Delay-based Rate Control Algorithms for the Real-time Video Transmission Application

By

Kun Meng

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

In this thesis, a new delay-based rate-control algorithm is proposed for a real-time video conferencing application. By implementing the new algorithm, the sender detects the current network condition based on the one-way delay variation trend of packets transmitted. Then the sender chooses the appropriate sending rate based on the current network condition.

The new delay-based rate-control algorithm is implemented in a video conferencing application. The application uses a simple rate-control algorithm to control the sending rate. The sender detects the network condition based on the packet loss and the packet's out-of-order feedback.

In this thesis, the performances of both the new algorithm and an old one are tested on a test bed. Based on the test results, the new algorithm improves the performance of the application significantly compared with the old algorithm.

Acknowledgements

I would like to thank my supervisor Professor Thomas Kunz for his great guidance and support of the thesis. With his help, I searched papers, determined research topics, collected experimental data and finally finished the thesis. He taught me how to do research, how to write an academic thesis as well as some valuable life lessons.

I would also like to thank Secure City Solutions for providing the source code and the guide of their video conferencing application on which my thesis worked.

I would like to thank Mike, a staff member of Secure City Solutions, and Cyrus, an undergraduate student of Carleton University; with their help, I learned and understood the operating mechanism of the video application used in the experiment. With Cyrus help, I set up the test bed in the thesis.

Finally, I would like to thank my parents for their love, understanding, encouragement and comprehensive support. Without their help, I would not be able to overcome difficulties and achieve my accomplishments today.

Table of Contents

List of Tables

List of Figures

List of Acronyms

BTC	Bulk Transfer Capacity
DCCP	Datagram Congestion Control Protocol
ECN	Explicit Congestion Notification
FEC	Forward Error Correction
FPS	Frame Per Second
GCC	Google Congestion Control
IETF	Internet Engineering Task Force
MPU	Minimum Packet Unit
NADA	Network Assisted Dynamic Adaptation
OWD	One Way Delay
PCT	Pairwise Comparison Test
PDT	Pairwise Difference Test
POV	Point of View
QoE	Quality of Experience
QoS	Quality of Service
QP	Quantization Parameter
RFC	Request for Comments
RTP	Real-time Transport Protocol
RTT	Round Trip Time
RTCP	RTP Control Protocol
SLoPS	Self Loading Periodic Streams

TBF	Token Bucket Filter
TCP	Transmission Control Protocol
TFRC	TCP Friendly Rate Control
TOPP	Trains of Packet Pairs
UDP	User Datagram Protocol

1 Chapter: Introduction

1.1 Research Objective

There are many kinds of video streaming applications that run in the global network. Video data capacity often exceeds network bandwidth. Therefore, codecs can be used to compress the video data to transmit the data in the correct form (at an appropriate frame rate, data rate, resolution, etc.).

Consequently, we will experience some common issues when we are sending the video stream in the network. The network conditions can vary. When network bandwidth is high and stable, we can send the video stream at a relatively high rate, with low compression level, resulting in higher quality signals being derived to users. When network bandwidth is smaller and variable, transmission data rates must be decreased. This decreases the video quality and requires more compression. A streaming video application must send data at an appropriate rate based on the current network condition. The application must decide the appropriate sending rate automatically and adapt the streaming rate based on network conditions. In the worst case, the application cannot stream video at all because of congestion in the network, and users then are forced to change the sending rate manually. We want to build an automatic feedback cycle in which the receiver sends to the sender feedback packets that tell the sender the current network condition. Based on such feedback packets, the sender determines the appropriate sending rate based on the current network conditions.

There are two kinds of video applications: video-on-demand applications and real-time applications. Video-on-demand applications, such as YouTube, are time insensitive, that is, they can tolerate some network delays because users can buffer contents at any time and then watch a stream. Video providers upload video content to a server for users download them. Non-real-time applications often use the TCP protocol to transmit data. TCP retransmits data when packets are lost or corrupted. TCP is good for transmitting time-insensitive video streams. But it is less suited when delay of packets cannot be tolerated. For example, real-time applications such as video conferencing applications do not use the TCP protocol. TCP has its own mechanism for congestion control, adjusting packet-sending rate based on current network conditions. The TCP congestion control mechanism is designed to allow several applications to share network resources fairly.

TCP congestion-control mechanism guarantees that the sending rate is appropriate under different network conditions. When network conditions change, the applications use the TCP protocol to adjust their sending rate automatically.

