THIRD-PARTY TCP RATE CONTROL

by

Dushyant Bansal

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2005

Author's Declaration For Electronic Submission of a Dissertation

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

ABSTRACT

The Transmission Control Protocol (TCP) is the dominant transport protocol in today's Internet. The original design of TCP left congestion control open to future designers. Short of implementing changes to the TCP stack on the end-nodes themselves, Internet Service Providers have employed several techniques to be able to operate their network equipment efficiently. These techniques amount to shaping traffic to reduce cost and improve overall customer satisfaction.

The method that gives maximum control when performing traffic shaping is using an inline traffic shaper. An inline traffic shaper sits in the middle of any flow, allowing packets to pass through it and, with policy-limited freedom, inspects and modifies all packets as it pleases. However, a number of practical issues such as hardware reliability or ISP policy, may prevent such a solution from being employed. For example, an ISP that does not fully trust the quality of the traffic shaper would not want such a product to be placed in-line with its equipment, as it places a significant threat to its business. What is required in such cases is third-party rate control.

Formally defined, a third-party rate controller is one that can see all traffic and inject new traffic into the network, but cannot remove or modify existing network packets. Given these restrictions, we present and study a technique to control TCP flows, namely triple-ACK duplication. The triple-ACK algorithm allows significant capabilities to a third-party traffic shaper. We provide an analytical justification for why this technique works under ideal conditions and demonstrate via simulation the bandwidth reduction achieved. When judiciously applied, the triple-ACK duplication technique produces minimal badput, while producing significant reductions in bandwidth consumption under ideal conditions. Based on a brief study, we show that our algorithm is able to selectively throttle one flow while allowing another to gain in bandwidth.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my family and loved ones for their support and assistance in moulding me into who I am today. I am indebted to my parents, Mr. Uttam Bansal and Mrs. Usha Bansal, for their love and guidance and for serving as my role models in life. They nurtured my gift for deeper thinking and encouraged me to aspire to greater things. My sisters, Aasthaa Bansal and Iti Bansal, helped me in keeping the humour alive in our lives. The love and support provided by my fiancée, Meera Chawla, in helping me carve out my own path in life was invaluable.

I would like to express my gratitude to my supervisor, Dr. Paul Ward, for harnessing my enthusiasm and providing me with the opportunity to do a Masters under him. His constructive criticism along the way helped shape my thesis into its current form. Thanks to his mentoring, I have developed a global perspective on life and see the role of science and technology in a broader context.

I would like to thank my examiners, Dr. Srinivasan Keshav and Dr. Sagar Naik, for taking the time to read my thesis. Their feedback was very useful in improving the quality of my thesis.

It was a joy engaging in passionate technical debates and explorations with some of the graduate students in the Networks and Distributed Systems research group. In particular, I would like to thank Evan Jones, Amol Shukla, and Kamran Jamshaid. Interacting with some of the professors in the same research group was an eye-opening experience.

I appreciate the funding provided by the Natural Sciences and Engineering Research Council of Canada, Sandvine, and Dr. Ward. The funding came in the form of the Industrial Postgraduate Scholarship, and research and teaching assistantships. Without these funds, this project would not have been possible.

Finally, I would like to acknowledge everyone else who has made a difference in my life and helped me reach where I am. In the context of my accomplishments, your influence on my life, positive or negative, direct or indirect, will always be appreciated.

For The Further Exploration of Space and Time,

Two constraints by which we, as Computer Engineers, must abide.

CONTENTS

FIGURES

TABLES

1 INTRODUCTION

Internet Service Providers (ISPs) purchase bandwidth from backbone network operators and sell the bandwidth to individual customers for Internet access. Every packet transition from one ISP's network to another, or between ISPs and backbones costs money. Thus, ISPs desire to limit the amount of outgoing traffic that would incur such cost, while still providing good service to their customers. Excess traffic causes problems on a technical level too. Excessive packets sent in a network create long packet queues at routers, which leads to delays and may lead to buffer overflows. This results in retransmissions, which consequently reduces the efficiency of the network. Many strategies are used to control the volume of traffic:

1. More bandwidth and better equipment can be acquired. However, bandwidth and new equipment are costly. Furthermore, the bandwidth-hungry applications will eventually consume the newly available bandwidth, thus only delaying the problems caused by excessive traffic [14].

2. A proxy cache can be used to cache files that are downloaded by users. However, this does not mitigate file-transfer requests from outside the ISP's domain.

3. ISPs that shape their traffic typically install a traffic shaper inline with their router. This shaper may form part of newer routers. This mode of deployment is shown in Figure 1.1. Inline traffic shaping can be used for the following purposes:

   (a) Block all ports used by bandwidth-hungry applications.

   (b) Implement transfer caps to limit the total number of bytes downloaded or uploaded by the user.

   (c) Limit the instantaneous bandwidth consumed by the user.

   (d) Drop packets belonging to bandwidth-hungry applications.

   (e) Downgrade the transmission priority of packets belonging to bandwidth-hungry applications.

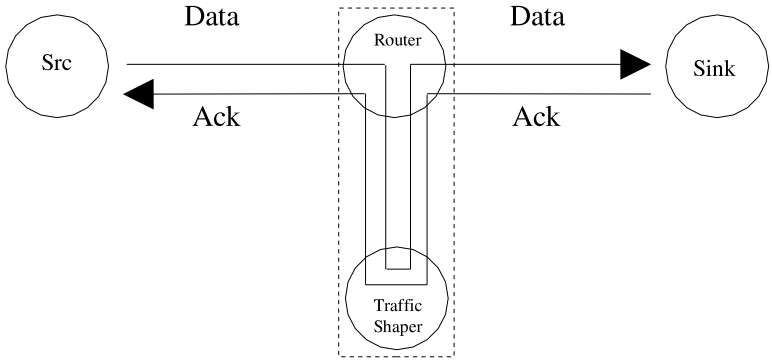


Figure 1.1: Inline traffic shaper deployment

(f) Monitor all protocol activity and determine which users have certain files. When a new file-transfer request is encountered from within the ISP domain, the traffic shaper can proxy the request to a user who has the same file and is within the same, or a peer, ISP.

(g) To limit upload bandwidth, limit how many connections can be made from outside the ISP's domain with subscribers in the ISP's domain.

1.1 MOTIVATION

There are at least five factors an ISP would consider when purchasing traffic-shaping equipment: reliability, correctness of operation, non-malicious behaviour, future upgrades, and customer support. Reliability means that the software and hardware remains functional all the time. Redundancy can be used to aid in this. Correct operation means that correct flows are identified for traffic shaping and the traffic shaping techniques are applied correctly. Non-malicious behaviour means that the traffic shaper would not collect personal information, such as credit card numbers and passwords. In order to retain its customers, an ISP must provide competitive service at all times. This means that if traffic patterns change, the ISP must be able to upgrade its equipment with new functionality, and if any equipment fails, the ISP must be able to receive prompt customer

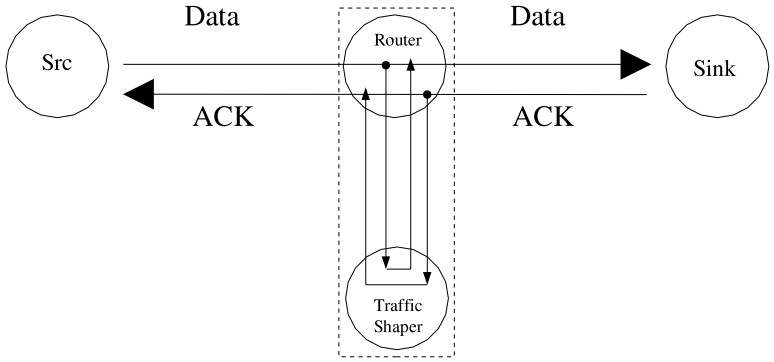


Figure 1.2: Third-party shaper deployment

support from the equipment manufacturer.

A big ISP would only purchase an inline traffic shaper from a reputable vendor (within budget constraints). A reputable vendor would have an established history of making quality products and good customer support. However, it would be difficult for the ISP to trust a small vendor with no brand name recognition. An approach that can be used in such situations is to install the traffic shaper in parallel with the router equipment, as shown in Figure 1.2. We call this the "third-party" mode of deployment.

In third-party mode, the traffic shaper receives copies of all packets and can inject new packets into the flow, but it cannot modify, drop, or delay the existing packets themselves. This means that a third-party traffic shaper cannot prevent the original packets from reaching their intended destination. Some advantages of a third-party deployment are as follows:

1. It is quite possible that the ISP may already have an inline traffic shaper and may be using the third-party traffic shaper only for additional "nice-to-have" traffic shaping features. Thus, third-party deployment provides an alternative to test new experimental techniques for traffic shaping.

2. In the same light, even if the third-party traffic shaper were to fail, Internet connectivity with basic traffic shaping would still be alive.

3. The cost of developing a piece of software that operates correctly 99.999% of the time would be significantly more than the cost of one that operates correctly 95% of the time. An inline traffic shaper would need to be highly reliable, since it lies directly in the path of the traffic. However, a third-party traffic shaper does not lie directly in the path of the traffic. Hence, it may not have to be as reliable. This would further reduce the cost of a third-party traffic shaper.

4. By providing a venue for small companies to enter the traffic shaping market, third-party deployment spurs competition.

5. A third-party deployment mitigates the problem of loss of connectivity by decoupling router operation and failure from the reliability of the traffic shaper. This helps small ISPs that do not wish to invest in equipment redundancy but do want to shape their traffic.

1.2 GOAL

The goal of this thesis is to show that third-party traffic shaping is possible. To this end, we design and simulate a technique for third-party traffic shaping.

Today, up to 60% of total traffic on a residential network is consumed by Peer-to-Peer (P2P) file-sharing [14]. Other venues for file downloads are the File Transfer Protocol (FTP) and the HyperText Transfer Protocol (HTTP). P2P file-sharing, FTP and HTTP typically occur over the Transmission Control Protocol (TCP). Hence, our work deals only with solutions based on TCP. Furthermore, file transfers are typically long-lived flows. Thus, our study will only look at long-lived TCP flows.

Any serious start-up company wanting to enter the traffic shaping market through third-party deployment would steer clear of malicious behaviour. Hence, this thesis focuses only on the correctness of operation.

1.3 CONTRIBUTION

This thesis provides the following contributions to scientific and engineering knowledge:

1. We have designed an algorithm that performs rate control in third-party mode

based on triple-duplicate ACKs. The pseudocode for this algorithm is provided in this thesis.

2. Through simulation we have shown that our algorithm produces significant bandwidth and goodput reduction under ideal conditions.

3. The relative latencies between the sender, router and receiver are irrelevant for bandwidth and goodput reduction with our algorithm.

4. Our bandwidth and goodput reduction as well as badput only depend on the bottleneck bandwidth and end-to-end RTT and not the individual delays and bandwidth differences encountered along an end-to-end path.

5. The bandwidth reduction with our algorithm is purely a function of bandwidth·RTT product and not individual bandwidths and RTTs.

6. We have studied the behaviour of our algorithm under different triple-ACK frequencies. We have determined that there are three regions of bandwidth reduction: zero reduction, rise, and saturation. We have determined that each frequency results in a unique segment pattern. We have used this segment pattern to explain the maximum bandwidth reduction achieved for each frequency. We find that the bandwidth and goodput reduction produced by our algorithm increase steadily from frequency 2 to frequency 6 and then steadily drop up to frequency 200. A triple-ACK frequency of 1 does not produce any bandwidth reduction. The badput decreases steadily from frequency 1 to frequency 200.

7. We have discovered that maximum bandwidth reduction is achieved with a triple-ACK frequency of 6. For this case, we have provided a mathematical model for the bandwidth and goodput reduction achieved by our algorithm under ideal conditions. We have also quantified the badput produced.

8. Under ideal conditions, using a maximum window size of 20 segments and triple-ACK frequency of 6, we achieved a bandwidth reduction of 4%-85% for client-server downloads and 12%-85% for P2P downloads. At the same time, we produced a badput on the order of 17%. Using a triple-ACK frequency of 200 in

our algorithm, the badput can be reduced to as little as 0.5%. However, then the maximum bandwidth reduction also reduces to 21%.

9. Our triple-ACK algorithm can selectively throttle one flow and allow another to gain in bandwidth consumption when the two flows are running over a bottleneck link. Our algorithm is successful under ideal conditions as well as in the presence of 5% chance of segment loss or delay.

1.4 ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter 2: We describe the basics of TCP and existing techniques for rate control.
- Chapter 3: We present our novel approach for third-party TCP rate control based on threefold acknowledgement duplication to force congestion control. We also outline the pseudocode of our algorithm.
- Chapter 4: We describe the variables that we considered in our simulations, the file-downloading environment we attempted to model, the simulation models we used, and the specific parameters we used in our simulation experiments.
- Chapter 5: We describe the results of simulating an upload model with computational elements typical of a high-speed Internet deployment. We use different frequencies of triple-ACK duplication and obtain numbers for bandwidth reduction and badput. Then we describe the results of simulating a simplified version of the previous model and show that the results are equivalent. Next, we simulate a download model, where the sender and receiver are reversed with respect to the router. We see that the results are equivalent to the upload model, implying that the relative latencies do not matter. Finally, we simulate a model that is symmetric with respect to bandwidth and latency. We obtain results for bandwidth and goodput reduction, and badput.
- Chapter 6: We use a wider range of triple-ACK frequencies with our symmetric model and determine the maximum bandwidth and goodput reduction and the

amount of badput generated in the process. We explain the numbers in terms of the segment patterns that we observed over multiple round trips. We observe that there is no bandwidth reduction for smaller bandwidth·RTT products and list some factors that would need to be modeled in order to determine at what point bandwidth reduction would start. We also explain how the frequency 1 case behaves differently compared to other frequencies.

- Chapter 7: On discovering that a triple-ACK frequency of 6 gives us the maximum bandwidth reduction, we analyze this case in more detail in this chapter. We start with a packet-level analysis to understand how the segment pattern for frequency 6 arises. We model the bandwidth and goodput reduction mathematically. Finally, we vary the TCP window limit and observe its effect on bandwidth reduction.

- Chapter 8: In this chapter, we run our simulations using lower bandwidths, which would be typical of P2P file-sharing.

- Chapter 9: We generalize the behaviour observed in the process of triple-ACK-based third-party rate control under ideal conditions.

- Chapter 10: In this chapter, we run two flows over a bottleneck link and selectively throttle one of them. We show that this allows the other flow to gain bandwidth under both ideal conditions as well as when we use a 1% probability of segment loss and a conservative 4% probability of segment delay.

- Chapter 11: Finally, we describe the conclusions we draw from our simulation studies and outline what work remains for the triple-ACK technique to be applicable for deployment.

2 BACKGROUND AND RELATED WORK

In this chapter we describe the basic operation of TCP and discuss some existing methods for TCP congestion control. Some fundamental definitions are provided in the process, which are used in the rest of the thesis.

2.1 BASIC MECHANICS OF TCP

TCP is a byte-stream-based connection-oriented protocol that uses sliding windows and acknowledgements to provide reliable, in-order delivery of data to the receiving application. It has facilities for flow control and multiplexing multiple flows between the same pair of hosts [27]. We will use the term "packet" to refer to a set of bits that are transmitted over a network as a single collection of information. The bits in the packet define the intended protocol context of the packet and its communicating end-points. We define a number of other terms below that we make use of in the rest of this thesis.

Segment A TCP segment is one that contains the payload data and TCP/IP headers [3]. A TCP acknowledgement packet (hereafter, ACK) is also called a segment, even though there is no payload. The TCP header is shown in Figure 2.1. As shown, TCP uses a 32-bit sequence number field and a 32-bit acknowledgement number field. Both these numbers are of byte granularity. The sequence number indicates the numerical sequence of the first payload byte carried in that segment, while the acknowledgement number indicates the numerical sequence of the first payload byte expected after the last correctly received in-sequence data byte. The header can have a number of options, thus making the total header length variable. Thus, the 4-bit header length serves the purpose of telling the receiver how long the header really is. The last item in the TCP segment is the data, as shown in Figure 2.1.

Sender Maximum Segment Size (SMSS) This is the maximum size segment without the TCP/IP headers and options that the TCP sender can transmit [3]. The value of SMSS is based in part on the value of the Receiver Maximum Segment Size.

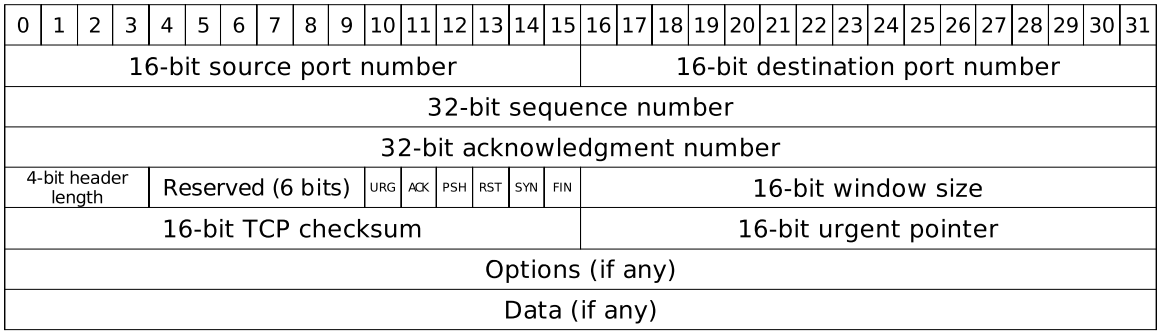


Figure 2.1: TCP header [27, 32]

Receiver Maximum Segment Size (RMSS) This is the maximum size segment without the TCP/IP headers and options that the TCP receiver can accept [3]. By default, RMSS has a value of 536 or 512 bytes [33] but can be overridden using the Maximum Segment Size (MSS) option sent by the receiver in the SYN or SYN-ACK segment during connection setup. Connection setup is described in Section 2.1.1.

Full-sized segment This is a data segment that contains SMSS bytes. This is the largest segment there can be.

Receiver window ($rwnd$) As part of receiver-imposed flow control, the receiver can advertise how much space it has available in its receive-buffer. The sender must not send more data than this advertised value. This advertised value is called the receiver window [3] and is shown in Figure 2.1 as the 16-bit window size field. Data received outside this window is dropped.

Congestion window ($cwnd$) As part of sender-imposed flow control [33], the congestion window variable limits how much data the TCP sender can send. Combining the limitations placed by the congestion and receiver windows, the condition is stated in the form that the sender TCP cannot send data with a sequence number greater than the sum of the highest acknowledged sequence number and the minimum of $rwnd$ and $cwnd$ [3].

Initial window (IW) The initial size of the congestion window is called the initial window [3]. TCP requires a three-way handshake to setup a connection between two TCP entities. This handshake procedure is described in Section 2.1.1. Once the handshake is done, the sender TCP starts sending data by transmitting IW many data segments, subject to the limit placed by $rwnd$. It is allowed to be up to 2 full segments in size, although other studies present the case to increase the value of IW to between 2 and 4 full segments, up to a maximum of 4380 bytes [2, 25]. However, studies have found that 42% of the Web servers on the Internet use an IW of 1 segment, while 54% use 2 [21].

Loss window (LW) This is the value assigned to the congestion window, when the sender TCP times out waiting for an ACK for a data segment [3]. It is set to 1 segment [3].

Flight-size This is the amount of data that has been transmitted but not acknowledged yet [3]. The flight-size is limited by the minimum of the $cwnd$ and $rwnd$.

Slow-start threshold ($ssthresh$) As the value of $cwnd$ increases, $ssthresh$ acts as the switching point between slow-start mode and congestion avoidance mode. These two modes are described in Section 2.1.3. The initial value of $ssthresh$ may be set arbitrarily high, although the size of the advertised receiver window, $rwnd$, may be used [3].

2.1.1 ESTABLISHING AND TEARING DOWN CONNECTIONS

TCP connections are set up using a three-way handshake. The handshake begins with the initiator sending a synchronize (SYN) segment to the receiver, the receiver acknowledging it with a SYN-ACK segment, and finally the sender acknowledging the SYN-ACK with an ACK segment. To send a SYN, the initiator turns on the SYN bit in the header and selects its initial sequence number. To send a SYN-ACK, the other end turns on both SYN and ACK bits and selects its own initial sequence number. Finally, to send an ACK, the initiator turns on only the ACK bit. Thereafter, data can start being exchanged by following the congestion control algorithms described in Section 2.1.3. When one side is finished sending data to the other, it closes its sending side of the connection by

sending a finish (FIN) segment to the other side. The other side responds with an ACK. The other side also sends its own FIN segment once it has completed sending data. It expects an ACK in return. A FIN segment is sent by turning on the FIN bit in the TCP header. The SYN, ACK, and FIN bits are shown in Figure 2.1.

2.1.2 DATA TRANSFER

Data transfer takes place using a sliding window protocol. A sliding window means that a TCP sender maintains a limit on how many data bytes it can transmit before receiving an ACK for any one of them, and a TCP receiver maintains a limit on how many data bytes it can buffer before the higher protocol layer consumes the data. On the sending end, the left edge of the window as shown in Figure 2.2 slides forward in the direction shown, when a contiguous set of data packets starting from the left edge are acknowledged. On the receiving end, the right edge of the window as shown in Figure 2.2 slides forward in the direction shown when a contiguous set of data packets starting from the right edge are consumed by the higher software layer. Likewise, when the TCP sender transmits new data segments, the right edge of its window slides forward, and when a new data segment is received by the TCP receiver, the left edge of its window slides forward. The right edges of the two windows are grown and shrunk in response to changing network and end-host conditions.

2.1.3 CONGESTION CONTROL ALGORITHMS

In order to avoid congestion on the Internet, a TCP flow must follow a “conservation of packets” principle [16]. This means that once running without packet loss with a full window of data in transit, no new packet is injected into the network until an old packet leaves. The “conservation of packets” principle is violated if the TCP end-points misbehave and inject less or more than a window-full of segments into the network or if, due to resource limitations along the network path, the TCP end-points cannot inject a window-full of segments into the network. To deal with each of these issues, four inter-twined algorithms were designed for TCP, *i.e.*, slow start, congestion avoidance, fast retransmit and fast recovery [3, 16].

1. Slow start: TCP is a self-clocking protocol [16]. This means that transmitted data

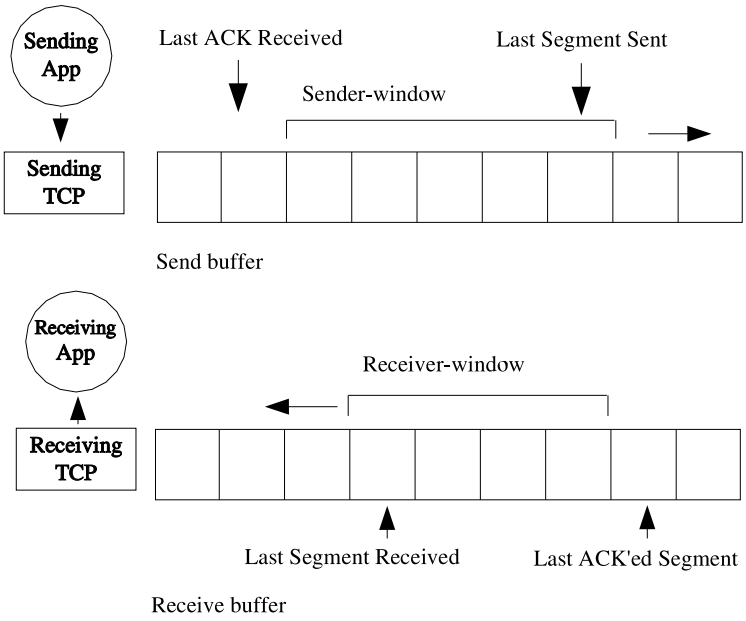


Figure 2.2: TCP sliding-window protocol [24]

segments cause ACKs to be transmitted by the receiver on reception of these data segments. These ACKs cause more data segments to be transmitted by the sender on reception. Thus, the ACKs act as a clock for the data stream. However, a TCP network typically consists of more than just the communicating end-hosts. There may be intermediate routers and other network elements. These elements may need to queue packets, which means that they have a finite-sized buffer allocated for this purpose. Since the buffer is of finite size, it will run out of space when there is faster incoming traffic than the bandwidth of the outgoing link. As a result, if the TCP sender starts with a large transmission and there are slower links on the inter-network between the two end-points, it may result in lost packets, which would affect the throughput. To avoid this situation, TCP follows an algorithm called “slow start”, which allows it to probe the network for available capacity by transmitting a small number of data segments, steadily increasing the number of data segments in flight [33].

Slow-start is achieved by starting the $cwnd$ at a value of IW. For every ACK received, $cwnd$ is increased by at most SMSS bytes, or 1 full-sized segment, for every ACK that acknowledges new data. This results in two more data segments being transmitted, one data segment due to the fact that the sender has received an ACK and another due to the fact that $cwnd$ has been increased by 1 segment [3, 16]. Thus, the flight-size is doubled every Round Trip Time (RTT), resulting in an exponential $cwnd$ increase vs. RTT. This growth continues until the $cwnd$ reaches or exceeds the value of the $ssthresh$, or the capacity of the network is reached and an intermediate router starts discarding packets. When $cwnd > ssthresh$, the TCP switches to the congestion avoidance algorithm. When $cwnd == ssthresh$, either slow start or congestion avoidance may be used [3]. When a data segment is lost, either the retransmit timer for that data segment will expire or TCP will receive three duplicate ACKs, as per the Fast Retransmit algorithm below. In both cases, the sending TCP will react by retransmitting the data segment understood to be lost. This is further discussed below. Slow-start was performed in the 4.3BSD Tahoe release only if the receiver was on a different network. However, starting with the 4.3BSD Reno release, slow start has always been performed [32].

2. Congestion avoidance: In congestion avoidance mode, the value of $cwnd$ is increased as per Equation (2.1) if $cwnd$ is specified in bytes [3].

$$cwnd\ (bytes) += \frac{SMSS * SMSS}{cwnd\ (bytes)} \tag{2.1}$$

If $cwnd$ is specified in segments, $cwnd$ grows as per Equation (2.2) [16].

$$cwnd\ (segments) += \frac{1}{cwnd\ (segments)} \tag{2.2}$$

Equation (2.2) would necessarily require the TCP implementation to use a floating-point representation for $cwnd$ to achieve a similar granularity as Equation (2.1). The effect of congestion avoidance is to increment $cwnd$ by approximately 1 full segment every RTT, resulting in a linear $cwnd$ increase vs. RTT.

When packet loss occurs, the value of $ssthresh$ is set to no more than that specified

in Equation 2.3 [3].

$$ssthresh = max(\frac{flightsize}{2}, 2 * SMSS) \tag{2.3}$$

As was defined earlier, flight-size is the amount of data in transit and is limited by the minimum of the $cwnd$ and $rwnd$. It should also be noted that this equation would set the value of $ssthresh$ to a minimum of 2 full segments.

A sender can detect packet loss by two means: a timeout or receiving three duplicate ACKs. If a timeout occurs, the $cwnd$ is set to LW. When another ACK arrives acknowledging new data, the value of $cwnd$ will be incremented by at most SMSS many bytes if it is in slow-start mode, or as per Equation (2.1) if it is in congestion-avoidance mode. By reducing its rate in this way, this algorithm infers network congestion from packet loss. This is because this algorithm assumes that the only other cause of packet loss, signal corruption, has a very low probability ($\ll 1\%$) [16, 33].

The value of the retransmit timeout (RTO) needs to be set appropriately so that retransmissions are not caused due to delayed ACKs and yet dropped segments are detected promptly. The RTT between two nodes is the time interval between sending a data segment and receiving an ACK for it. The RTO is estimated dynamically based on measurements of the the mean and variance of the RTT [15]. The timestamp option of TCP can be used towards this end. The variance itself varies according to changing network conditions, such as load. Various studies have presented algorithms to accurately estimate the RTO [16, 23, 27].

3. Fast retransmit: When the receiver TCP receives an out of sequence data segment, it is required to transmit a duplicate of the last ACK it sent out [33]. Thus, it would acknowledge receiving data bytes up to the last in-order segment, but not those in the one received out of order. It follows that when a data segment is lost in the network, the receiver would send out a duplicate ACK for every subsequent data segment it receives. Since a duplicate ACK could then be caused by a reordering of data segments, or by an actual segment drop, or due to replication of ACK or data segments by the network, the sender waits for a small number of duplicate ACKs

to be received to decide what to do [3]. Three or more duplicate ACKs in a row are used to infer that a segment has been lost. The sender TCP then retransmits the missing segment immediately, without waiting for the retransmit timer to expire, thus making TCP more efficient at transmitting data. This process of immediate retransmission on receiving three duplicate ACKs is called fast retransmit. It first appeared in the 4.3BSD Tahoe release and it was followed by slow start [33].

4. Fast recovery: Starting with the 4.3BSD Reno release, once the sender has retransmitted the missing data segment in response to the three duplicate ACKs, it enters congestion-avoidance mode rather than slow-start mode, if it is not already in congestion-avoidance mode [33]. This algorithm is called fast recovery. The rationale for not going into slow start is that the three duplicate ACKs were received in response to three other data segments that have left the network now. This means that data flow is still alive so going into slow-start mode would reduce the data flow too severely.

   The mechanism of going into congestion avoidance involves two steps. First, $ssthresh$ is set to half the current value of $cwnd$, but no less than two segments. Second, $cwnd$ is artificially inflated to $ssthresh+3*SMSS$ to reflect the three additional data segments that are now in the receiver TCP's buffer. Thus, the sender TCP's $cwnd$ is greater than its $ssthresh$, which puts it in congestion avoidance mode. Each subsequent duplicate ACK further inflates the $cwnd$ by SMSS, again to reflect another data segment that has reached the receiver TCP's buffer. If the new $cwnd$ and $rwnd$ allow, new data segment(s) are transmitted. When the missing data segment reaches the receiver TCP, the receiver TCP sends out a cumulative ACK acknowledging all contiguous data received up to that point. When the sender receives this ACK acknowledging new data, it deflates its $cwnd$ to $ssthresh$.

TCP stacks have been named based on the version of the BSD operating system in which the different congestion control algorithms appeared. Thus, TCP with only slow start, congestion avoidance, and fast retransmit is called Tahoe TCP, while TCP with fast recovery as well as the first three algorithms is called Reno TCP [9]. It has been found that Reno TCP is unable to recover efficiently from multiple segment losses within the same RTT [9]. The NewReno extension and the Selective ACK (SACK) option for TCP

were developed to deal with this [12, 20]. However, this thesis uses Reno TCP. Hence, we will not investigate the NewReno extension and the SACK option.

2.2 EXISTING TECHNIQUES FOR RATE CONTROL

Techniques for TCP-specific rate control require one or both of the following:

- Exploit existing TCP features to modify the existing flow so that end-nodes react appropriately and reduce their transmission rate.
- Install new features on the end-node TCP stacks so that transmission speeds can be reduced on an end-to-end basis.

To exploit existing TCP features, existing TCP segments can be modified, delayed or dropped, or new TCP segments can be injected into the flow. Only an inline traffic shaper can modify, delay or drop a segment. Both inline and third-party traffic shapers can inject new segments. Inline traffic controllers would use the more-direct means rather than injecting new segments, if an equivalent impact can be achieved. However, a third-party controller can only pursue the packet-injection option to reduce bandwidth.

Implementing new features on an end-TCP stack would involve persuading users to install those features. An ISP would have little capacity to pursue this option. Thus, a third-party controller has to make use of features that are already implemented on end-node TCP stacks to achieve its goal.

In this section, we describe some of the existing rate-control techniques. This serves the purpose of giving the reader an overview of the rate-control space, and also allows us to detail what parameters might be controlled by the traffic shaper.

2.2.1 IN-NETWORK RATE CONTROL

An inline traffic shaper has control over all the packets passing through that router. It can modify the contents, including the header, of the packets, drop packets, or delay them in order to reduce the bandwidth. The sending rate of any TCP sender is controlled by the following five things: availability of data, the congestion window size, the receiver window size, the round trip time (RTT), the rate of acknowledgements, and how often it incurs timeouts.

It is not possible to control what data an end-user actually holds. However, the amount of data transmitted out of or into an ISP domain can be. The traffic volume depends on the nature of the applications. If data requested by an ISP customer can be fetched from another internal user instead of an outside source, it cuts down on download traffic. If data requests coming from outside the ISP domain can be limited, it cuts down on the upload traffic. Furthermore, the view of world as seen by internal users or external users can be controlled by modified control information, such as search hits. However, any manipulation of this nature is inherently application specific. Thus, for each new application, a new shaper technique must be developed, and is thus costly to implement and deploy. This approach has been used to control Peer-to-Peer traffic-flow rates [14].

The congestion-window size of a sender TCP, if smaller than the receiver-window size, can limit the rate of TCP traffic flows [3]. The Random Early Detection (RED) technique [13] operates at the gateway level. It manipulates the congestion window size by monitoring average queue size for each output queue and preemptively discarding selected TCP segments (it can alternatively mark a bit in the packets but that requires co-operation from the end host's TCP stack). The probability that a segment of a particular connection is discarded is proportional to its share of the throughput through the gateway. A discarded segment would result in three duplicate ACKs or a retransmit timeout, causing the sender to go into fast recovery or slow start, depending on the TCP variant used. In either case, the congestion window would be reduced. Our triple-ACK duplication mechanism, described in Section 3.1, is based on this approach. However, not being in the direct path of the traffic, we cannot drop segments.

Similarly, the receiver window size, if smaller than the congestion-window size, can limit the rate of TCP traffic flows. Several techniques have therefore been developed based on artificially reducing the size of the receiver window below that which is advertised by the receiver [5, 6, 17]. We investigated a similar technique in third-party mode, where we injected a duplicate ACK with the receiver window size set to zero. Unfortunately, this technique did not work, as we will describe in Section 3.2.

TCP is proportionally fair to the inverse of the round trip time (RTT) [17]. The longer the RTT, the lower the bandwidth for the stream. The RTT can be increased, if the traffic shaper introduces a delay between subsequent ACKs, thus slowing down the rate at which the sender TCP clocks its output data, thereby reducing bandwidth. Similarly,

the faster the rate of ACKs, the faster the sender TCP's congestion window will expand, resulting in a larger data-output rate overall.

There are a number of other inline flow-control techniques that are independent of the transport protocol used, including token and leaky buckets [35] and maintaining some unused bandwidth on a link at all times [1] to provide a buffer for bursty traffic. None of these approaches seem to provide insight for the design of third-party shapers.

2.2.2 END-NODE RATE CONTROL

Rather than introduce in-network elements to shape current traffic, the protocol can be updated on the end-hosts. An updated TCP stack allows one side to inform the other of congestion, for one side to infer impending congestion from the current network dynamics [7, 8, 19], or for the receiver to selectively acknowledge which data segments it has received to avoid unnecessary retransmissions and speed up transmissions of the necessary segments, as done in the SACK options [20]. It does not appear that a third-party rate controller can implement or use approaches based on the end node.

2.2.3 MIXED-METHOD RATE CONTROL

A mixed approach to bandwidth management is one that combines the first two techniques. Some algorithms are implemented on the end-node stacks and an in-network element is also used. The two would work together to control network utilization. A typical example in this area is the Explicit Congestion Notification (ECN) scheme [28], a modification of RED that avoids unnecessary packet drops. This scheme sets an ECN bit in the TCP/IP headers. The transmitter is expected to respond to these ECN bits in the same way as fast recovery without the segment retransmission. However, studies have found that 93% of Web servers are not ECN-capable and that only 0.2% of the clients advertise the capability to use Explicit Congestion Notification (ECN) [21]. Clearly, by being dependent on network as well as end-node support, ECN becomes a less desirable choice for traffic rate control.

Other techniques that fall in the mixed category include Random Early Marking [4] and source-quench Internet Control Message Protocol (ICMP) messages [26]. Source-quench messages were designed to be sent by a router or destination host to the sending

host when it started approaching its capacity limit or if it discarded a segment, so that the sending host would slow down [26]. However, vendors started implementing source quench to alternatively indicate a burst causing massive overload or a burst slightly exceeding reasonable load, causing problems for end-nodes as to what they should understand a source quench message to mean [18]. Furthermore, middleboxes today tend to suppress ICMP messages of any format, so source quench messages may not always achieve their desired purpose [21].

3 PROPOSED SOLUTIONS

We have designed a mechanism for third-party flow-control using triple-ACK duplication. It is based on the indirect manipulation of the sender's congestion window threshold, and is inspired by the RED mechanism. We also investigated another technique, but found that it was infeasible. This technique was based on an injection of zero-window ACKs to manipulate the sender's view of the receiver window size. We now describe the triple-ACK duplication algorithm in detail, and show how it differs from existing techniques.

3.1 TRIPLE-ACK DUPLICATION

Our triple-ACK duplication algorithm manipulates the congestion window and threshold of the sending TCP entity. The technique is illustrated in Figure 3.1. To do so, it requires that the sender TCP agent is executing the fast-retransmit protocol. The manipulation is caused by sending out three duplicate ACKs for some of the ACKs the sender has seen. When the TCP sender receives four ACKs with the same sequence and acknowledgement numbers, it will retransmit the data segment that it understands to be lost and invoke congestion control. Depending on the particular variant, it will either enter slow start (Tahoe) or fast recovery (Reno). In either case, the congestion window and threshold are reduced. However, studies have found that 94% of Web servers successfully tested for congestion window halving do in fact do so [21]. Hence, we will assume that our end-host TCP stacks have Reno-style fast recovery. The reduction in the sizes of the congestion window and threshold should reduce the rate at which the sender transmits data, thus reducing bandwidth consumption. However, we must note that the retransmissions caused by invocation of fast retransmit will mitigate the bandwidth reduction to some degree, as well as reducing the goodput of the flow. Such a situation would be equivalent to a network that experienced frequent packet re-ordering.

