock appears.

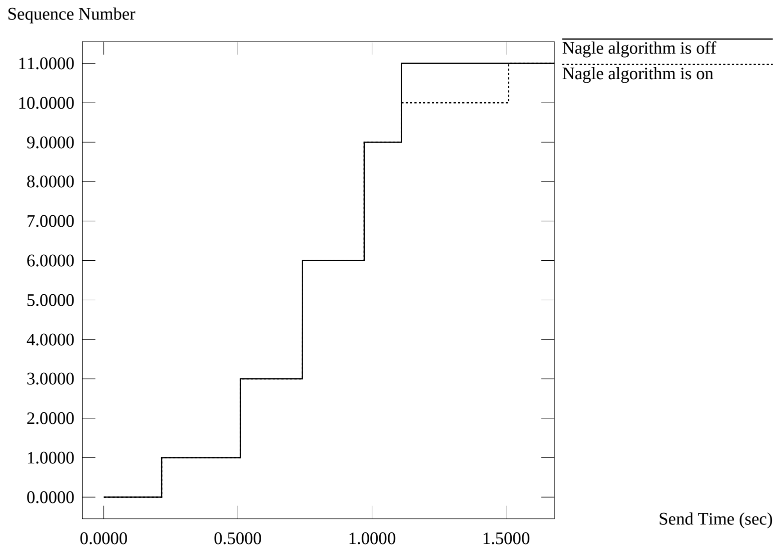

Figure 3.13: Performance of TCP with Nagle algorithm is on and off.

Figure 3.13 shows the performance comparison of TCP with Nagle algorithm on and off on the server side. The dotted line and the solid line represent situations when the Nagle algorithm is enabled and disabled respectively. In both cases, the delayed acknowledgment is enabled on the client side. The simulation this time is carried out by sending 11 data segments from the server to the client and checks the time required for the transmission. We can see from Figure 3.13 that in the period between 0.00 second to 1.11 seconds, the two lines overlap. However, beyond 1.11 seconds, when Nagle algorithm is enabled, there will be a temporary deadlock. The deadlock will last for about 200 ms. Nevertheless, no deadlock will appear when the Nagle algorithm is disabled.

The ultimate cause of the deadlock is that Nagle algorithm prohibits web server from sending any non full-sized segment, while delayed acknowledgments prohibit client's browser from acknowledging any segment received which is not in pair. In fact, Nagle algorithm is intended for small-packet and two-way traffic, while TCP is used by HTTP persistent connection for a series of requests and responses.

3.4.3 Initial Value of *ssthresh*

As mentioned before, the slow-start threshold, *ssthresh*, is a critical point to trigger either slow-start and congestion avoidance algorithm to control data transmission. Default values for the parameters will be adopted when the server TCP starts up in the absence of any details of the client and network capacity. There is possibility that the initial value of *ssthresh* is arbitrarily high, say the size of the advertised window or simply 64KB. But the value will reduce automatically when congestion occurs. Nevertheless, under this mechanism, the server will send excessive segments too rapidly and thus unnecessary loss of some multiple segments in the same window, which often takes much time to recover, may occur.

Figure 3.14 and Figure 3.15 illustrate the situations at different level of initial threshold. In Figure 3.14, the initial threshold is set to a relatively high level, equivalent to 22-segment size. In Figure 3.15, the initial threshold is set to a relatively low level, equivalent to 10-segment size. Under the same traffic condition, TCP with high level of initial threshold results in multiple segment loss. Eventually, there will be retransmission timeout and the transmission returns to slow-start. On the contrary, in relatively low threshold, the degree of segments loss is relatively less serious such that no

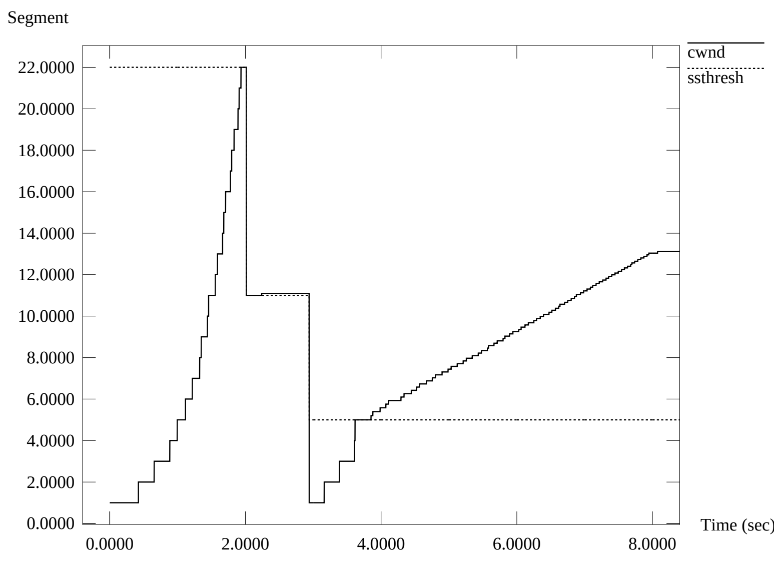

Figure 3.14: TCP with large initial *ssthresh*.

retransmission timeout will be induced and fast recovery is good enough to handle the loss.

Figure 3.16 is the results of a simulation for comparing the performance of TCP with a relatively large and small initial threshold. Total 200 segments are sent by the server for testing purpose. The finding is: in the large-threshold case, it takes 7.74 seconds for the transmission; in the small-threshold case, it takes 6.67 seconds.

3.4.4 Validation of Congestion Window

It is common in current TCP implementations that it does not attempt to find out whether the previous value of the *cwnd* has been fully utilized before the congestion window is enlarged upon receiving ACKs, as long as the advertised window of the client and the slow-start or congestion avoidance permit. Since the amount of data transmitted is less than the amount allowed by *cwnd*, *cwnd* will be inaccurate in showing the network condition in the application-limited period.

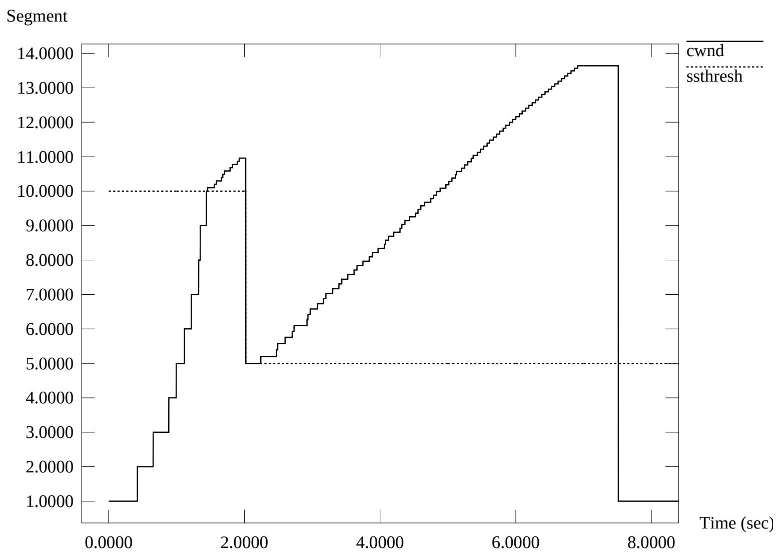

Figure 3.15: TCP with small initial *ssthresh*.

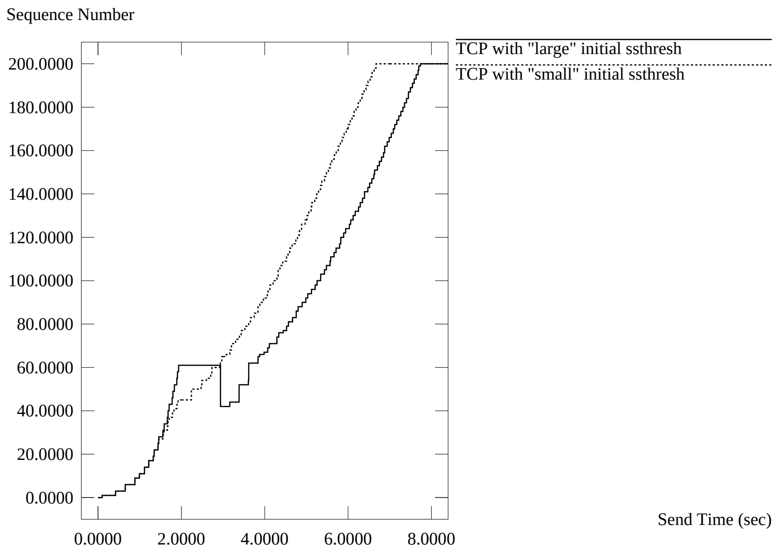

Figure 3.16: Performance of TCP with large and small initial *ssthresh*.