Real-time applications comprise video conferencing, video chat and so on. They cannot tolerate delay in packet transmission, but they can tolerate moderate levels of packet loss. These applications often use UDP to transmit data. UDP has no congestion-control mechanism. Real-time video stream applications need to choose an appropriate transmission rate under different network conditions. If network bandwidth is limited, sending rate must be decreased to avoid network congestion. If the sending rate is too low, the application is not using network resources efficiently. When there are both TCP

and UDP applications running in the network, we need to allocate the network resources fairly to each application. So when we run a real-time video streaming application, the application itself needs some algorithms to help it send the data at an appropriate rate under different network conditions. When the network conditions change, the application needs to adapt to that change.

This thesis addresses the problem of how to choose the appropriate sending rate of a video stream for different kinds of network conditions, when running real-time video applications. As UDP does not have a congestion-control mechanism, the work must be done by the applications themselves. I summarize our approach for this and review the related work in later parts of the thesis.

As a secondary objective, our congestion control algorithm must work well with other TCP applications in the network. Network resources need to be allocated fairly among different applications. For example, if an UDP-based video conferencing application and a TCP-based file transmission application are running concurrently in the network, network bandwidth should be allocated fairly between two applications, without causing network congestion.

1.2 Contribution of the Thesis

The contributions of this thesis comprise the following:

- Delay-based feedback: I use packet delay as a new feedback for the real-time video transmission applications to determine current network conditions. A lot of

video transmission applications only use packet loss to determine network conditions. When there is packet loss detected in the network, the applications decide that the current network is congested and decrease their sending rate. In some cases, only packet loss cannot reflect the current network condition accurately. In this thesis, I add a new feedback called packet delay to the rate control algorithm. I also implement PCT and PDT test to measure the trend of one-way delay variation of packets. The applications determine current network conditions based on both packet loss and packet delay. Based on the test results, the new rate control algorithm can detect changes of current network conditions more accurately and quickly.

- Rate control algorithm: I successfully add a delay-based algorithm to a real-time video transmission application. I test both the new algorithm and the original algorithm of the application on the test bed (I set up a test bed to simulate a real network environment). The new algorithm changes its states based on packet loss and packet delay measurement. The final sending rate is determined based on states of the algorithm. Based on the test results, the new algorithm improves the application's performance compared with the old one, as the new algorithm enables the application to control the sending rate in different network conditions more accurately and effectively. The new algorithm also helps the application to share the network bandwidth more fairly with a concurrent TCP flow. The application can send the video packets at an appropriate sending rate when there is another TCP flow in the network.

1.3 Organization of the Thesis

In Chapter 2, related work on congestion-control algorithms with several different approaches is discussed. Chapter 3 introduces the test bed used for the whole experiment. In Chapter 4, I describe how the application runs on the test bed and show several drawbacks of running the current application on the test bed. In Chapter 5, I implement my approach in the application. Based on the experimental results, my new algorithm improves the performance of the application in several ways. Chapter 6 concludes the thesis and makes recommendations for future work.

2 Chapter: Related Work

Some papers discuss how to implement rate-control algorithms in real-time video applications, and some of them introduce delay-based control algorithms. A video application called OmniSHIELD is used to implement my rate-control algorithm in this thesis. OmniSHIELD is a general video-conferencing application with a server and clients. The server helps to establish and disconnect the connection between clients. The clients use H.264 as a codec and stream video between the sender and the receiver at different sending rates. The application uses PacketLoss and PacketOutOfOrder statistics to determine the network condition. Based on the network condition, the application controls the sending rate by using a simple rate control algorithm. The structure and the performance of OmniSHIELD are discussed in detail in Chapter 4. As the application uses only PacketLoss and PacketOutOfOrder statistics to determine the network status, it is good to add a delay parameter to measure the network condition. In Section 2.1, I introduce the problems associated with transmitting the video stream in real-time applications. In Section 2.2, I discuss the network performance metrics. In Section 2.3, RTP and RTCP protocol are introduced. In Section 2.4, I introduce papers that discuss delay-based congestion-control algorithms. A summary of work that is related to this study is provided in Section 2.5.

2.1 Issues Associated with Sending a Real-Time Video Stream

Regarding transmitting video streams for real-time applications, there are two issues. First, we need to use some kind of a codec to help us transmit video data. Therefore,

making the codec adaptive, that is, enabling the codec to change the sending rate to different levels, is one of the problems.

In [1], the authors do not only consider the data rate of a video stream, but they also quantify the video quality using Quality of Experience (QoE). QoE is a very important concept in video transmission; it comes from the Quality of Service (QoS). Many video metrics influence the quality of a video. From the users' side, two video streams running at the same sending rate in one network may have different qualities. The frame rate, resolution, compression level and many other characteristics all influence the QoE of the video stream. In the paper, the authors run several tests to show the influence of different video characteristics on the video quality experienced by the users in different network conditions.