Our triple-ACK duplication algorithm is similar to the RED technique, in that it forces a segment retransmission, and causes the congestion window and threshold to be reduced. However, it differs in that the segment retransmitted has, in fact, already

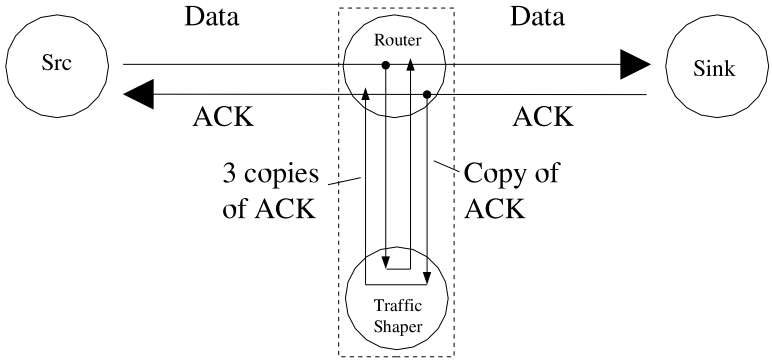


Figure 3.1: Triple-ACK duplication algorithm

been seen by the receiver. As such, there is only a brief interval during which the three duplicate ACKs can be transmitted to the sender. If they do not arrive before the next ACK from the receiver, the sending TCP entity will likely ignore them. In the case of the RED technique, the segment is genuinely dropped, and the remainder of the RED technique is simply the normal TCP reaction to the loss of a segment. As a result, the RED technique will work regardless of which TCP flavour is used. We must emphasize, however, that the RED technique is not feasible in third-party deployment, as it requires the removal of a packet from the packet stream, which is impossible for a third-party.

We also note that others have observed the behaviour of TCP Tahoe using a combination of duplicate ACKs and retransmit timeouts [10]. However, our work is different, because we are not using Tahoe TCP but Reno TCP, which has fast recovery. Furthermore, we deliberately inject three duplicate ACKs and by virtue of operating in third-party mode, do not suffer from retransmit timeouts. Finally, our motive is different, as our goal is to reduce bandwidth-consumption and not to study any unusual behaviour observed of TCP.

Our algorithm exploits the fact that TCP requires all three duplicate ACKs to be exact duplicates, preventing it from knowing which segment those ACKs were sent in response to [31]. TCP may not have even sent out any additional segments, thus allowing a third-party to spoof duplicate ACKs.

We propose to apply our algorithm at various frequencies. The frequency refers to

how many ACKs the traffic shaper sees before it generates a triple-ACK duplicate. Thus, a frequency of n means every n^{th} ACK is duplicated. This is done regardless of whether that ACK itself is a receiver-generated duplicate ACK or not. However, if the ACK has already been duplicated by our algorithm in the past, then we do not duplicate the current ACK and simply reset our ACK count to find the n^{th} ACK that we see from then on.

Our algorithm is shown below.

```
//Initialization

1. set ACK_count to -1
2. set oldACK to 0
        //remember last ACK triple-duplicated
3. set ACK_array[ACKCOUNT] to 0
        //tracks last ACKCOUNT bytes
      //flag byte ACK_array[ACK number] when ACK
      //duplicated
4. set source to 0
        //first SYN sets this
5. set dest to 0
        //first SYN sets this
6. set syn_received to 0
        //set when first SYN received
7. set fin_received to 0
        //set when FIN received
8. set ack_rolloverCount to 0
        //tracks how many times ACK_array has
        //wrapped around
9. set start_tripleAcking to 0
        //turns triple-ACK on or off
10. set ack_freq to 0
        //triple-duplicate every ack_freq'th ACK
```

```
//Operation

11. receive segment
12. send segment to segment.dest
        //traffic-shaper and router on same node,
        //and cannot delay segments, since third-party

13. if (segment.header & SYN_flag){
14.     if (syn_received == 0){
15.        source = segment.source
16.        dest = segment.dest
17.     }
18.     set syn_received to 1
19. }
20. if (segment.header & FIN_flag)
21.     set fin_received to 1

22. if (start_tripleAcking AND segment.src == dest AND
23.     segment.header & ACK_flag AND
24.     !(segment.header & SYN_flag) AND
25.     !(segment.header & FIN_flag) AND
26.     segment.length == 40) {
        //triple-ACK is on and it is a pure ACK
27.    if (fin_received)
28.        return
29.    if (segment.ACK_number == oldACK)
30.        return
31.    if (++ACK_count % ack_freq == 0)
32.        set ACK_count to 0

33.    if (segment.ACK_number/ACKCOUNT > ack_rollOverCount){
           //wrapping around
```

```
34.         reset ACK_array to 0
35.         while (segment.ACK_number/ACKCOUNT >
                    ack_rollOverCount)
36.              ack_rollOverCount++
            //need while loop since triple-ACK can
            //start in middle
37.     }

38.     if (ACK_array[segment.ACK_number-
             ack_rolloverCount*ACKCOUNT]
          ==1)
39.         return   //the ACK has been duplicated
40.     else
41.         ACK_array[segment.ACK_number-
               ack_rolloverCount*ACKCOUNT]=1
42.     old_ACK = segment.ACK_number
43.     for (count = 1 to 3) {
44.         newSegment = segment.copy()
45.         send newSegment to newSegment.dest
46.     }
47. }
```

3.1.1 ASSUMPTIONS

The Internet consists of many networks that mostly cooperate with each other. A network is comprised of links and routers and other equipment required to bring the Internet connection to the home. As a result, the factors that could complicate an analysis include the changing bandwidth along each individual link, the latency introduced by distance, other competing flows, the delay introduced by individual computational elements such as routers (processing and queuing delays), buffer overflows in computational elements that can cause packet loss, and delays and packet reordering due to route changes. All these factors may or may not introduce complications in our research study. However,

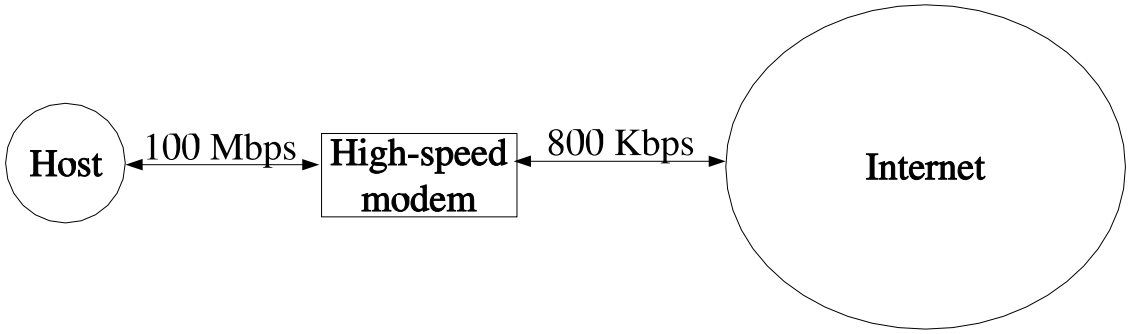


Figure 3.2: Setup of high-speed wireline connection to the Internet

for the purposes of our basic simulation experiments, we will assume no packet losses, no packet delays, and no packet reordering.

3.2 ZERO-WINDOW SIZE INVESTIGATION

Our second technique had two parts. Both parts relied on sending out a zero receiver-window-size duplicate ACK for every ACK seen. The first part of the technique was designed to manipulate the receiver-window size as seen by the sender TCP in an attempt to disallow the sender from sending any more data. The second part of the technique was based on the realization that data sent out by TCP would in fact be queued at various levels before being sent out, namely, the TCP stack, the network card driver, and the network card itself. Our zero receiver-window-size ACKs could be used as a signal to retract data queued up in one of the buffers.

The first part of this technique did not work, since TCP reacts to each ACK as it receives it. If the ACK acknowledges a new data segment, the TCP sender sends two segments in slow-start mode or one segment in fast-recovery mode, provided that the sliding window has free space in it. As a result, by the time the duplicate ACK reaches the sender, the sender has already sent out the required amount of data. Furthermore, the network pipeline typically contains many data segments, subject to congestion and the sender's window size. The resulting ACKs from the receiver would advertise the correct receiver window size, which would open up the receiver window as seen by the sender.

The second part of this technique also turned out to be a futile approach. Given a

typical high-speed wireline Internet home-user, who shares files over P2P networks, he is likely to have a 100 Mbps network card attached to his computer. The network card, in turn, would be attached to a high-speed modem, as shown in Figure 3.2. At such a high bandwidth connection between the end-host and the high-speed modem, there would be no queue build-up in TCP's send buffer. As a result, there would be no data segments to retract.

3.2.1 BUG IN NS-2.27

Through investigating the zero-window size algorithm, we discovered a defect in ns-2.27 for handling duplicate ACKs. We discovered that once the congestion window reaches the window cap, the FullTCP agent of ns-2 does not reset the duplicate-ACK count on receiving cumulative ACKs. As a result, zero-window ACKs belonging to different ACK numbers were able to trigger congestion control.

4 SIMULATION SETUP

In this chapter, we describe how we set up our simulations. Specially, we describe the variables we used in our simulations, the file-download environment that we modeled our simulations on, the simulation models themselves, the simulator and parameter values we used, and finally some terms we use throughout our thesis.

4.1 VARIABLES

The variables in our simulations are:

- frequency of triple ACK duplication,
- the size of the memory buffer allocated to a connection by the ns-2 TCP module (*i.e.* the window cap in ns-2),
- the round trip time (RTT) between the two end-hosts, and
- the bottleneck bandwidth between the two hosts.

Intuitively, if the traffic shaper generates triple duplicate ACKs too frequently, the bandwidth reduction and goodput will be offset by excessive retransmission. Conversely, if this happens too infrequently, the average congestion window and threshold will not be adequately reduced, as after being reduced, the window and threshold will build up again. Again, on an intuitive basis, the RTT and bandwidth between the end-hosts would play a role in terms of the bandwidth·RTT product. The larger the bandwidth·RTT product, the larger the bandwidth savings of throttling the sender's sliding window size could be.

4.2 ENVIRONMENT

There are two types of file-transfers that we are interested in throttling. One type takes place between end-hosts and large-scale servers, while the other type takes place between the end-hosts themselves. In the first case, the download bandwidth of the end-host is the

bottleneck in the file-transfer, while in the second, it is the upload bandwidth of the end-host serving the file. A survey of the current Internet service offerings from Rogers.com for home-users shows that the download offering varies from 128 Kbps to 5 Mbps and the upload offering varies from 64 Kbps to 800 Kbps [29].

To gauge the RTT numbers in a client-server environment, we pinged the University of Toronto's web server at www.utoronto.ca from a computer at the University of Waterloo. Over 140 ping messages, the average RTT was found to be 28.6 ms with the minimum being 6.2 ms. In order to get an upper bound on RTT measurements in a client-server environment, we decided to measure the RTT to a site in India. We chose the Indian Institute of Technology in New Delhi, India, accessible on the Web at http://www.iitd.ernet.in/. Since the website does not respond to ping messages, we decided to load up the page in a browser and capture the packets using Ethereal. The time-difference between sending the SYN segment and receiving the SYN-ACK segment was deemed to be equivalent to the RTT time-period. This was discovered to be about 405 ms. A more accurate modeling of client-server RTTs is outside the scope of this thesis.

In order to gauge the RTT values between end-hosts in a P2P environment, we conducted a P2P search using a publicly available P2P file-sharing client. We then pinged the IP addresses returned as hits. Typical RTTs observed varied from 75 ms to 400 ms. A more accurate model would require statistics from ISP-traffic-monitoring companies such as Sandvine [30].

4.3 MODELS

We want to focus on the simple symmetric model shown in Figure 4.1 to simulate our triple-ACK algorithm when applied to ideal conditions. In the symmetric model, we modeled our router and traffic-shaper on one node, as any communication delay could be simulated using a timer. However, in order to show that this model is apt for analysis, we started by simulating a simplified model of the Internet, as shown in Figure 4.2. We based our model on Asymmetric Digital Subscriber Line (ADSL), a dominant wireline technology for Internet access at the home. ADSL provides higher download bandwidths than upload bandwidths. In order to use ADSL service, the customer requires a DSL transceiver, colloquially called a "DSL modem". The DSL service provider has a DSL

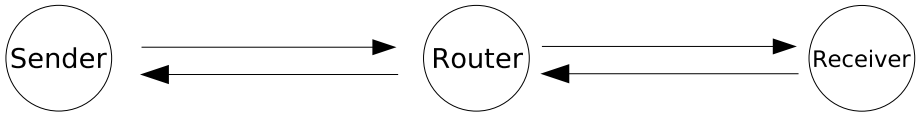


Figure 4.1: Simplified symmetric setup

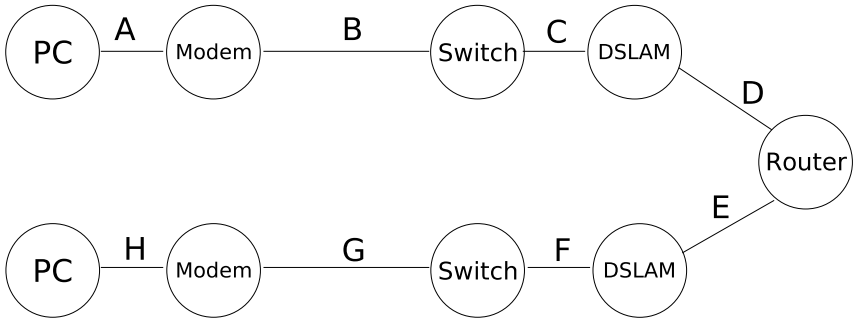


Figure 4.2: Simulating all components

Access Multiplexer (DSLAM) that multiplexes multiple customer connections on the transmitter side, and demultiplexes signals and forwards them to the appropriate DSL connections on the receiving side. The DSLAM is connected to the ISP router, which in turn connects to the routing infrastructure of the Internet.

Then we simplified this model to the upload model shown in Figure 4.3 to show that equivalent results are obtained. We reversed the link latencies to obtain the simplified download model shown in Figure 4.4 to show again that the results are the same. Finally, we ran our simulation experiments on the symmetric model of Figure 4.1, saw that it was equivalent to the upload and download models, and performed our analyses using that model. In these three models, we modeled our router and traffic-shaper on one node, as any communication delay could be simulated using a timer.

4.4 SIMULATION

We used the ns-2 simulator [34] to perform our simulation experiments. Both the source and sink nodes have FullTCP agents attached to them. The rest of the nodes are only

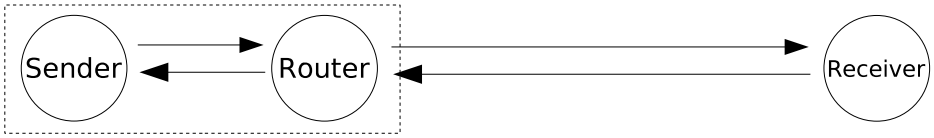


Figure 4.3: Simplified upload model

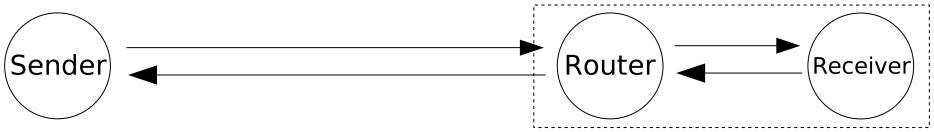


Figure 4.4: Simplified download model

required to implement IP and link-layer functionality to route messages from the sender to the receiver. In ns-2 the FullTCP agent differs from the regular TCP agent in that it has support for connection setup and tear-down, header flags, and bidirectional data transfer, and the sequence numbers are specified in bytes rather than packets [34]. The header flags were necessary so that we could recognize ACK segments. The source node also had an application generating a steady stream of data using the "File Transfer Protocol" (FTP) attached to the TCP agent. Rather than implementing the FTP protocol, the application simply sent out a continuous stream of data. This is in keeping with the original motivation of the project, which is to manage large TCP flows. Unless otherwise specified, default ns-2 parameters were used. This includes ns-2's window cap feature that models the maximum memory buffer per TCP session and has a default value of 20 segments. We used an MSS of 1360 bytes. This is in keeping with studies that have found that 94% of Internet clients use an MSS of between 1300 and 1460 bytes, with two-thirds using 1460 bytes [21].

Unless otherwise specified, we ran both the normal as well as triple-ACK cases for 130 seconds. Triple-ACKing was turned on only at 60 seconds so as to allow the $cwnd$ to grow to its maximum allowed size. We allowed 3 seconds for the new equilibrium under triple-ACK to be reached. We parsed the produced ns-2 packet trace between 63 and 124 seconds.

Some snippets of our raw data are available in Appendix G. This data is only meant

to give the reader an idea of what our experimental data looks like.

4.5 TERMS

This section defines some key terms we use throughout this thesis. We measure the impact of our algorithms in terms of these quantities.

Bandwidth is calculated as the rate of sending raw bytes in a given period of time, including the TCP/IP headers and payload.

Throughput is calculated as the rate of sending all the payload bytes in a given period of time. Given the constant ratio between headers and payload, the throughput reduction would be the same as the bandwidth reduction. Hence, throughput reduction is not studied in this thesis.

Goodput is the rate of transmitting the net TCP payload, *i.e.* without the duplicates.

Badput is the percentage of duplicate segments generated by TCP.

5 A SYMMETRIC MODEL IS ADEQUATE

In this chapter, we simulate various models, starting with a full model that includes all computational elements typical of a high speed deployment and ending with a highly simplified model with a symmetric setup. Our goal is to show that our symmetric setup is equivalent to the full model under ideal conditions and hence, is sufficient for our detailed experiments.

5.1 SIMULATING A FULL MODEL

In this section, we obtain the results of applying our triple-ACK-duplication algorithm to a highly simplified model of the Internet. Without including any of the complicating factors, we modeled the upload case based on the DSL architecture, as shown in Figure 5.1. We assumed that we have zero packet-loss, our route is static, the inter-ACK delay is constant, and packets do not get reordered. We assigned appropriate bandwidths and latencies to the individual links. We treated the bandwidth and latency inside the infrastructure as static quantities but varied the last-mile bandwidth between 1 and 5 Mbps. We ran our initial simulations for RTTs up to 400 ms, as per the expected RTTs in our scenarios, but found that the bandwidth reduction saturated by about 200 ms. As a result, we performed subsequent simulations for RTTs up to 250 ms. Our setup for the various links is described below. The Personal Computer (PC) node connected to link A sends the application data over TCP and the other PC node receives the application data over TCP.

Links A and H are the connections between the user's computer and his high-speed modem. This is most likely to be a 100 Megabits per second (Mbps) connection and of very low latency. Pinging our home and office routers, which are connected directly to our computers, gave us numbers on the order of 0.250 milliseconds (ms). Hence, we used a single-direction latency of 0.125 ms.

Links B and G are the last-mile links to the home. We varied the single-direction values for these links between 0.5 and 7.5 ms in increments of 1 ms. This latency would

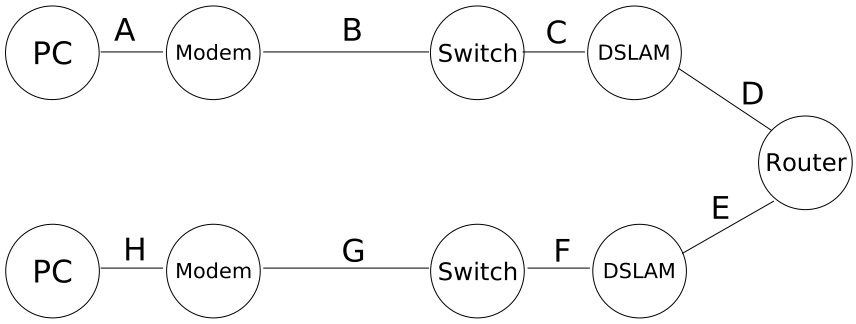


Figure 5.1: Simulating all components

be a combination of the distance factor as well as the time spent by various pieces of equipment in processing packets. We used a bandwidth range of 1 to 5 Mbps in increments of 1 Mbps, which effectively simulates a client-server scenario, where the file transfer is limited by the client's download bandwidth.

Links C and F are inside the switching office, where the data and voice traffic is demultiplexed. We expect this to be an optical connection with a bandwidth on the order of gigabits per second (Gbps). We assigned a value of 1 Gbps to these links. We expect the latency to be on a similar scale as that for links A and H. So we used a single-directional latency value of 1 ms, which would also take into account the computational load on the system.

Links D and E are links to the IP routing infrastructure. We only used 1 router to model the entire routing infrastructure, since there would be a bandwidth bottleneck in the end-to-end connection along the routers somewhere and because we assumed no route changes, no packet losses and no packet delays. Both sender and receiver hosts could be within the same router domain or they may be multiple router-hops away from each other. Hence, we varied the one-directional latency from 1 to 55 ms in increments of 1 ms, which would also include the computational time involved at every router along the route. A possible ISP scenario may consist of a POP size of 10,000 customers and average upload bandwidths of 500 Kbps for each customer. Since ISPs typically expect only a certain fraction of their

customers to be connected at once, they usually under-provision connectivity to the Internet. This process is called aggregation. Assuming a 20:1 level of aggregation, this would mean that an ISP in our scenario would have to provision a link to the Internet with 250 Mbps of bandwidth. Thus, we modeled links D and E with 250 Mbps.

For each latency value of links B and G, we varied the latency value of links D and E. Adding up the RTTs, the total end-to-end RTT varied from 10.5 to 254.5 ms, in increments of 4 ms. We conducted our experiments with Fast Recovery enabled. Our other controls are the maximum memory buffer allocated to a TCP session at each node, which specifies the maximum number of unacknowledged full data segments the sender can have at any given time, and the frequency with which we generate triple duplicate ACKs.

Since we have represented our router network using just one router, for simplicity, our triple-ACK model attaches the third-party traffic shaper to the sending side DSLAM. For the purpose of the simulation, the DSLAM and traffic shaper are modeled as a single node. We generated triple duplicate ACKs at frequencies of 1-8. A frequency of 1 means that every ACK was triple-duplicated, and that of 8 means that only one in 8 ACKs was triple-duplicated. We ran the normal case for 60 seconds and parsed the ns-2 packet trace between 30 and 50 seconds. We ran the triple-ACK case for 110 seconds, where we actually turned on triple-ACKing only at the 60-second point. We parsed the ns-2 packet trace between 70 and 100 seconds.

5.1.1 RESULTS

Our bandwidth reduction results at link B for frequencies 2–8 are summarized in Table 5.1. Due to the high starting point of our round-trip latencies (10.5 ms), we could not capture RTT values at which our bandwidth reduction exceeds zero. Hence, this information has not been shown in the table. We leave our analysis of the goodput to when we simulate the symmetric-latency model in Section 5.4.

Table 5.2 shows the observed badput as a percentage of the total transmission volume for each duplication frequency. It shows that careful optimization of the frequency of triple-ACK duplication is critical to the efficient operation of our algorithm. The badput

is a flat line for each frequency and its value is independent of bandwidth and latency. This is due to the fact that the badput is a function of ACK segments and that bandwidth and RTT affect both fresh data transmissions and repeated data transmissions alike.

A frequency of 1 does not produce any bandwidth reduction. This is explained further in Section 6.1. The bandwidth reduction results for frequencies 1–8 are shown graphically in Figures A.1 and A.2. The curves for goodput reduction for all frequencies are shown in Figures A.3 and A.4. We note further that the bandwidth consumed by ACKs going from the router to the sender also increases.

Quantity	Freq	1Mbps	2Mbps	3Mbps	4Mbps	5Mbps
Maximum bandwidth reduction	2	60%	60%	60%	60%	60%
	3	79%	79%	79%	79%	79%
	4	80%	80%	80%	80%	80%
	5	83.5%	83.5%	83.5%	83.5%	83.5%
	6	85%	85%	85%	85%	85%
	7	83%	83%	83%	83%	83%
	8	81%	81%	81%	81%	81%
Starting RTT (ms) of bandwidth reduction plateau	2	198.5	98.5	66.5	50.5	42.5
	3	198.5	98.5	66.5	50.5	42.5
	4	198.5	98.5	66.5	50.5	42.5
	5	202.5	102.5	66.5	50.5	42.5
	6	198.5	98.5	66.5	50.5	38.5
	7	198.5	98.5	66.5	50.5	42.5
	8	198.5	98.5	66.5	50.5	42.5
Starting bandwidth·RTT product (#segments) of bandwidth reduction plateau	2	17.72	17.59	17.81	18.04	18.97
	3	17.72	17.59	17.81	18.04	18.97
	4	17.72	17.59	17.81	18.04	18.97
	5	18.08	18.3	17.81	18.04	18.97
	6	17.72	17.59	17.81	18.04	17.19
	7	17.72	17.59	17.81	18.04	18.97
	8	17.72	17.59	17.81	18.04	18.97

Table 5.1: Bandwidth reduction for full model with window cap 20

Freq	Badput
1	50.0%
2	37.0%
3	30.7%
4	25.0%
5	20.0%
6	16.6%
7	14.2%
8	12.5%

Table 5.2: Badput for full model with window cap 20

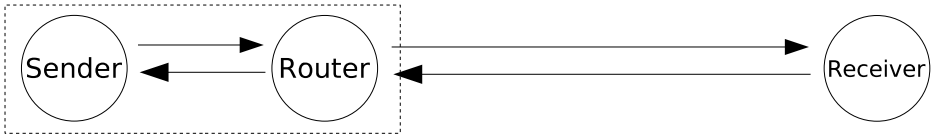


Figure 5.2: Simplified upload model

5.2 SIMULATING A SIMPLIFIED UPLOAD MODEL

In this section, we simulate a simpler version of the full model. Our goal is to simulate a file-upload without using all the intermediate elements and show that the results obtained with this model are equivalent to those obtained using a full model. Upload is defined as where the sending node is inside the traffic-shaping router's administrative domain. We model all computational and link latencies as latencies between one router and the sending and receiving nodes. Our model is shown in Figure 5.2.

We wanted results for up to 250 ms of RTT. Thus, we used latency values of 0.5 to 15.0 ms in each direction for the internal link and 1.0 to 125.0 ms in each direction for the external link, in increments of 1 ms. The link bandwidths were varied from 1 to 5 Mbps, in increments of 1 Mbps.

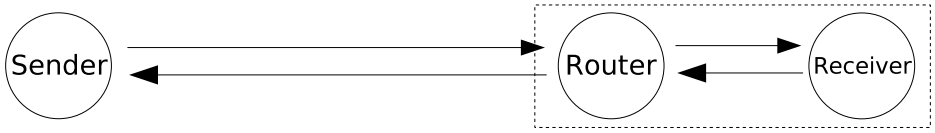


Figure 5.3: Simplified download model

5.2.1 RESULTS

The results obtained for bandwidth reduction are shown in Tables 5.3. The differences between these results and those obtained for the full model are also calculated. As can be seen, there is virtually no difference between the two models, with the small differences being due to the specific latency values used in the simulations. Furthermore, a frequency of 1 does not produce any bandwidth reduction even with a simplified upload model. The bandwidth reduction results for frequencies 1–8 are shown graphically in Figures B.1 and B.2. The goodput reduction curves are shown in Figures B.3 and B.4. A more detailed analysis of these numbers is left to Section 5.4. The badput results were identical to the full model case. Thus, we do not repeat that data in this section. Also as before, the badput was a constant line, independent of bandwidth and RTT.

5.3 SIMULATING A DOWNLOAD MODEL

We are interested in controlling the rates of not just uploads but also downloads. This section determines whether changing the direction of transfer affects the bandwidth reduction. Download is defined as where the receiving node is inside the traffic-shaping router's administrative domain. This is shown in Figure 5.3. Thus, we keep our setup similar to as it was in the upload case, but we reverse the link latencies. Again, we used default ns-2 settings, a window cap of 20 segments, and triple-ACK frequencies of 1-8. Just as with the upload case, we used latency values of 0.5 to 15.0 ms in each direction for the link internal to the router and 1.0 to 125.0 ms in each direction for the link external to the router, in increments of 1 ms in both cases. The link bandwidths were varied from 1 to 5 Mbps in increments of 1 Mbps.

Quantity	f	1Mb	%Δ	2Mb	%Δ	3Mb	%Δ	4Mb	%Δ	5Mb	%Δ
Starting RTT (ms) of >zero bandwidth reduction	2	35	-	19†	-	11	-	11†	-	7	-
	3	11	-	7	-	7†	-	3	-	3†	-
	4	23	-	11	-	11†	-	7	-	7†	-
	5	11	-	7	-	7†	-	3	-	3†	-
	6	11	-	7	-	7†	-	3	-	3†	-
	7	11	-	7	-	7†	-	3	-	3†	-
	8	11	-	7	-	7†	-	3	-	3†	-
Starting bw· RTT product (#segs) of >zero bandwidth reduction	2	3.13	-	3.39	-	2.95	-	3.93	-	3.13	-
	3	0.98	-	1.25	-	1.88	-	1.07	-	1.34	-
	4	2.05	-	1.96	-	2.95	-	2.5	-	3.13	-
	5	0.98	-	1.25	-	1.88	-	1.07	-	1.34	-
	6	0.98	-	1.25	-	1.88	-	1.07	-	1.34	-
	7	0.98	-	1.25	-	1.88	-	1.07	-	1.34	-
	8	0.98	-	1.25	-	1.88	-	1.07	-	1.34	-
Maximum bandwidth reduction	2	60%	0%	60%	0%	60%	0%	60%	0%	60%	0%
	3	79%	0%	79%	0%	79%	0%	79%	0%	79%	0%
	4	80%	0%	80%	0%	80%	0%	80%	0%	80%	0%
	5	83.5 %	0%	83.5 %	0%	83.5 %	0%	83.5 %	0%	83.5 %	0%
	6	85%	0%	85%	0%	85%	0%	85%	0%	85%	0%
	7	83%	0%	83%	0%	83%	0%	83%	0%	83%	0%
	8	81%	0%	81%	0%	81%	0%	81%	0%	81%	0%
Starting RTT (ms) of bandwidth reduction plateau	2	199	-0.2%	99	-0.5%	67	-0.75%	51	-1.0%	43†	-1.2%
	3	203	-2.3%	103	-4.6%	67	-0.75%	51	-1.0%	43†	-1.2%
	4	203	-2.3%	103	-4.6%	67	-0.75%	51	-1.0%	43†	-1.2%
	5	199	1.7%	103	-0.5%	67	-0.8%	51	-1.0%	43†	-1.2%
	6	195	1.8%	99	-0.5%	67	-0.8%	51	-1.0%	39	-1.3%
	7	199	-0.2%	99	-0.5%	67	-0.8%	51	-1.0%	43†	-1.2%
	8	199	-0.2%	99	-0.5%	67	-0.8%	51	-1.0%	43†	-1.2%
Starting bw· RTT product (#segs) of bandwidth reduction plateau	2	17.8	-0.3%	17.7	-0.5%	18.0	-0.8%	18.2	-0.9%	19.2	-1.2%
	3	18.1	-2.3%	18.4	-4.6%	18.0	-0.8%	18.2	-0.9%	19.2	-1.2%
	4	18.1	-2.3%	18.4	-4.6%	18.0	-0.8%	18.2	-0.9%	19.2	-1.2%
	5	17.8	1.7%	18.4	-0.5%	18.0	-0.8%	18.2	-0.9%	19.2	-1.2%
	6	17.4	1.8%	17.7	-0.5%	18.0	-0.8%	18.2	-0.9%	17.4	-1.3%
	7	17.8	-0.3%	17.7	-0.5%	18.0	-0.8%	18.2	-0.9%	19.2	-1.2%
	8	17.8	-0.3%	17.7	-0.5%	18.0	-0.8%	18.2	-0.9%	19.2	-1.2%

†Difference as compared to symmetric case due to coarser granularity

Table 5.3: Bandwidth reduction for simplified upload model with window cap 20

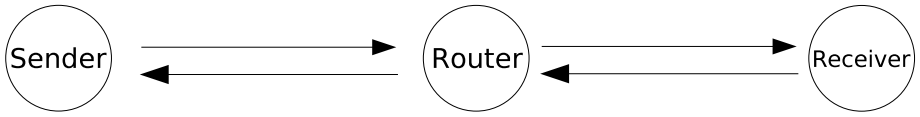


Figure 5.4: Simplified symmetric setup

5.3.1 RESULTS

The results obtained for bandwidth reduction are shown in Tables 5.4. As can be seen, there is absolutely no difference between the two models. The bandwidth reduction results for all frequencies are shown graphically in Figures C.1 and C.2. All the goodput reduction curves are shown in Figures C.3 and C.4. The badput results were identical to the upload model, hence they are not repeated in this section. As before, the badput curves were horizontal lines, independent of bandwidth and RTT. A more detailed analysis of these numbers is left to Section 5.4, where we simulate a symmetric-latency setup.

Due to the similarity in results obtained for simplified upload and download results, we conclude that injecting triple duplicate ACKs right after an ACK anywhere along the end-to-end path would give the same results, provided that all segments arrive at the destination in the same order. This implies that we can perform our experiments using a symmetric setup, where both links can have the same latency configured. This allowed us to perform our simulations faster, since both links have the same latency value.

5.4 SIMULATING A SYMMETRIC MODEL

Keeping a simplified view of the network, we set up symmetric connections between the two hosts and the router. This is shown in Figure 5.4. Again, we used default ns-2 settings and a window cap of 20 segments but this time, we used a wider range of duplication frequencies, *i.e.*, 1-10 in increments of 1, 15-25 in increments of 5, and 50-200 in increments of 50. Each of the links was assigned a bandwidth of 1 to 5 Mbps in increments of 1 Mbps, and the total RTT was varied from 1 to 250 ms in increments of 1 ms, *i.e.*, the latency assigned to each link was incremented by 0.25 ms at a time.

Quantity	Freq	1Mb	2Mb	3Mb	4Mb	5Mb
Starting RTT (ms) of >zero bandwidth reduction	2	35	19†	11	11†	7
	3	11	7	7†	3	3†
	4	23	11	11†	7	7†
	5	11	7	7†	3	3†
	6	11	7	7†	3	3†
	7	11	7	7†	3	3†
	8	11	7	7†	3	3†
Starting bandwidth·RTT product (#segments) of non-zero bandwidth reduction	2	3.13	3.39	2.95	3.93	3.13
	3	0.98	1.25	1.88	1.07	1.34
	4	2.05	1.96	2.95	2.5	3.13
	5	0.98	1.25	1.88	1.07	1.34
	6	0.98	1.25	1.88	1.07	1.34
	7	0.98	1.25	1.88	1.07	1.34
	8	0.98	1.25	1.88	1.07	1.34
Maximum bandwidth reduction	2	60%	60%	60%	60%	60%
	3	79%	79%	79%	79%	79%
	4	80%	80%	80%	80%	80%
	5	83.5%	83.5%	83.5%	83.5%	83.5%
	6	85%	85%	85%	85%	85%
	7	83%	83%	83%	83%	83%
	8	81%	81%	81%	81%	81%
Starting RTT (ms) of bandwidth reduction plateau	2	199	99	67	51	43†
	3	203	103	67	51	43†
	4	203	103	67	51	43†
	5	199	103	67	51	43†
	6	195	99	67	51	39
	7	199	99	67	51	43†
	8	199	99	67	51	43†
Starting bandwidth·RTT product (#segments) of bandwidth reduction plateau	2	17.77	17.68	17.95	18.21	19.20
	3	18.13	18.39	17.95	18.21	19.20
	4	18.13	18.39	17.95	18.21	19.20
	5	17.77	18.39	17.95	18.21	19.20
	6	17.41	17.68	17.95	18.21	17.41
	7	17.77	17.68	17.95	18.21	19.20
	8	17.77	17.68	17.95	18.21	19.20

†Difference as compared to symmetric-latency case due to coarser granularity

Table 5.4: Bandwidth reduction for simplified download model with window cap 20

5.4.1 BANDWIDTH REDUCTION

The results obtained for bandwidth reduction are summarized in Tables 5.5. Clearly, the bandwidth reduction results do show some differences, compared to the upload and download models. This can be attributed to the finer granularity that we used in incrementing the latency values. The bandwidth and goodput reduction curves are shown for frequencies 1–8 in Figures D.1 and D.2, and D.3 and D.4, respectively. It can be seen that the curves show the same behaviour as with the previous models. The badput results were identical to the previous models and were flat lines for each frequency, independent of bandwidth and latency. Hence, the data and graphs for badput are not shown.

5.4.1.1 BANDWIDTH·RTT PRODUCT

In Figure 5.5, we show the bandwidth reduction results for frequency 6 plotted as a function of RTT. In Figure 5.6, we show the bandwidth reduction results for frequency 6 plotted as a function of the bandwidth·RTT product (segments). As can be seen, all 5 bandwidths coincide, thus showing that our results are purely a function of the bandwidth·RTT product. The theory behind this is as follows. The total time it takes for the first full data segment to reach the destination is given by Equation (5.1).

$$\text{time to destination} = \frac{TCP/IP\ Segment\ size\ \frac{bytes}{segment} * 8\ \frac{bits}{byte}}{Bandwidth\ in\ Mbps} + \frac{RTT}{2} \qquad (5.1)$$

As determined by Equation (5.1), throughput is directly proportional to the link bandwidth and inversely proportional to the RTT. This means that a higher bandwidth and lower RTT connection will be able to send out more data segments in a fixed period of time, whether under normal transfer or with our triple-ACK algorithm turned on, as compared to a lower bandwidth and higher RTT connection. However, the bandwidth·RTT product limits how many data and ACK segments can exist in the end-to-end link at a time. Thus, with a cap on the growth of the congestion window, our throughput is limited by the smaller of the window cap and the bandwidth·RTT product.