Figure 3.17 illustrates how congestion window fails to reflect the network condition accurately during application-limited period. The dotted line and solid line represent the size of congestion window and sequence number respectively. From 2.2 seconds to 2.5 seconds, the server sends one data segment to the client for every 0.1 second. Like most of the current implementations, TCP increases the value of congestion window for each ACK it receives. Since the amount of data transmitted is far below the amount allowed by the congestion window, it does not reflect the amount of data segments that the network can actually afford.

However, when the server tries to send 30 more data segments to the client at 2.6 seconds, this burst of data segments overflow the intermediate routers. As this amount of data segments exceeds the congestion window limit, TCP will only transmit that maximum amount. Unfortunately, the value of *cwnd* is misleading during the application-limited period (between 2.2 seconds to 2.5 seconds) since TCP does not check whether the previous value of *cwnd* has been fully utilized before the congestion window is inflated whenever an ACK is received. As a result, transmitting a large burst of data allowed by *cwnd* after application-limited period may lead to data segment loss and eventually the overall performance deterioration.

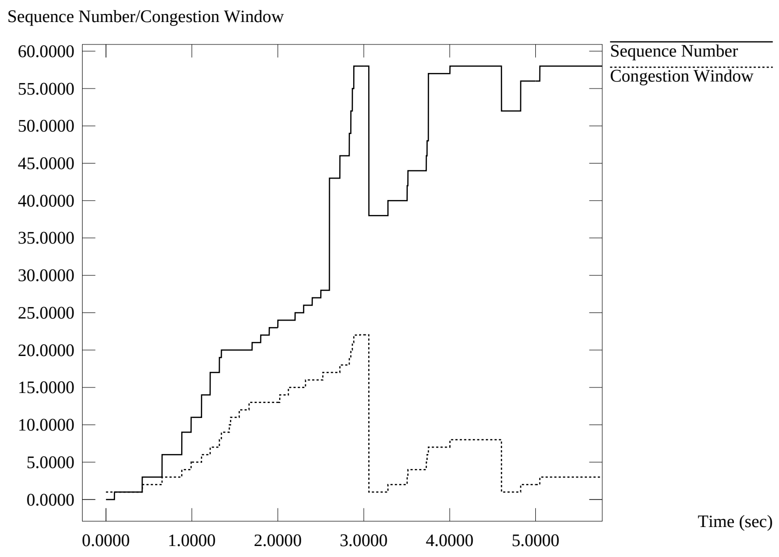

Figure 3.17: Validation of Congestion Window.

Chapter 4

Proposed Solution

In previous chapter, we have evaluated some related works proposed by other researchers to resolve the problem of restarting idle TCP connections. In addition, we have also discussed a number of problems with HTTP persistent connections running over existing TCP implementations. In this chapter, we propose a *Cocktail* approach to resolve the problems and describe the rationales behind.

4.1 A *Cocktail* Approach

The problem of restarting idle TCP connections frequently happens in HTTP persistent connections. In the previous chapter, a number of simulation experiments have been conducted and it reveals that SSR is the best solution in restarting an idle TCP connection when the network is extremely busy during the idle period when RTT is at medium to low level. Since this level range is close to the real Internet/Intranet environment, it is proposed that SSR can be used to handle the problem.

To further enhance the solution by tackling other generic problems discussed in Section 3.4, some more elements can be added to supplement SSR.

1. Increasing the size of initial window from one to two

2. Using TCP NewReno with Initial *ssthresh* Estimation

3. Limiting the growth of congestion window

4. Disabling Nagle algorithm

As the set of solution is a mix, we can call it *Cocktail* Approach.

4.2 Slow-Start Restart of Idle Connections

In Section 3.2, approaches to handle the restarting of idle TCP connections, namely SSR, RBP, and CWV, have been explored. The basic idea of RBP is attempting to maintain the transmission rate to pre-idle rate even after the idle period. CWV, on the other hand, is an attempt to decay the *cwnd* for every RTO. However, these two solutions have their hidden problems because they are assumed that the network traffic condition fluctuates mildly rather than drastically. So theoretically, RBP and CWV can improve transmission performance if there is no drastic change in the network condition during the idle period.

Unfortunately, in reality, traffic in the Internet is dynamic and unpredictable. There is no guarantee that the traffic condition in this moment is the same as the next moment, especially in a highly congested environment. If the network turns very congested during the idle period, RBP and CWV may inject too many segments into the network, exceeding its handling capacity. Eventually, this is lead to retransmission timeout at the early stage when the connection is restarted. Hence transmission performance is degraded.

Compared with SSR, RBP and CWV are more aggressive solutions. If the number of aggressive senders is few, they will be better off. However, if more and more clients adopt the two approaches to handle idle connections, the network *as a whole* may end up with more congestion. Thus it may be more prudent to adopt SSR to avoid this potential drawback.

4.3 Other Enhancements

4.3.1 Increasing the Size of Initial Window

One of the easiest ways to improve the performance of slow-start is to increase the size of initial window from one to two. Suppose the initial window is of one segment, a client who adopts delayed acknowledgment cannot create an acknowledgment until there is a timeout. If the initial window is of two segments or more, however, the client will generate an acknowledgment after the second segment arrives. The unnecessary waiting for the timeout can be eliminated.

Figure 4.1 illustrates the situation when the size of the initial window is increased from one to two. When the initial window is two, the server is able to send out two segments at the initial stage. Upon receiving these two segments, the client can immediately send out the acknowledgment and the 200 ms timeout can be eliminated, even if the delayed acknowledgment is enabled.

One of the potential tradeoffs of the change is that the network can tolerate more congestion, provided that more and more clients use two segments as the initial window size. Internet Engineering Task Force (IETF) has conducted a study on the effectiveness of the modification [18]. The study eventually found no evidence that the modification resulted in more congestion.

On the other hand, RFC 2414 [2] proposed to increase TCP's initial window size from one segment to between two and four segments according to the following equation:

$$\min(4MSS, \max(2MSS, 4380bytes) \tag{4.1}$$

Apart from eliminating the need to wait for the timeout due to the delayed acknowledgment at the start-up stage, there are several other ad-

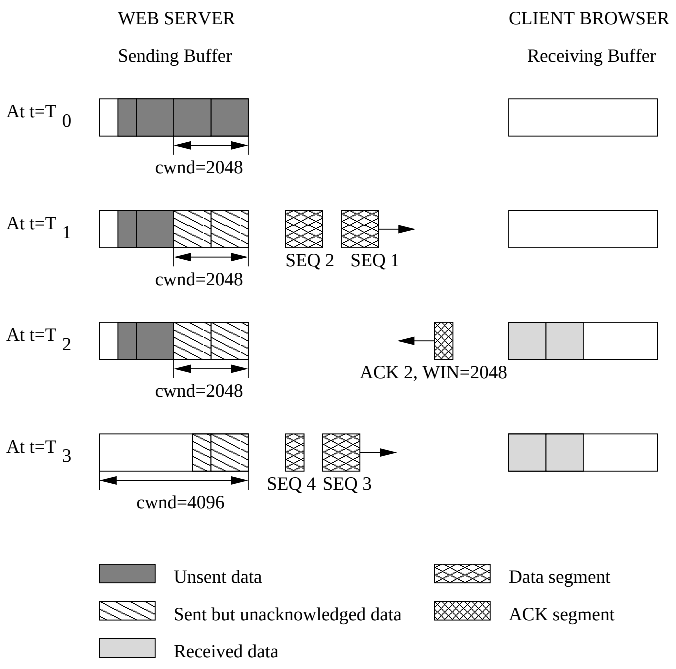

Figure 4.1: Increasing the size of initial window from one to two.

vantages of using initial window sized larger than one. Firstly, it minimizes the transmission time for connections transmitting a small amount of data. Secondly, it eliminates up to three RTTs and a delayed a delayed acknowledgment timeout in slow-start for connections which can use large windows. For high-bandwidth large-propagation-delay TCP connections, for instance, those in satellite links, it will become particularly useful.