[2] discusses how to do rate control in real-time H.264 video applications. The video-sending rate adapts to the network status using different kinds of feedback information. Two rate control schemes are present in the paper: The first one uses available bandwidth feedback to help H.264 in selecting a proper Quantization Parameter (QP), which determines the video compression rate. The second one uses packet-loss feedback. The goal of the paper is to find the right QP value in different network conditions to encode the video packet as a trade-off between video rate and video quality.

[3] introduces the On2 video codec. This is a commercial video codec, which is used in Skype. It is a typical adaptive video codec that supports real-time video coding. It can

adapt its rate to different network conditions by changing the video's resolution, frame rate and bit rate. When we send video data through On2, it encodes a video with different parameters. For example, when the network is in an optimal condition, On2 encodes a video with a high bit rate, video resolution and frame rate.

In summary, advanced codecs of video applications can adapt to different levels of video sending rate by changing the compression rate of the video data. In H.264, the codec changes QP to get different levels of compression rate and data rate. For users, the video bit rate is not the single parameter influencing the quality of video, but there are other parameters such as the frame rate and resolution. All these parameters influence the QoE of a video. My thesis does not focus on the codec part. We need to only know that when we use an adaptive codec (H.264 or On2) to encode a video stream, we can get different levels of video quality by using different levels of the sending rate. My thesis focuses on finding a rate-control algorithm that can adapt the video sending rate to current network conditions.

2.2 Network Performance Metrics

There are many different network performance metrics we can get. [4] discusses about several different network performance metrics including packet loss, delay, delay variation (jitter), raw and restricted capacity (bandwidth). These metrics can reflect the network performance. [4] provides two different kinds of point of view (POV).

- Network Characterization: This point of view (POV) looks inward toward the network. It describes the network performance from the network's side.

- Application Performance Estimation: This POV looks outward, toward the users. It describes the network performance from the application's and the user's side.

For the packet loss, setting the timeout value of packets is important. The timeout value is used to differentiate the true packet loss and the long delay packet. Another issue is that we always treat errored packet as lost packet. If there are significant packet losses detected, the network performance is bad.

For the delay and delay variation (jitter), the network performance is bad if the delay of packets increases or the delay variation has an increasing trend. The one-way delay of packets consists of the transmission delay, queuing delay and network jitter. Only the queuing delay reflects whether the current network is congested. When the network is congested, packets are queued in the buffer of the routers. The queuing delay will increase, which causes the increase of one-way delay of packets. When discussing the measurement of delay trends, the clock synchronization between the sender and the receiver must be considered. If we only use the measurement value of one-way delay of packets to determine the network conditions, the clock must be synchronized precisely. So the measurement value is the real one-way delay time of packets after packets are transmitted from the sender to the receiver. If we consider the delay variation of packet pairs, the clock synchronization is not necessary. If the delay variation of packet pairs shows an increasing trend, the network is considered congested. However, the clock drift cannot be tolerated. If the clock drift happens at the sender or the receiver, the delay variation cannot reflect the network conditions. For example, if the clock at the receiver

continuously drifts, the delay variation can show a continuously increasing or decreasing trend although the network conditions are not changed.

Network bandwidth can also reflect the network performance. There are different kinds of definitions of the network bandwidth, which are discussed in [5].

- Network bandwidth or raw bandwidth: This is the total available data rate the network can transmit in unit time.
- Available bandwidth: This is the data capacity that is not used in the network in unit time. Both the network raw bandwidth and network load determine the available bandwidth.
- TCP throughput: [6] defines Bulk-Transfer-Capacity (BTC) calculation metrics. BTC represents the largest throughput a TCP link can get. This link must have all TCP congestion control algorithms. This is different from the available bandwidth.

For the network operators, they need to focus on the network throughput. This can give them information of the network quality. For the users, they focus more on the available bandwidth estimation that means the bandwidth they can use now.

There are many network bandwidth estimation tools. [7] describes a widely used bandwidth estimation method called Self-Loading Periodic Streams (SLoPS). The router buffers the passed packets in the queue. If the packet arrival rate is larger than the packet leave rate, packets will queue in the buffer and increase the sending delay. Based on this,

when we send packets to the network, if the sending rate is larger than the available bandwidth, the delay will increase. So the available bandwidth can be replaced by the critical point of the sending rate that nearly causes the increasing delay. Based on SLoPS, there are different kinds of available bandwidth estimation tools.

[8] describes an available bandwidth estimation tool called PathLoad. PathLoad tool is based on SLoPS. The sender periodically sends UDP data packets sequence. The sender sends same length and fixed interval test packets as a packet train at rate R. On receiver side we can know the one-way delay (OWD) of the received test packets. If R is larger than the network available bandwidth, the test packets will cause the congestion of the network. The delay will have an obvious increasing trend. The sender gets feedback of this information, and it can change the sending rate R. We repeat this measurement for many times, until the delay is stable, which means R is nearly equal to the available bandwidth.