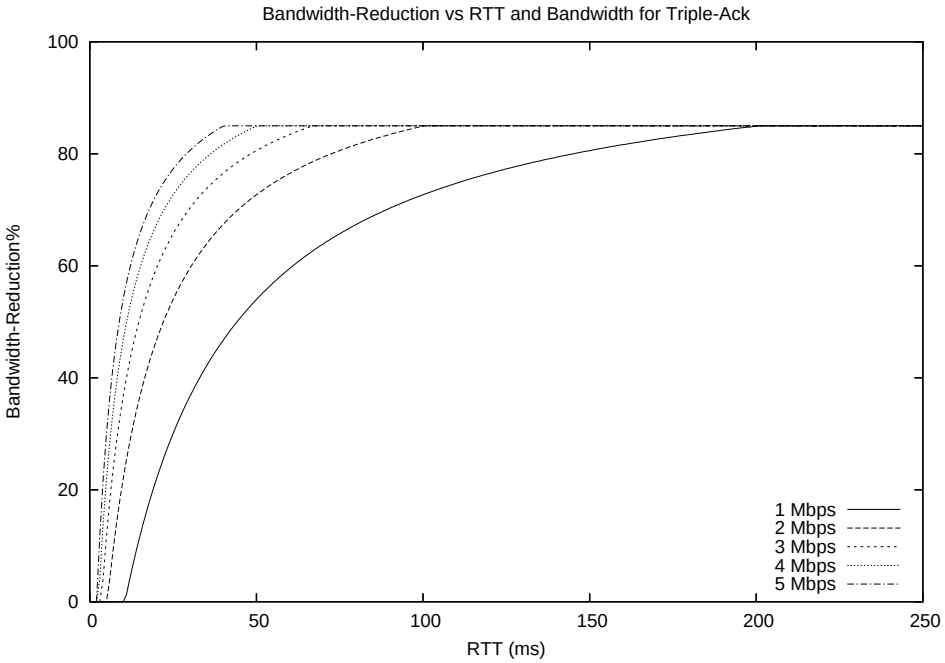


Figure 5.5: Bandwidth reduction vs. RTT with symmetric setup, freq 6, wincap 20

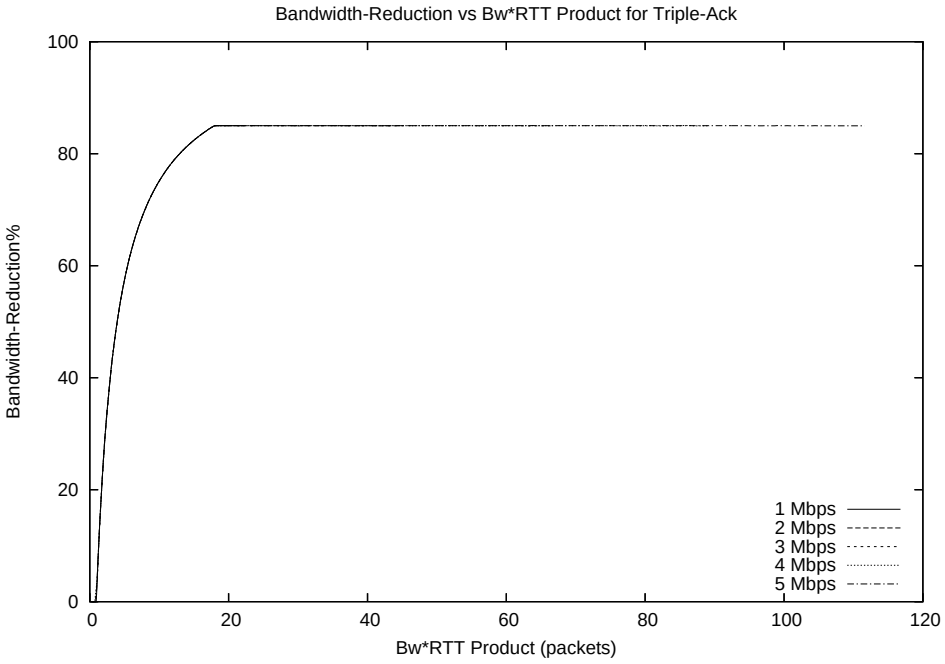


Figure 5.6: Bandwidth reduction vs. bandwidth·RTT with symmetric setup, frequency 6, wincap 20

Quantity	Freq	1Mb	2Mb	3Mb	4Mb	5Mb
Starting RTT (ms) of >zero bandwidth reduction	2	33	17	11	9	7
	3	11	6	4	3	3
	4	22	11	8	6	5
	5	11	6	4	3	3
	6	11	6	4	3	3
	7	11	6	4	3	3
	8	11	6	4	3	3
Starting bandwidth·RTT product (#segments) of non-zero bandwidth reduction	2	2.95	3.04	2.95	3.21	3.13
	3	0.98	1.07	1.07	1.07	1.34
	4	1.96	1.96	2.14	2.14	2.23
	5	0.98	1.07	1.07	1.07	1.34
	6	0.98	1.07	1.07	1.07	1.34
	7	0.98	1.07	1.07	1.07	1.34
	8	0.98	1.07	1.07	1.07	1.34
Maximum bandwidth reduction	2	60%	60%	60%	60%	60%
	3	79%	79%	79%	79%	79%
	4	80%	80%	80%	80%	80%
	5	83.5%	83.5%	83.5%	83.5%	83.5%
	6	85%	85%	85%	85%	85%
	7	83%	83%	83%	83%	83%
	8	81%	81%	81%	81%	81%
Starting RTT (ms) of bandwidth reduction plateau	2	197	99	66	50	40
	3	202	101	67	51	41
	4	201	101	67	51	41
	5	200	100	67	50	40
	6	201	101	67	51	41
	7	201	101	67	51	41
	8	201	101	67	51	41
Starting bandwidth·RTT product (#segments) of bandwidth reduction plateau	2	17.59	17.68	17.68	17.86	17.86
	3	18.04	18.04	17.95	18.21	18.3
	4	17.95	18.04	17.95	18.21	18.3
	5	17.86	17.86	17.95	17.86	17.86
	6	17.95	18.04	17.95	18.21	18.3
	7	17.95	18.04	17.95	18.21	18.3
	8	17.95	18.04	17.95	18.21	18.3

Table 5.5: Bandwidth reduction for symmetric model with window cap 20

5.4.2 GOODPUT REDUCTION

In this section, we describe the results we obtained for goodput reduction. Table 5.6 summarizes our minimum goodput reduction results and badput for frequencies 2–8. Comparing these two quantities, we can see that our minimum goodput reduction is the same as the badput. Table 5.7 summarizes the rest of our goodput results for frequencies 2–8. Comparing these results with Table 5.5, we can see that the bandwidth·RTT product at which goodput reduction exceeds the badput and that at which it reaches a saturation point are the same as the results obtained for the bandwidth reduction case. Within these numbers, we notice some differences across bandwidths for any given frequency. These differences are due to the coarseness of our RTT granularity.

Freq	1Mbps	2Mbps	3Mbps	4Mbps	5Mbps	Badput
2	37.0%	37.0%	37.0%	37.0%	37.0%	37.0%
3	30.7%	30.8%	30.8%	30.8%	30.8%	30.7%
4	25.0%	25.0%	25.0%	25.0%	25.0%	25.0%
5	20.0%	20.0%	20.0%	20.0%	20.0%	20.0%
6	16.7%	16.7%	16.7%	16.7%	16.7%	16.6%
7	14.3%	14.3%	14.3%	14.3%	14.3%	14.2%
8	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%

Table 5.6: Minimum goodput reduction for symmetric model with window cap 20

Quantity	Freq	1Mbps	2Mbps	3Mbps	4Mbps	5Mbps
	2	36	18	12	9	8
Starting RTT	3	11	6	4	3	3
(ms) of	4	22	11	9	6	5
goodput	5	11	6	4	3	3
reduction >	6	11	6	4	3	3
badput	7	11	6	4	3	3
	8	11	6	4	3	3
Starting	2	3.21	3.21	3.21	3.21	3.57
bandwidth·RTT	3	0.98	1.07	1.07	1.07	1.34
product	4	1.96	1.96	2.41	2.14	2.23
(#segments) of	5	0.98	1.07	1.07	1.07	1.34
goodput	6	0.98	1.07	1.07	1.07	1.34
reduction >	7	0.98	1.07	1.07	1.07	1.34
badput	8	0.98	1.07	1.07	1.07	1.34
	2	75.3%	75.3%	75.3%	75.3%	75.3%
	3	85.5%	85.5%	85.5%	85.5%	85.5%
Maximum	4	85.0%	85.0%	85.0%	85.0%	85.0%
goodput	5	86.9%	86.9%	86.9%	86.9%	86.9%
reduction	6	87.5%	87.5%	87.5%	87.5%	87.5%
	7	85.3%	85.3%	85.3%	85.3%	85.3%
	8	83.3%	83.3%	83.3%	83.3%	83.3%
	2	201	102	67	51	41
Starting RTT	3	202	101	67	52	41
(ms) of	4	202	101	67	51	41
goodput	5	201	101	67	51	41
reduction	6	201	101	67	51	41
plateau	7	200	100	67	50	40
	8	200	100	67	50	40
Starting	2	17.95	18.21	17.95	18.21	18.30
bandwidth·RTT	3	18.04	18.04	17.95	18.57	18.30
product	4	18.04	18.04	17.95	18.21	18.30
(#segments) of	5	17.95	18.04	17.95	18.21	18.30
goodput	6	17.95	18.04	17.95	18.21	18.30
reduction	7	17.86	17.86	17.95	17.86	17.86
plateau	8	17.86	17.86	17.95	17.86	17.86

Table 5.7: Goodput reduction for symmetric model with window cap 20

6 FREQUENCY VARIATION

Table 6.1 summarizes the maximum reduction in bandwidth achieved with frequencies 2 and up, when the window cap is set to 20 segments. Frequency 1 is not shown, as it results in a bandwidth-increase. The goodput reduction results for frequencies 2 and higher are summarized in Table 6.2. The badput results are summarized in Table 6.3. The badput is an indication of how efficiently we use the available network resources. We observe that starting at frequency 4, the badput becomes a function of the frequency, as given by Equation (6.1).

$$Badput = \frac{1}{frequency} \tag{6.1}$$

Figure 6.1 shows the bandwidth reduction, goodput reduction, and badput curves overlayed on top of each other. This helps us compare the three quantities directly. As we can see, the reduction in bandwidth and goodput are very close to each other. When trying to produce the maximal reduction in bandwidth, ISP policy would decide how to balance off the reduction in goodput and the amount of badput created in the process.

Freq	Maximum bandwidth reduction	Freq	Maximum bandwidth reduction
2	60%	10	83.3%
3	79%	15	76%
4	80%	20	75.3%
5	84%	25	69.5%
6	85%	50	59%
7	83%	100	40%
8	81%	150	29%
9	82%	200	21%

Table 6.1: Maximum bandwidth reduction vs. frequency, wincap 20

Freq	Maximum goodput reduction	Freq	Maximum goodput reduction
2	75.3%	10	85.0%
3	85.5%	15	77.4%
4	85.0%	20	76.5%
5	86.9%	25	70.8%
6	87.5%	50	60.1%
7	85.3%	100	40.7%
8	83.3%	150	30.1%
9	84.0%	200	21.9%

Table 6.2: Maximum goodput reduction vs. frequency, wincap 20

6.1 SEGMENT PATTERNS INDUCED BY FREQUENCY

Table 6.4 summarizes the pattern of total segments transmitted that emerges with each frequency when using a window cap of 20 segments. Table 6.5 summarizes the pattern of segments retransmitted that emerges with each frequency when using a window cap of 20 segments. The Round Trip is the number of the RTT in which $Total$ many segments are sent out, of which $Retx$ many are retransmitted segments. We determined these patterns by manually analyzing our experimental data obtained when using 5 Mbps as the link bandwidth and 200 ms as the RTT. In order to discern the pattern more easily, we would need a sufficiently large bandwidth·RTT product that would accommodate the largest number of transmissions in a single round trip itself. For example, we see up to 25 data segments being transmitted in one round trip. In order to accommodate all of them together, we needed a bandwidth·RTT product of at least 25 segments in size. Our link settings give us a bandwidth·RTT product of 89 segments. We have already seen in Section 5.4.1.1 that the bandwidth reduction is only dependent on the bandwidth·RTT product. We also know that ACKs act as a clock for TCP. Thus, we are confident that we would get the same segment pattern for a given frequency for all bandwidths and RTTs. If the bandwidth·RTT product is large enough to accommodate the largest number of segments in any round trip, as shown in Table 6.4, we would see the same distribution of segments over each round trip. If it is smaller, the same segment pattern would emerge

Freq	Badput	Freq	Badput
1	50.0%	10	10.0%
2	37.0%	15	6.7%
3	30.6%	20	5.0%
4	25.0%	25	4.0%
5	20.0%	50	2.0%
6	16.7%	100	1.0%
7	14.0%	150	0.7%
8	12.5%	200	0.5%
9	11.0%		

Table 6.3: Badput vs. frequency, wincap 20

but the distribution of segments over each round trip may vary.

Based on the observed patterns, we calculated the expected saturation in bandwidth reduction and badput. The saturation in bandwidth reduction is calculated using Equation (6.3) and the badput is calculated using Equation (6.4). The results are summarized in Tables 6.6 and 6.7. By comparing these values with the ones obtained using simulations, we can see that our pattern-based approach is valid.

$$Max\ bw\ reduction = \frac{(total\ segs)_{normal} - (total\ segs)_{tripleACK}}{(total\ segs)_{normal}} \quad (6.2)$$

$$= \frac{\#round\ trips * wincap - (total\ segs)_{tripleACK}}{\#round\ trips * wincap} \quad (6.3)$$

$$Badput = \frac{(total\ retx\ segments)_{tripleACK}}{(total\ segments)_{tripleACK}} \quad (6.4)$$

6.2 CROSSING ZERO BANDWIDTH REDUCTION

While it was fairly easy to develop a formulaic approach to predict the saturation in bandwidth reduction, a formulaic approach to determine when bandwidth reduction crosses the 0 mark is a lot more difficult. This variable depends on the exact dynamics of the

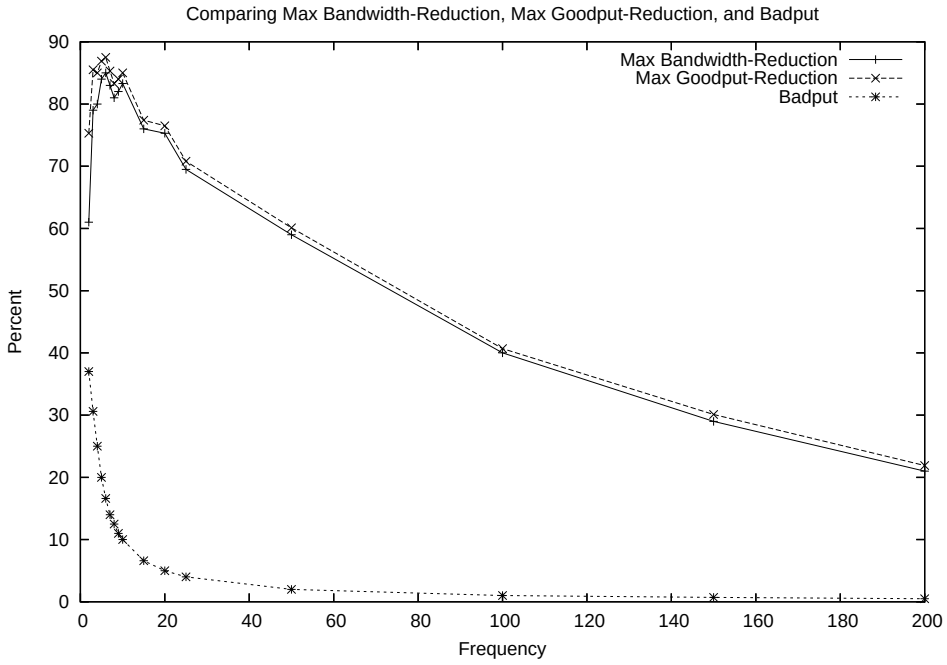


Figure 6.1: Comparing bandwidth & goodput reduction, and badput vs. freq, wincap 20

system. As we will see in the analysis of frequency 6 in Chapter 7, queues of segments can build up that can cause the pipeline to stay full or ACKs can be generated that do not cause data transmission. Below, we list some factors that need to be taken into account when determining the value of this variable:

- Queuing effects
- ACKs that result in new segments
- Symmetry across round trips in terms of number of segments
- Symmetry across round trips in terms of which ACKs cause segment transmission and which do not

Round Trip / Frequency	1	2	3	4	5	6	7	8	9	10
1	25	20	25	20	25	20	23	22	20	
2	6	10	6	8	10	6	10	7	8	10
3	3	6	4							
4	4									
5	3	4	3							
6	3	3								
7	4	3								
8	4	4								
9	4	4	3	3	4					
10	3	4	3							
15	5	5	5							
20	5	4	5	6						
25	7	7	5	6						
50	6	7	8	9	10	10				

Table 6.4: Pattern of total segments transmitted with a window cap of 20

6.3 FREQUENCY 1

The case of frequency 1 is unique, as it actually increases the bandwidth consumption. Using Figure D.1(a), we can see that our operating point of 5 Mbps with 200 ms RTT is actually in a relatively more stable portion of bandwidth-increase for frequency 1. Since the pattern only shows a maximum of 25 segments in transmission at any given time, we should see a stabilization in bandwidth-increase when the bandwidth·RTT reaches 25 segments. At 5 Mbps, this would happen at 56 ms of RTT. Based on our experiments, the inflection point occurs at 56 ms, after which the bandwidth-increase quickly approaches the expected value of 11%, which is the same value that we see in Table 6.6. Due to this pattern, the number of retransmitted segments continues to be at 50% of the total number of segments. For bandwidth·RTT products up to 18 segments, our triple-ACK algorithm is able to keep the pipeline full, just as the normal case would have. Thus, we see no bandwidth reduction until we hit a bandwidth·RTT product of 18 segments. Between 18 and 25, we see a sudden drop in bandwidth reduction, as the number of segments transmitted by our triple-ACK algorithm exceeds that in the normal case.

Round Trip / Frequency	1	2	3	4	5	6	7	8	9	10
1	18	7	13	12	8	17	3	20	2	
2	4	1	4	2	4	3	2	5	1	4
3	1	1	2							
4	1									
5	1	1	0							
6	1	0								
7	1	0								
8	1	4								
9	1	0	1	0	0					
10	0	0	1							
15	0	0	1							
20	1	0	0	0						
25	0	1	0	0						
50	0	0	0	0	0	1				

Table 6.5: Pattern of segments retransmitted with a window cap of 20

Frequency	Total segments with TripleACK	Total segments under normal	Expected saturation	Actual saturation	%Difference
1	200	180	-11.1%	-10.8%	-2.78%
2	81	200	59.5%	59.8%	0.50%
3	13	60	78.3%	78.5%	0.25%
4	4	20	80.0%	80.0%	0.00%
5	10	60	83.3%	83.4%	0.12%
6	6	40	85.0%	85.0%	0.00%
7	7	40	82.5%	82.6%	0.12%
8	8	40	80.0%	80.2%	0.25%
9	18	100	82.0%	82.0%	0.00%
10	10	60	83.3%	83.3%	0.00%
15	15	60	75.0%	75.2%	0.27%
20	20	80	75.0%	75.1%	0.13%
25	25	80	68.8%	68.9%	0.15%
50	50	120	58.3%	58.6%	0.51%

Table 6.6: Expected saturation in bandwidth reduction based on segment patterns

Frequency	Total segments	Total retx segs	Expected badput	Actual badput	%Difference
1	200	100	50.0%	49.9%	-0.20%
2	81	30	37.0%	37.0%	0.00%
3	13	4	30.8%	30.7%	-0.33%
4	4	1	25.0%	25.0%	0.00%
5	10	2	20.0%	20.0%	0.00%
6	6	1	16.7%	16.5%	-1.21%
7	7	1	14.3%	14.3%	0.00%
8	8	1	12.5%	12.5%	0.00%
9	18	2	11.1%	11.1%	0.00%
10	10	1	10.0%	10.0%	0.00%
15	15	1	6.7%	6.6%	-1.52%
20	20	1	5.0%	5.0%	0.00%
25	25	1	4.0%	4.0%	0.00%
50	52	1	2.0%	2.0%	0.00%

Table 6.7: Expected badput based on segment patterns

7 ANALYZING FREQUENCY 6

In this chapter, we analyze the triple-ACK frequency 6 case with a window cap of 20 segments. We chose frequency 6 since it produces the maximum bandwidth reduction.

7.1 PACKET-LEVEL ANALYSIS

This section explains the experimental results for the case of window cap 20 and triple-ACK frequency 6. For TCP, the ACKs act as a clock that provides cycles to TCP to increase or decrease its sliding window, send out data segments, and update byte sequence pointers accordingly. In an ideal environment where a flow can keep a link full at all times, the RTT and end-to-end bandwidth are constant, and there are no route changes and no router delays, all ACKs will have the same delay between them. The inter-ACK delay would be a function of the size of the data segments and the bottleneck bandwidth. Furthermore, the ACKs will always be a minimum of RTT behind the data segments in terms of sequence number. In other words, an ACK received now was sent in response to a data segment sent RTT amount of time ago. Due to this close relationship between ACKs and data segments sent, it makes more sense to analyze the dynamics of our system in terms of the packet flow, rather than specific timing information.

Table E.1 shows the full packet-flow analysis when the window cap is 20 and the triple-ACK frequency is 6. It breaks the packet flow down to understand the fresh and repeat transmissions that are taking place, the acknowledgement numbers of ACKs that are generated in response, and the growth of the sender's congestion window. We start tracing the segments only once triple ACK duplication is turned on, which we only do once the sender's congestion window has reached the maximum allowed size. ACKs and data segments are labeled as *An* and *Dn* respectively, where ACKs use *n* to indicate the next data segment expected by the receiver and data segments use *n* to indicate the n^{th} data segment. Packets labeled with $'$ are retransmitted segments with the same sequence or acknowledgement number, depending on whether they are data or ACK segments, respectively.

As soon as we turn on triple ACK duplication, the first ACK seen is triple duplicated,

as it corresponds to an ACK count of 0, as per our triple-ACK-duplication algorithm. Thus, we label this ACK as 0. This means that the data segment it acknowledges has to be labeled as D_{-1}. After the last step shown in Table E.1, we repeat from step 45 and enter a cycle. For convenience, we repeat the cyclic steps 45–49 in Table 7.1.

Table 7.1: Packet-flow cycle, frequency 6, wincap 20

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
45.	Normal ACK A31	2	2.5	Sent in response to D30 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` Send 1 new data segment D32, since only 1 data segment still unACKed ($< cwnd$).	+1 New
46.	Normal ACK A32	2	2.9	Sent in response to D31 `cwnd += 1/cwnd` Send 2 new data segments D33 & D34, since only 1 unACKed data segment ($< cwnd$).	+2 New
47.	Normal ACK A33	2	3.553	Sent in response to D32 `cwnd += 1/cwnd` Send 1 new data segment D35, since only 2 unACKed data segments ($< cwnd$).	+1 New
48.	Normal ACK A34	2	3.834	Sent in response to D33 `cwnd += 1/cwnd` Send 1 new data segment D36, since only 2 unACKed data segments ($< cwnd$).	+1 New

Continued on next page...

Table 7.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**

49.	Triple ACKs A34'	2	2	6 ACKS have passed by since the last ACK we triple duplicated. Plus, the ACK number of this ACK is not the same as the last ACK that we triple duplicated. Hence, we triple duplicate this one.
				Send 1 retx data segment D34'
				`cwnd=cwnd/2=1`
				`thresh=cwnd=1`
				But $ssthresh$ must be ≥ 2
				Thus, `thresh=2`
					+1 Old

			5	`cwnd = thresh+3`, due to artificial $cwnd$ inflation	

The same segment flow can be represented pictorially, as shown in Figures E.1, E.2, E.3, and E.4. In these figures, we have combined some of the steps in the previous analysis for brevity. At the end of the analysis, it can be seen that step 29 is the same as step 24 and that the system goes into a cycle of repetition there on. When the tabular and pictorial analyses are closely observed, the following steps can be discerned:

1. A normal ACK results in a data segment being transmitted. Call these A0 and D1.
2. A normal ACK results in two data segments being transmitted. Call these A1, D2 and D3.
3. A repeat ACK results in no data segments being transmitted. Call this ACK A2′.
4. A normal ACK results in a data segment being transmitted. Call this A2 and D4.
5. A normal ACK results in a data segment being transmitted. Call this A3 and D5.

6. Three duplicates of A3 result in an old data segment being transmitted. Call the triple-ACK sequence A3′ and the data segment D3′.

7. A normal ACK results in no data segments being transmitted. Call this ACK A4.

It can be seen that under the new equilibrium, there are only 3 data segments in the data stream at any given time. There are, in fact, two types of three-data-segment sets. These are listed below:

1. The first set of 3 data segments is sent in steps 45 and 46 of Table 7.1, namely, D0, D1 and D2.

2. The second set of three segments in the pipeline comes from the data segments transmitted in steps 47–49 of Table 7.1, namely D3, D4, and D2′.

We realized that the analysis for frequency 6 becomes simpler compared to some other frequencies, as there is a lot of inherent symmetry in the frequency 6 case. The factors contributing to the symmetry are:

1. The pattern of segments occurs over an even number of RTTs.

2. The number of data segments in each RTT is the same.

3. Packet queues are small and occur once per RTT.

4. The ACKs that do not result in transmissions are symmetric across RTTs.

7.2 BANDWIDTH REDUCTION

Based on frequency 6 curves in Figures D.1 and D.2, we notice that there are three regions of bandwidth reduction: zero reduction, rise, and saturation. Each of these cases is explained in the sections below, although we do not follow the exact same order.

7.2.1 ZERO REDUCTION

Zero reduction in bandwidth would happen, when the data transmission generated by our triple-ACK algorithm is able to keep the network pipeline full whenever the normal case would also do so. We try to determine when zero reduction takes place by analyzing when reduction would actually start appearing. Based on the pictorial analysis of frequency 6, we noticed that the data stream under equilibrium with triple-ACK is not continuous. We found three sources of transmission gaps:

1. There is a delay between A3 and its three duplicates, A3′, reaching the sender. Given that step 5 only causes one data segment, D5, to be transmitted, the transmission of D3′ is preceded by a gap. One sample run at 1 Mbps showed us that the gap between A3 and the three A3′s is 0.96 ms, which is significant relative to small RTTs but is negligible for large RTTs.

2. Step 7 does not produce any new data transmission. This is because the three duplicates of A3, namely, A3′, cause the congestion window to shrink. This disallows new segments from being sent in response to a subsequent ACK until $cwnd$ has grown sufficiently.

3. A2′ does not cause new data transmission. However, step 2 causes two new data segments, D2 and D3, to be transmitted. D3 can keep the link full for bw·RTT products ≤ 1 segment. Thus, the transmission gap caused by A2′ will only become apparent for bandwidth·RTT products > 1 segment.

7.2.2 SATURATION

Under normal conditions, as the sender's congestion window starts building up, it sends out 2 data segments for every ACK it receives, growing the $cwnd$ by 1 every time. In essence, this results in a doubling of the $cwnd$ every RTT on average. If there is a time-gap between the last data segment being placed on the link and an ACK coming back from the receiver, it would result in an incomplete usage of the available capacity of the link, as given by the bandwidth·RTT product. Given that our triple-ACK duplication algorithm results in a cyclic behaviour, we would achieve a maximum in bandwidth reduction as soon as a time gap starts appearing between groups of window cap data

segments. When the window cap is 20 segments, a time gap starts appearing when the bandwidth·RTT product is equal to 18 segments. This is because, of the remaining 2 segments in the window cap, 1 data segment is at the receiver and 1 ACK segment is at the sender. With a frequency of 6, we were able to maintain a constant number of 3 data segments in the network in any given round trip. Thus, by applying Equation (6.3), the bandwidth reduction is capped at 85% for bandwidth·RTT products greater than 18 segments. This is exactly what we see in Table 5.5.

7.2.3 RISE

When the bandwidth·RTT product of the end-to-end link is less than the window cap, a maximum of bandwidth·RTT segments can remain in transit. Of the remaining segments, 1 data segment is being processed by the receiver, 1 ACK segment is being processed by the sender, and the rest of the data segments are queued at the sender. Under our ideal conditions, we would not need any queued segments to have full link utilization. Thus, the equivalent window cap would be bandwidth·RTT + 2. Any bandwidth reduction we see before saturation can be calculated based on this equivalent window cap and can be expressed in equation form as shown in Equation (7.1).

$$Bandwidth\ reduction = \frac{((Bandwidth * RTT + 2) - 3)}{Bandwidth * RTT + 2} \tag{7.1}$$

A zoomed in view of the rise in bandwidth reduction is shown for the first 50 ms in Figure 7.1. This is the region where the bandwidth reduction is still rising and has not reached the plateau yet. It can be seen that when the bandwidth·RTT product is small, the reduction is noticeably less.

7.3 WINDOW-CAP VARIATION

Given the fact that our algorithm was able to create a repetitive pattern in data transmission, we would like to understand if the same pattern takes place regardless of the size of the window cap. The role that the window cap plays in our algorithm is at the beginning before the system reaches equilibrium, and in between as the $cwnd$ ramps up until the next occurrence of triple-ACK duplication. Initially, the window cap and frequency

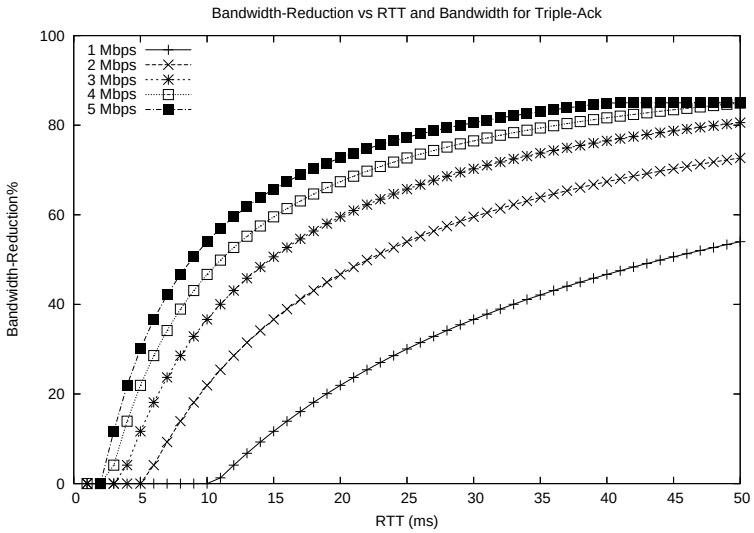


Figure 7.1: First 50 ms of bandwidth reduction, frequency 6, window cap 20

of triple-ACK duplication determine how many triple-ACK events actually take place within the first window cap many ACKs. This is given by Equation (7.2).

$$\#TripleACK\ invocations = \left\lfloor \frac{wincap}{f} \right\rfloor + 1 \tag{7.2}$$

The initial triple-ACK events would generate as many duplicate segments. With our frequency case of 6, if we were to reduce our window cap to 7 segments, our pictorial analysis of the ensuing transmissions until equilibrium is reached is shown in Figures F.1, F.2, and F.3. As can be quite clearly seen, step 22 is the same as step 18 and the pattern repeats here onward. Clearly, over 2 round trips, we transmit 6 segments, of which 1 is a retransmission.

However, if the window cap is smaller than our equilibrium value of 3, then we may actually incur a bandwidth increase if in addition to new segments, the sender also transmits duplicate segments. Our pictorial analysis for the case of window cap 2 is provided in step-by-step form in Figures F.4, F.5, and F.6. As can be seen from the analysis, Step 24 is the same as Step 14. The pattern repeats itself here onwards. Thus, despite the window cap, an equilibrium is reached. As can also be seen from the pattern,

the number of data segments in the link varies between 2 and 3, while in the normal case, there would only be 2 data segments in the system at a time. Specifically, the number of data segments per round trip in a pattern is 3, 2, 3, 2, and 2 respectively. In the first and third round trips, we see 1 retransmission. Thus, over 12 segments, 2 segments are retransmissions, giving us the same badput as before, *i.e.*, 16.7%. Under normal conditions, we would only see 10 data segments over 10 round trips. Thus, the bandwidth increase is 20%.

For window cap 7, the bandwidth reduction is given by Equation (7.3). We would like to see if Equation (7.3) is applicable to other window cap values.

$$Bandwidth\ reduction = \frac{window\ cap - 3}{window\ cap} \tag{7.3}$$

The TCP header provides a 16-bit field for the window size [32]. Hence, the maximum size of the window can be 65,535 bytes. With an MSS of 1360 bytes, this translates to 48 segments, after applying the floor function. Thus, we used window caps of 1-10 in increments of 1 and 15-50 in increments of 5. The window-scaling option provides an 8-bit scaling factor to indicate the number of bits by which TCP should shift the advertised value [15]. This effectively multiplies the advertised window by $2^{scalingfactor}$. When no bit is set, this translates to a scale factor of $2^0 = 1$. The highest shift value allowed is 14. This gives us a maximum window size of 1,073,725,440 bytes. Translating this to segments, we get a maximum window size of 789,504 segments. Studies have found that the mean window advertised by clients is almost 44 KB [21]. With an MSS of 1360 bytes, this translates to about 33 segments. Furthermore, only 26.6% of clients advertise support for TCP's window-scaling option [21]. However, over 97% of the clients that advertise support for the window-scaling option, in fact, use a value of 0, which translates to no scaling [21]. Thus, although far short of the theoretical maximum, we extended our experiments to include window caps of 55, 60 and 65 only.

Table 7.2 summarizes the saturation in bandwidth reduction for the different window caps, as given by theory and as determined by simulations. The badput remains constant at 16.6% for all cases greater than 1. The close match between theory and simulation strongly suggests that the same equilibrium pattern in segment flow is achieved for different window caps once we turn on our triple-ACK algorithm. For a window cap of 1, neither bandwidth reduction nor badput were observed.

Window-cap (segments)	Theoretical saturation	Experimental saturation	%Difference
1	-	0.0%	-
2	-%	-20.0%	0.0%
3	0.0%	0.0%	0.0%
4	25.0%	25.0%	0.0%
5	40.0%	40.0%	0.0%
6	50.0%	50.0%	0.0%
7	57.1%	57.1%	0.0%
8	62.5%	62.5%	0.0%
9	66.7%	66.7%	0.0%
10	70.0%	70.0%	0.0%
15	80.0%	80.0%	0.0%
20	85.0%	85.0%	0.0%
25	88.0%	88.0%	0.0%
30	90.0%	90.0%	0.0%
35	91.4%	91.4%	0.0%
40	92.5%	92.5%	0.0%
45	93.3%	93.3%	0.0%
50	94.0%	94.0%	0.0%
55	94.5%	94.5%	0.0%
60	95.0%	95.0%	0.0%
65	95.4%	95.4%	0.0%

Table 7.2: Saturation in bandwidth reduction vs. window cap, frequency 6

The bandwidth reduction is plotted vs. window cap in Figure 7.2. Only window caps > 2 segments are shown. It can be clearly seen that as we increase the window cap, the maximum bandwidth reduction keeps increasing. At our mean window size of 33 segments, the bandwidth reduction is 91%. Graphs for bandwidth and goodput reduction per window cap are shown in Figures H.1–H.10.

7.4 FORMULA FOR BANDWIDTH REDUCTION

As we have seen throughout our analysis so far, while triple ACK duplication with a frequency of 6 limits the maximum number of data segments in the network to 3, the re-

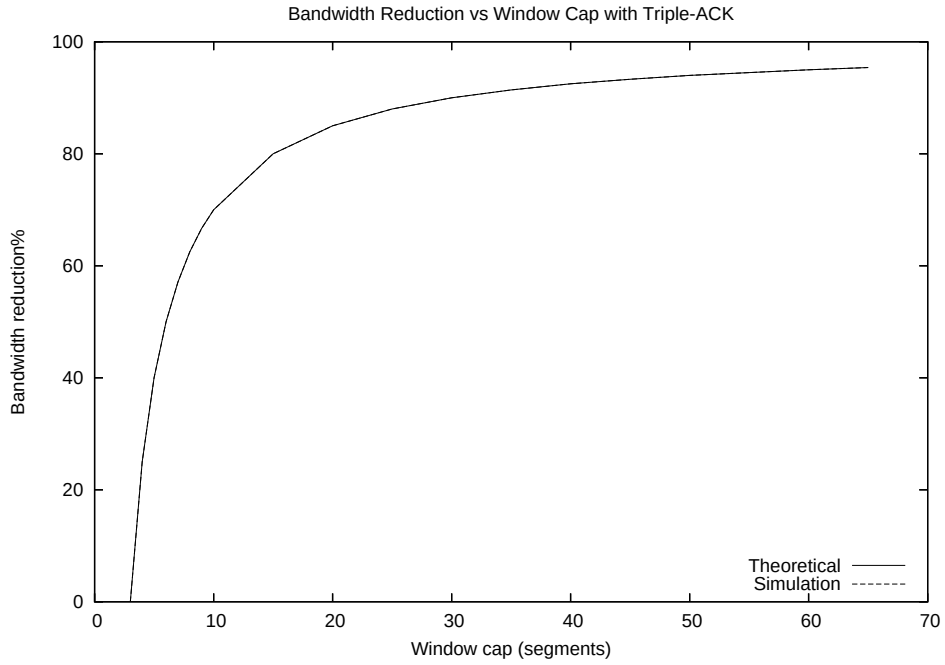


Figure 7.2: Maximum bandwidth reduction vs. window cap with frequency 6

peating pattern actually extends over 2 RTTs. We have also seen that we need a minimum bandwidth·RTT product of approximately 1 data segment before we start seeing bandwidth reduction. In our analysis of the frequency 6 case, we were able to reduce the total number of data segments in the link to 3. Thus, as the number of segments in the link under normal conditions is increased, we would expect to see a rise in bandwidth reduction until we hit the limit of the window cap. Conversely, if we hit the bandwidth·RTT limit of the link before the window cap limit, we would see a corresponding limit to how much we can reduce the bandwidth. Furthermore, the bandwidth reduction will be capped when the bandwidth·RTT product reaches a value 2 data segments less than the window cap. We also need a minimum window cap of 3 to see bandwidth reduction. With a window cap less than 3, the number of fresh data transmissions is constrained, and the added retransmissions would cause a bandwidth increase, as compared to the normal case. Thus, the formula we get for predicting bandwidth reduction with frequency 6 is as given in Equation (7.4).

For a window cap ≥ 3:

$$Bandwidth\ reduction = \begin{cases} 0 & 0 < bw * RTT \leq 1 \\ \frac{2*(bandwidth*RTT+2)-6}{2*(bandwidth*RTT+2)} & 1 < bw * RTT \leq (wincap-2) \\ \frac{2*wincap-6}{2*wincap} & bw * RTT > (wincap-2) \end{cases} \tag{7.4}$$

Clearly, this can be simplified to Equation (7.5).

$$Bandwidth\ reduction = \begin{cases} 0 & 0 < bw * RTT \leq 1 \\ \frac{bandwidth*RTT-1}{bandwidth*RTT+2} & 1 < bw * RTT \leq (wincap-2) \\ \frac{wincap-3}{wincap} & bw * RTT > (wincap-2) \end{cases} \tag{7.5}$$

7.4.1 COMPARING SIMULATION WITH THEORY

The theoretical and experimental bandwidth reduction values for frequency 6 and window cap 20 are compared in Figure 7.3. The theoretical values were derived using Equation (7.5). We ran our simulations for 1 to 5 Mbps in increments of 1 Mbps, with corresponding RTT values that would give us the bandwidth·RTT product desired. We obtained the same reduction in bandwidth for all bandwidths for a given bandwidth·RTT product, thus we only show the curves for 5 Mbps. The deviation of theory from simulation starts at -8.75% at a bandwidth·RTT product of 1.5 segments and reduces to 0% by 18 segments. This higher percentage of error for lower bandwidth·RTT products was discussed in Section 7.2.1.

7.5 GOODPUT REDUCTION

This section begins with a theoretical analysis of the goodput reduction we achieved. Goodput is the rate of transmitting the TCP payload without the retransmissions. TCP/IP segments contain 40 bytes of TCP/IP headers and up to SMSS bytes of payload. Given that all the data segments are full segments, Equation (7.6) defines the goodput for the normal case. We note that we have zero retransmission in the normal case.