In contrast, when the environment is highly congested, large initial windows will result in retransmission timeout at the early stage of slow-start. Though it can recover in the absence of a retransmission timeout, the transition from slow-start to the congestion avoidance phase of the window increase algorithm will occur too early.

Therefore, choosing two as the TCP's initial window size, rather than three or four, is more appropriate as it is a way to compromise the pros and cons of the modification. TCP's initial window two-segment size is large enough to eliminate the temporary deadlock caused by the delayed acknowledgment at the initial slow-start phase. In addition, this size can highly reduce the chance of retransmission timeout occurring at the early stage of slow-start that a large initial window may introduce.

4.3.2 Using TCP NewReno with Initial *ssthresh* Estimation

Apart from estimating a reasonable value of *ssthresh*, as discussed in Section 2.1.5, TCP NewReno can efficiently recover from multiple segment losses in the start-up period. Therefore, TCP NewReno with initial *ssthresh* estimation is an essential element for our *Cocktail* approach to improve the performance of HTTP persistent connections.

Slow-start threshold, *ssthresh*, is the critical point to decide whether to use slow-start or congestion avoidance algorithm to control data transmission. For most TCP implementations, initial *ssthresh* value is arbitrary. Some use the size of the advertised window. Some may simply take 64KB as the initial value. In fact, if the initial *ssthresh* is too low, *cwnd* will rise exponentially and reach *ssthresh*, hence switching to the additive increase mode, too early. The transmission will be very slow, downgrading the performance. So ideally, it will be preferable to estimate the value of *ssthresh* to a level close to the full capacity of the link so that the growth of the congestion window can slow down once the full capacity is achieved.

Hoe [10] proposed a simple way to estimate the *ssthresh*. Closely packed segments are sent to the client at the rate of the bottleneck link bandwidth. Provided that the ACKs are roughly equally spaced, an estimation of the bandwidth can be estimated using the ACKs and the time at which they arrive. To roughly estimate the round-trip delay, record the timestamp when sending the segment and do it again when the corresponding ACK arrives. Hence the bandwidth-delay product can then be computed. The mechanism is basically like this. The *ssthresh* is initialized to 64 segments.

Then the SYNC segment's round-trip time is recorded. Then the bandwidth is estimated by least-square method with figures from three closely spaced ACKs.

It is important that the value of *ssthresh* chosen by such estimation is better than choosing arbitrarily, say 64KB. Whether the *ssthresh* chosen by estimation is absolutely reliable is not important. If the estimated value is higher than the true value, it still makes sense to estimate rather than choosing arbitrarily. For example, if the arbitrarily chosen value is 64KB, the window will be opened too aggressively, resulting in much loss in segments. In contrast, if the value estimate is too low, the server may be to conservative. The result may even be worse than losing multiple segments and waiting for a retransmit timeout to recover. For the network performance as a whole, a more conservative server is more desirable than an aggressive one.

4.3.3 Limiting the Growth of Congestion Window

The next element in the cocktail approach is limiting the growth of the congestion window. Recapping the discussion from previous sections, when the congestion window is increased during the application-limited periods and the previous value of window has not been fully utilized, there will be an invalid congestion window. However, if the client's advertised window and either the slow-start or congestion window algorithm permits, all the existing TCP implementations automatically enlarges the congestion window upon receiving the ACKs without checking to ensure the previous value of the window has not been used.

RFC 2861 [8] proposes that the window increase algorithm should not be invoked during application-limited periods. The server should not increase the congestion window when the server is application-limited (in other words, not fully utilizing the current window). It aims at limiting the growth of the window so that it will not be arbitrarily large without ensuring that the network can support that large window size.

4.3.4 Disabling Nagle Algorithm

Referring back to Section 3.4.2, when Nagle algorithm is enabled on the server side while delayed acknowledgment are also enabled on the client side, temporary deadlock will occur. From the server's point of view, it does not know whether the delayed acknowledgment is enabled. Thus the simplest way to solve the deadlock problem is to completely disable the Nagle algorithm [17].

Figure 4.2 demonstrates the situation when the Nagle algorithm is disabled. We can see that the server is still able to send the last non full-sized segment when the acknowledgment segment (ACK 2) is received, even though some unacknowledged data exists.

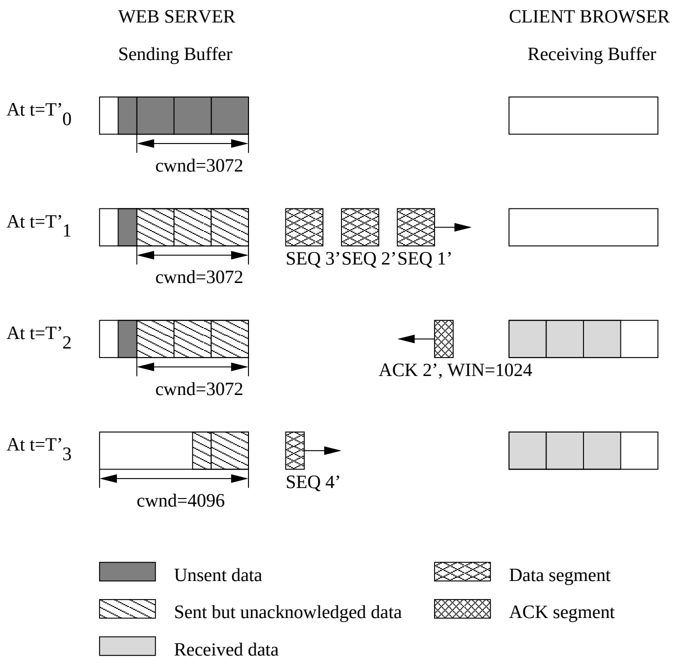

Figure 4.2: Disabling Nagle Algorithm

Minshall [14] proposed a modified version of Nagle algorithm that can avoid the deadlock problem even when it is enabled. The idea of the solution is that the ACK will be delayed only if the unacknowledged data is smaller than one full-sized segment. Transmitting short segments when there is any unacknowledged data will not happen. The advantage is that it prevents flooding the network with many small segments, which is the original purpose of the algorithm, and without increasing delay to the transmission of responses which requires segments in pairs.

Unfortunately, the pitfall of the modified Nagle algorithm is that if the amount of data sent fluctuates much, one large segment, followed by a small one and then a large one again, the original Nagle algorithm will give better performance than the modified version. The original version will send at most one segment, delaying the middle small amount of data and give way to the following large segment. Hence the algorithm will send out two small segments, the middle and the last ones.

Because of the pitfall, the modified solution cannot completely solve the problem. Under HTTP persistent connection, disabling the Nagle algorithm will not significantly affect the performance of the web server. This is because most of the segments exchanged are full-sized. Thus, simply disabling the Nagle algorithm is more preferable to adopting the modified version to solve the problem. To do this, the web server can simply turn on the `TCP_NODELAY` socket option [23]. No modification to the existing TCP implementation is necessary.

Chapter 5

Simulation Experiment

In the previous chapter, we have proposed a *Cocktail* approach to improve the performance of HTTP persistent connections over the existing TCP implementations. In this chapter, we are going to evaluate the performance of the approach by conducting a series of simulation experiments at microscopic and macroscopic levels.

5.1 Microscopic Comparison

Cocktail approach comprises several elements and each element addresses a specific problem that degrades the performance of HTTP persistent connections. The purpose of this section is to illustrate how individual element resolves the problem one by one.

Since comparison of SSR with other solution has been discussed in detail the chapters before, this section will concentrate on other elements in *Cocktail*.