[9] describes an available bandwidth estimation tool called PathChirp. PathChirp tool is based on Trains of Packet Pairs (TOPP). TOPP is very similar to SLoPS. TOPP estimates available bandwidth by sending a series of test packets sequences. The time intervals of test packets sequences are decreased in exponent. If the sending rate is larger than the available bandwidth, the packet sequence will queue in the network. Then at the receiver side, the interval between two test sequences will increase. If the sending rate is smaller than the available bandwidth, the interval between two test sequences will not change. At

receiver side, based on the information that the intervals between two packet sequences begin to increase, we can calculate the available bandwidth.

[10] tests the performance of PathLoad and PathChirp bandwidth estimation tools. The conclusions show that the PathLoad (SIoPS based) needs to send test packets several times until it gets critical rate. The sender needs to get feedback from receiver. So PathLoad needs more time to get the final result and uses considerable bandwidth of the network although it has accurate measurement results. The PathChirp (TOPP based) does not need to get feedback. It uses less time to get results. But the size of the test packet is large, which causes significant bandwidth usage of the network.

[11] describes a new algorithm to measure the available bandwidth. By using Round Trip Time (RTT), the test time is much shorter and the accuracy of the measurement can also be increased. In [12], the authors work on FEAT (Fish-Eye Available-bandwidth Tool). It has a new packet train structure. Comparing to the stream in PathChirp, a fisheye stream has more sampling frequency around the center of the focus region. Therefore, it is faster to get the measurement result. In [13], the authors work on a tool called PathPair. It replaces OWD (One-way Delay) in pathLoad by AOWD (Accurate One-way Delay), which is an optimization of OWD. The OWD testing is often influenced by the real environment and jitter. PathPair optimizes the OWD results and make it more accurate and efficient to measure the available bandwidth. In [14], the authors combine the Variable Packet Size Probing (VPS) model and Self-Loading Periodic Streams (SLoPS)

model to propose a bandwidth estimation tool called PathQuick. The packet train of PathQuick has short length and covers wide probing range.

2.3 RTP and RTCP Protocol

[15] describes RTP (Real-time Transport Protocol). RTP is widely used for real-time audio and video transmission applications. RTP multiplexes several real-time data streams into one UDP data packet. It has several features to benefit real-time multimedia transmission. The RTP header has the following format:

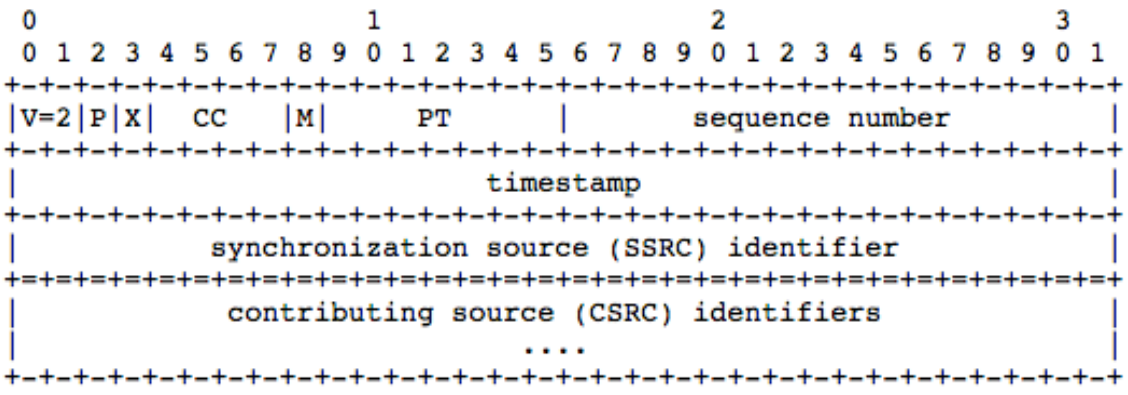


Figure 2.1: RTP Header [15]

PT (Payload type) indicates which encoding algorithm the data uses. If the sender knows the network metrics such as delay, jitter, bandwidth, congestion or other things, the encoding algorithm can be changed to adapt the network condition. When the network bandwidth is high, applications can use better encoding algorithms. The PT field tells receiver which encoding algorithm the data uses, so it is available to change the encoding algorithm dynamically. The sequence number is a counter. Every time a RTP packet is

sent, the counter adds by one. Every data packet has a sequence number in RTP. The receiver can determine whether there is any packet loss by analyzing the sequence number. The timestamp value indicates the time interval between the current packet and the first packet. This value can be used to reduce the influence of the network jitter.