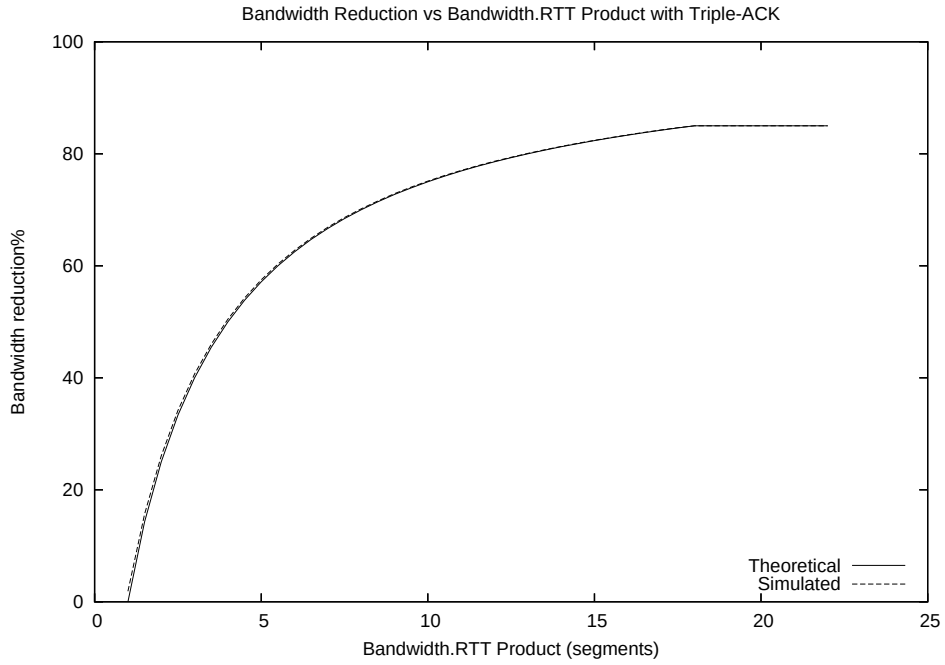


Figure 7.3: Theoretical vs. simulated bandwidth reduction, frequency 6, wincap 20

$$Goodput_{normal} = \frac{bytes_{total} - bytes_{retx}}{time} * \frac{SMSS}{SMSS + 40} \quad (7.6)$$

$$= \frac{bytes_{total}}{time} * \frac{SMSS}{SMSS + 40} \quad (7.7)$$

Equation (7.8) defines the goodput for the triple-ACK case.

$$Goodput_{tripleACK} = \frac{bytes'_{total} - bytes'_{retx}}{time} * \frac{SMSS}{SMSS + 40} \quad (7.8)$$

The goodput reduction is defined as the ratio of the difference between the goodput under normal and triple-ACK conditions, and the normal goodput. This is expressed in Equation (7.9).

$$Goodput_{reduction} = \frac{Goodput_{normal} - Goodput_{tripleACK}}{Goodput_{normal}} \quad (7.9)$$

$$= \frac{bytes_{total} - bytes'_{total} + bytes'_{retx}}{bytes_{total}} \quad (7.10)$$

Based on Equation (7.10), we can see that if $bytes_{total} = bytes'_{total}$, then our equation would reduce to Equation (7.11).

$$Goodput_{reduction} = \frac{bytes'_{retx}}{bytes'_{total}} \quad (7.11)$$

This would also be the percentage of badput in the transmitted data. This would happen only as long as the triple-ACK algorithm is able to keep the pipeline as full as the normal case, which according to our theory for frequency 6, is up to 1 data segment of bandwidth·RTT product. As soon as periods of zero-transmission start appearing, this simplification will no longer hold true. Realizing the fact that the condition $bytes_{total} \geq bytes'_{total}$ will always hold true, this also means that the minimum amount of goodput reduction we expect to see is the percentage of badput in the transmission stream.

Given that the segments are of the same size in the normal as well as the triple-ACK cases, we can divide Equation (7.10) by 1400 bytes/segment and obtain Equation (7.12).

$$Goodput_{reduction} = \frac{segments_{total} - segments'_{total} + segments'_{retx}}{segments_{total}} \quad (7.12)$$

As with Equation (7.11), we can divide Equation (7.12) by 1400 bytes/segment and obtain Equation (7.13).

$$Goodput_{reduction} = \frac{segments'_{retx}}{segments'_{total}} \quad (7.13)$$

We can calculate the goodput reduction in terms of segments. For this, we first need to determine how many segments traversed the pipeline in the normal case and how

many in the triple-ACK case. An approximate method of calculating these numbers is as follows. We first determine how many RTTs fit within the given time period. One RTT will only contain up to $wincap$ data segments at a time. However, a gap begins to appear in the pipeline under normal conditions when the bandwidth·RTT product hits $wincap - 2$ segments (since at that value of bandwidth·RTT product, we have 1 segment being processed by each node). Beyond $wincap - 2$ segments, we would start seeing a plateau in the goodput reduction. Thus, in a given period of time, the total number of data segment transmissions we expect to see is given by Equation (7.14).

$$Max\ \#segments \quad = \quad \frac{time}{RTT} * min(\frac{bw * RTT}{8 * 1400}, wincap - 2) \tag{7.14}$$

Our reduction in bandwidth would translate directly to a reduction in total number of segments transmitted. Thus, the reduction in segments can be calculated by multiplying our bandwidth reduction with the total number of segments. This is given by Equation (7.15).

$$Segment\ reduction \quad = \quad bw_{red} * \frac{time}{RTT} * min(\frac{bw * RTT}{8 * 1400}, wincap - 2) \tag{7.15}$$

For frequency 6, we derived the expression for bandwidth reduction as Equation (7.5). The formula only applies for $wincap > 3$ segments. For $bandwidth * RTT \leq 1$ segment, we theoretically expect no bandwidth reduction. For $bandwidth * RTT > 1$ segment, Equation (7.5) can be re-written as Equation (7.16).

$$bandwidth\ reduction = \frac{min(wincap, \frac{bw*RTT}{8*1400} + 2) - 3}{min(wincap, \frac{bw*RTT}{8*1400} + 2)} \tag{7.16}$$

Naturally, the ratio of actual bandwidth after reduction would then obtained by subtracting the bandwidth reduction ratio from 1. This is given by Equation (7.17).

$$bandwidth\ ratio = 1 - \frac{min(wincap, \frac{bw*RTT}{8*1400} + 2) - 3}{min(wincap, \frac{bw*RTT}{8*1400} + 2)} \quad (7.17)$$

$$= \frac{3}{min(wincap, \frac{bw*RTT}{8*1400} + 2)} \quad (7.18)$$

When we multiply the total number of segments under normal conditions with the bandwidth-ratio from Equation (7.18), we get the total number of segments with our triple-ACK algorithm. This is given by Equation (7.19).

$$total\ segments_{tripleACK} = total\ segments_{normal} * bandwidth\ ratio \quad (7.19)$$

When the total number of segments with our triple-ACK algorithm is multiplied by the badput ratio, which is 16.7% with frequency 6, we get the number of retransmitted segments. This is given by Equation (7.20).

$$retx\ segments_{tripleACK} = total\ segments_{tripleACK} * badput\ ratio \quad (7.20)$$

7.5.1 COMPARING SIMULATION WITH THEORY

We plot our simulation results for 1 Mbps along with those predicted by our theory in Figure 7.4. As can be seen, our theory is a good match. We note that we incur a larger error at lower latencies. At a bandwidth·RTT product of 1.07 segments, the error is 7.74%. At 2.05 segments, it is 2.11%, and at 3.04 segments, 1.04%. This is due to the inaccuracy in our model at lower RTTs, as discussed in Section 7.1. By 22 segments, the error reduces to 0.04%.

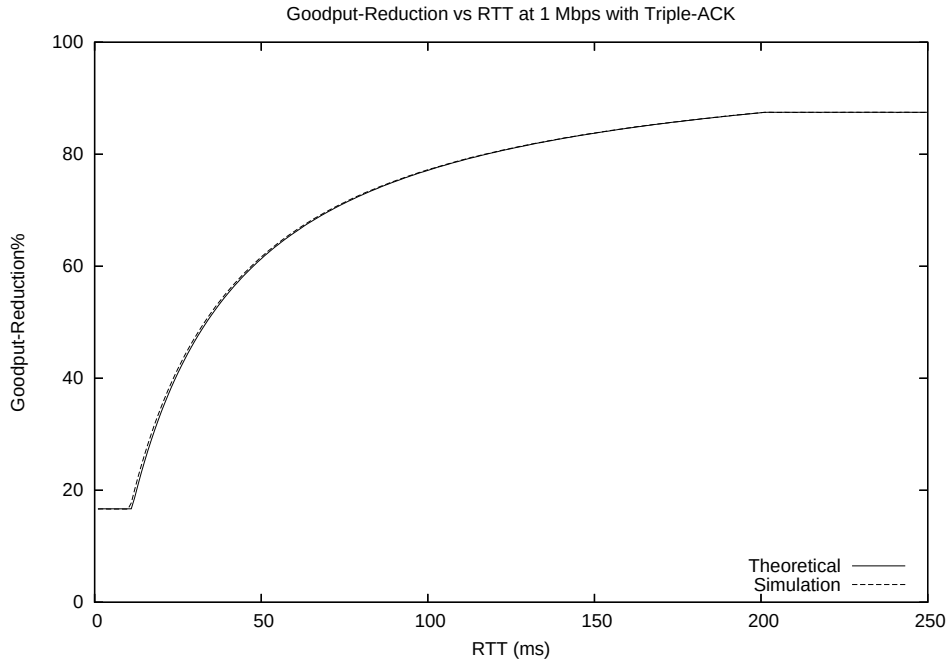


Figure 7.4: Comparing theoretical and simulated goodput reduction with 1 Mbps, frequency 6, window cap 20

8 LOWER BOTTLENECK BANDWIDTHS

In a P2P environment, the bottleneck bandwidth during a file-transfer will be the upload bandwidth of the sending node. Thus, keeping the same simplified symmetric experimental setup as before, we repeated our simulation experiments, varying the last-mile bandwidth from 200 Kbps to 800 Kbps in increments of 200 Kbps. At 200 Kbps, a bandwidth·RTT product of 18 segments translates to 1008 ms. Thus, we ran our experiments for RTTs up to 1100 ms in increments of 4 ms.

The bandwidth reduction results for maximum bandwidth reduction are summarized in Table 8.1. As can be seen, the results are the same as the higher bandwidths in terms of the saturation reached for bandwidth reduction and the bandwidth·RTT product at that point. Graphs for bandwidth reduction with frequencies 1-8 are shown in Figures I.1 and I.2. Graphs for goodput reduction are shown in I.3 and I.4.

We note that the RTT at which maximum bandwidth reduction is achieved is quite high for lower bandwidths. Typical RTT values for P2P file-sharing may be limited to only 200 ms. At that value, we expect to see the bandwidth reduction values shown in Table 8.2. A graph plotting the same is shown in Figure 8.1.

The bandwidth reduction at 200 ms varies across frequencies in the same way as the saturation values shown in Table 8.1. Thus, we may be able to use a ratio-based approach, when using different frequencies to control flows in real-time.

Quantity	Freq	0.2Mb	0.4Mb	0.6Mb	0.8Mb
Starting RTT (ms) of >zero bandwidth reduction	2	168	84	56	44
	3	56	28	20	16
	4	108	56	36	28
	5	56	28	20	16
	6	56	28	20	16
	7	56	28	20	16
	8	56	28	20	16
Starting bw· RTT product (#segs) of >zero bandwidth reduction	2	3.00	3.00	3.00	3.14
	3	1.00	1.00	1.07	1.14
	4	1.93	2.00	1.93	2.00
	5	1.00	1.00	1.07	1.14
	6	1.00	1.00	1.07	1.14
	7	1.00	1.00	1.07	1.14
	8	1.00	1.00	1.07	1.14
Maximum bandwidth reduction	2	60.0%	60.6%	60.7%	60.7%
	3	78.7%	79.2%	79.0%	79.1%
	4	79.6%	80.0%	80.1%	80.1%
	5	83.5%	83.5%	83.6%	83.6%
	6	84.9%	85.0%	85.0%	85.0%
	7	83.0%	82.9%	82.9%	82.9%
	8	80.9%	81.0%	81.0%	81.0%
Starting RTT (ms) of bandwidth reduction plateau	2	1008	516	336	252
	3	1004	508	336	252
	4	1008	508	336	252
	5	1004	500	336	252
	6	1004	504	336	252
	7	1004	508	336	252
	8	1004	504	336	252
Starting bandwidth·RTT product (#segments) of bandwidth reduction plateau	2	18.00	18.43	18.00	18.00
	3	17.93	18.14	18.00	18.00
	4	18.00	18.14	18.00	18.00
	5	17.93	17.86	18.00	18.00
	6	17.93	18.00	18.00	18.00
	7	17.93	18.14	18.00	18.00
	8	17.93	18.00	18.00	18.00

Table 8.1: Bandwidth reduction with lower bandwidths, window cap 20

Quantity	Freq	0.2Mb	0.4Mb	0.6Mb	0.8Mb
Bandwidth reduction	2	2.2%	21.7%	39.6%	52.3%
	3	31.2%	56.2%	67.8%	74.6%
	4	29.7%	56.8%	68.8%	75.6%
	5	44.1%	65.0%	74.6%	80.0%
	6	46.7%	67.4%	76.5%	81.6%
	7	42.9%	63.9%	73.6%	79.2%
	8	39.7%	60.8%	71.0%	76.9%

Table 8.2: Bandwidth reduction at 200 ms

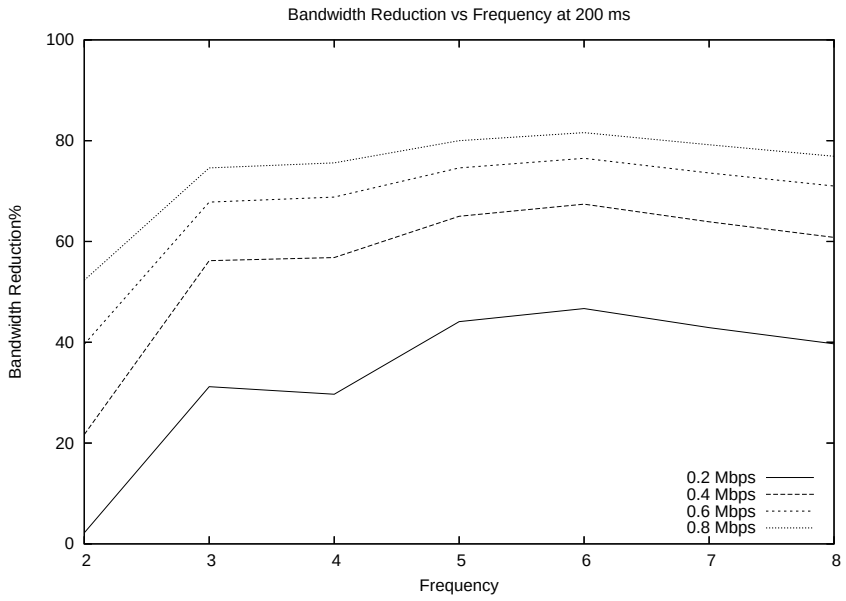


Figure 8.1: Bandwidth reduction at 200 ms for lower bandwidths

9 GENERALIZATION

Our analysis of the triple-ACK algorithm under ideal conditions can be generalized as follows. We start with a given window cap on how much the congestion window can grow and how much maximum buffer space the receiver can advertise. Let us call this cap w. We allow the congestion window to grow to the value of w, assuming the receiver window is not the bottleneck. After this, we start injecting triple duplicate ACKs with a frequency of f. The result is the following. First the threshold, $ssthresh$, will be halved and set to $w/2$, $cwnd$ will be set to $ssthresh + 3$ (where the three comes from the three duplicate ACKs), and the corresponding data segment will be retransmitted. It will be ensured that if halving the $ssthresh$ results in $ssthresh$ being less than 2, it is first set to 2, and then added to 3 to give us the new value for $cwnd$. When the sender gets a full ACK, the $cwnd$ gets deflated to $ssthresh + 1/ssthresh$. Also, as can be seen from the analysis, once the triple-ACK algorithm is initiated, we will be able to invoke triple-ACK duplication on the first window cap many ACKs coming back to the sender as many times as given in Equation (9.1).

$$\#TripleACK\ invocations = \left\lfloor \frac{wincap}{f} \right\rfloor + 1 \tag{9.1}$$

The number of triple-ACK invocations means that the sender will send out as many duplicate data segments. There may also be a few excess ACKs that are excluded by the floor function in the expression above. These ACKs may also cause additional transmissions, depending on the current value of $cwnd$ and the number of currently outstanding unacknowledged data segments.

Due to reduction in $cwnd$, the window is now perceived as full, since there are more unacknowledged data segments than $cwnd$ allows. However, every ACK increases the value of $cwnd$ by $1/cwnd$ and it takes $cwnd + 1$ many new ACKs (not duplicate ACKs) before the $cwnd$ increases by 1. When this happens, another data segment can be sent out.

After a triple-duplicate ACK, however, if $f < cwnd + 1$, our triple-ACK duplication algorithm will duplicate another ACK three times before the sender becomes ready to

transmit fresh segments. At this point, the $cwnd$ and $ssthresh$ will get further halved. Conversely, if $f \geq cwnd+1$, the sender TCP will be able to send out at least $f-(cwnd+1)+1$, or $f - cwnd$, fresh data segments out before triple-ACK takes place and reduces the $ssthresh$ and $cwnd$ again.

Furthermore, since the retransmissions are redundant, they would result in the receiver sending out a duplicate of the last ACK it sent out. These ACKs may also end up being triple-duplicated by our third party traffic shaper, since our traffic shaper does not take into account uniqueness when counting ACKs passing by. If three redundant data segments happen to be transmitted in a row, they would generate three duplicate ACKs in a row, which in turn would trigger congestion control on the sender.

Assuming zero packet loss, if duplicate ACKs are injected into the ACK channel by the receiver immediately after three other duplicate ACKs were received by the sender (whether due to our algorithm or three genuine duplicate ACKs), it would result in the $cwnd$ to be artificially inflated and a new data segment to be sent out per additional duplicate ACK, according to the rules of TCP [3]. This artificially inflated $cwnd$ will be shrunk back to $ssthresh + 3$ as soon as a full ACK is received, where 3 is added to the $ssthresh$ to reflect the three data segments that left the network and elicited the three duplicate ACKs.

10 SIMULATING MULTIPLE FLOWS

In this chapter, we present the results of simulating two flows running over a bottleneck link. We wish to show that by throttling one flow, our algorithm is able to increase the bandwidth consumption of the other. Our model is shown in Figure 10.1. In this model, S1 and S2 are the two TCP senders, D1 and D2 are the two TCP sinks, and R1 and R2 are the two routers. Our traffic shaper is located at router R1. We provide enough queue buffer so that the queue itself does not drop segments. We throttled S1 and compared the impact our triple-ACK algorithm has on the S2 flow in terms of bandwidth reduction. The parameters given below were used.

1. Window cap of 20 segments
2. Triple-ACK frequency of 6
3. Link S1-R1: Bandwidth of 5 Mbps, one-way latency of 5 ms
4. Link S2-R1: Bandwidth of 5 Mbps, one-way latency of 5 ms
5. Link R1-R2: Bandwidth of 0.5 Mbps, one-way latency of 1–41 ms, in increments of 5 ms
6. Link R2-D1: Bandwidth of 5 Mbps, one-way latency of 5 ms
7. Link R2-D2: Bandwidth of 5 Mbps, one-way latency of 5 ms

We only assigned 0.5 Mbps to the R1-R2 link as we wanted to make the bw·RTT product of the link the bottleneck, not the window cap. Note that the end-to-end RTT varied from 22 to 102 ms. We ran each simulation for 130 seconds and turned on triple-ACK at 60 seconds. The behaviour without triple-ACK was determined by parsing the ns-2 trace between 5 and 60 seconds and that with triple-ACK turned on was obtained by parsing the ns-2 trace between 63 and 124 seconds.

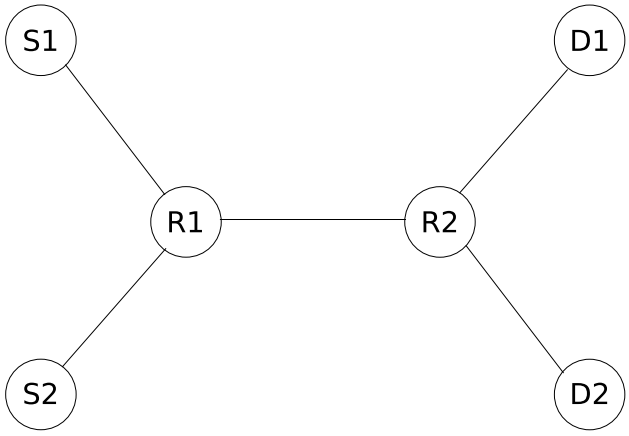


Figure 10.1: Multiple flow model

10.1 IDEAL CONDITIONS

We first obtained our results under ideal conditions without any segment loss or delay. Before triple-ACK was turned on, each flow consumed about 250 Kbps at all RTT values. After triple-ACK was turned on, the victim flow only consumed 65 Kbps of bandwidth, while the other flow was able to ramp its bandwidth consumption up to 435 Kbps at all RTT values. This translates to a bandwidth reduction as well as bandwidth gain of 74%.

10.2 WITH SEGMENT LOSS AND DELAY

Our next set of experiments tested to see if we would be able to throttle one flow while allowing another to gain in bandwidth even under non-ideal conditions. To simulate a non-ideal environment, we manually controlled the probability of segment loss and delay. We used a hypothetical mix of the two. We simulated a 1% probability of picking a segment to drop and a conservative 4% probability of picking a segment to delay. We simulated a segment delay of 1 ms, 3 ms, and 200 ms with equal probability. At a bottleneck bandwidth of 0.5 Mbps, the interACK delay would be 22.4 ms. A delay of 1 or 3 ms would not cause a timeout while a delay of 200 ms would be enough to cause a timeout in our current setup. We made sure not to drop or delay SYN and FIN segments. Note that we operated in non-ideal conditions both before and after triple-

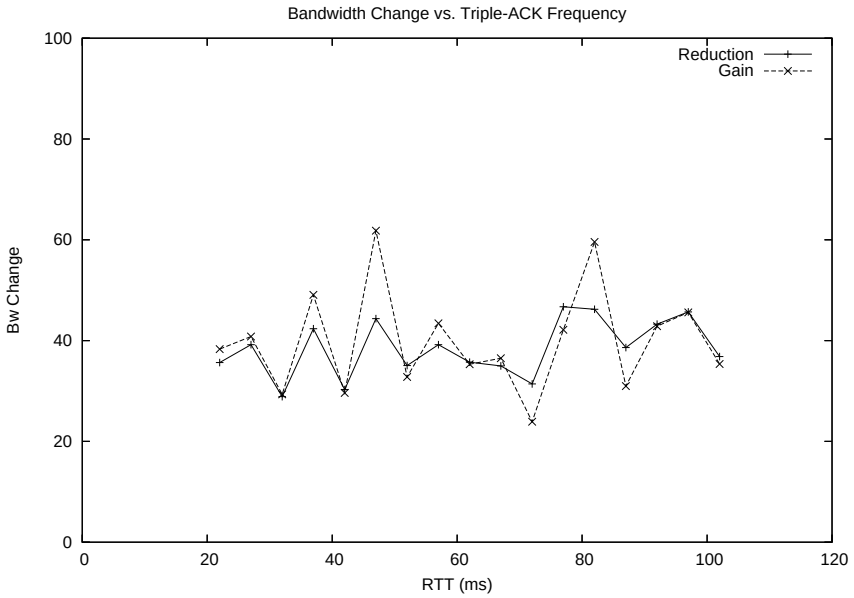


Figure 10.2: Bandwidth change with erroneous conditions

ACK was turned on. Further study with different probabilities of segment loss and delay is left as future work.

Figure 10.2 shows the average bandwidth reduction on the target flow and the average bandwidth gain on the other flow that we found over five simulation runs. As can be clearly seen, whenever we achieved a reduction in bandwidth on the target flow, there was a bandwidth increase on the other flow. An analysis of why the reduction and gain are not equal at all times is left as future work. Figures 10.3 and 10.4 show the average bandwidth reduction and average bandwidth gain with errorbars. The errorbars indicate the minimum and maximum change in bandwidth that we observed. An analysis of why the bandwidth change varies so much is also left as future work.

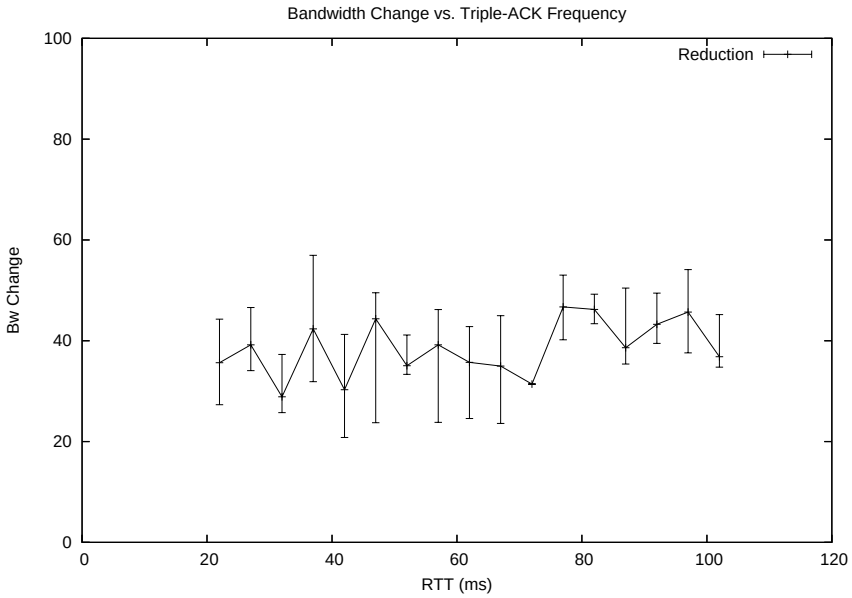


Figure 10.3: Bandwidth reduction with erroneous conditions

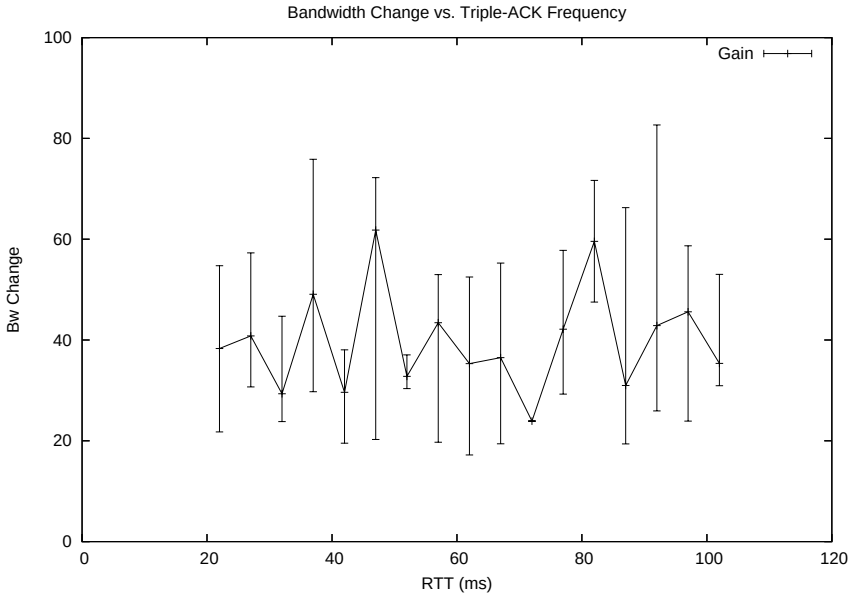


Figure 10.4: Bandwidth gain with erroneous conditions

11 CONCLUSION AND FUTURE WORK

Based on our simulation experiments and analyses, we have demonstrated that it is possible to control TCP flows with a third-party traffic shaper. We draw the following conclusions from our work:

1. The bandwidth and goodput reduction and the badput only depend on the end-to-end bandwidth·RTT product and not individual bandwidths and RTTs.

2. The bandwidth and goodput reduction and the badput vary with triple-ACK frequency. We have determined that each frequency results in a unique segment pattern. We find that the bandwidth and goodput reduction produced by our algorithm increase as we increment the frequency from 2 to 6 and then drop as we increment the frequency to 200. A triple-ACK frequency of 1 does not produce any bandwidth reduction. At sufficiently high bandwidth·RTT products, frequency 1 actually increases the link consumption. At a frequency of 2 and window cap 20, the bandwidth reduction is 60%, at frequency 6, it is 85%, and at frequency 200, it is 21%. At a more realistic window cap of 33 segments, our triple-ACK duplication algorithm can reduce bandwidth consumption by up to 91% under ideal conditions. The badput decreases steadily from frequency 1 to frequency 200. Starting at 50% at frequency 1 and window cap 20, the badput reduces to 17% at frequency 6, and 0.5% at frequency 200.

3. There are three regions of bandwidth reduction: zero reduction, rise, and saturation. The region of zero bandwidth reduction depends on queuing effects, ACKs that result in new data segments and those that do not, and the number of data segments transmitted per round trip, as well as the symmetry in all these factors. This varies with each frequency, as each frequency produces different dynamics. The maximum bandwidth reduction achieved for each frequency can be explained in terms of the segment pattern. For a given frequency, the rise in bandwidth reduction depends on the link consumption, which depends on the window cap and bandwidth·RTT product under ideal conditions.

4. The bandwidth and goodput reduction can be modeled mathematically for frequency 6 under ideal conditions. The variables involved are bandwidth, RTT, and window cap. The badput produced can be understood on the basis of segment patterns. In order to derive formulas for other frequencies, we would have to analyze the zero-reduction region.

5. Under ideal conditions, using a maximum window size of 20 segments and triple-ACK frequency of 6, we achieved a bandwidth reduction of 4%-85% for client-server downloads and 12%-85% for P2P downloads. This is based on the RTTs and bandwidths we observed between clients and servers, and between peers. At the same time, we produced a badput on the order of 16.6%.

6. Starting with a triple-ACK frequency of 4, the badput becomes an inverse function of the frequency.

7. Given two TCP flows running over a bottleneck link, our algorithm, operating at a triple-ACK frequency of 6, is able to allow one flow to consume more bandwidth by throttling the other. This works under ideal conditions as well as in the presence of some segment loss and delay.

11.1 FUTURE WORK

A number of items of future work remain. This future work can be classified into three categories: immediate follow-on, variations in the system, and other work.

11.1.1 IMMEDIATE FOLLOW-ONS

1. Model the delay introduced by our traffic shaper and the impact it has on the exact bandwidth·RTT product where bandwidth reduction begins.

2. Analyze the exact dynamics of triple-ACK frequencies other than 6, and see how a segment pattern is formed and maintained.

3. Understand the behaviour of our algorithm in the presence of segment loss and timeout. An analysis is required when we only have one flow in the system as well as when we have multiple flows in the system.

4. Implement a prototype of our triple-ACK duplication scheme to determine the degree to which our simulation results work in practice. This could be built on top of the netfilter system [22].

5. Once a prototype is shown to be working, test the scalability limits of third-party rate control by seeing how many flows it can tolerate running over the same bottleneck link. This involves determining the average processing time per ACK as well as the average time between ACKs.

6. Design and simulate a controller that has feedback-control-based bandwidth adjustment. The controller will measure the current bandwidth utilization and compare it against the target level. The controller could also measure the probability of segment delay and loss occurring in the network. The frequency of triple-ACK duplication will be adjusted accordingly. Once the controller is fine-tuned, it could be implemented using netfilter.

11.1.2 VARIATIONS

1. Modify our triple-ACK-duplication algorithm such that the third of the three duplicate ACKs is transmitted with a zero window-size. This would suppress any badput, since a zero-sized receiver window signals to the sender to stop transmitting anything. It is necessary to explore real-world TCP stacks to determine if such a modified ACK is recognized as a duplicate.

2. "Delayed ACKs" may affect the net bandwidth reduction we achieve by running our algorithm at any given frequency. "Delayed ACKs" are ACKs sent by the receiver in response to $k > 1$ data segments at a time [15]. Since our triple-ACK algorithm only counts actual ACKs, "delayed ACKs" may reduce the bandwidth reduction we achieve.

3. Investigate the impact of our algorithm on NewReno TCP and TCP with the SACK options. A superficial analysis with NewReno and SACK using 5 Mbps link bandwidth and 200 ms RTT revealed that NewReno would need an modification of our basic mechanism to produce bandwidth reduction in third-party mode while SACK would need a lower triple-ACK frequency to produce viable bandwidth reduction.

NewReno is known to result in a bandwidth-increase if the three duplicate ACKs are caused by packet reordering [12]. This is because NewReno's fast-recovery mechanism requires transmission starting from the first unacknowledged segment. With our triple-ACK algorithm, this would correspond to the data segment whose ACK immediately follows our three duplicate ACKs. As a result, we end up retransmitting data that was already on its way to being acknowledged, resulting in useless retransmissions. A frequency of 6 results in 60% increase in bandwidth with approximately 50% badput, while a frequency of 20 results in 45% increase in bandwidth also with approximately 50% badput. On the other hand, SACK produces 0.4%, 0.8%, and 69% reduction in bandwidth at triple-ACK frequencies of 4, 6, and 20. It is unclear at present how bandwidth reduction with SACK occurs.

11.1.3 OTHER WORK

1. In the context of non-switched Local Area Networks, it would be possible for a system administrator to monitor user's packets and inject triple duplicate ACKs as per our scheme to maintain a cap on bandwidth consumption. This mode of traffic-shaping may also be employed by 802.11 wireless routers to throttle specific flows to ensure fairness in bandwidth allocation among flows.

2. In the context of network protocol security, if a user can snoop on other users' packets, he may also be able to inject triple duplicate ACKs with spoofed IP addresses, causing a Denial-of-Service attack. Thus, it may be worthwhile investigating what protection mechanisms we may provide in TCP to protect against our own scheme.

   (a) If the TCP stack notices that the inter-ACK arrival time of the three duplicate ACKs is much smaller than that of regular ACKs, it may alert the user of a possible intrusion.

   (b) Rather than measuring inter-ACK arrival times, it may be possible to infer the same anomaly from RTT measurements. The timestamp option has been proposed to improve upon RTT measurement techniques [15]. With the timestamp option, both the sender and the receiver maintain independent virtual clocks and timestamp each transmitted segment with their current clock

value. Each side also echoes back the timestamp of the last segment it received. However, an ACK for an out-of-order data segment contains the timestamp from the most recent data segment *that advanced the window*. Given a dropped data segment, three duplicate ACKs would contain the same timestamp echo reply. Since our triple-ACK algorithm would replicate the timestamp information along with everything else in the target ACK, we expect that our triple-ACK scheme would escape undetected with the timestamp option.

(c) The sender TCP could be modified to only act on the three duplicate ACKs, if there were indeed as many unacknowledged data segments in its buffer. If there were already 3 or more outstanding data segments, a nonce-based approach could help detect spoofed duplicate ACKs. In this approach, the sender embeds a nonce in every data segment it transmits and requires the receiver to echo the same value back in its ACK [31].

(d) Three duplicate ACKs could arise if the network delayed one data segment and allowed three others to reach the receiver first. The sender would receive three duplicate ACKs followed by a full ACK. The full ACK would have an acknowledgement number that is at least 3*RMSS bigger (assuming full data segments) than that of the three duplicate ACKs and not just RMSS bigger, as is the case in our experiments. Thus, the sender TCP could perform this check on the acknowledgement numbers, realize that the three duplicate ACKs did not arise as a result of reordering, and undo any congestion window halving it performed. The idea of undoing congestion window-halving on detecting packet reordering has been previously proposed [11].

A FULL MODEL RESULTS

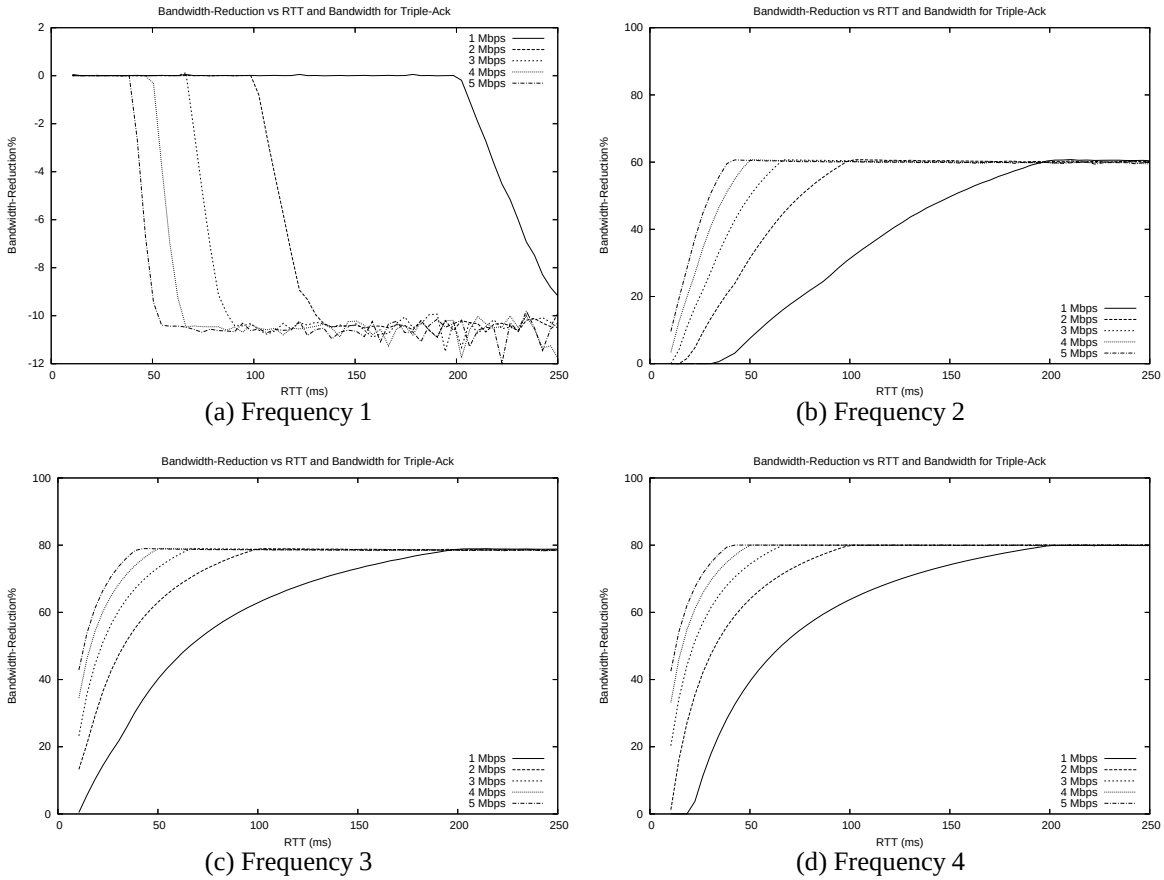
This section contains graphs for the simplified full-model case. Graphs are shown for bandwidth and goodput reduction for frequencies of 1-8. Experiments were run for the following ranges.