5.1.1 Increasing the Size of Initial Window

In our *Cocktail* approach, we propose to increase the size of initial window from one to two. This simple modification can eliminate the unnecessary

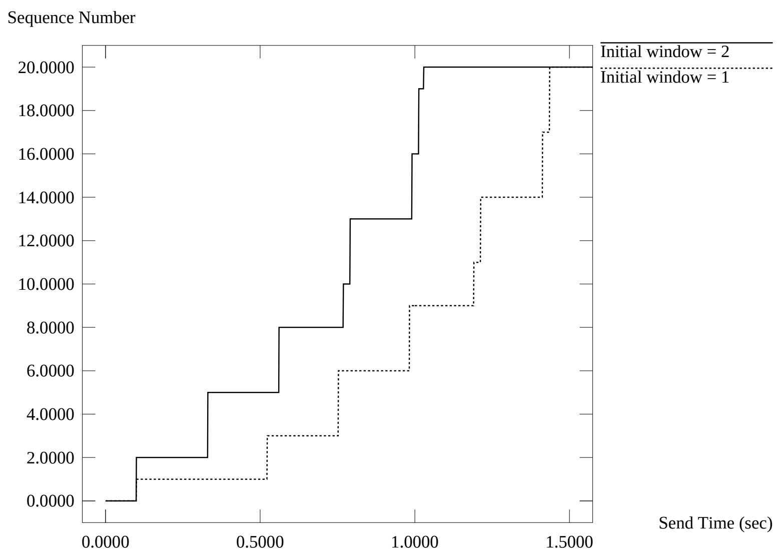

Figure 5.1: Performance comparison of TCP before and after increasing the initial window size from one to two. Delayed acknowledgment is enable on client side.

waiting time for the timeout due to the delayed acknowledgment at the early stage of slow-start. It aims at improving the performance of slow-start.

A simple simulation experiment is set up to show the performance of TCP before or after the modification. The server sends 20 data segments to the client at 0.1 second. Figure 5.1 shows the result of the simulation that assumes delayed acknowledgment is enabled on the client side. The dotted line and solid line represent the sequence number of the data segments when the initial window is one and two respectively against the send time.

In the situation where the initial window is one, the server takes 1.34 seconds to send 20 data segments. On the other hand, when the initial window is two, the server takes 0.93 second to send 20 data segments. The improvement is 43.8%, quite substantial, especially when the burst size is small.

Apart from eliminating the temporary deadlock at the early stage of

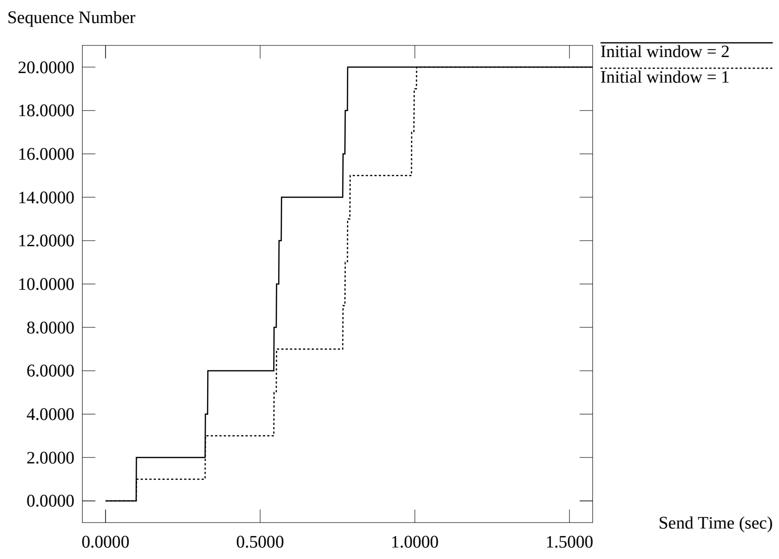

Figure 5.2: Performance comparison of TCP before and after increasing the initial window size from one to two. Delayed acknowledgment is disable on client side.

slow-start caused by enabling delayed acknowledgment on the client side, increasing the size of initial window can improve the performance of TCP even if delayed acknowledgment is disabled on the client side. Figure 5.2 shows the result of simulation experiments that assumes delayed acknowledgment is disabled on the client side. When the initial window is one, the server takes 0.91 second to send 20 data segments. On the other hand, when the initial window is two, the server takes 0.68 second to send 20 data segments. The improvement is 24.6%. It is a remarkable improvement, although this figure is not as significant as that when the delayed acknowledgment is enabled on client side. Therefore, we can conclude that increasing the size of initial window from one to two can eliminate the unnecessary waiting time for the timeout due to the delayed acknowledgment at the early stage of slow-start.

5.1.2 Using TCP NewReno with Initial *ssthresh* Estimation

As discussed in the previous chapter, TCP NewReno with initial *ssthresh* estimation is able to improve the startup performance of TCP. Now we set up a simple simulation experiment to illustrate its effect on the performance of TCP.

In the simulation experiments, the server is trying to send a total of 100 data segments to the client in three different ways: (1) TCP NewReno with initial *ssthresh* estimation; (2) TCP NewReno without initial *ssthresh* estimation; and (3) TCP Reno. The experiments are performed under both congested and less congested environments as well.

Figure 5.3 shows the results of the simulation experiments under a congested environment and compares the time required for different approaches to complete the transfer of the 100 data segments. It is found that the server takes 6.10 seconds, 6.52 seconds and 7.33 seconds for TCP NewReno with initial *ssthresh* estimation, TCP NewReno without initial *ssthresh* estimation and TCP Reno respectively to transfer 100 data segments to the client. The result shows that TCP NewReno with initial *ssthresh* estimation can improve the TCP performance efficiently under a highly congested environment.

However, as shown in Figure 5.4, under a less congested environment, the transmission time is 3.88 seconds, 2.06 seconds and 2.46 seconds for TCP NewReno with initial *ssthresh* estimation, TCP NewReno without initial *ssthresh* estimation and TCP Reno respectively.

5.1.3 Limiting the Growth of Congestion Window

We repeat the simulation experiment that has been performed in Section 3.4.4. The procedures and configurations are exactly the same except CWV is used. Figure 5.5 shows the results. The dotted line and solid line represent the size of congestion window and sequence number respectively. From 2.2 seconds

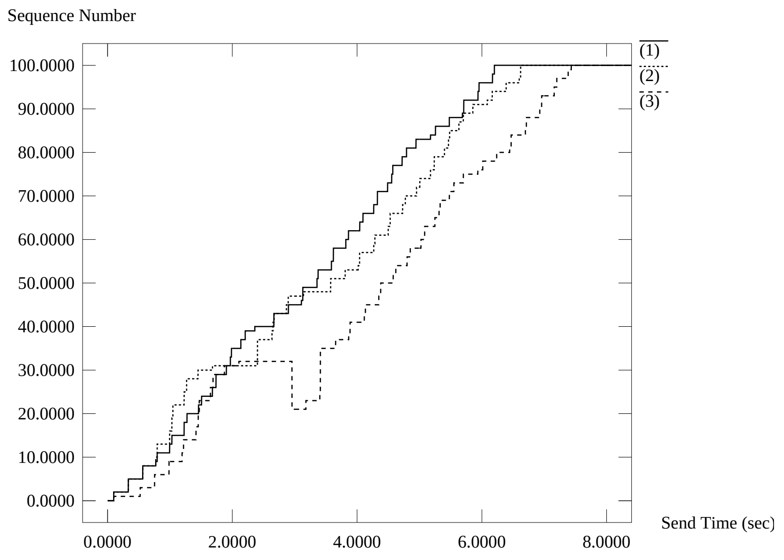

Figure 5.3: Performance comparison of different TCP implementations working under a congested environment: (1) TCP NewReno with *ssthresh* estimation; (2) TCP NewReno without *ssthresh* estimation; and (3) TCP Reno.

to 2.5 seconds, the server sends one data segment to the client for every 0.1 second. For most of the current implementations, TCP increases the value of congestion window for each ACK it receives. In our case, as CWV is used, TCP inhibits the growth of congestion window for each ACK received as the previous value of the cwnd has not been fully utilized. At 2.6 seconds, the server tries to send 30 more data segments to the client. As TCP limits the growth of congestion window during the application-limited period, it prohibits TCP from injecting ”too many” data segments into the network that may overflow the network.