RTCP (Real-time Transport Control Protocol) is designed to cooperate with RTP and provide reliable streaming data transmission. It also provides feedback information of the network condition. We can use these feedbacks to do congestion control. [15] defines five kinds of RTCP packets. The SR (Sender Report) and RR (Receiver Report) provide network performance metrics such as packet loss, packet delay of RTP packets. Applications can change sending rate based on those metrics to avoid network congestion. There are other three kinds of RTCP packets. They provide other functions such as synchronization, media source description and disconnection notice.

[16] describes a new kind of RTCP packets called minimal compound RTCP feedback packets. It only contains mandatory information. When the network bandwidth is limited, applications can use this kind of RTCP packets to reduce the network usage. [16] also defines three feedback modes. The first one is called immediate feedback mode. In this mode, we do not consider the bandwidth usage of the feedback information. The feedback report is sent immediately after generated. The second one is called early RTCP mode. In this mode, feedback reports are no longer sent immediately after generated. But feedback can still be given sufficiently often so that it allows the sender to adapt the sending rate relatively quickly to increase the media quality. The third one is called

regular RTCP mode. In this mode, applications do not trace network changes rapidly. We need to send feedback packets in a comparative large time interval.

[17] defines reduced-size RTCP feedback packets. It just sends feedback without other parts of a compound RTCP packet. Then it uses less bandwidth and takes less delay to transmit this packet through the network. [17] indicates benefits to use reduced-size RTCP feedback packets. A reduced-size RTCP packet can become at least 70-80 bytes smaller than the compound packet. For limited network bandwidth, the reduced-size RTCP packet is less likely to be dropped. It also takes less time to transmit a reduced-size RTCP packet. For high network bandwidth, the benefit is limited, but exist. The reduced-size RTCP packet takes less bandwidth and is less complexity.

[18] and [19] describe how the video sending rate can be controlled using TCP-friendly rate-control protocols when sending a real-time video stream. When we send UDP or RTP packets via TCP-friendly rate-control protocols (like TFRC or DCCP), these protocols use TCP congestion-control rules to control the video sending rate. As I discussed in Chapter 1, when we send data using the TCP protocol, we do not deal with issues of controlling the data-sending rate. TCP congestion-control algorithms guarantee that the sending rate is appropriate under the current network condition.

2.4 Congestion-Control Algorithms Used in Real-Time Video Applications

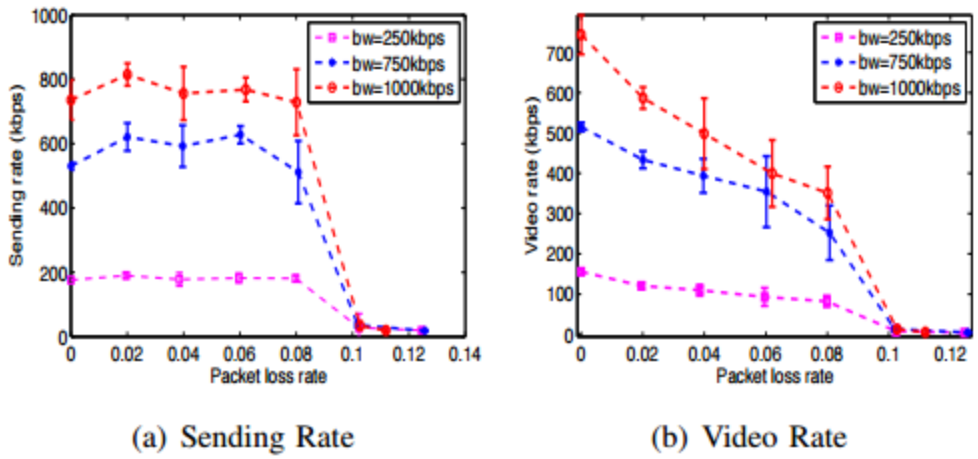
There are many video applications around the world, of which the most famous one is Skype. Skype is a commercial software; it is not open source. So it is hard for people to

know in detail how the software works. Some people also implement test beds to evaluate its behavior and congestion-control mechanism. They treat Skype's client and server as a black box and test their behavior in various network conditions. In [20], the authors test Skype's performance and draw the following conclusions:

1. Skype can adapt the sending rate to various network bandwidth.
2. Skype uses Forward Error Correction to reduce the packet loss. This technology is useful in real-time video or audio applications that use UDP.
3. Skype is TCP-friendly and can compete fairly for bandwidth resources with other TCP flows in the same network.

[21] discusses how congestion control algorithms of Skype work in the network. The authors run a lot of experiments and establish a number of observations. First, Skype adapts the video through varying frame rate, frame quality and video resolution. Second, Skype also employs an adaptive FEC rate related to packet loss. Third, the sending rate adapt to the available bandwidth after a long time, but does not exploit all available bandwidth. We should also note that Skype is not restricted to RTCP feedback.