Links A and H 100 Mbps bandwidth; 0.125 ms latency per direction

Links B and G 1 to 5 Mbps bandwidth, in increments of 1 Mbps; 0.5 to 7.5 ms latency per direction, in increments of 1 ms

Links C and F 1 Gbps bandwidth; 1 ms latency per direction

Links D and E 250 Mbps bandwidth; 1 to 55 ms latency per direction, in increments of 1 ms



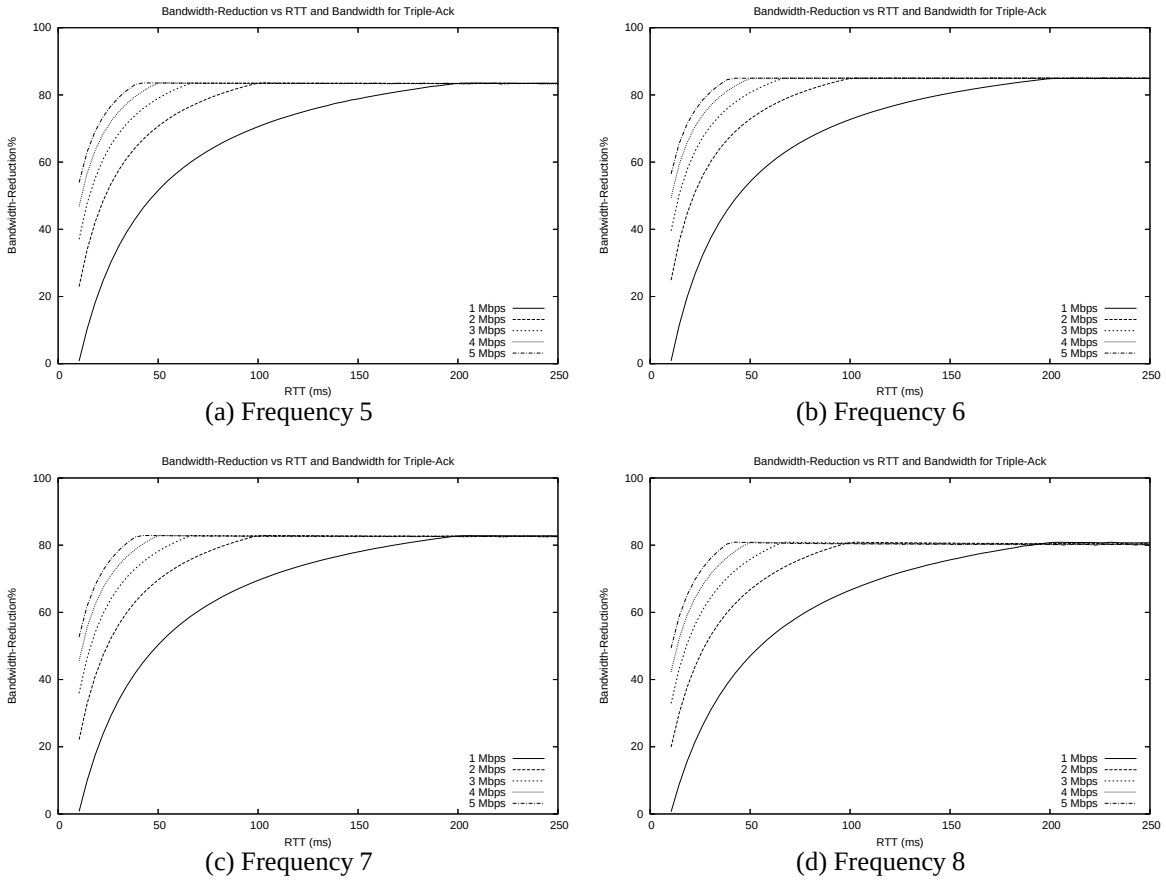
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure A.1: Bandwidth reduction for full model with freq 1-4 and wincap 20



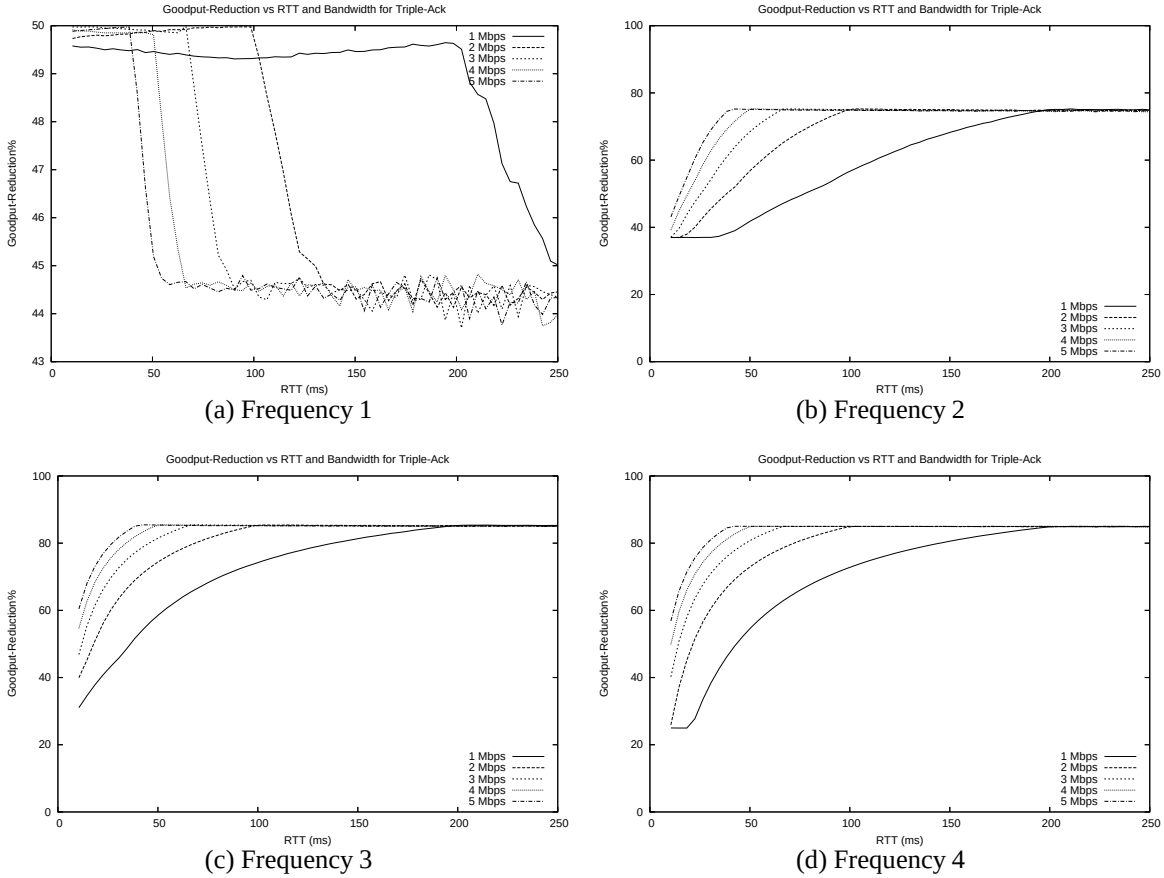
(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure A.2: Bandwidth reduction for full model with freq 5-8 and wincap 20



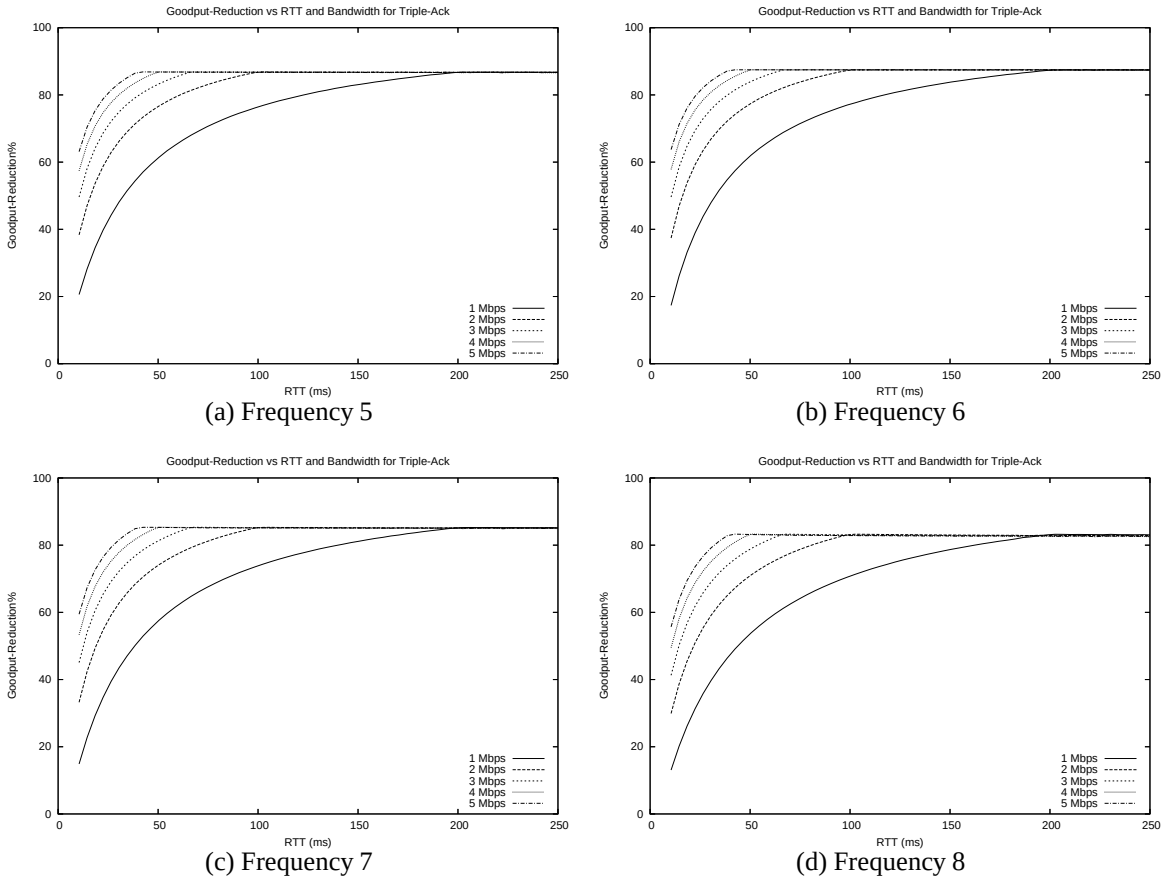
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure A.3: Goodput reduction for full model with freq 1-4 and wincap 20



(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure A.4: Goodput reduction for full model with freq 5-8 and wincap 20

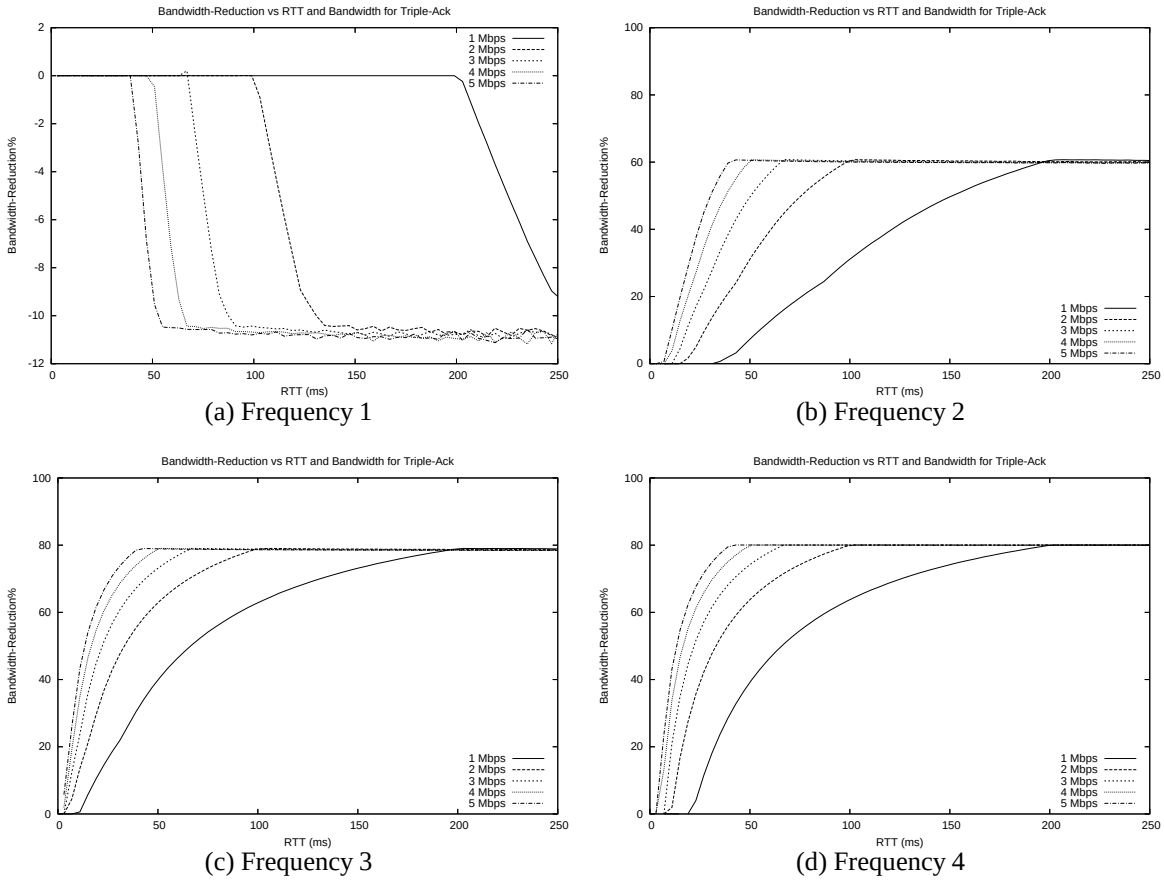
B SIMPLIFIED UPLOAD RESULTS

This section contains graphs for the simplified upload case. Graphs are shown for frequencies of 1-8 and for bandwidth and goodput reduction. Experiments were run for the following ranges.

Bandwidth 1.0 to 5.0 Mbps, in increments of 1 Mbps

Sender-side latency 0.5 to 15.0 ms per direction, in increments of 1 ms

Receiver-side latency 1.0 to 125.0 ms per direction, in increments of 1 ms



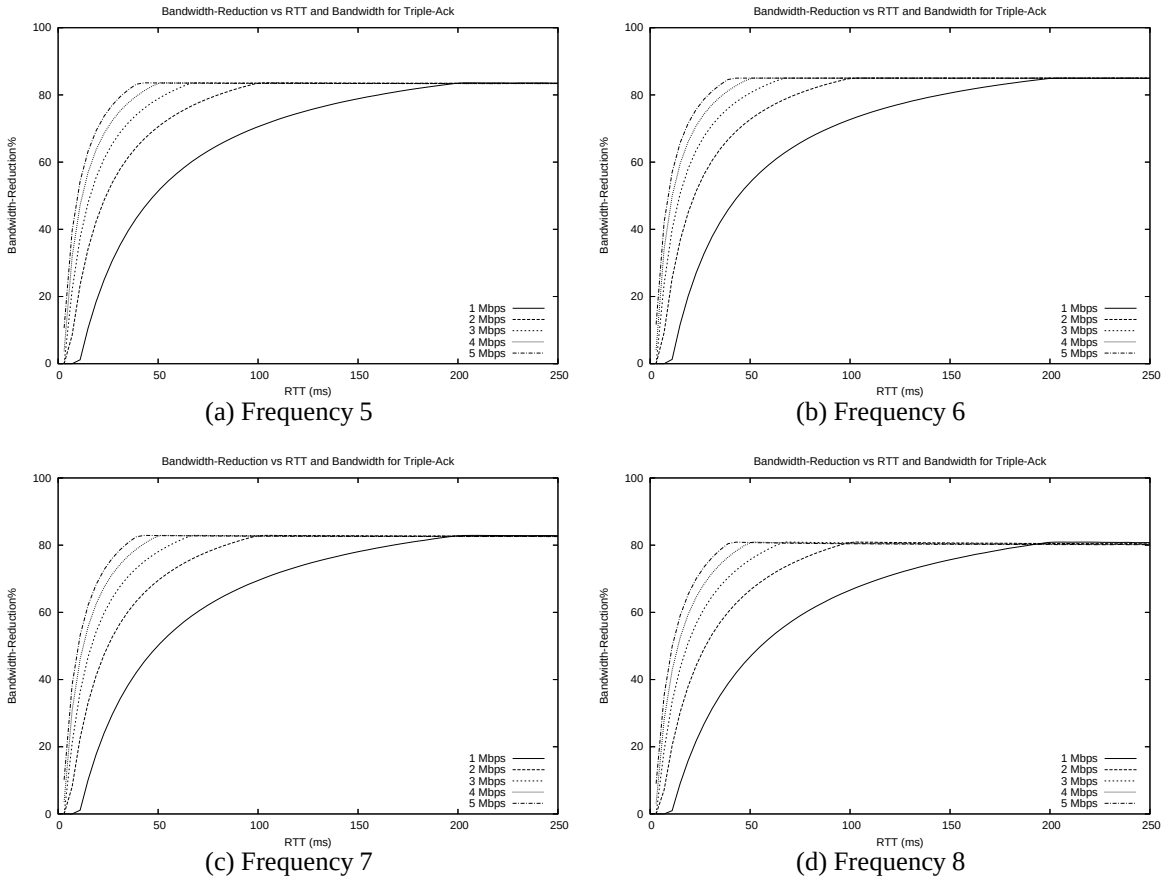
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure B.1: Bandwidth reduction for upload model with freq 1-4 and wincap 20



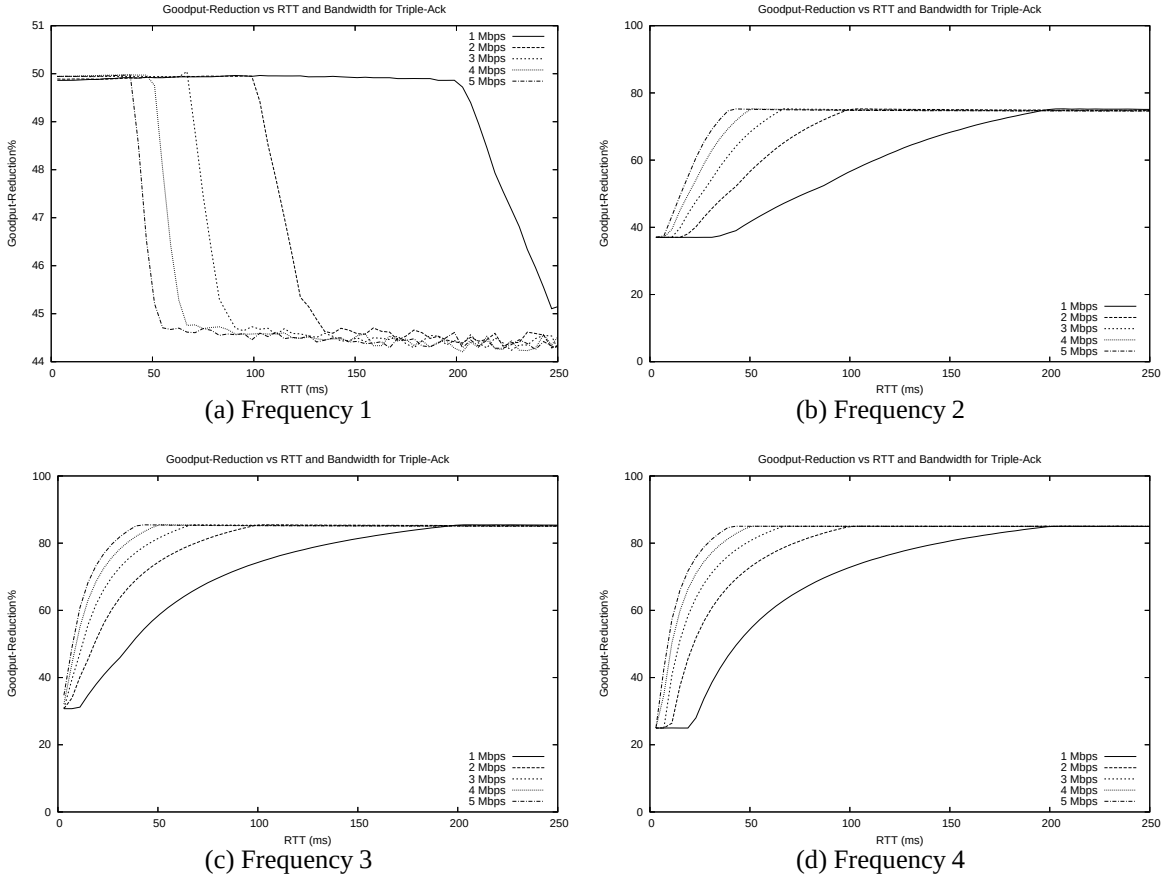
(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure B.2: Bandwidth reduction for upload model with freq 5-8 and wincap 20



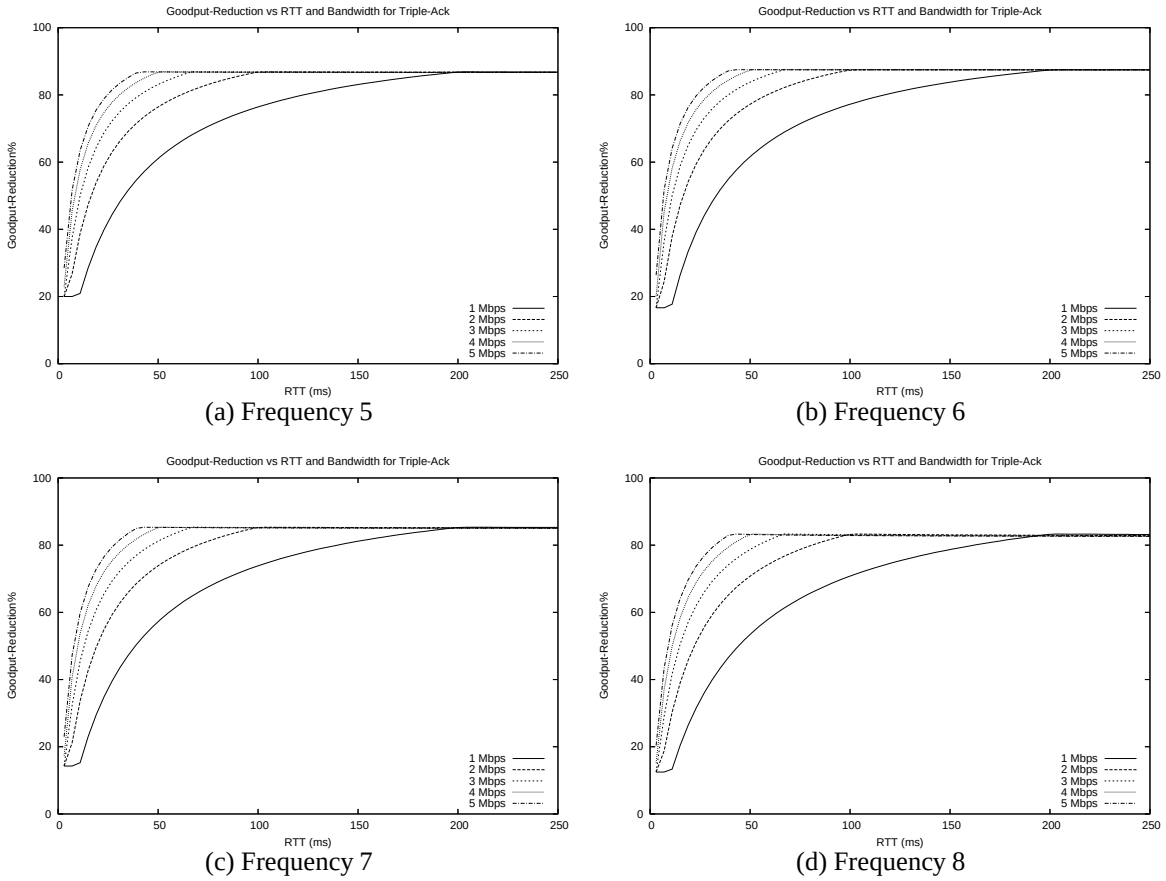
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure B.3: Goodput reduction for upload model with freq 1-4 and wincap 20



(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure B.4: Goodput reduction for upload model with freq 5-8 and wincap 20

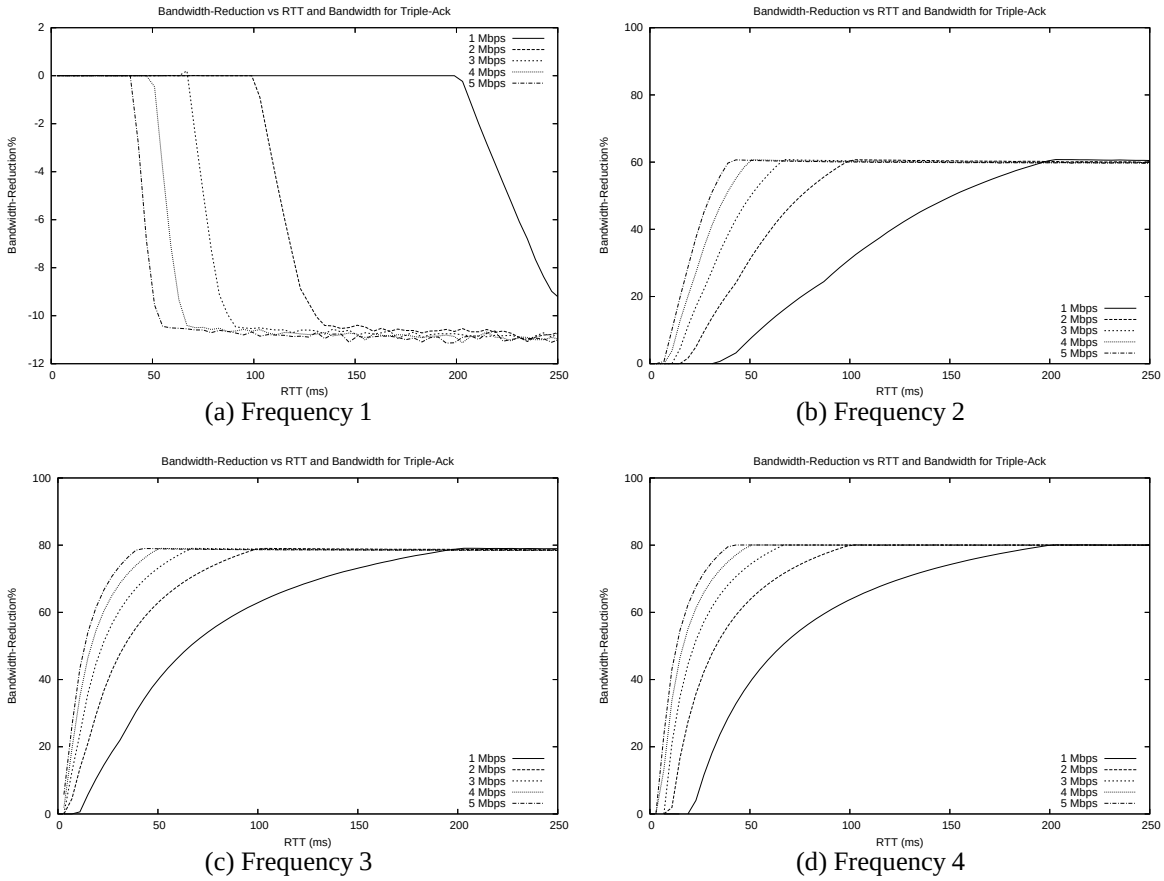
C SIMPLIFIED DOWNLOAD RESULTS

This section contains graphs for the simplified download case. Graphs are shown for frequencies of 1-8 and for bandwidth and goodput reduction. Experiments were run for the following ranges.

Bandwidth 1.0 to 5.0 Mbps, in increments of 1 Mbps

Sender-side latency 1.0 to 125.0 ms per direction, in increments of 1 ms

Receiver-side latency 0.5 to 15.0 ms per direction, in increments of 1 ms



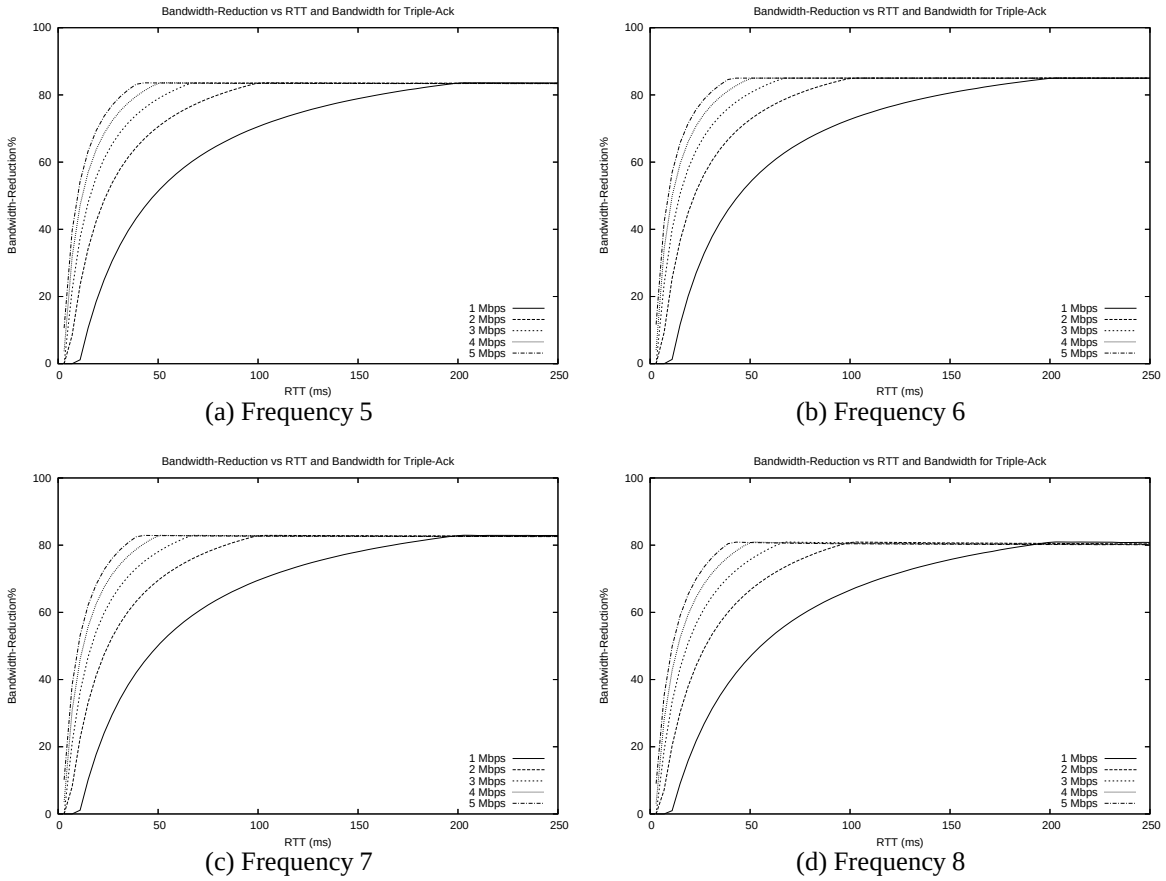
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure C.1: Bandwidth reduction for download model with freq 1-4 and wincap 20



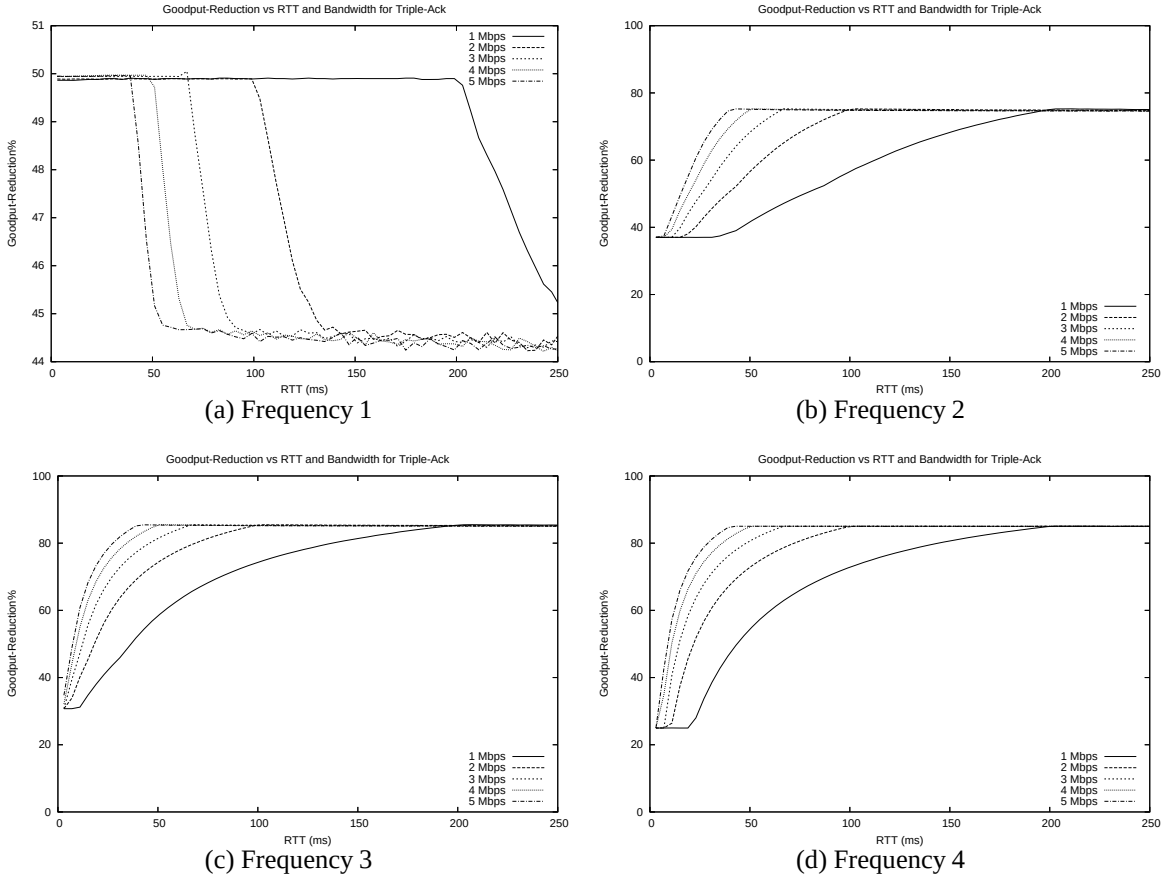
(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure C.2: Bandwidth reduction for download model with freq 5-8 and wincap 20



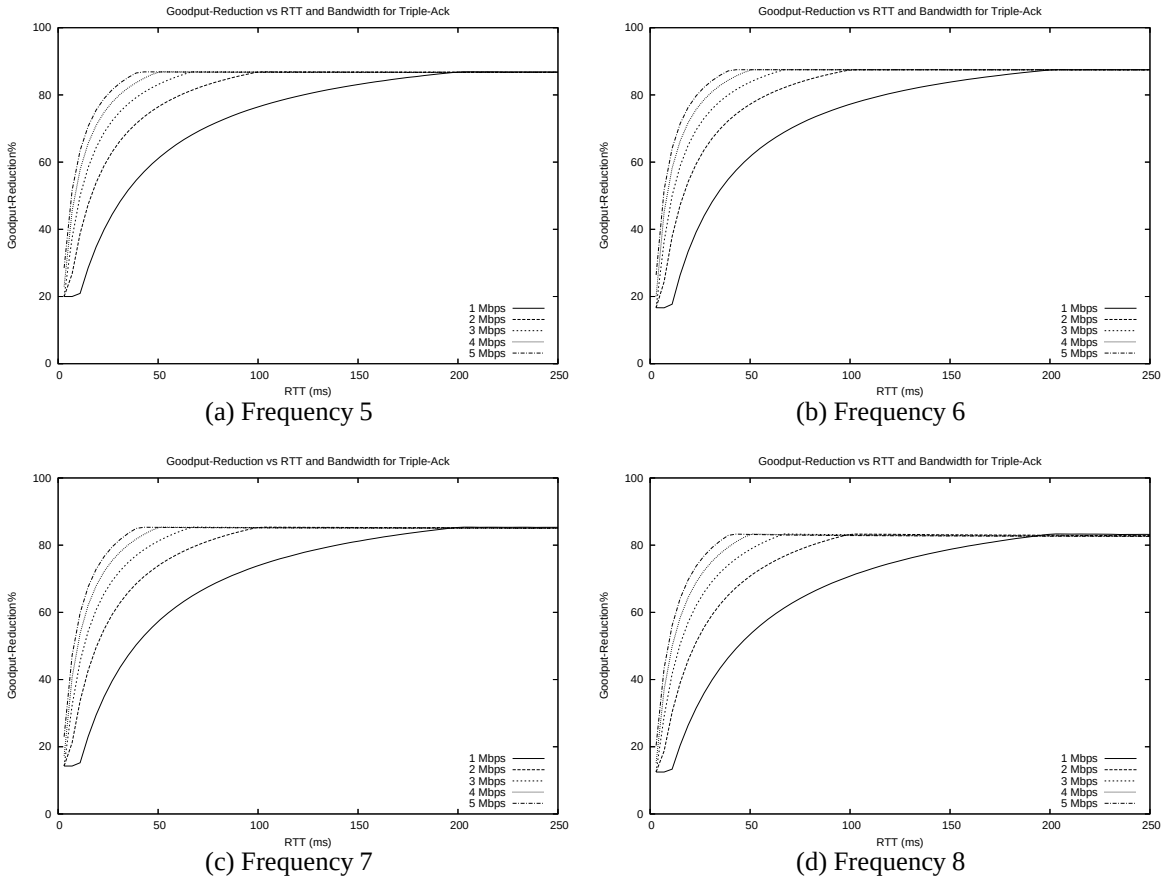
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure C.3: Goodput reduction for download model with freq 1-4 and wincap 20



(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

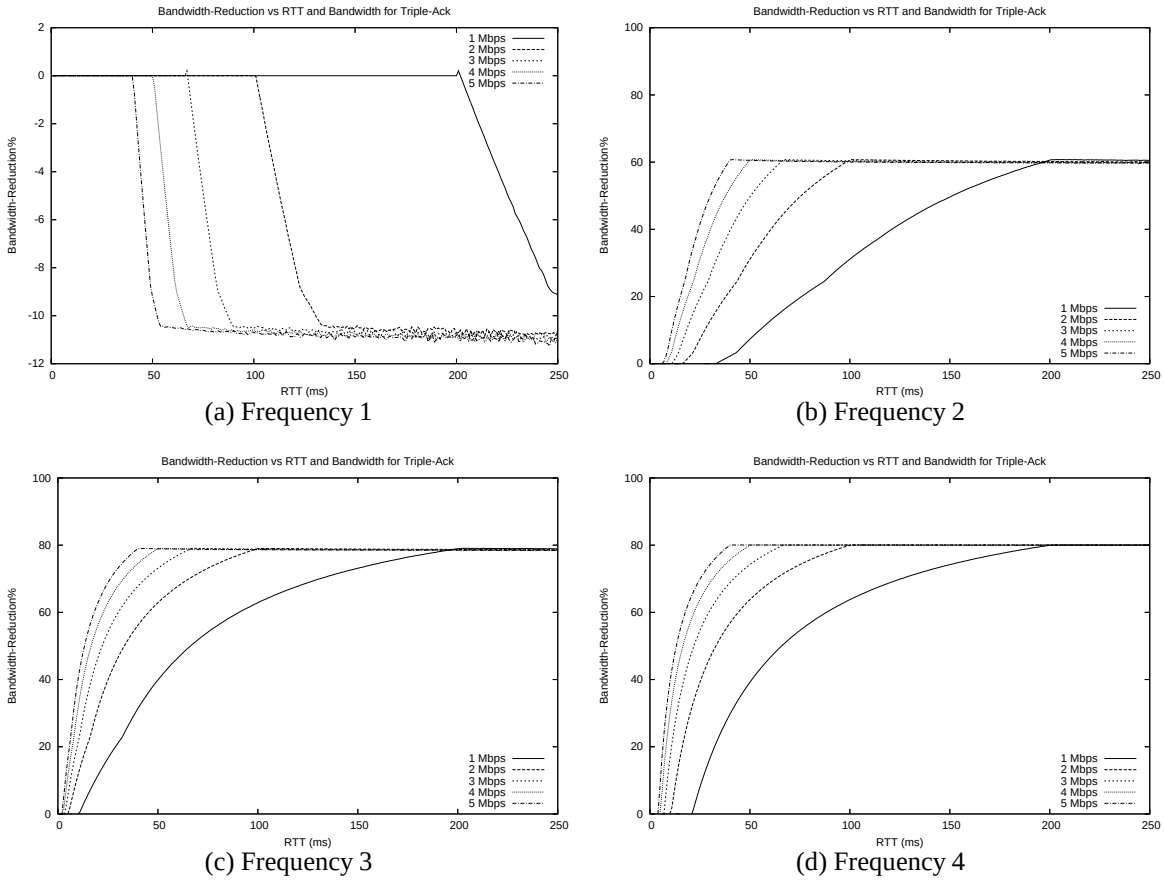
Figure C.4: Goodput reduction for download model with freq 5-8 and wincap 20

D SYMMETRIC LATENCY RESULTS

This section contains graphs for the symmetric latency case. Graphs are shown for frequencies of 1-8 and for bandwidth and goodput reduction. Experiments were run for the following ranges.