Figure 5.6 is the performance comparison of TCP with CWV enabled and disabled. When CWV is disabled, the server completes sending the last data segment at 4.90 seconds whereas it takes 3.10 seconds when CWV is enabled.

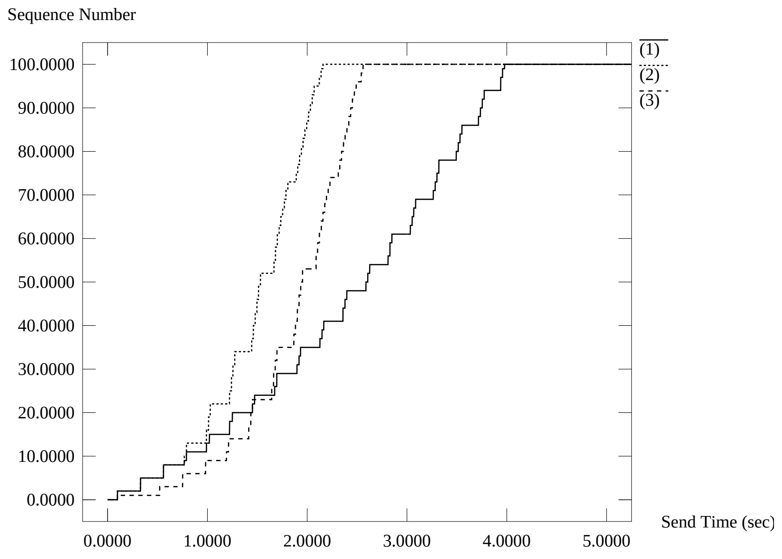

Figure 5.4: Performance comparison of different TCP implementations working under a less congested environment: (1) TCP NewReno with *ssthresh* estimation; (2) TCP NewReno without *ssthresh* estimation; and (3) TCP Reno.

5.1.4 Disabling Nagle Algorithm

Disabling Nagle algorithm is the last element of our *Cocktail* approach. This modification intends to eliminate the temporary deadlock caused by the interaction of Nagle algorithm and delayed acknowledgment.

A simple simulation experiment is set up to compare the performance of TCP with Nagle algorithm on and off on the server side. The dotted line and the solid line represent situations when the Nagle algorithm is enabled and disabled respectively. In both cases, the delayed acknowledgment is enabled on the client side. The simulation is carried out by sending 11 data segments from the server to the client and checking the time required for the transmission. Figure 5.7 shows the results of the simulation experiments. When the Nagle algorithm is on, the server takes 1.29 seconds to send 11 data segments. On the other hand, when the Nagle algorithm is disabled, it

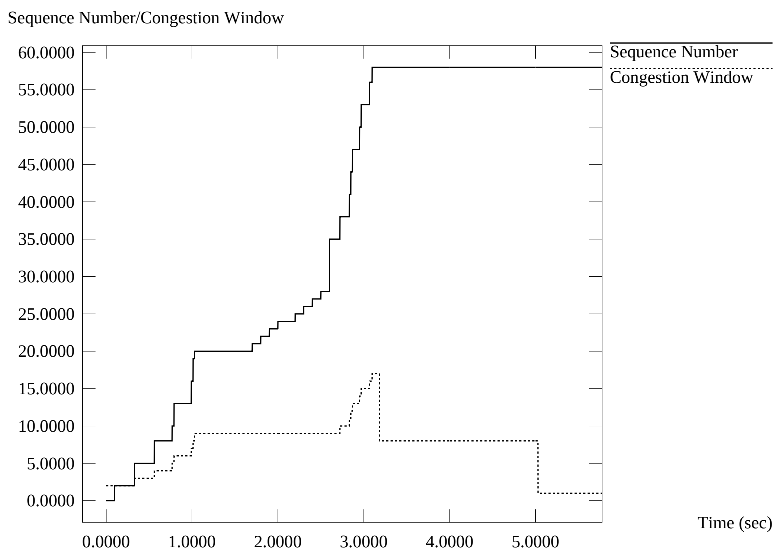

Figure 5.5: Effect of TCP Congestion Window Validation.

takes only 0.69 seconds. The 46.5% improvement is significant. As a result, it proves that disabling Nagle algorithm is able to eliminate the problem caused by the interaction of Nagle algorithm and delayed acknowledgment.

5.2 Macroscopic Comparison

In the previous section, we have shown how individual element works for in the *Cocktail* at the microscopic level. As the *Cocktail* is a mixture of individual elements, we also have to evaluate they work as a whole. In this section, we are going to do this evaluation and compare it with other existing proposed solutions. The procedures and configurations are exactly the same as those for evaluating SSR, RBP, and CWV in Section 3.3. The simulation experiments are shown in Table 5.1 to Table 5.4.

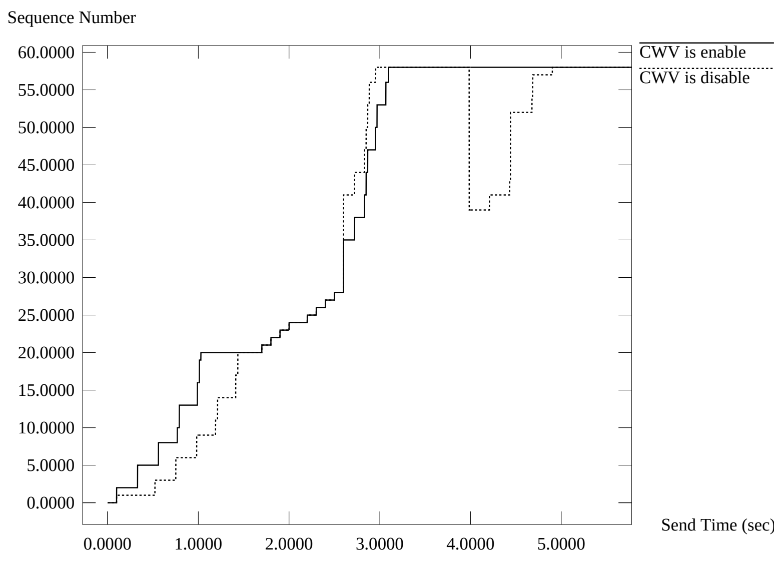

Figure 5.6: Performance comparison of TCP with CWV is enabled and disabled.

At various propagation delay

Referring to Figure 5.8 to Figure 5.15, it is discovered that when *Cocktail* is used, the transmission time is the shortest among the proposed solutions at low RTT. In other words, *Cocktail* works best in low RTT environment, regardless of the burst size and the queue management approach adopted.

However, as shown in Figure 5.12 and Figure 5.14, when *Cocktail* is used in a high RTT environment, the transmission time is significantly higher than that required by the other proposed solutions when the burst size is as large as 500 or 1000 segments and drop-tail queue is used. On the hand, the transmission time is the shortest when RBP is adopted in a high RTT environment with any burst size and any queue management approach.

At various burst size

From the figures, it is observed that the transmission time required for *Cocktail* is the shortest at low RTT environment. In other words, the per-

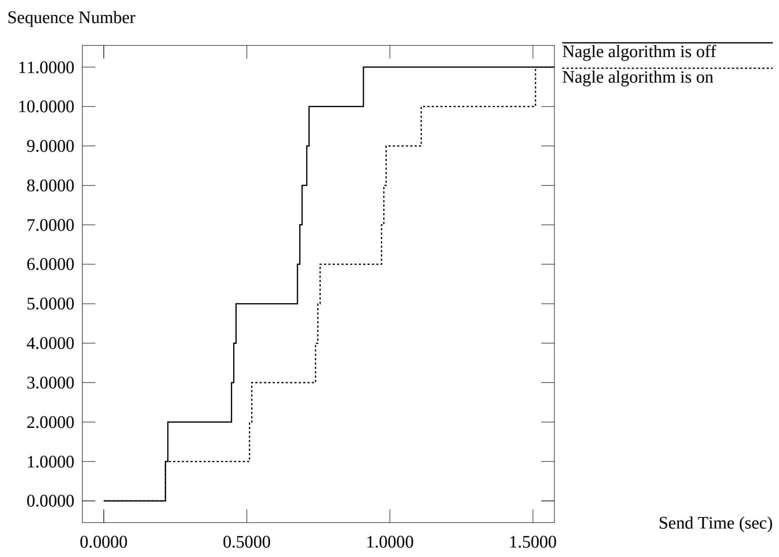

Figure 5.7: Performance comparison of TCP before and after disabling Nagle Algorithm. Delayed acknowledgment is enabled on client side.

formance of *Cocktail* is the best at low RTT environment. The burst size does not affect its performance.