[22] discusses how Skype works to adapt the video quality dynamically in different network conditions. The authors run a lot of experiments on their test bed. Figure 2.2 shows the impact of packet loss in Skype.



(a) Sending Rate (b) Video Rate

Figure 2.2: Impact of Packet Loss in Skype

We can see that in Skype, when packet loss is less than 10%, it will not influence the sending rate and video rate. When packet loss is large than 10%, Skype works in another state, where it reduces the sending rate and the video rate to very low levels because the packet loss is too high for Skype to transmit any packets. When packet loss is less than 10%, Skype's sending rate almost remains constant while its video rate drops almost linearly with packet loss rate. The gap between sending rate and video rate increases when the packet loss rate increases. When packet loss rate increases, Skype reduces its video rate and allocates more bandwidth to FEC packets.

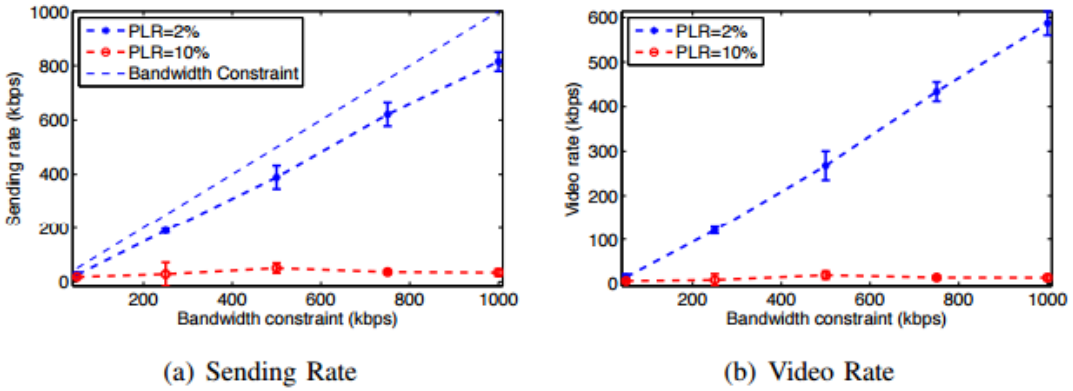


Figure 2.3: Impact of Available Bandwidth in Skype

Figure 2.3 shows the impact of available bandwidth in Skype. When packet loss is less than 10%, Skype is in NORMAL state and increases its sending rate proportionally as the available bandwidth increases. Similar trend can be detected in the video rate. The video rate in NORMAL state changes linearly with available bandwidth. Skype can closely keep track of the available network bandwidth without causing excessive congestion. But it is also noticed that Skype does not fully utilize all available bandwidth.

In the last part of the paper, the authors also run experiments to show that algorithms running in Skype is TCP-friendly. When Skype runs with TCP applications in the network, the bandwidth is allocated fairly among different applications.

Another important real-time video application is called WebRTC, and IETF and Google support it. It is used in the Chrome web browser to support real-time video conferencing. It is open source, and is described in detail in some IETF drafts. Many other people also test its behavior under a real network or the test bed. It has its own congestion-control

algorithm called the Google Congestion Control (GCC). The main feature of the GCC is that it implements a loss-based congestion control mechanism at the sender's side and a delay-based congestion control mechanism at the receiver's side. They work together to adapt the sending rate of the video frame to the current network condition. The receiver collects network performance metrics: packet loss and packet delay. The receiver sends feedback reports to the sender only including packet loss. The sender determines the rate A_s based on the packet loss at the sender side. At the receiver side, the receiver determines rate A_r based on the packet delay and sends another kind of feedback reports to the sender including rate A_r. Finally, the sender decides the final sending rate based on the measurement values of A_s and A_r. In this way, we say that the loss-based controller at the sender side and the delay-based controller at the receiver side work together to calculate the final sending rate.

[23] and [24] describe how GCC works. There is a loss-based congestion controller at the sender side. GCC uses RTP to transmit packets. So it is easy for the sender to get feedback about packet loss with RTCP reports. After getting the RTCP report including the packet loss rate $f_l(t)$, the sender uses this parameter to compute the sending rate $A_s(t)$ as follows:

$$A_s(t_k) = \begin{cases} \max\{X(t_k), A_s(t_{k-1})(1 - 0.5 f_l(t_k))\} & f_l(t_k) > 0.1 \\ 1.05(A_s(t_{k-1}) + 1\text{kbps}) & f_l(t_k) < 0.02 \\ A_s(t_{k-1}) & \text{otherwise} \end{cases}$$

When the rate of packet loss is larger than 10%, the network is overused, which causes the GCC to decrease the sending rate to $A_s(t)*(1 - 0.5f_l(t))$. Nevertheless, the lower bound of the sending rate as derived from TFRC [18] is X(t). TCP Friendly Rate Control (TFRC) is used in the RTP packet transmission to calculate the sending rate of UDP packets, which uses an approach similar to TCP's sending rate calculation. Using TFRC, the sender can send UDP packets that fairly compete for the bandwidth with other TCP flows in the network.