Bandwidth 1.0 to 5.0 Mbps, in increments of 1 Mbps

Latency 0.25 to 62.50 ms per direction for each link, in increments of 0.25 ms



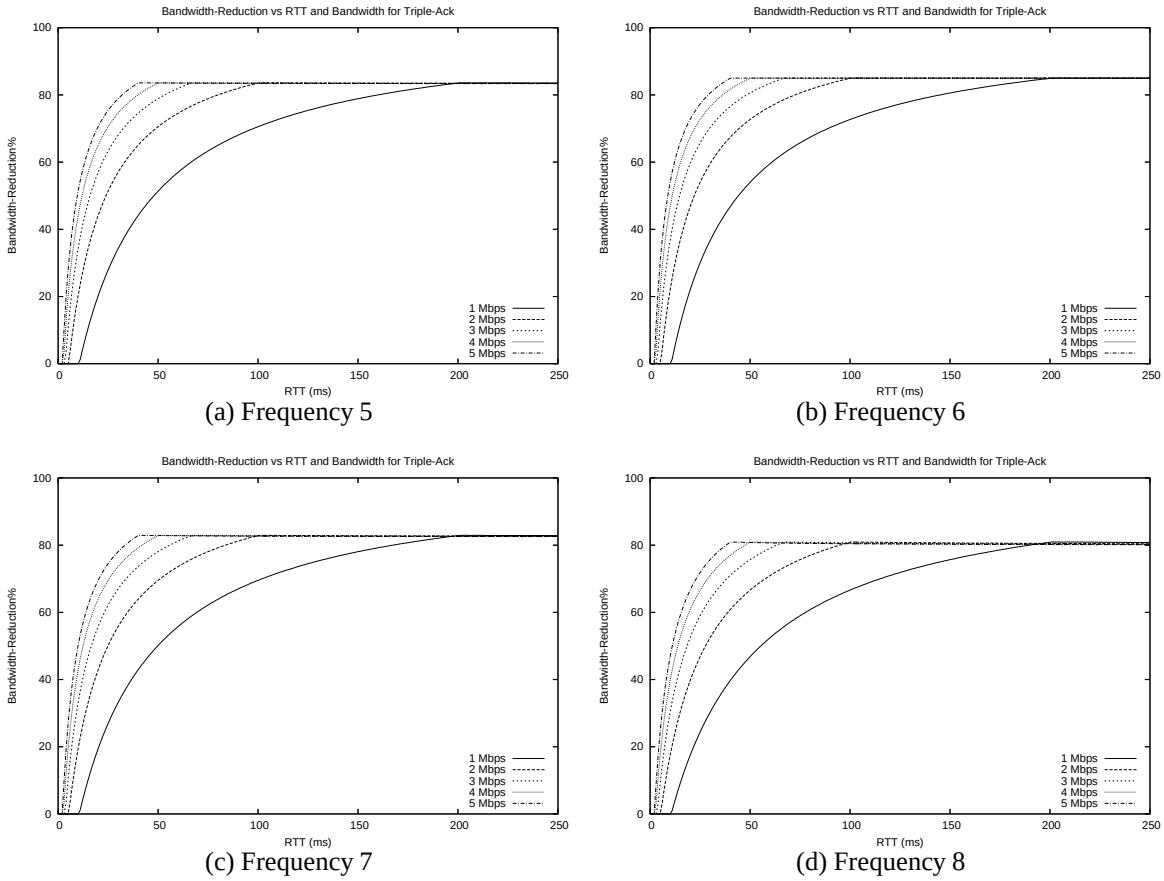
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure D.1: Bandwidth reduction for symmetric model with freq 1-4 and wincap 20



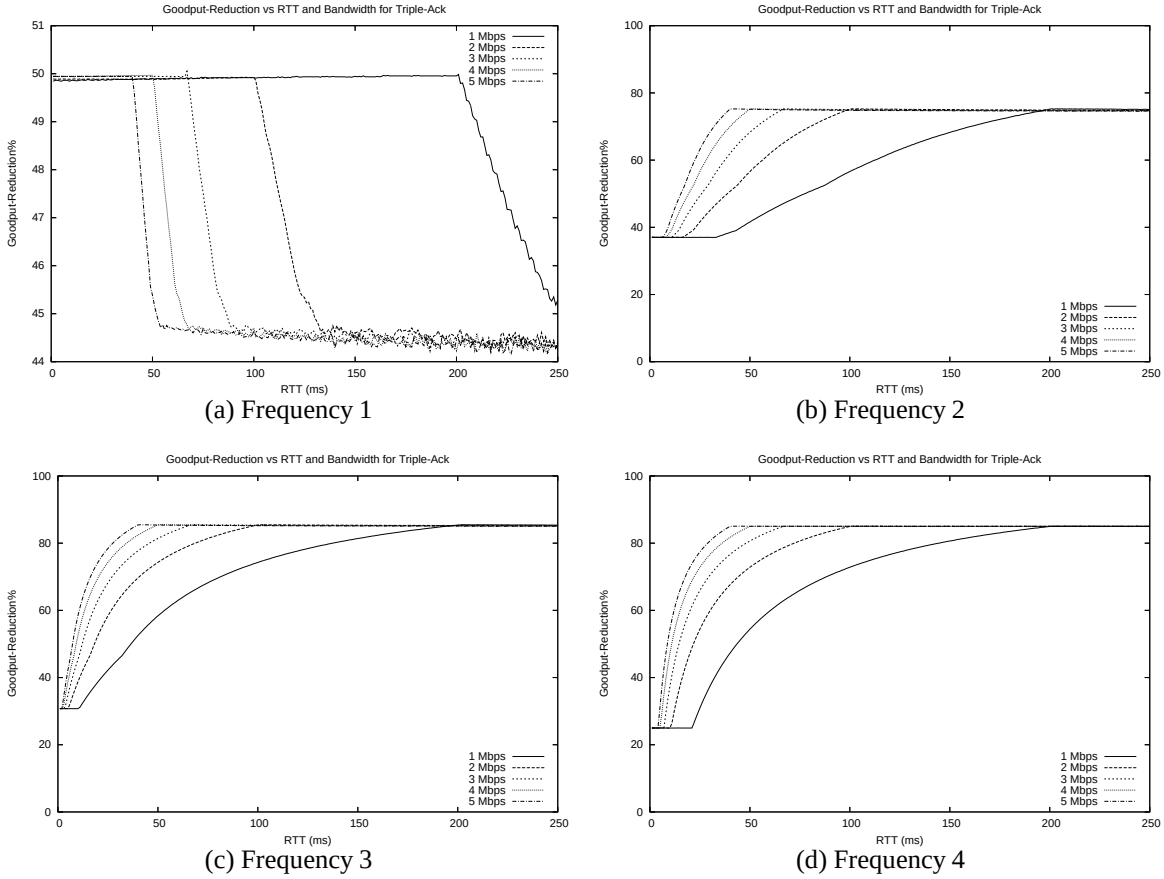
(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure D.2: Bandwidth reduction for symmetric model with freq 5-8 and wincap 20



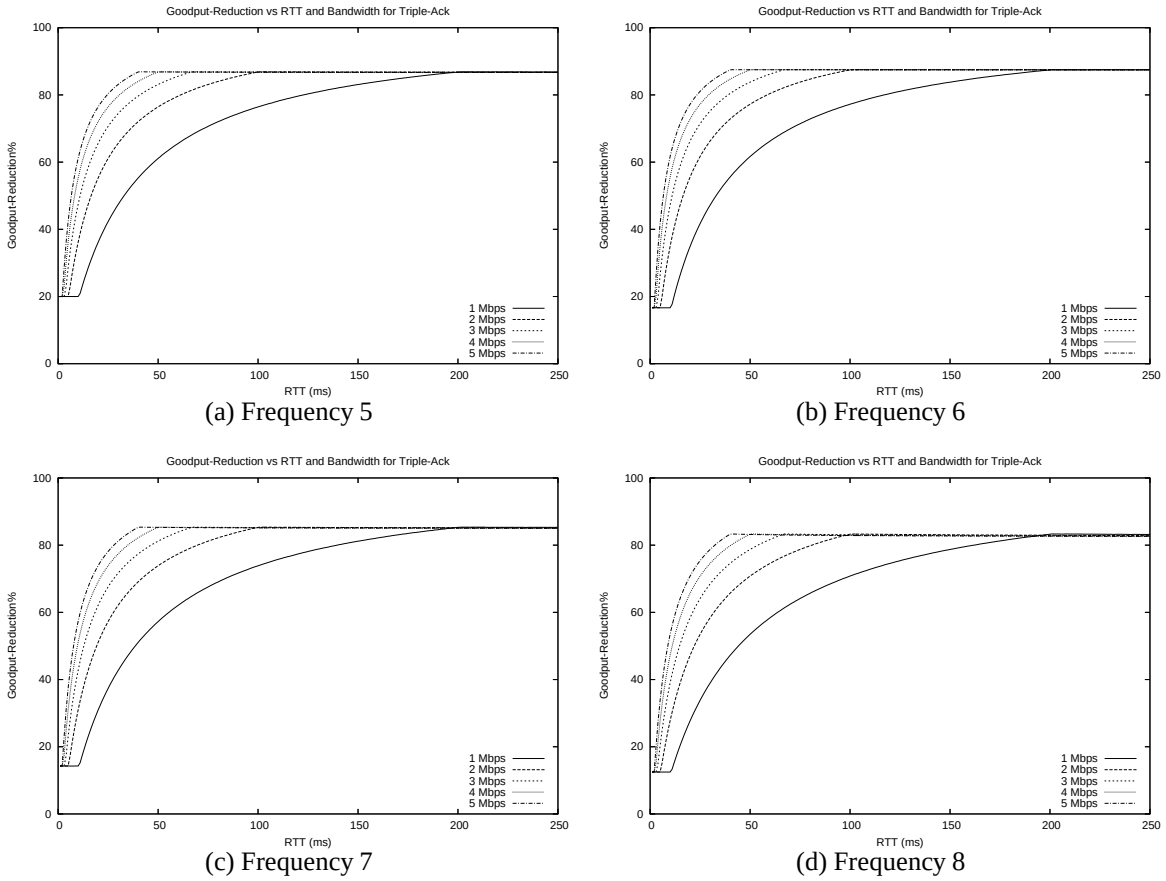
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure D.3: Goodput reduction for symmetric model with freq 1-4 and wincap 20



(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure D.4: Goodput reduction for symmetric model with freq 5-8 and wincap 20

E PACKET-FLOW ANALYSIS

This section contains an experimental analysis of the triple-ACK case of frequency 6, when the window cap is 20 segments. The analysis begins with the first of those 20 ACK segments coming back. After the last step shown, we start repeating what happened at step 45 and enter a cycle.

Table E.1: Packet-flow analysis, frequency 6, wincap 20

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
1.	Normal ACK A0	20	20	Sent in response to D_{-1}. Send 1 new data segment D19, since only 19 data segments still unACKed ($< cwnd$).	+1 New

2.	Triple ACKs A0'	10	10
			
			13
				Send 1 retx data segment D0'
				`cwnd=cwnd/2`
				`thresh=cwnd`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

3.	Normal ACK A1	10	10.1	Sent in response to D0 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` No new transmission, since 19 data segments still unACKed ($\geq cwnd$).	

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
4.	Normal ACK A2	10	10.199	Sent in response to D1 `cwnd += 1/cwnd` No new transmission, since 18 data segments still unACKed ($\geq cwnd$).	
5.	Normal ACK A3	10	10.297	Sent in response to D2 `cwnd += 1/cwnd` No new transmission, since 17 data segments still unACKed ($\geq cwnd$).	
6.	Normal ACK A4	10	10.394	Sent in response to D3 `cwnd += 1/cwnd` No new transmission, since 16 data segments still unACKed ($\geq cwnd$).	
7.	Normal ACK A5	10	10.490	Sent in response to D4 `cwnd += 1/cwnd` No new transmission, since 15 data segments still unACKed ($\geq cwnd$).	
8.	Normal ACK A6	10	10.586	Sent in response to D5 `cwnd += 1/cwnd` No new transmission, since 14 data segments still unACKed ($\geq cwnd$).	

9.	Triple ACKs A6'	5	5	Send 1 retx data segment D6'
				`cwnd=cwnd/2`
				`thresh=cwnd`
					+1 Old

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
			8	`cwnd = thresh+3`, due to artificial $cwnd$ inflation	
10.	Normal ACK A7	5	5.2	Sent in response to D6 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` No new transmission, since 13 data segments still unACKed ($\geq cwnd$).	
11.	Normal ACK A8	5	5.392	Sent in response to D7 `cwnd += 1/cwnd` No new transmission, since 12 data segments still unACKed ($\geq cwnd$).	
12.	Normal ACK A9	5	5.578	Sent in response to D8 `cwnd += 1/cwnd` No new transmission, since 11 data segments still unACKed ($\geq cwnd$).	
13.	Normal ACK A10	5	5.757	Sent in response to D9 `cwnd += 1/cwnd` No new transmission, since 10 data segments still unACKed ($\geq cwnd$).	
14.	Normal ACK A11	5	5.931	Sent in response to D10 `cwnd += 1/cwnd` No new transmission, since 9 data segments still unACKed ($\geq cwnd$).	

Continued on next page. . .

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
15.	Normal ACK A12	5	6.099	Sent in response to D11 `cwnd += 1/cwnd` No new transmission, since 8 data segments still unACKed ($\geq cwnd$).	

16.	Triple ACKs A12'	3	3
			
			6
				Send 1 retx data segment D12'
				`cwnd=cwnd/2`
				`thresh=cwnd`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

17.	Normal ACK A13	3	3.333	Sent in response to D12 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` No new transmission, since 7 data segments still unACKed ($\geq cwnd$).	
18.	Normal ACK A14	3	3.633	Sent in response to D13 `cwnd += 1/cwnd` No new transmission, since 6 data segments still unACKed ($\geq cwnd$).	
19.	Normal ACK A15	3	3.909	verb—cwnd += 1/cwnd— No new transmission, since 5 data segments still unACKed ($\geq cwnd$).	

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
20.	Normal ACK A16	3	4.164	Sent in response to D15 `cwnd += 1/cwnd` No new transmission, since 4 data segments still unACKed ($\geq cwnd$).	
21.	Normal ACK A17	3	4.405	Sent in response to D16 `cwnd += 1/cwnd` Send 1 new data segment D20, since only 3 data segments still unACKed ($< cwnd$).	+1 New
22.	Normal ACK A18	3	4.632	Sent in response to D17 `cwnd += 1/cwnd` Send 1 new data segment D21, since only 3 data segments still unACKed ($< cwnd$).	+1 New

23.	Triple ACKs A18'	2	2
			
			5
				Send 1 retx data segment D18'
				`cwnd=cwnd/2`
				`thresh=cwnd`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

24.	Normal ACK A19	2	2.5	`cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` No new transmission, since 4 data segments still unACKed ($\geq cwnd$).	

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
25.	Normal ACK A20	2	2.9	Sent in response to D19 `cwnd += 1/cwnd` No new transmission, since 4 data segments still unACKed ($\geq cwnd$).	
26.	Repeat ACK A20'	2	2.9	Sent in response to old D0'	
27.	Repeat ACK A20'	2	2.9	Sent in response to old D6'	

28.	Repeat ACK A20'	2	2
			
			5
				Sent in response to old D12'
				Third duplicate ACK, so it invokes congestion control.
				Send 1 retx data segment D20'
				`cwnd=cwnd/2=1`
				`thresh=cwnd=1`
				But $ssthresh$ must be ≥ 2
				Thus, `thresh=2`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

29.	Normal ACK A21	2	2.5	Sent in response to D20 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` Send 1 new data segment D22, since only 1 unACKed data segment ($< cwnd$).	+1 New

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**

30.	Triple ACKs A21'	2	2
			
			5
				6 ACKs have passed by since the last ACK we triple duplicated. Plus, the ACK number of this ACK is not the same as the last ACK that we triple duplicated. Hence, we triple duplicate this one.
				Send 1 retx data segment D21'
				`cwnd=cwnd/2=1`
				`thresh=cwnd=1`
				But $ssthresh$ must be ≥ 2
				Thus, `thresh=2`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

31.	Normal ACK A22	2	2.5	Sent in response to D21 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` Send 1 new data segment D23, since only 1 unACKed data segment ($< cwnd$).	+1 New
32.	Repeat ACK A22'	2	2.5	Sent in response to old D18'.	
33.	Repeat ACK A22'	2	2.5	Sent in response to old D20'.	
34.	Normal ACK A23	2	2.9	Sent in response to D22 `cwnd += 1/cwnd` Send 1 new data segment D24, since only 1 unACKed data segment ($< cwnd$).	+1 New
35.	Repeat ACK A23'	2	2.9	Sent in response to old D21'.	

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
36.	Normal ACK A24	2	3.245	Sent in response to D23 `cwnd += 1/cwnd` Send 2 new data segments D25 & D26, since only 1 unACKed data segment ($< cwnd$).	+2 New

37.	Triple ACKs A24'	2	2
			
			5
				6 ACKS have passed by since the last ACK we triple duplicated. Plus, the ACK number of this ACK is not the same as the last ACK that we triple duplicated. Hence, we triple duplicate this one.
				Send 1 retx data segment D24'
				`cwnd=cwnd/2=1`
				`thresh=cwnd=1`
				But $ssthresh$ must be ≥ 2
				Thus, `thresh=2`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

38.	Normal ACK A25	2	2.5	Sent in response to D24 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` No new transmission, since 2 data segments still unACKed ($\geq cwnd$).	

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
39.	Normal ACK A26	2	2.9	Sent in response to D25 `cwnd += 1/cwnd` Send 1 new data segment D27, since only 1 data segment still unACKed ($< cwnd$).	+1 New
40.	Normal ACK A27	2	3.245	Sent in response to D26 `cwnd += 1/cwnd` Send 2 new data segments D28 & D29, since only 1 unACKed data segment ($< cwnd$).	+2 New
41.	Repeat ACK A27'	2	3.245	Sent in response to old D24'.	
42.	Normal ACK A28	2	3.553	Sent in response to D27 `cwnd += 1/cwnd` Send 1 new data segment D30, since only 2 unACKed data segments ($< cwnd$).	+1 New
43.	Normal ACK A29	2	3.834	Sent in response to D28 `cwnd += 1/cwnd` Send 1 new data segment D31, since only 2 unACKed data segments ($< cwnd$).	+1 New

Continued on next page...

Table E.1 – Continued

Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**

44.	Triple ACKs A29'	2	2
			
			5
				6 ACKS have passed by since the last ACK we triple duplicated. Plus, the ACK number of this ACK is not the same as the last ACK that we triple duplicated. Hence, we triple duplicate this one.
				Send 1 retx data segment D29'
				`cwnd=cwnd/2=1`
				`thresh=cwnd=1`
				But $ssthresh$ must be ≥ 2
				Thus, `thresh=2`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

45.	Normal ACK A31	2	2.5	Sent in response to D30 `cwnd = thresh`, due to full ACK `cwnd += 1/cwnd` Send 1 new data segment D32, since only 1 data segment still unACKed ($< cwnd$).	+1 New
46.	Normal ACK A32	2	2.9	Sent in response to D31 `cwnd += 1/cwnd` Send 2 new data segments D33 & D34, since only 1 unACKed data segment ($< cwnd$).	+2 New

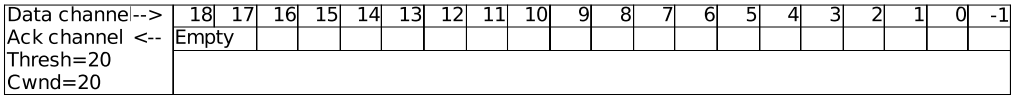
Continued on next page...

Table E.1 – Continued

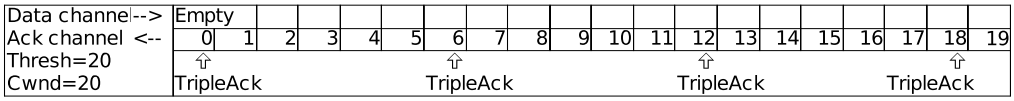
Step	**Event**	$ssthresh$	$cwnd$	**Action**	**Data tx**
47.	Normal ACK A33	2	3.553	Sent in response to D32 `cwnd += 1/cwnd` Send 1 new data segment D35, since only 2 unACKed data segments ($< cwnd$).	+1 New
48.	Normal ACK A34	2	3.834	Sent in response to D33 `cwnd += 1/cwnd` Send 1 new data segment D36, since only 2 unACKed data segments ($< cwnd$).	+1 New

49.	Triple ACKs A34'	2	2
			
			5
				6 ACKS have passed by since the last ACK we triple duplicated. Plus, the ACK number of this ACK is not the same as the last ACK that we triple duplicated. Hence, we triple duplicate this one.
				Send 1 retx data segment D34'
				`cwnd=cwnd/2=1`
				`thresh=cwnd=1`
				But $ssthresh$ must be ≥ 2
				Thus, `thresh=2`
				`cwnd = thresh+3`,
				due to artificial $cwnd$ inflation
					+1 Old

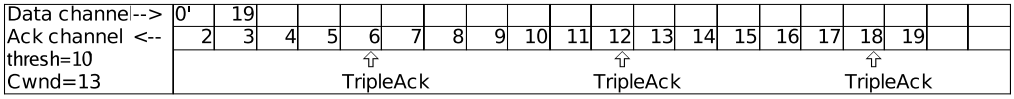
In Figures E.1, E.2, E.3, and E.4, we show the same packet-flow analysis in pictorial form. This time, we take time into account. For convenience, we have labeled the first ACK we duplicate as ACK 0. This means that we had to label the first data segment as -1. Also note the two separate channels for the data and ACK segments. To conserve space, we have labeled the segments simply according to the sequence number (in terms of segments) for data segments and acknowledgement numbers (in terms of segments) for ACK segments. The values of the threshold and congestion window at the end of each step are also shown. We assume that our end-to-end link has a bandwidth·RTT product larger than 20 segments so that we can see all the packets lined up back-to-back. The ACKs that will incur triple-duplication are marked.



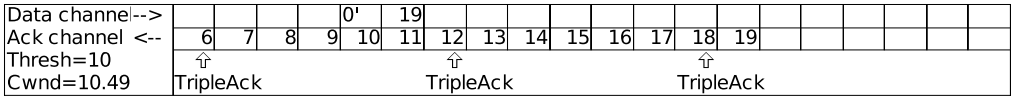
(a) Step 1



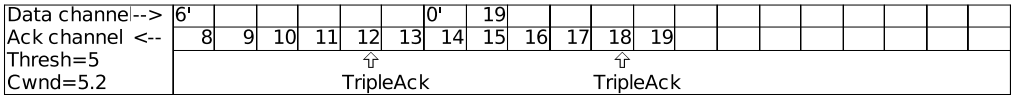
(b) Step 2



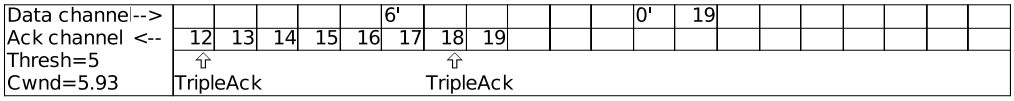
(c) Step 3



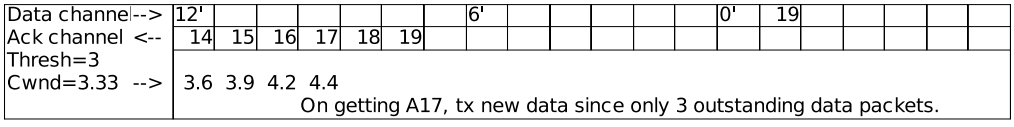
(d) Step 4



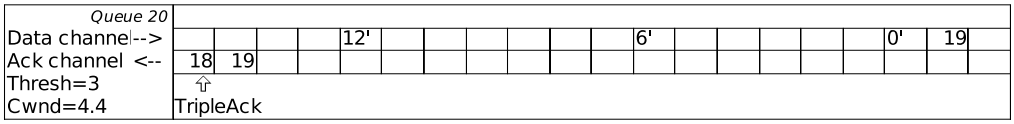
(e) Step 5



(f) Step 6

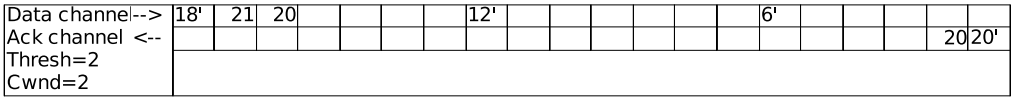


(g) Step 7

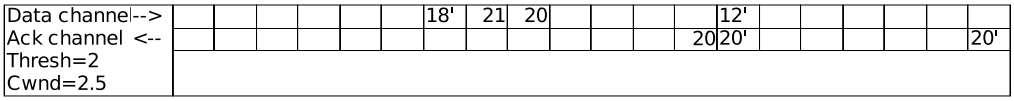


(h) Step 8

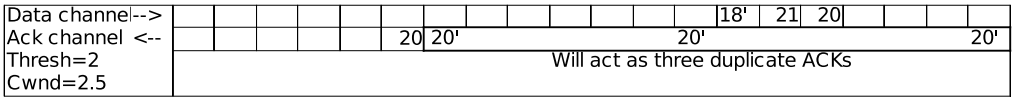
Figure E.1: Pictorial analysis of frequency 6 with window cap 20, steps 1-8



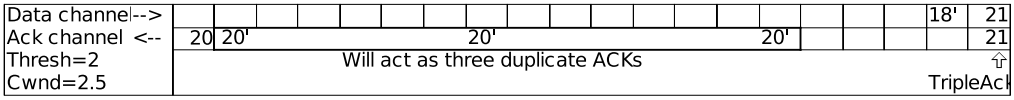
(a) Step 9



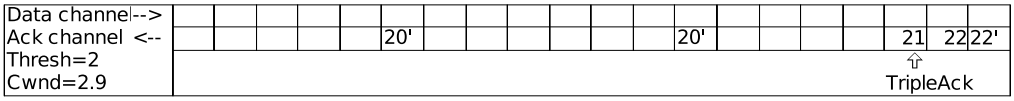
(b) Step 10



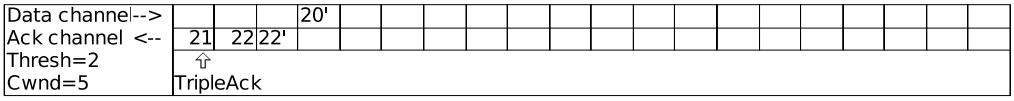
(c) Step 11



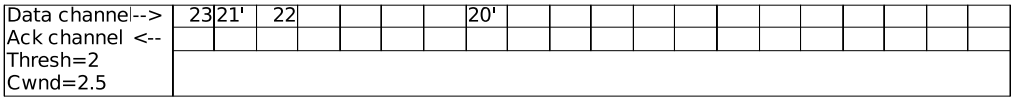
(d) Step 12



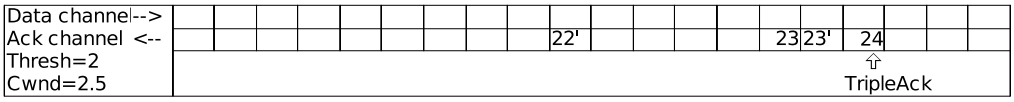
(e) Step 13



(f) Step 14

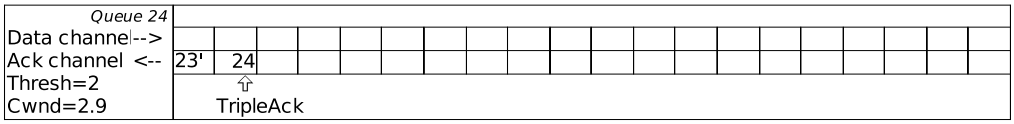


(g) Step 15

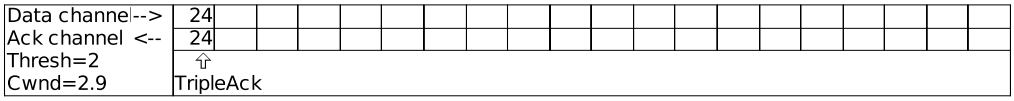


(h) Step 16

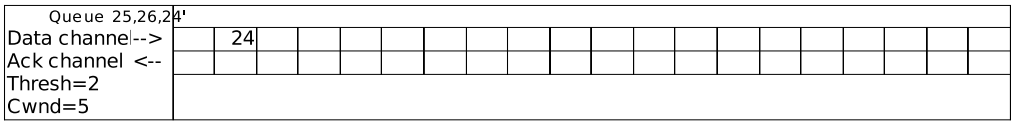
Figure E.2: Pictorial analysis of frequency 6 with window cap 20, steps 9-16



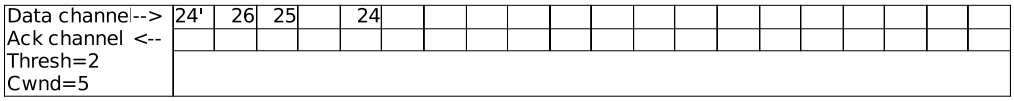
(a) Step 17



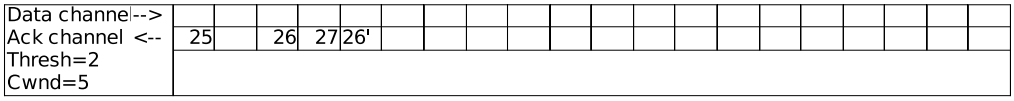
(b) Step 18



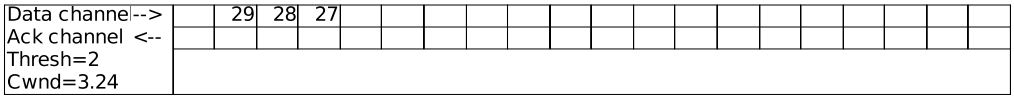
(c) Step 19



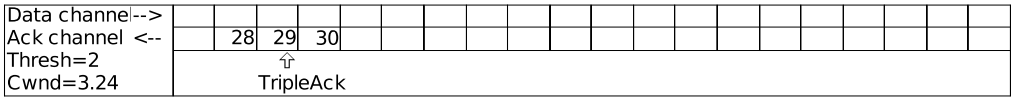
(d) Step 20



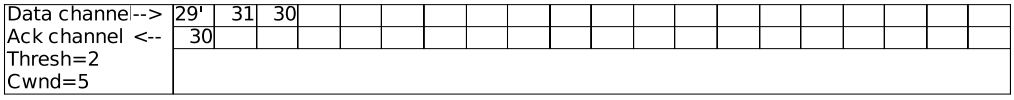
(e) Step 21



(f) Step 22

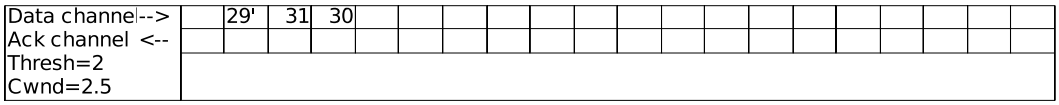


(g) Step 23

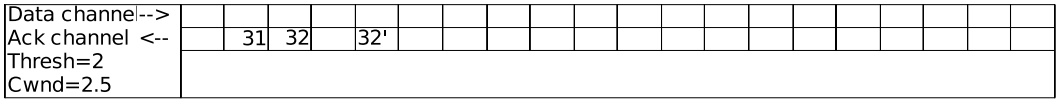


(h) Step 24

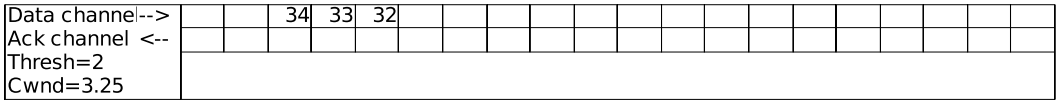
Figure E.3: Pictorial analysis of frequency 6 with window cap 20, steps 17-24



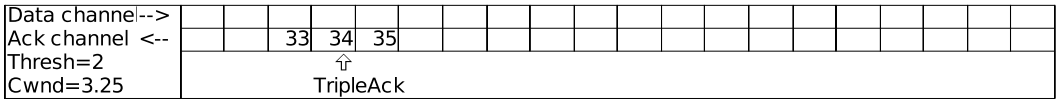
(a) Step 25



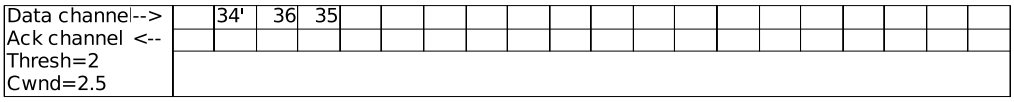
(b) Step 26



(c) Step 27



(d) Step 28



(e) Step 29

Figure E.4: Pictorial analysis of frequency 6 with window cap 20, steps 25-29

F PACKET-FLOW ANALYSIS WITH OTHER WINDOW CAPS

This chapter gives a pictorial analysis of the packet flow when we use a triple-ACK frequency of 6 and window caps of 7 and 2. The same principles apply as those used in Appendix E. It can be seen with a window cap of 7 that the end result is the same as with a window cap of 20 segments. With a window cap of 2 segments, the duplicate segment that increases bandwidth consumption can be seen in steps 15 and 19.

F.1 WINDOW CAP 7

In this section, we analyze the packet flow with a window cap of 7 segments. Figures F.1, F.2, and F.3 show the results.