However, by comparing the transmission time required for SSR, RBP and CWV, it is found that the transmission time required for *Cocktail* in a high RTT environment is relatively high when the burst size is as large as 500 or 1000 segments and drop-tail queue management approach is used.

At various queue management approaches

By comparing the results of the simulation experiments, we found that the transmission time is significantly reduced when RED queue is used in high RTT environment. However, we observed that RED queue does not help to reduce the transmission time when Cocktail is used in low RTT environment.

Delay	*SSR*		*RBP*		*CWV*		*Cocktail*	
10	2.498	(2.991)	3.895	(2.891)	4.112	(2.995)	2.610	(2.817)
20	4.238	(2.901)	4.390	(2.884)	4.045	(3.141)	2.597	(3.427)
30	2.699	(2.773)	3.932	(3.917)	3.322	(3.710)	2.813	(2.637)
50	3.534	(3.130)	4.301	(3.521)	3.678	(3.667)	2.701	(2.693)
100	3.330	(4.052)	4.398	(4.303)	5.123	(5.025)	3.390	(3.336)
150	5.998	(5.167)	5.480	(5.254)	6.590	(6.580)	4.816	(4.219)
200	7.219	(5.890)	5.912	(5.851)	8.262	(6.109)	7.531	(5.069)
300	10.596	(7.123)	2.888	(2.809)	12.187	(8.327)	11.410	(8.164)

Table 5.1: Time required for Cocktail, SSR, RBP, and CWV to transfer a burst of 100 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

5.3 Discussion

After examining the results of the simulation experiments, it is found that *Cocktail* can generally improve the performance of HTTP persistent connections in a low RTT environment. However, in a high RTT environment, RBP works best among the other proposed solutions.

In a low RTT environment, the total data transmission time is dominated by the unnecessary retransmission timeouts. Hence, *Cocktail* works best in a low RTT environment since it efficiently eliminates the factors that may lead to unnecessary retransmission timeouts.

On the other hand, as delay of the network increases, the total capacity of the network also gets larger. TCP therefore takes more RTTs to ramp up to the maximum window than for a low-delay connection. In fact, several researchers have noted that the TCP implementations now in use are not suitable for high delay-bandwidth networks, such as satellite links. RBP is

Delay	*SSR*		*RBP*		*CWV*		*Cocktail*	
10	4.843	(6.445)	7.430	(5.731)	7.587	(6.567)	5.236	(5.770)
20	6.987	(5.364)	8.430	(6.019)	7.937	(6.020)	4.751	(6.095)
30	5.435	(5.488)	7.023	(6.594)	6.548	(6.871)	5.269	(5.113)
50	6.186	(5.756)	7.765	(7.511)	7.891	(6.395)	5.326	(5.233)
100	5.762	(7.234)	6.768	(6.902)	7.653	(8.246)	5.921	(5.938)
150	10.542	(9.299)	8.765	(8.625)	10.301	(10.195)	9.104	(8.445)
200	12.987	(10.408)	11.399	(10.162)	13.114	(11.989)	12.529	(10.699)
300	18.436	(12.789)	5.801	(5.7111)	18.781	(15.041)	16.826	(12.773)

Table 5.2: Time required for Cocktail, SSR, RBP, and CWV to transfer a burst of 200 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

one of the possible solutions to the problem. As mentioned, instead of sending the entire window of segments in a single burst when the transmission is resumed after the connection becomes idle, the TCP sender sends out the segments in a steady stream over the entire course of a round-trip time. Therefore, RBP works best in high RTT environment.

Delay	*SSR*		*RBP*		*CWV*		*Cocktail*	
10	13.844	(15.778)	19.219	(15.975)	18.999	(16.755)	14.561	(13.234)
20	18.240	(14.667)	19.581	(15.102)	19.267	(15.803)	12.842	(11.703)
30	13.802	(13.223)	16.693	(15.021)	16.023	(17.624)	13.010	(12.067)
50	15.167	(14.810)	19.431	(18.435)	19.087	(17.019)	13.018	(12.193)
100	14.627	(15.010)	17.223	(18.511)	19.355	(18.395)	15.338	(14.824)
150	19.994	(18.129)	17.151	(17.511)	20.792	(19.925)	19.024	(17.451)
200	21.414	(20.186)	19.944	(20.278)	21.101	(22.129)	21.827	(20.195)
300	30.851	(27.398)	14.551	(14.556)	30.295	(27.872)	28.653	(28.016)

Table 5.3: Time required for Cocktail, SSR, RBP, and CWV to transfer a burst of 500 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

Delay	*SSR*		*RBP*		*CWV*		*Cocktail*	
10	23.548	(29.894)	38.215	(31.925)	39.614	(32.779)	26.973	(27.307)
20	39.104	(28.556)	40.248	(30.364)	41.112	(30.779)	29.957	(28.497)
30	27.530	(28.645)	33.275	(30.667)	32.987	(33.600)	25.955	(25.085)
50	28.521	(29.461)	37.997	(34.793)	37.669	(33.432)	25.445	(25.489)
100	29.995	(29.801)	32.651	(30.812)	30.889	(31.793)	28.523	(28.260)
150	33.925	(35.147)	32.831	(34.112)	33.127	(36.530)	32.900	(32.894)
200	41.173	(37.781)	38.912	(35.911)	42.194	(38.455)	40.275	(37.782)
300	55.916	(39.959)	30.123	(28.717)	55.319	(42.781)	53.736	(43.270)

Table 5.4: Time required for Cocktail, SSR, RBP, and CWV to transfer a burst of 1000 segments when the idle TCP connection is restarted. Figures outside and inside the brackets are the results when drop-tail and RED queue management is used respectively.

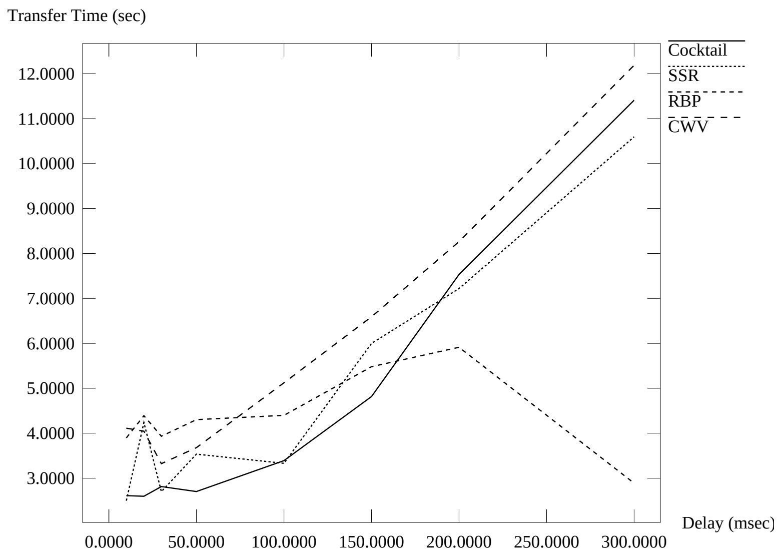

Figure 5.8: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delays when the burst size is 100 segments and the drop-tail queue management approach is used.