When the packet loss rate is smaller than 2%, the network is underused; the sending rate is, therefore, increased to $1.05*(A_s(t) + 1)$. When the packet-loss rate is between 2 and 10%, the sending rate is not changed.

The GCC also uses a delay-based controller at the receiver's side. It measures different values of the one-way delay of packets. If the value is increased above a threshold in past several tests, the application assumes that the network is overused. The receiver, therefore, sends this feedback to sender, which then decreases the sending rate.

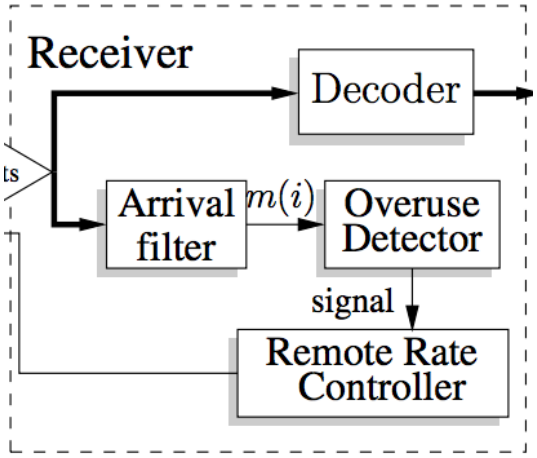


Figure 2.4: Structure of the Delay-based Congestion Controller in the GCC [23]

Figure 2.4 shows the structure of the delay-based congestion controller in the GCC. There are three parts in the delay-based controller, namely the over-use detector, overuse detector and remote rate controller. The receiver measures the one-way delay variation $d(t_i)$, which is affected by three factors: the transmission time, queuing time and network jitter. Only the queuing time reflects whether the current network is congested. The arrival-time filter separates the queuing time variation $m(t_i)$ from the one-way delay variation $d(t_i)$.

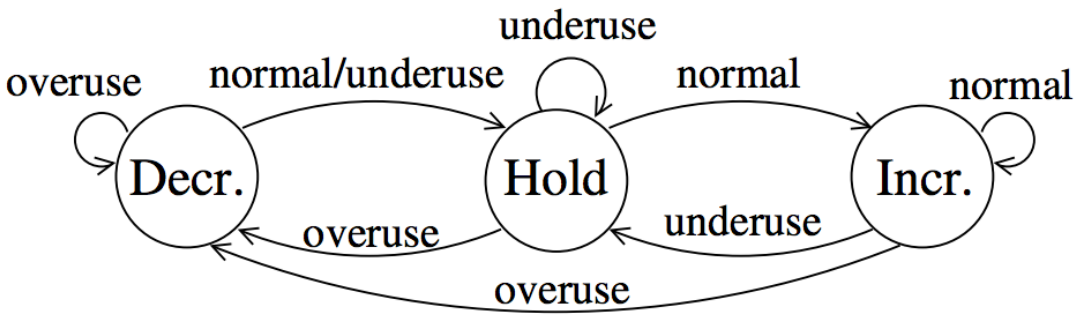


Figure 2.5: Network State Diagram in the GCC [24]

The overuse detector produces a signal to drive the diagram shown in Figure 2.5. The networked state is determined based on $m(t_i)$. If values of $m(t_i)$ show an increasing trend for a certain amount of time, the network is considered congested. So the overuse detector generates an "overuse" signal. If values of $m(t_i)$ show a decreasing trend for a certain period of time, the network is considered to be underused, and the "underuse" signal is generated. If values of $m(t_i)$ show neither an increasing nor a decreasing trend, the network is considered neither overused nor underused; the detector generates a "normal" signal. Based on signals generated, the network state varies among three given states ("Decrease", "Increase" and "Hold").

The function of the remote rate controller is to compute the sending rate measured by the delay-based controller at the receiver's side (A_r). The following is the measurement formula.

$$A_r(t_i) = \begin{cases} \eta A_r(t_{i-1}) & \text{Increase} \\ \alpha R(t_i) & \text{Decrease} \\ A(t_{i-1}) & \text{Hold} \end{cases}$$

When the state is "Increase", the rate increases by $\eta \in [1.005, 1.3]$. When the state is "Decrease", the rate decreases by $\alpha \in [0.8, 0.95]$. R(t) is the receiving rate measured during the last 500 ms at the receiver's side.