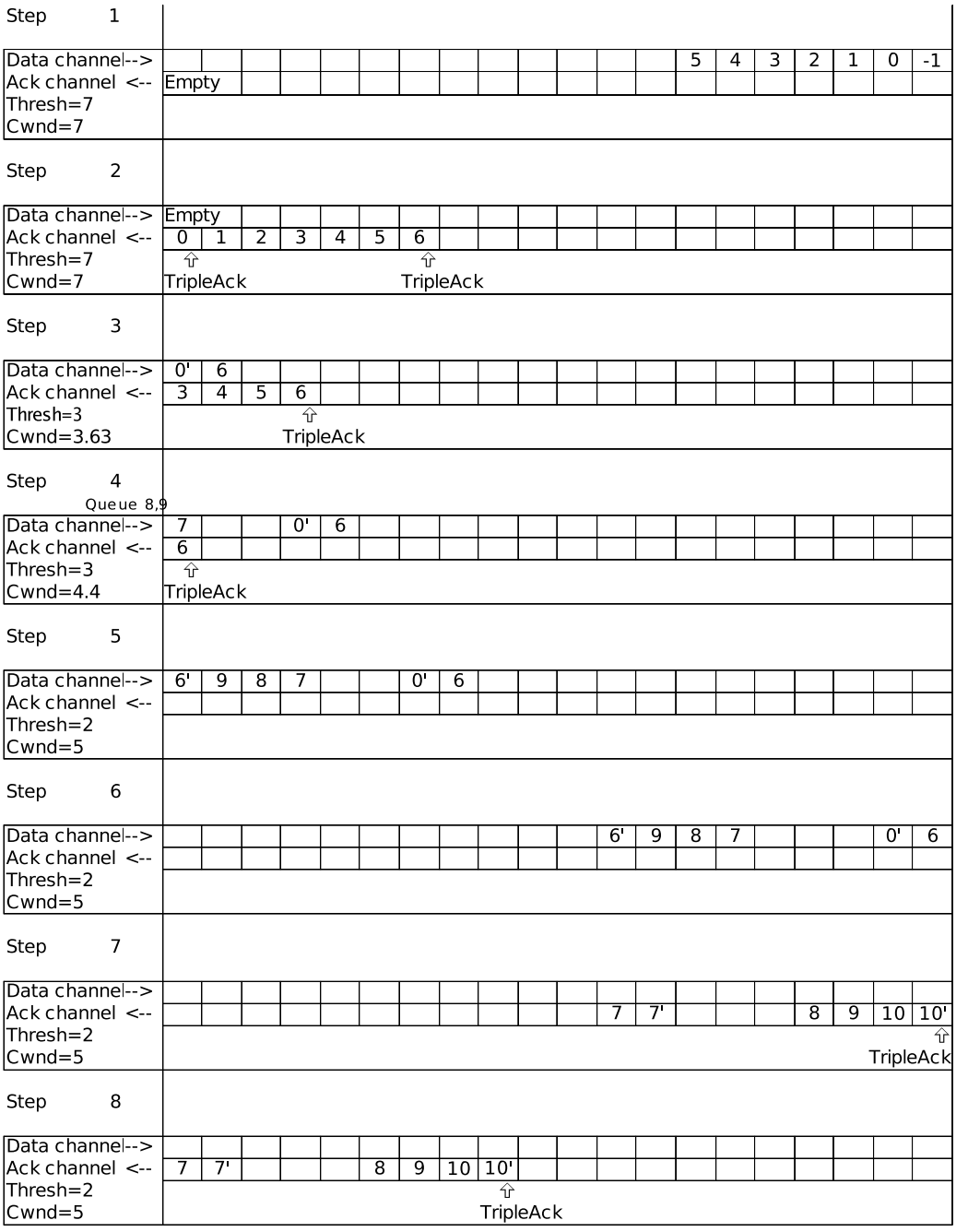


Figure F.1: Pictorial analysis of frequency 6 with window cap 7, steps 1-8

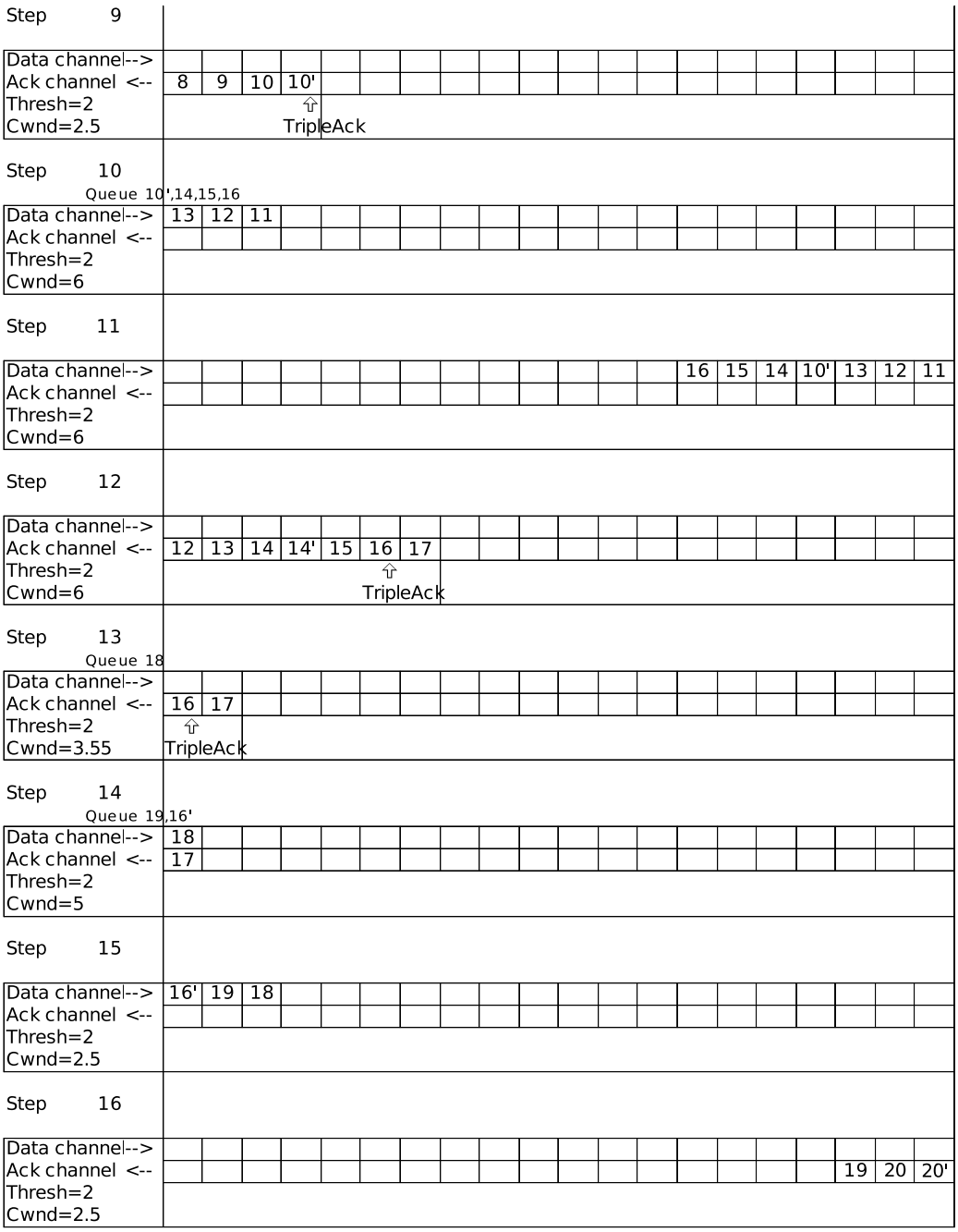


Figure F.2: Pictorial analysis of frequency 6 with window cap 7, steps 9-16

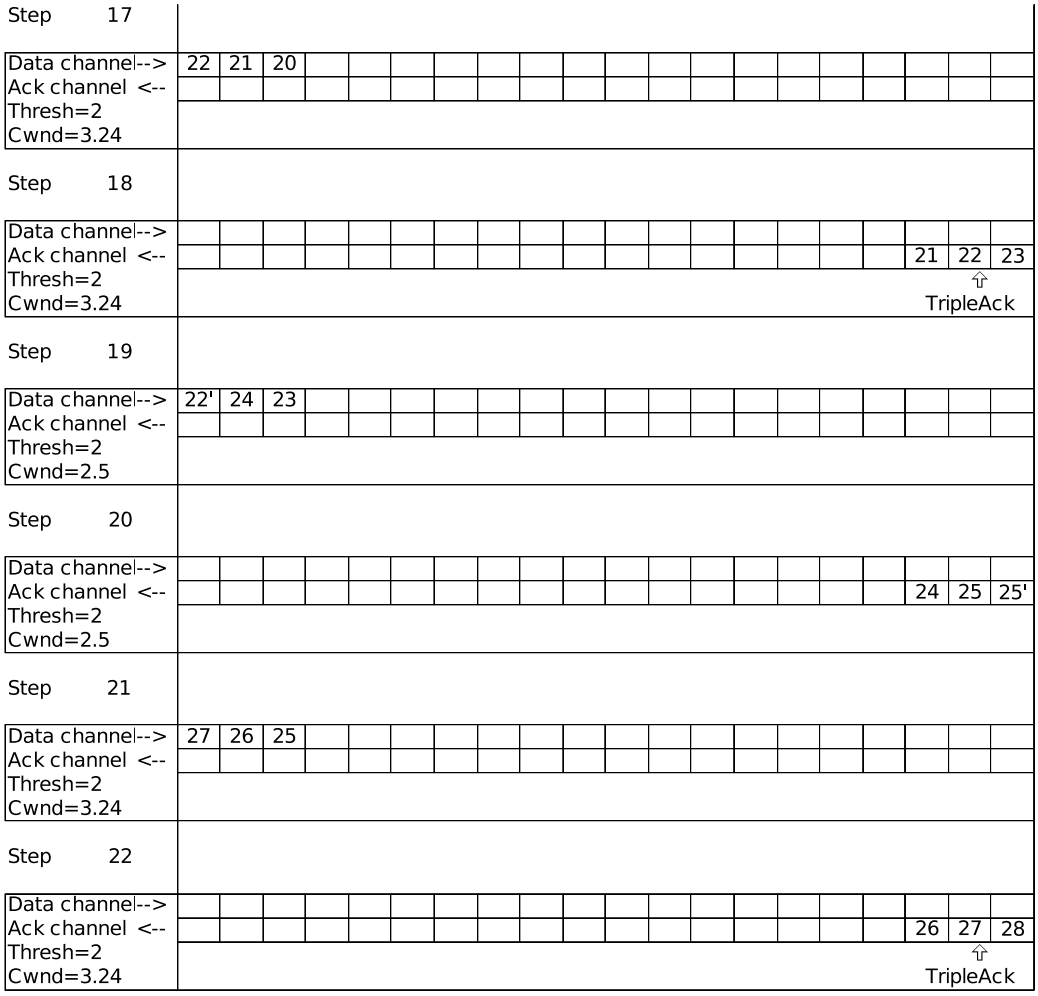


Figure F.3: Pictorial analysis of frequency 6 with window cap 7, steps 17-22

F.2 WINDOW CAP 2

In this section, we analyze the packet flow with a window cap of 2 segments. Figures F.4, F.5, and F.6 show the results.

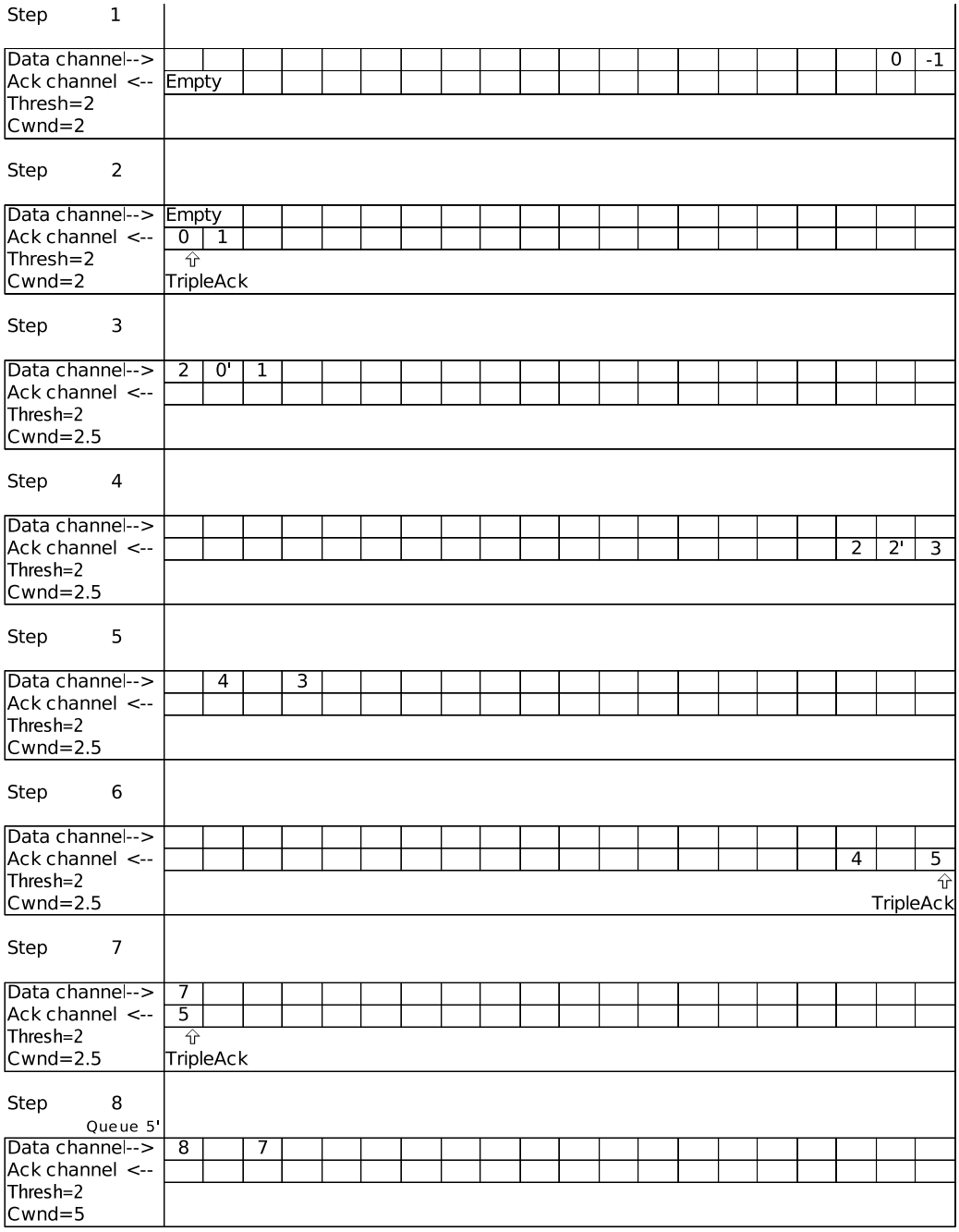


Figure F.4: Pictorial analysis of frequency 6 with window cap 2, steps 1-8

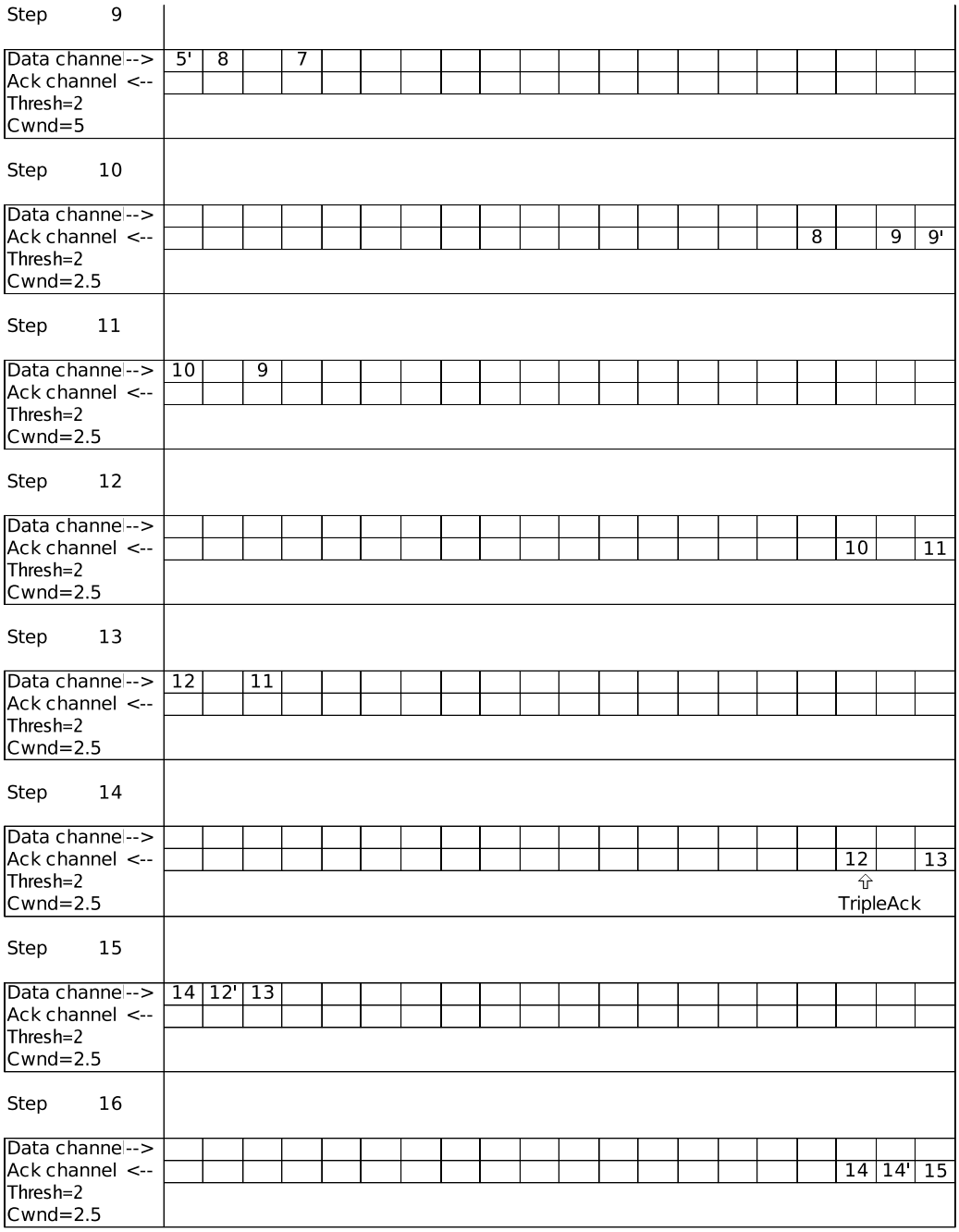


Figure F.5: Pictorial analysis of frequency 6 with window cap 2, steps 9-16

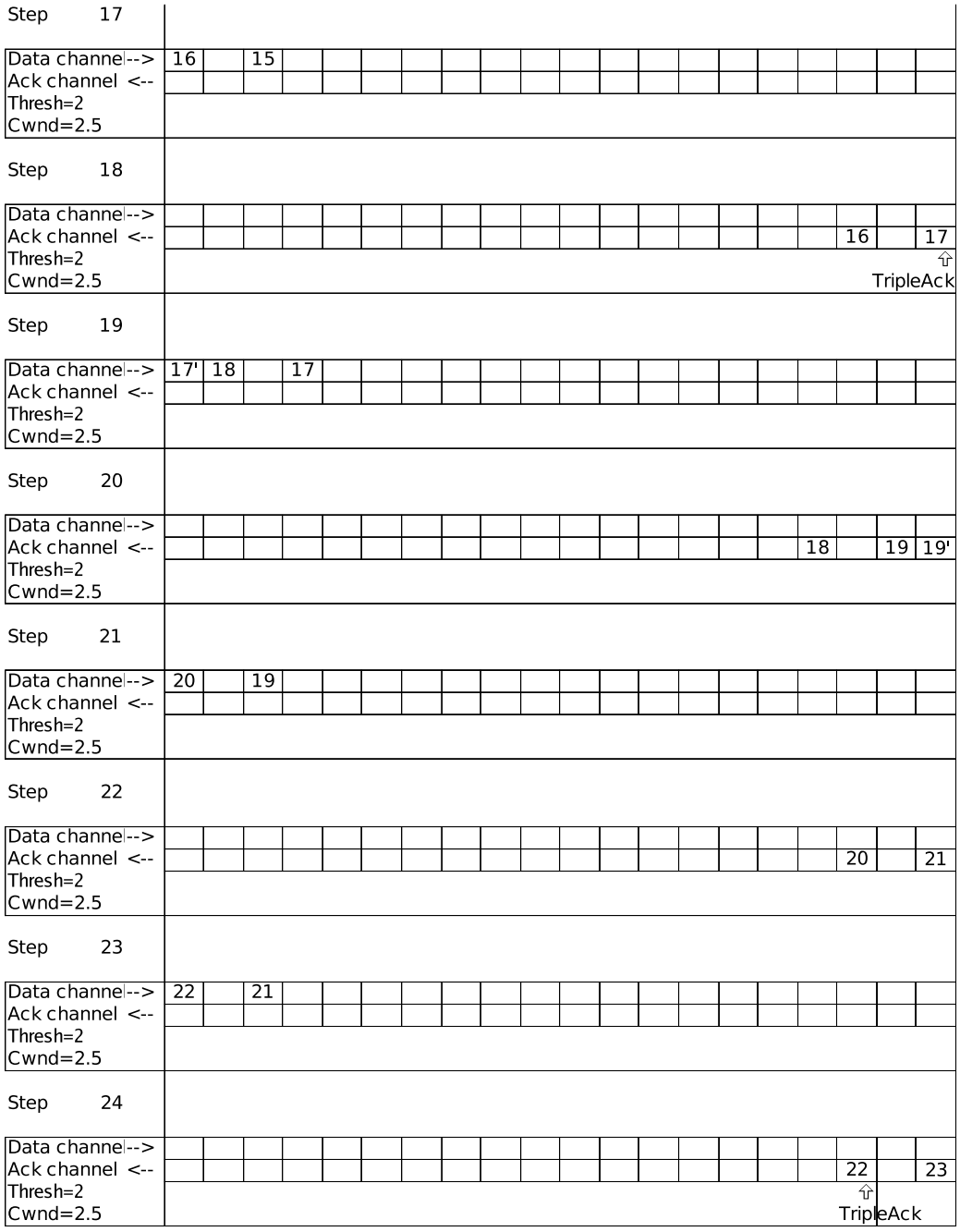


Figure F.6: Pictorial analysis of frequency 6 with window cap 2, steps 17-24

G SAMPLE RAW RESULTS

This chapter shows some sample results in their raw form. This section is not intended to be complete by any means. It simply serves to show the format of our data.

Data collected for normal and triple-ACK cases looks as shown below, with one entry for each combination of bandwidth and latency. Each line after the bandwidth and latency title refers to a particular link and contains back-to-back information in the form of ”field, value”.

Bandwidth(Mbps),1.0,Latency(ms),5.0
,,,,source,0,destination,1,time(s),61.00,bytes,5952800,bandwidth(kbps),780.65,packets,
4252,retxBytes,991200,retxPackets,708,throughput(kbps),758.34,goodput(kbps),632.07
,,,,source,1,destination,2,time(s),61.00,bytes,5954200,bandwidth(kbps),780.89,packets,
4253,retxBytes,992600,retxPackets,709,throughput(kbps),758.57,goodput(kbps),632.12
,,,,source,2,destination,1,time(s),60.99,bytes,170080,bandwidth(kbps),22.31,packets,
4252,retxBytes,28320,retxPackets,708,throughput(kbps),0.00,goodput(kbps),0.00
,,,,source,1,destination,0,time(s),61.00,bytes,255160,bandwidth(kbps),33.46,packets,
6379,retxBytes,113400,retxPackets,2835,throughput(kbps),0.00,goodput(kbps),0.00

The two sets of data, normal and triple-ACK, are then compared and calculations are done for badput and reduction in bandwidth, throughput, and goodput. This data looks as shown below.

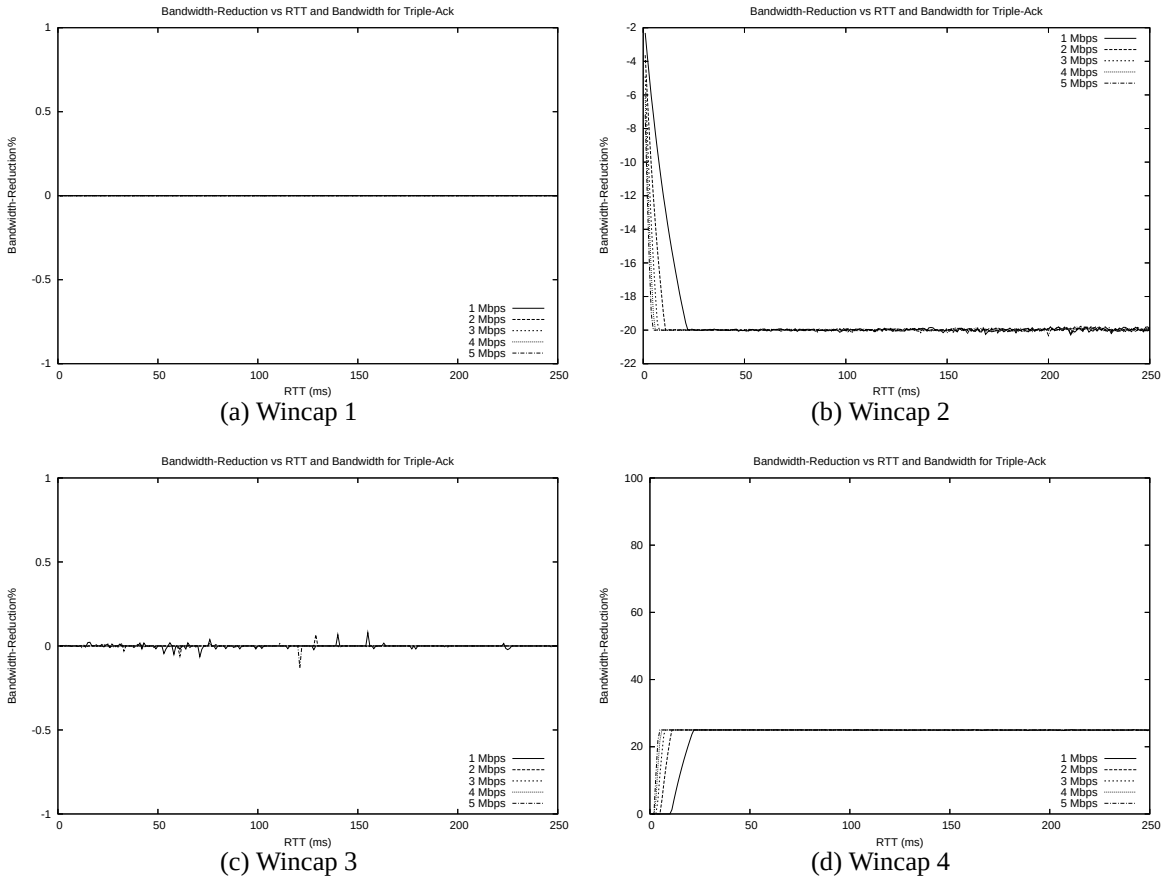
Bandwidth(Mbps),1.0,Latency(ms),45.5, 83.6106555738%
Bandwidth(Mbps),1.0,Latency(ms),11.75, 52.0199192032%
Bandwidth(Mbps),5.0,Latency(ms),50.25, 85.0094459016%
Bandwidth(Mbps),5.0,Latency(ms),49.25, 84.9861750322%
Bandwidth(Mbps),5.0,Latency(ms),26.5, 84.9920964236%
Bandwidth(Mbps),4.0,Latency(ms),34.25, 85.0150549681%
Bandwidth(Mbps),4.0,Latency(ms),9.5, 80.8039419606%
Bandwidth(Mbps),3.0,Latency(ms),8.25, 72.4685263639%

H WINDOW-CAP VARIATION RESULTS

This chapter shows the results obtained for bandwidth and goodput reduction by varying the window cap. The triple-ACK frequency used was 6. The window caps used were 1-10 in increments of 1 and 15-65 in increments of 5. Experiments were run using the symmetric setup for the following ranges.

Bandwidth 1.0 to 5.0 Mbps, in increments of 1 Mbps

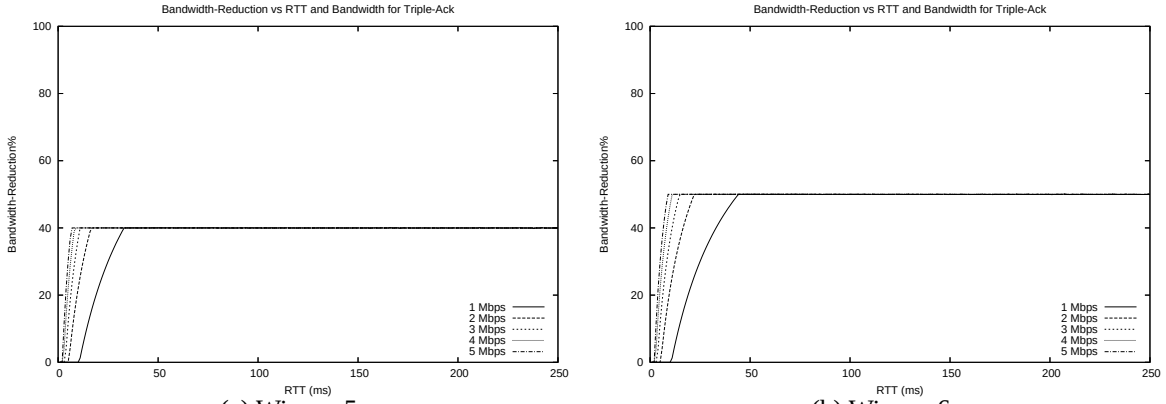
Latency 0.25 to 62.50 ms per direction for each link, in increments of 0.25 ms