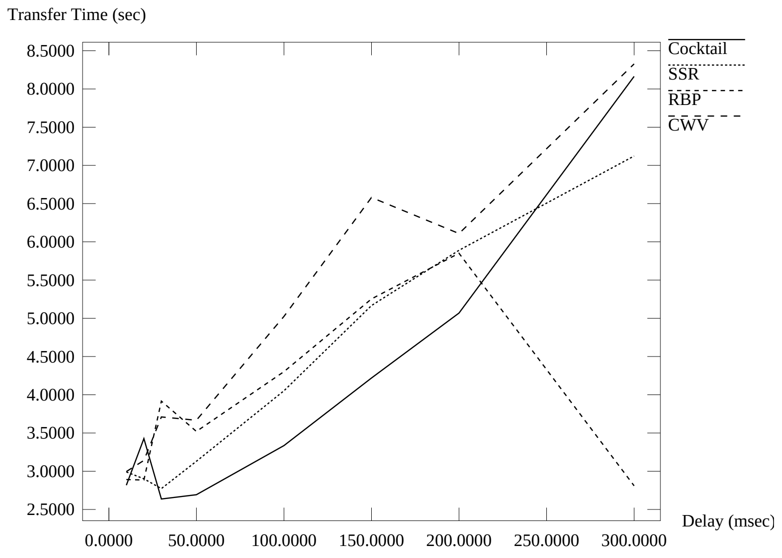

Figure 5.9: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delays when the burst size is 100 segments and the RED queue management approach is used.

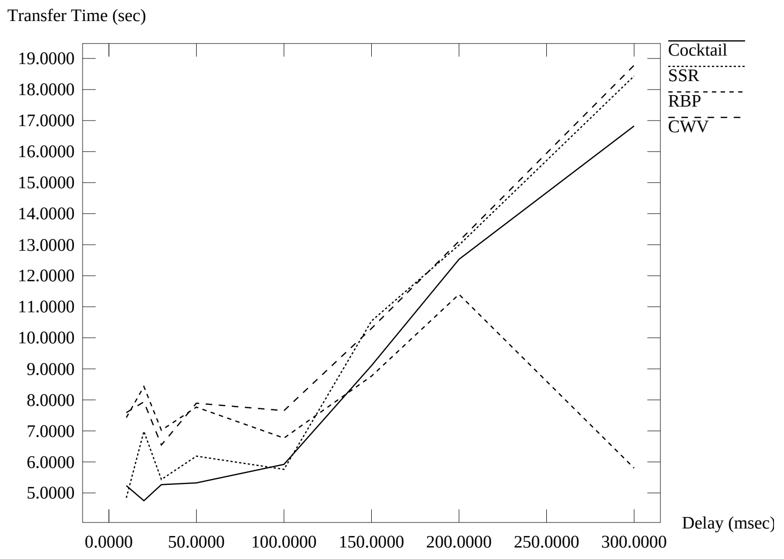

Figure 5.10: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delays when the burst size is 200 segments and the drop-tail queue management approach is used.

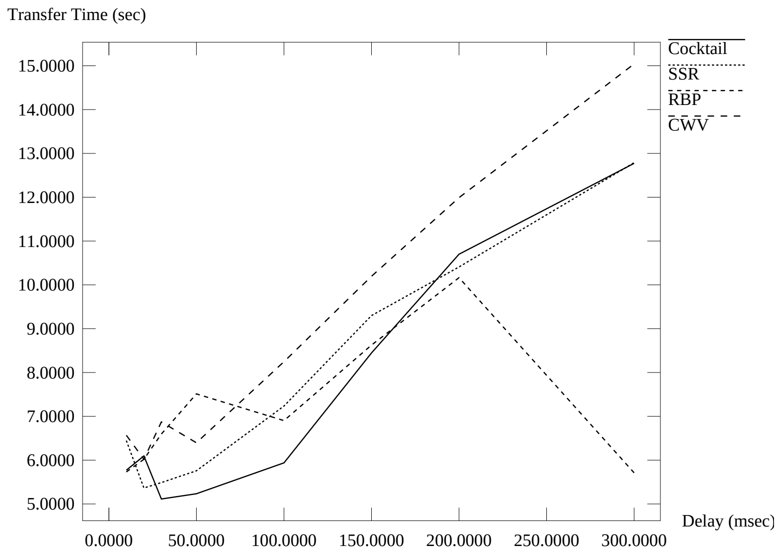

Figure 5.11: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delays when the burst size is 200 segments and the RED queue management approach is used.

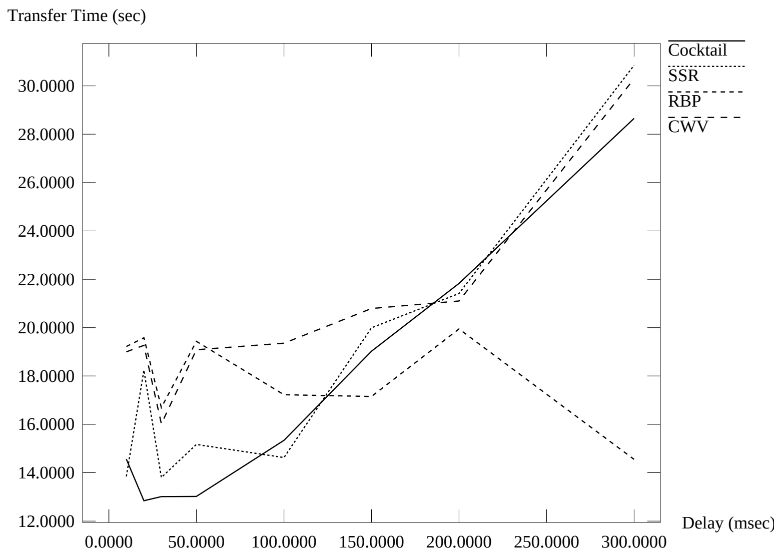

Figure 5.12: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delay when the burst size is 500 segments and the drop-tail queue management approach is used.

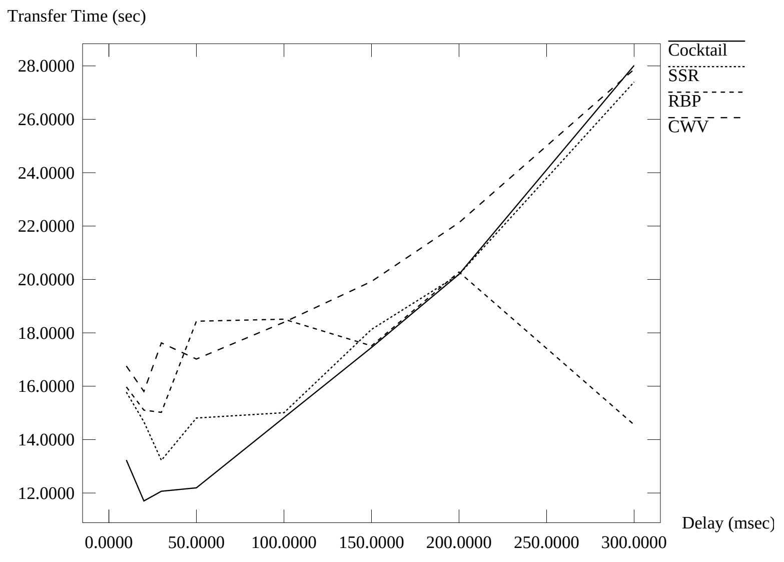

Figure 5.13: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delay when the burst size is 500 segments and the RED queue management approach is used.

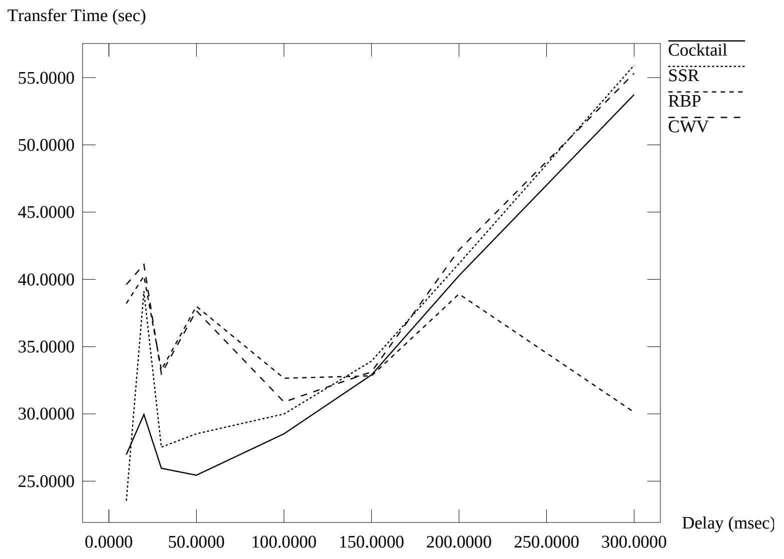

Figure 5.14: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delay when the burst size is 1000 segments and the drop-tail queue management approach is used.