(a) Wincap 1 (b) Wincap 2

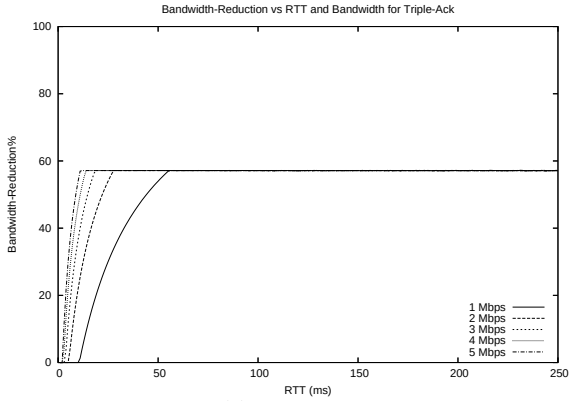
(c) Wincap 3 (d) Wincap 4

Figure H.1: Bandwidth reduction with frequency 6, window caps 1-4

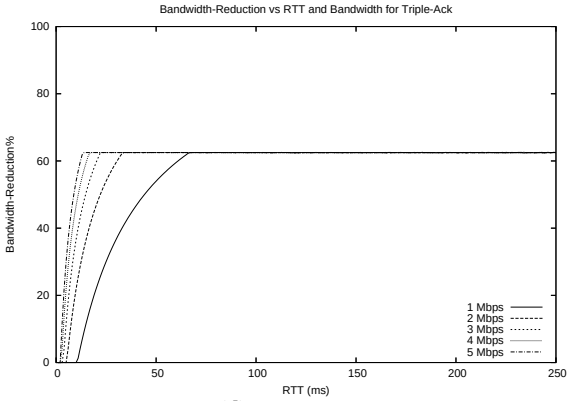


(a) Wincap 5

(b) Wincap 6

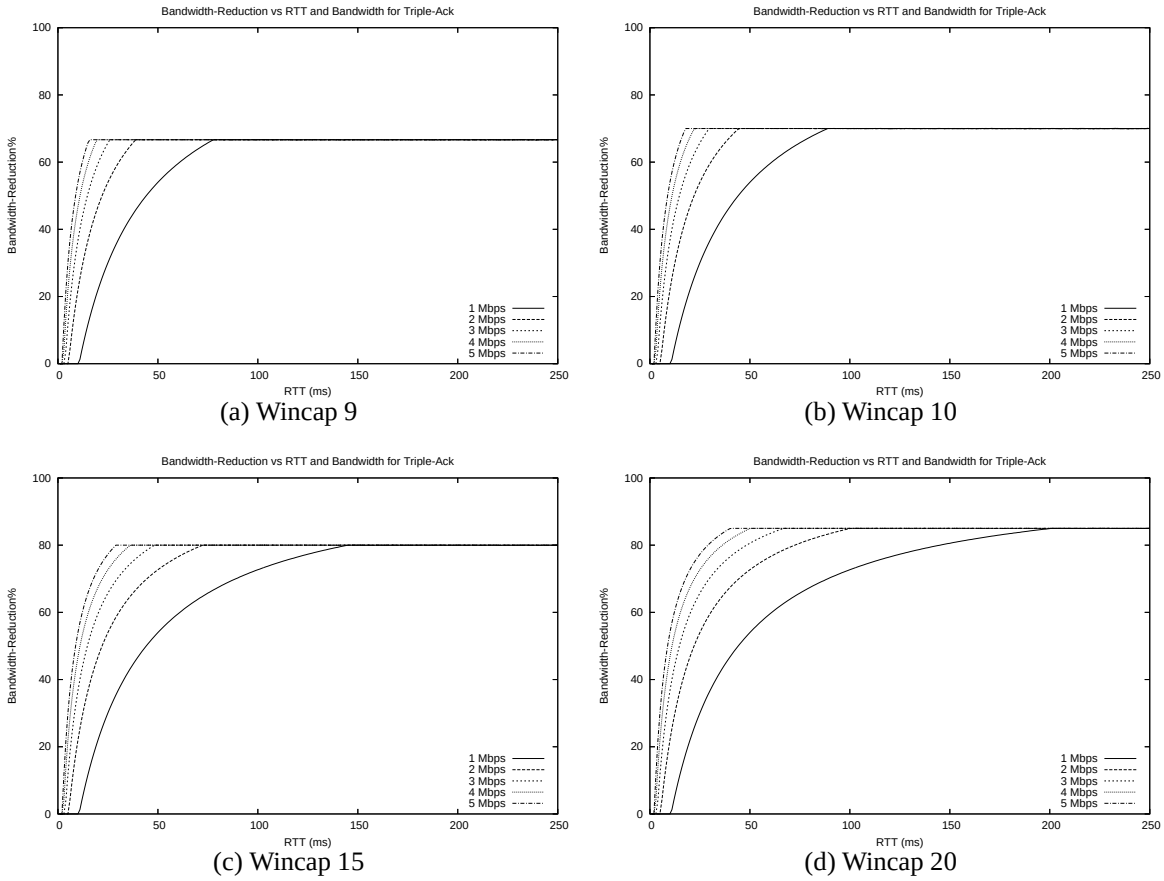


(c) Wincap 7



(d) Wincap 8

Figure H.2: Bandwidth reduction with frequency 6, window caps 5-8



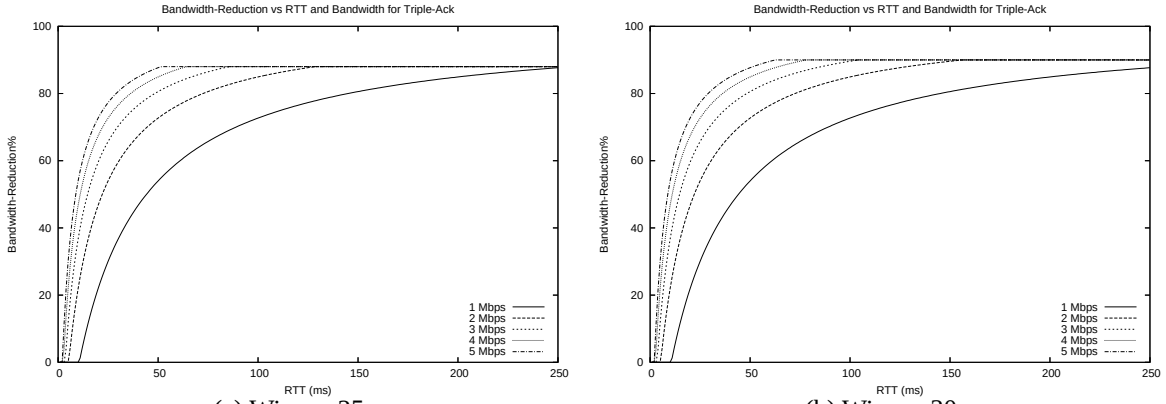
(a) Wincap 9

(b) Wincap 10

(c) Wincap 15

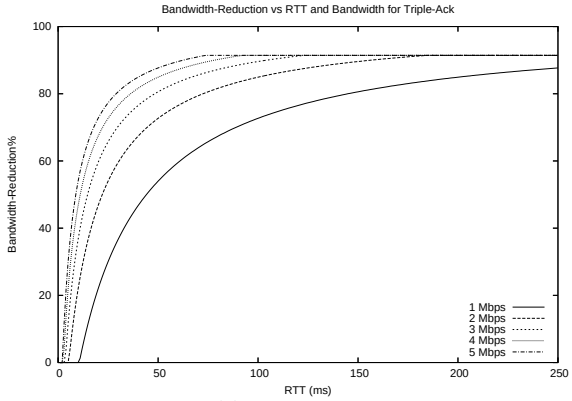
(d) Wincap 20

Figure H.3: Bandwidth reduction with frequency 6, window caps 9-20

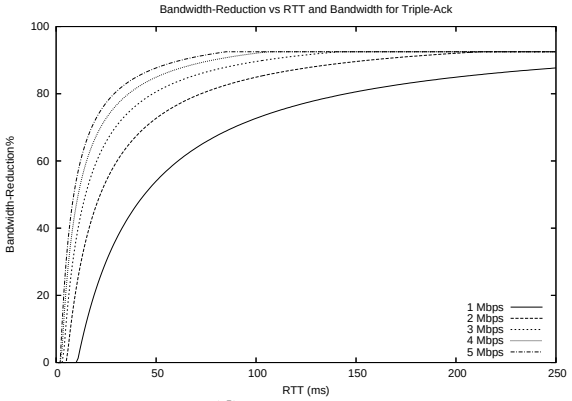


(a) Wincap 25

(b) Wincap 30

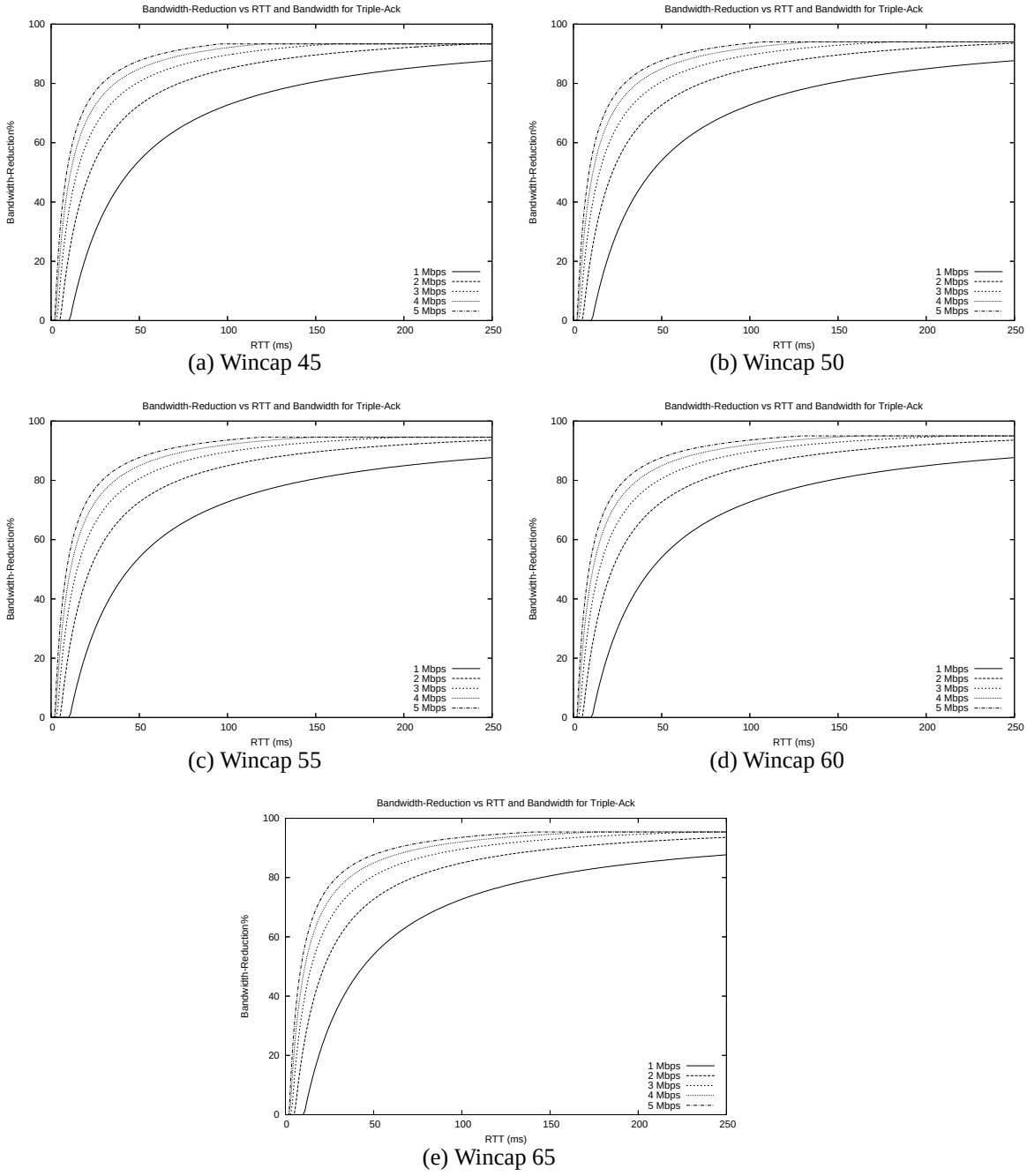


(c) Wincap 35



(d) Wincap 40

Figure H.4: Bandwidth reduction with frequency 6, window caps 25-40



(a) Wincap 45

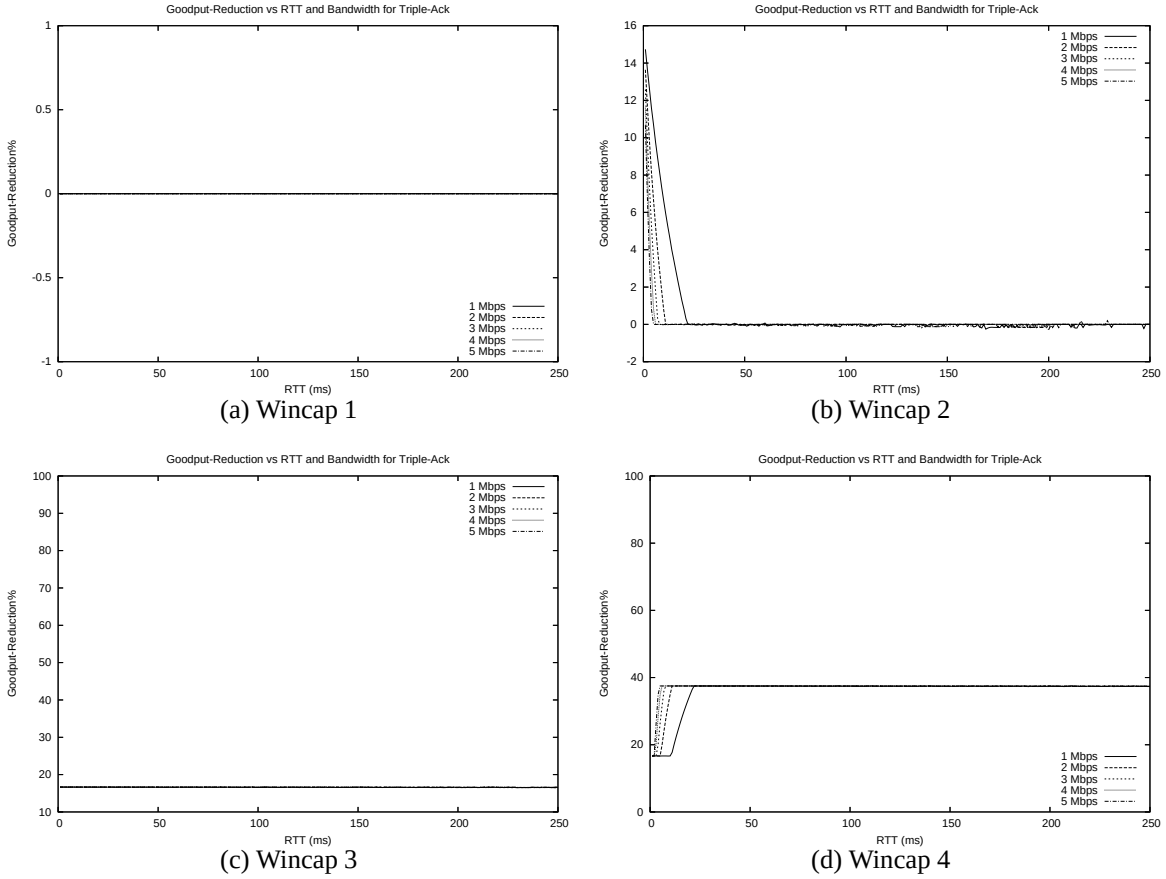
(b) Wincap 50

(c) Wincap 55

(d) Wincap 60

(e) Wincap 65

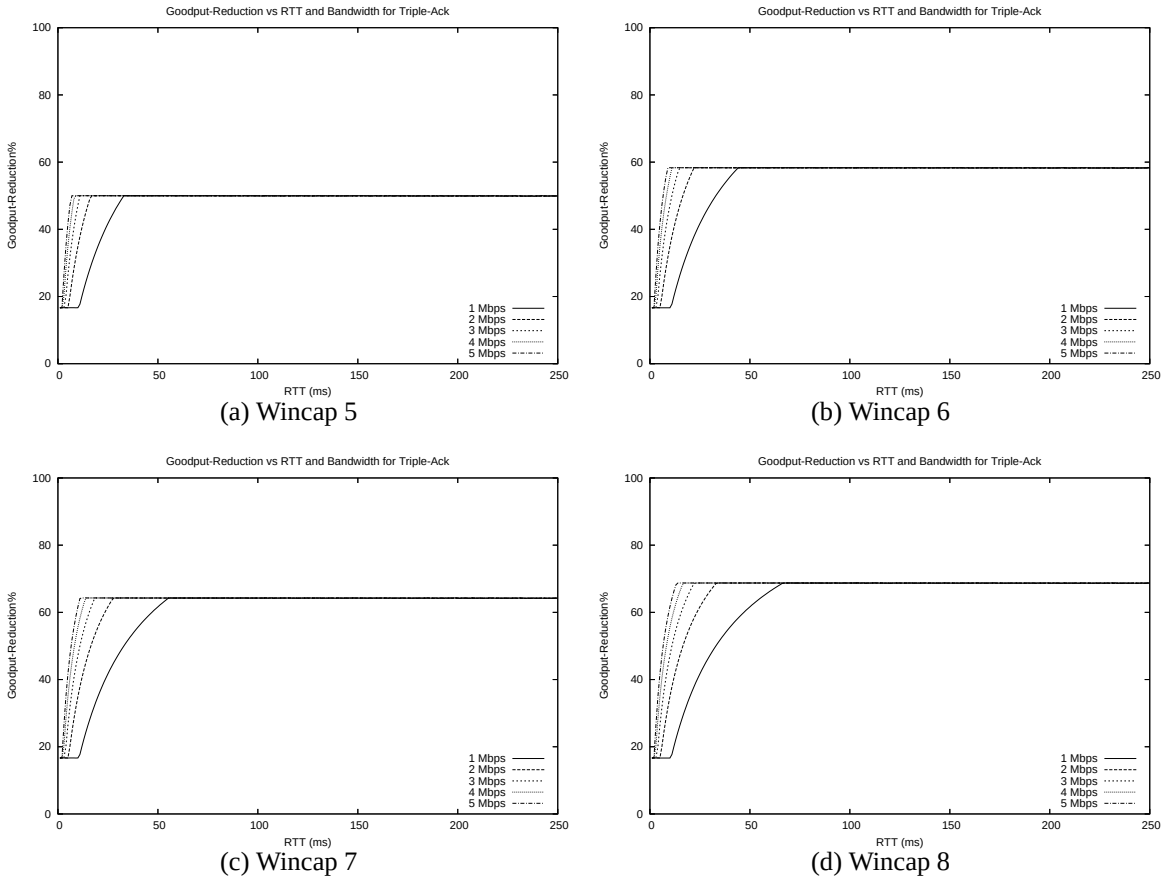
Figure H.5: Bandwidth reduction with frequency 6, window caps 45-65



(a) Wincap 1 (b) Wincap 2

(c) Wincap 3 (d) Wincap 4

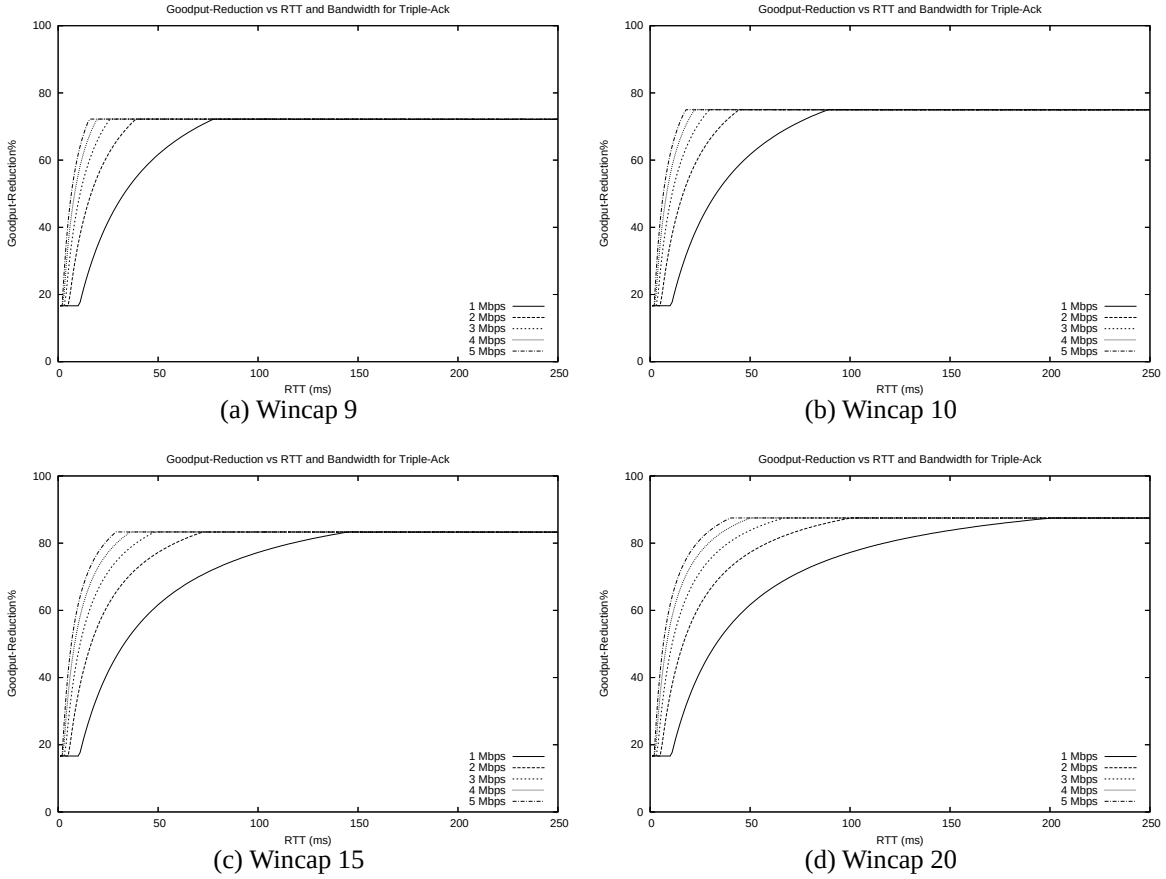
Figure H.6: Goodput reduction with frequency 6, window caps 1-4



(a) Wincap 5 (b) Wincap 6

(c) Wincap 7 (d) Wincap 8

Figure H.7: Goodput reduction with frequency 6, window caps 5-8



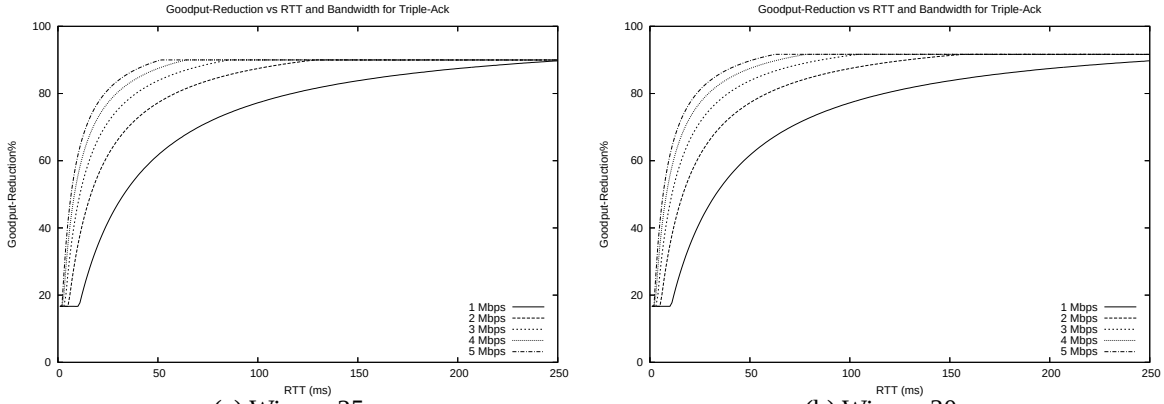
(a) Wincap 9

(b) Wincap 10

(c) Wincap 15

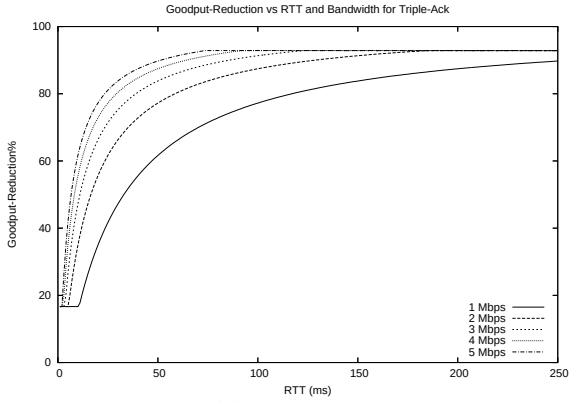
(d) Wincap 20

Figure H.8: Goodput reduction with frequency 6, window caps 9-20

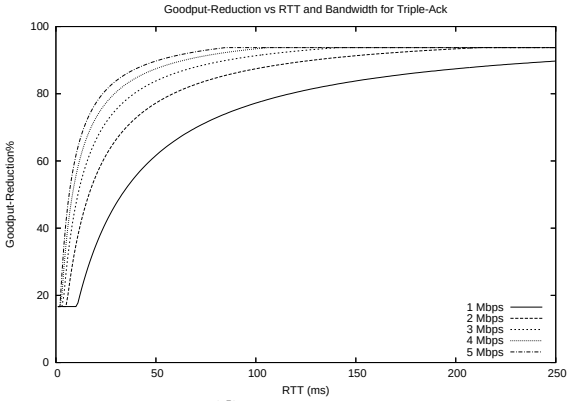


(a) Wincap 25

(b) Wincap 30

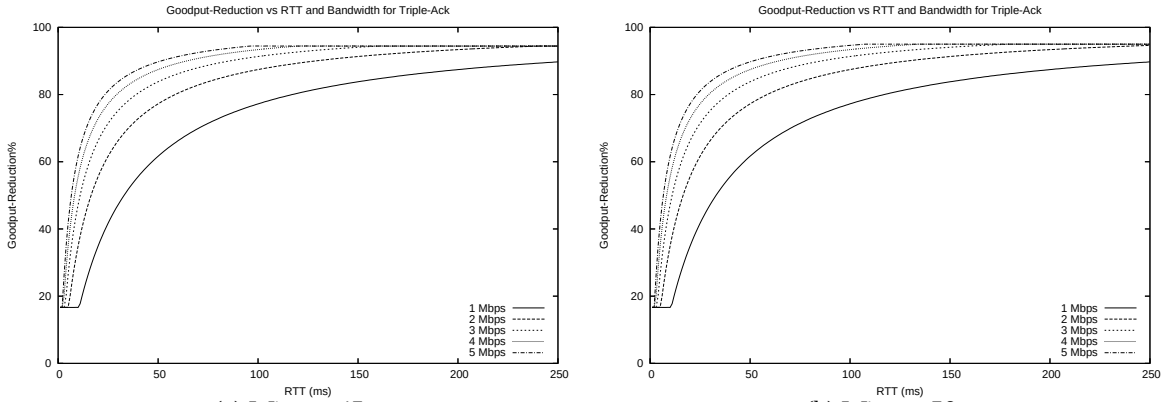


(c) Wincap 35



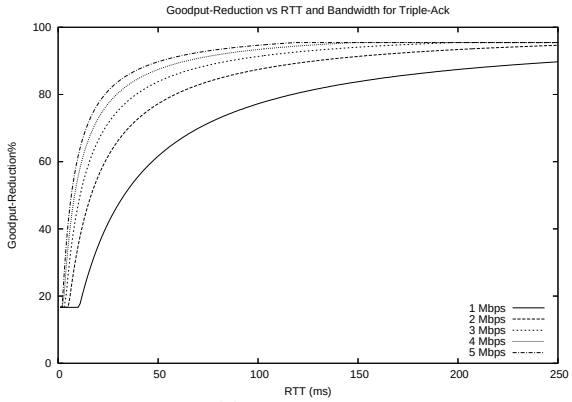
(d) Wincap 40

Figure H.9: Goodput reduction with frequency 6, window caps 25-40

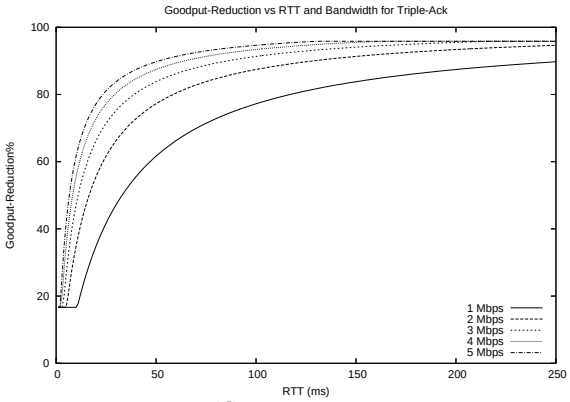


(a) Wincap 45

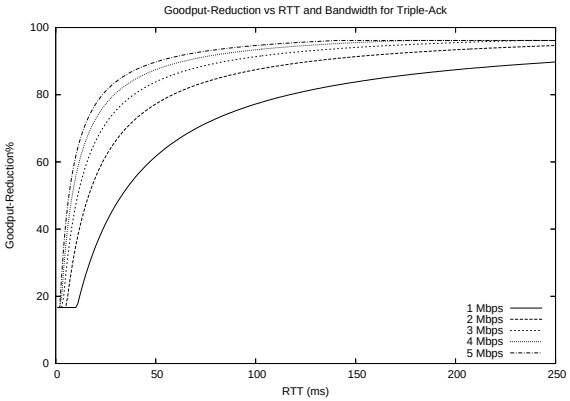
(b) Wincap 50



(c) Wincap 55



(d) Wincap 60



(e) Wincap 65

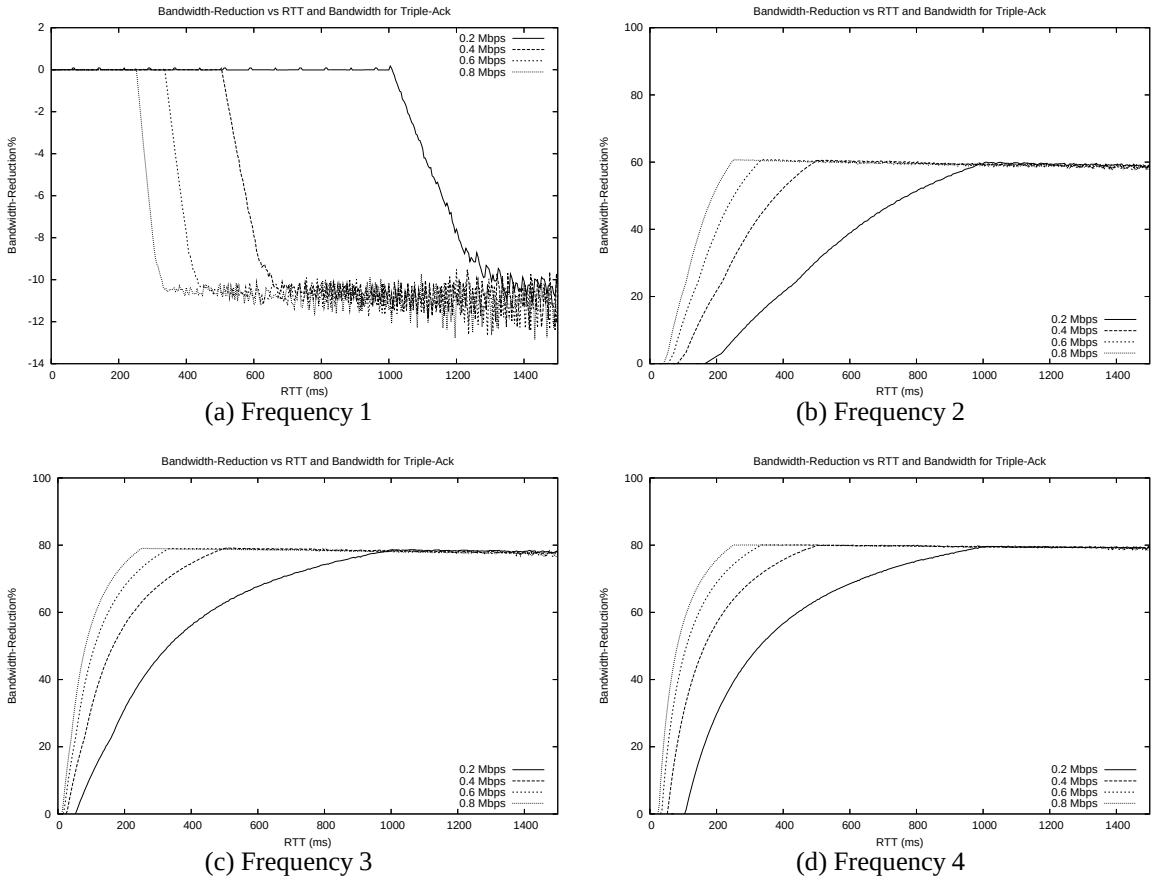
Figure H.10: Goodput reduction with frequency 6, window caps 45-65

I LOWER BANDWIDTH RESULTS

This section contains graphs for the lower bottleneck-bandwidth case. Graphs are shown for frequencies of 1-8 and for bandwidth and goodput reduction. Experiments were run using the symmetric setup for the following ranges.

Bandwidth 0.2 to 0.8 Mbps, in increments of 0.2 Mbps

Latency 1 to 1100 ms per direction for each link, in increments of 4 ms



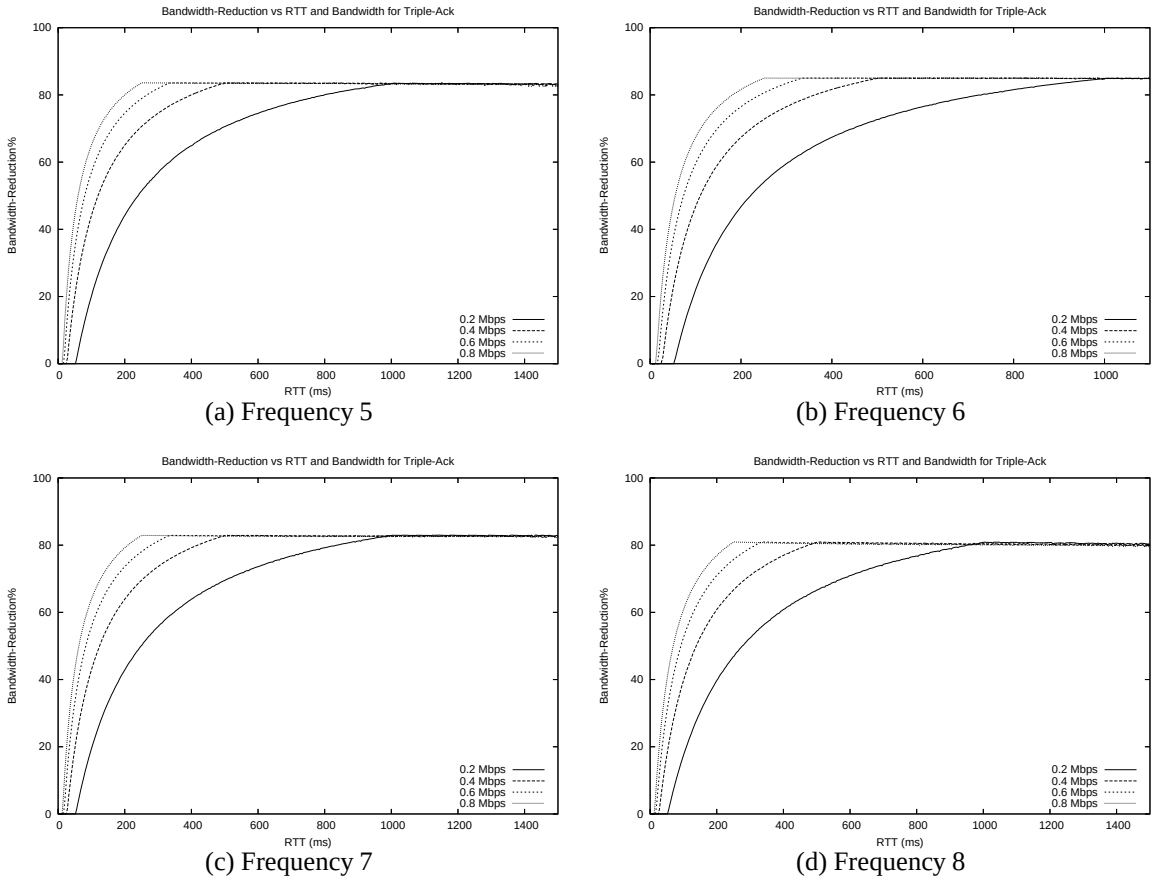
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure I.1: Bandwidth reduction for lower bandwidths with freq 1-4 and wincap 20



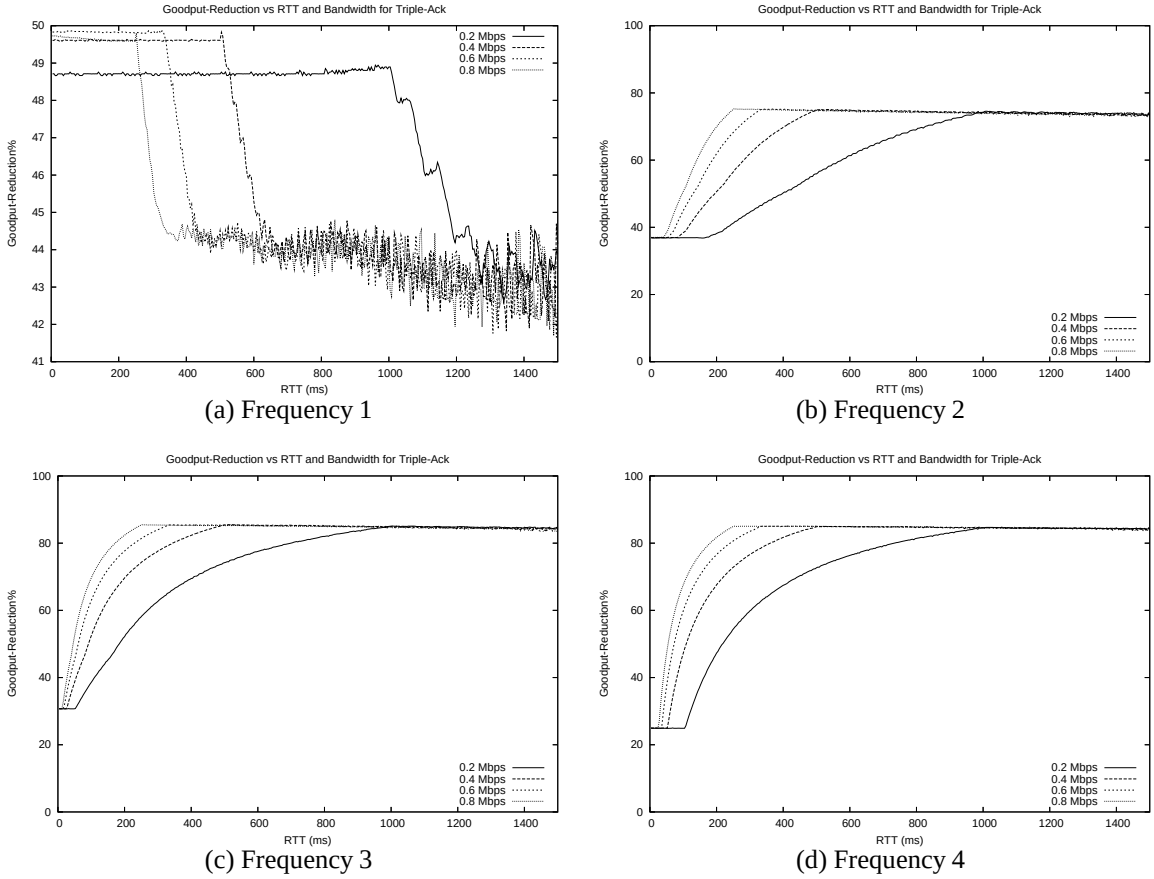
(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure I.2: Bandwidth reduction for lower bandwidths with with 5-8 and wincap 20



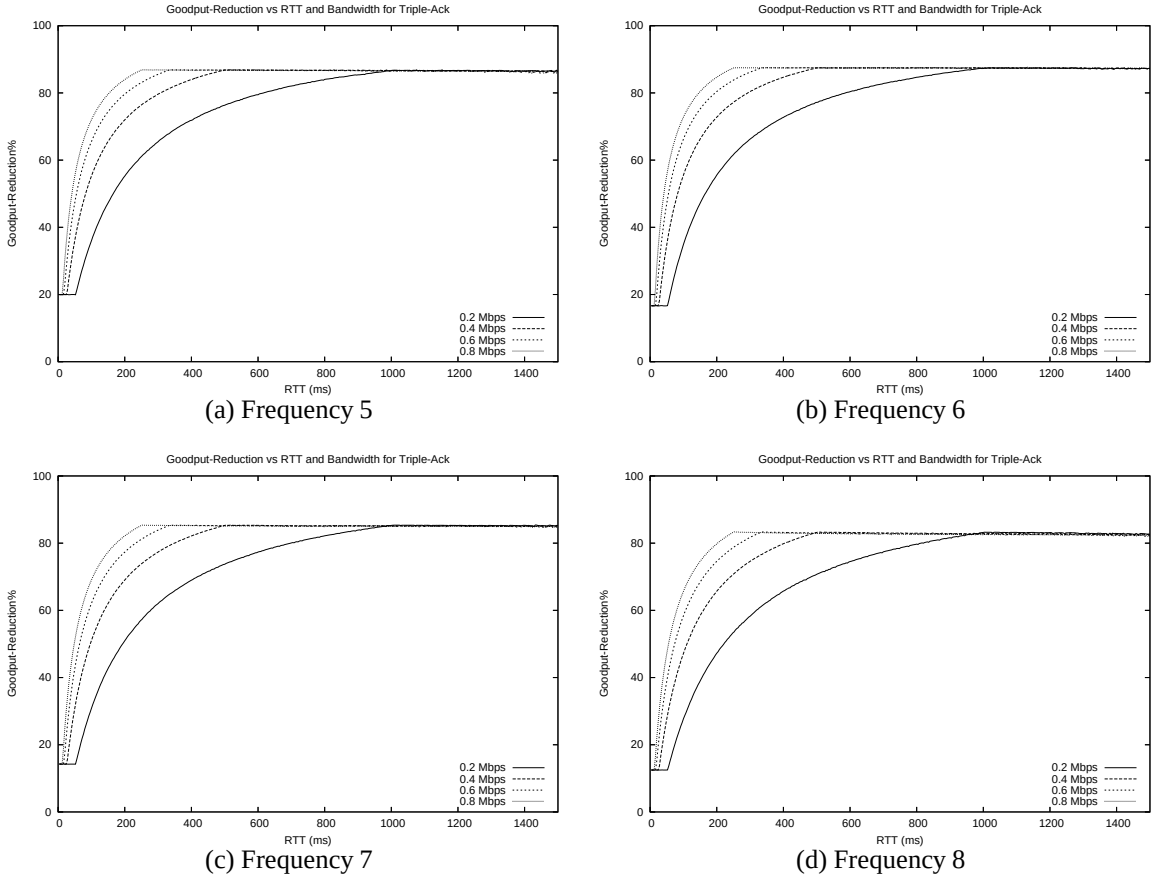
(a) Frequency 1

(b) Frequency 2

(c) Frequency 3

(d) Frequency 4

Figure I.3: Goodput reduction for lower bandwidths with with 1-4 and wincap 20



(a) Frequency 5

(b) Frequency 6

(c) Frequency 7

(d) Frequency 8

Figure I.4: Goodput reduction for lower bandwidths with with 5-8 and wincap 20

REFERENCES

[1] Yehuda Afek, Yishay Mansour, and Zvi Ostfeld. Phantom: A simple and effective flow control scheme. In *SIGCOMM '96: Conference proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 169–182. ACM Press, 1996.

[2] Mark Allman, Sally Floyd, and Craig Partridge. RFC 3390: Increasing TCP's initial window, October 2002.

[3] Mark Allman, Vern Paxson, and W. Richard Stevens. RFC 2581: TCP congestion control, April 1999.

[4] Sanjeewa Athuraliya, Steven H. Low, and David E. Lapsley. Random early marking. In *QofIS '00: Proceedings of the First COST 263 International Workshop on Quality of Future Internet Services*, pages 43–54. Springer-Verlag, 2000.

[5] James Aweya, Michel Ouellette, and Delfin Y. Montuno. Weighted proportional window control of TCP traffic. *Int. J. Netw. Manag.*, 11(4):213–242, 2001.

[6] James Aweya, Michel Ouellette, Delfin Y. Montuno, and Zhonghui Yao. Enhancing network performance with TCP rate control. In *Proc. of the IEEE Global Telecommunications Conference (GLOBECOM)*, volume 3, pages 1712–1718, San Francisco, CA, 2000.

[7] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *SIGCOMM*, pages 24–35, 1994.

[8] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[9] Kevin Fall and Sally Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[10] Sally Floyd. TCP and successive fast retransmits. Available at ftp://ftp.ee.lbl.gov/-papers/fastretrans.ps, May 1995.

[11] Sally Floyd. A report on some recent developments in TCP congestion control. *IEEE Communications Magazine*, 39(4):84–90, April 2001.

[12] Sally Floyd, Tom Henderson, and Andrei Gurtov. RFC 3782: The NewReno modification to TCP's fast recovery algorithm, April 2004.

[13] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[14] Sandvine Incorporated. Meeting the challenge of today's evasive P2P traffic: Service provider strategies for managing P2P filesharing. Available at http://www.-sandvine.com/general/getfile.asp?FILEID=16, September 2004.

[15] Van Jacobson, Robert Braden, and David Borman. RFC 1323: TCP extensions for high performance, May 1993.

[16] Van Jacobson and Michael J. Karels. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[17] Shrikrishna Karandikar, Shivkumar Kalyanaraman, Prasad Bagal, and Bob Packer. TCP rate control. *SIGCOMM Comput. Commun. Rev.*, 30(1):45–58, 2000.

[18] Allison Mankin and K.K. Ramakrishnan. RFC 1254: Gateway congestion control survey, August 1991.

[19] Jim Martin, Arne A. Nilsson, and Injong Rhee. Delay-based congestion avoidance for TCP. *IEEE/ACM Trans. Netw.*, 11(3):356–369, 2003.

[20] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. RFC 2018: TCP selective acknowledgement options, October 1996.

[21] Alberto Medina, Mark Allman, and Sally Floyd. Measuring the evolution of transport protocols in the Internet. *SIGCOMM Comput. Commun. Rev.*, 35(2):37–52, April 2005.

[22] netfilter. The netfilter/ip tables project. Available at http://netfilter.org/.

[23] Vern Paxson and Mark Allman. RFC 2988: Computing TCP's retransmission timer, November 2000.

[24] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, second edition, 2000.

[25] Kedarnath Poduri and Kathleen Nichols. RFC 2415: Simulation studies of increased initial TCP window size, September 1998.

[26] Jon Postel. RFC 792: Internet control message protocol, September 1981.

[27] Jon Postel. RFC 793: Transmission Control Protocol, September 1981.

[28] K.K. Ramakrishnan, Sally Floyd, and David L. Black. RFC 1254: The addition of explicit congestion notification (ECN) to IP, September 2001.

[29] Rogers. Rogers.com. Available at http://rogers.com/.

[30] Sandvine Incorporated. Sandvine incorporated: Welcome! Available at http://-sandvine.com/.

[31] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *Computer Communication Review*, 29(5), 1999.

[32] W. Richard Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, 1994.

[33] W. Richard Stevens. RFC 2001" TCP slow start, congestion avoidance, fast retransmit and fast recovery algorithms, January 1997.

[34] VINT Group. UCB/LBNL/VINT Network Simulator (ns) - version 2. Available at http://www.isi.edu/nsnam/ns.

[35] Cheng-Shong Wu, Ming-Hsien Hsu, and Kim-Joan Chen. Traffic shaping for TCP networks: TCP leaky bucket. In *Proc. of the IEEE Conference on Computers, Communications, Control and Power Engineering (TENCON)*, volume 2, pages 809–812, October 2002.