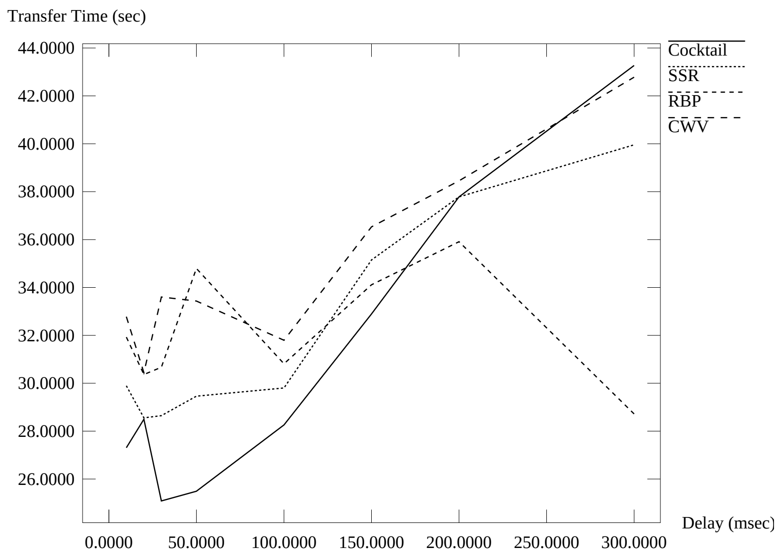

Figure 5.15: Performance comparison of Cocktail, SSR, RBP, and CWV at various propagation delay when the burst size is 1000 segments and the RED queue management approach is used.

Chapter 6

Conclusions

6.1 Conclusions

Cocktail approach is proposed to improve the overall performance of HTTP persistent connections. The changes only require modifications to a TCP sender, and a TCP with these changes is interoperable with any existing TCP implementation. *Cocktail* is composed of several elements. Each element is assigned to tackle a specific problem caused by the interactions between HTTP persistent connections and existing TCP implementations. A number of simulation experiments have been conducted to evaluate the performance of SSR, RBP and CWV against that of *Cocktail*. The results of the experiments show that *Cocktail* generally performs better than the others in a low RTT environment do. On the other hand, RBP works best in a high RTT environment.

6.2 Recommendations and Further Studies

6.2.1 A True *Cocktail* Approach

This study concludes that *Cocktail* is able to improve the overall performance of HTTP persistent connections in low RTT environment. However, in a high RTT environment, it is definitely not the rival of RBP and it is the weakest point of *Cocktail*. One possible way to deal with this shortcoming is to combine their strengths. That is, replacing the first element of *Cocktail* in this way: when RTT is lower than a certain level, use slow-start restart; otherwise, use RBP. The threshold value which determines whether *Cocktail* or RBP should be used should be tunable (manually or automatically) to optimize the overall TCP performance. Future studies can explore this issue in more detail.

6.2.2 Recent Developments in TCP Congestion Control

Recently, researchers have suggested two mechanisms to further improve the performance of TCP. The two mechanisms are: selective acknowledgement (SACK) [13] and explicit congestion notification (ECN) [19].

Selective Acknowledgment

Traditionally, TCP congestion control lies in cumulative acknowledgment scheme. By this, this client does not acknowledge any segments arrived that are not at the left edge of the receive window. There are two consequences:

1. The server, in order to find out each missing segment, has to wait for a roundtrip time; and
2. The server will retransmit those properly-received segments.

Both of these consequences are very inefficient in the sense that TCP will lose the ACK-based clock and hence the overall throughput will be dragged down due to the missing segments. Therefore, SACK is proposed. It works by having the client notifying the server about all the successfully-arrived segments. Then the server only has to retransmit the missing ones. However, the disadvantage of SACK is that it requires modifications to TCP on both the senders and receivers.

Explicit Congestion Notification

Another attempt to further enhance TCP's performance is adopting ECN. However, there are special conditions for running this mechanism.

First, it requires an ECN field with 2 bits. The server will specify the ECT bit to show that the end nodes of the transport protocol are able to handle ECN. In addition, the router will specify the CE bit to notify the end nodes about any congestion. Second, it requires support from the transport layer. To ensure all end nodes are ECN-capable, a negotiation procedure is necessary. It also requires the transport protocol to react properly upon receiving a CE segment. The client would notify the server about the segment. Under the above conditions, the TCP ECN works under the following mechanisms:

1. To ensure the end nodes are ECN capable, there will be negotiation among them at the setup stage;
2. To ensure the client notify the server upon receipt of a CE packet, there will be an ECN-echo flag in the TCP header; and
3. To enable the server to inform the client that the congestion window has been reduced, there will be a congestion window reduced flag (CWR) in the TCP header.

The disadvantage of ECN is that it only works under the above special conditions. In the other words, adopting ECN is impossible unless all the end nodes (i.e. senders and receivers) and intermediate routers are ECN-enabled. Hence, it increases the difficulties during the deployment.

6.3 Final Remarks

Although the above two methods to further enhance the performance of TCP are quite effective, they are, unfortunately, not readily available because extensive support and special conditions are necessary for the mechanisms to work. Instinctively, they are good ideas for improving the performance of TCP if they can work under less additional conditions. Therefore, there are rooms for further studies on enhancing the performance of TCP using SACK or ECN under less restricted environment.

Bibliography

[1] ns–The Network Simulator. http://www.isi.edu/nsnam/ns/index.html.

[2] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 2414, Internet Engineering Task Force, September 1998.

[3] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, Internet Engineering Task Force, April 1999.

[4] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, Internet Engineering Task Force, October 1989.

[5] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. SIGCOMM '88*, volume 18, pages 106–114, August 1988.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[7] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, Internet Engineering Task Force, April 1999.

[8] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC 2861, Internet Engineering Task Force, June 2000.

[9] John Heidemann. Performance Interactions Between P-HTTP and TCP Implementations. *Computer Communication Review*, 27(2):65–73, April 1997.

[10] Janey C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. *Computer Communication Review*, 26(4), October 1996.

[11] Amy Hughes, Joe Touch, and John Heidemann. Issues in TCP Slow-Start Restart After Idle. Internet Draft, Internet Engineering Task Force, March 1998.

[12] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. *Computer Communication Review*, 18(4):314–329, August 1988.

[13] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, Internet Engineering Task Force, October 1986.

[14] Greg Minshall. A Suggested Modification to Nagle's Algorithm. Internet Draft, Internet Engineering Task Force, December 1998.

[15] Greg Minshall, Yasushi Saito, Jeffrey C. Mogul, and Ben Verghese. Application Performance Pitfalls and TCP's Nagle Algorithm. In *Workshop on Internet Server Performance*, May 1999.

[16] John Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, Internet Engineering Task Force, January 1984.

[17] Henrik Frystyk Nielsen, Jim Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network Performance Effects of HTTP/1.1, CCS1, and PNG. In *Proc. SIGCOMM '97*, September 1997.

[18] K. Poduri and K. Nichols. Simulation Studies of Increased Initial TCP Window Size. RFC 2415, Internet Engineering Task Force, September 1998.

[19] K. Ramakrishnan and S. Floyd. A Proposal to Add Explicit Congestion Notification (ECN) to IP. RFC 2481, Internet Engineering Task Force, January 1999.

[20] W. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC 2001, Internet Engineering Task Force, January 1997.

[21] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison Wesley, 1994.

[22] Vikram Visweswaraiah and John Heidemann. Improving Restart of Idle TCP Connections. Technical Report 97-661, University of Southern California, November 1997.

[23] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated*, volume 2. Addison Wesley, 1995.