Every time there is A_r calculated from the receiver's side, the receiver sends a report including A_r back to the sender. When the sender gets the sending rate A_s by the loss-based controller at the sender's side and A_r by the delay-based controller at the receiver's side, it uses the smaller value of the two rates as the final sending rate.

The preceding are the loss-based and delay-based congestion control algorithms used in the GCC. The GCC also uses FEC (Forward Error Correction) [25] to reduce packet loss. The FEC corrects packet loss at the lower layers, so nothing happens at the transport layer.

In [26] and [27], there is another congestion control algorithm called the Network Assisted Dynamic Adaptation (NADA) congestion-control algorithm. This algorithm also

uses packet loss and delay as feedback information. It also changes the sending rate based on both implicit and explicit congestion notifications (ECN). The latter is obtained from the node itself. ECN works at the network layer. When a packet goes through the network, if congestion occurs and the packet is delayed in the router, the packet will set an ECN flag in the IP header to let the receiver know that there is congestion in the current network.

In [26], the authors describe different ways of measuring the delay of packets. The NADA obtains the one-way delay of every packet by adding a time stamp to packets in the sender's side. Then the one-way delay of the packets will be measured using the arriving time and sending time. However, there are some drawbacks to such a measurement. The most important thing is that the time clocks in both the sender and the receiver must be synchronized precisely. As the delay time of packets might be at the millisecond level, the synchronization accuracy must be at millisecond level as well, which is difficult to implement.

The GCC does not measure the one-way delay of every packet. Instead, it measures the one-way delay variation $d(t_i) = (t_i - t_{i-1}) - (T_i - T_{i-1})$, where T_i is the sending time stamp of the i-th video frame, and t_i is the receiving time stamp. Doing this makes clock synchronization unnecessary.

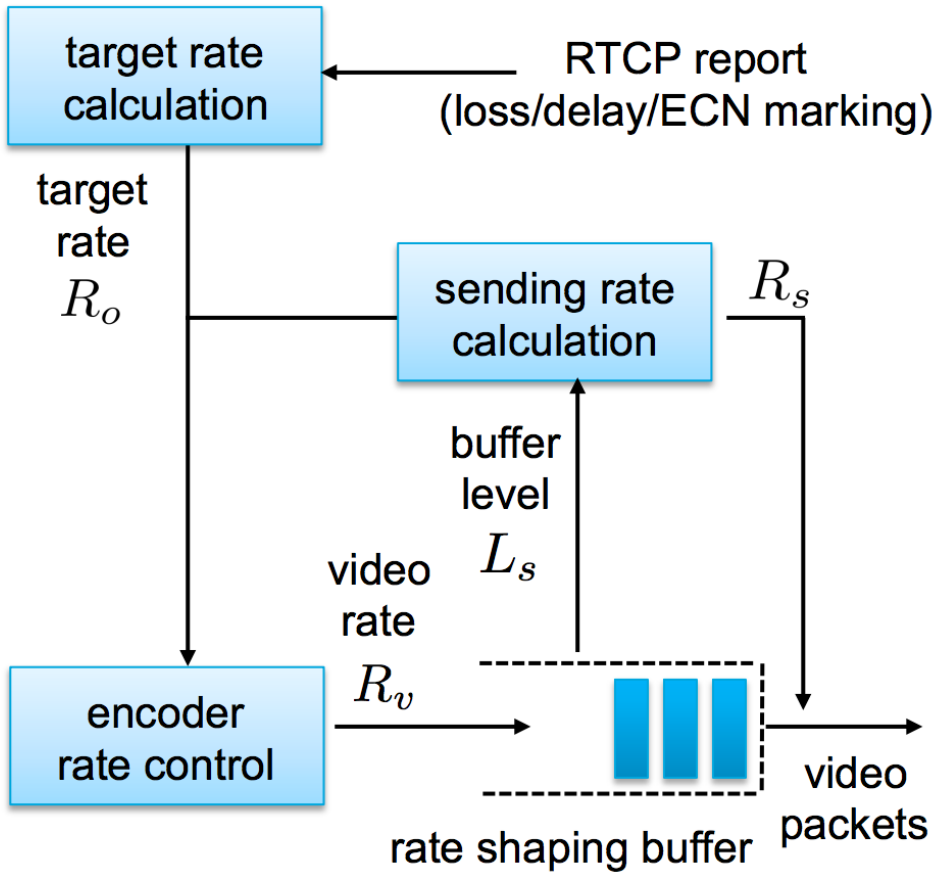


Figure 2.6: The NADA Algorithm [26]

Figure 2.6 shows how the sender changes the sending rate after receiving the RTCP report. By measuring packet loss, delay variation and potentially ECN (if ECN is supported by the IP protocol), the sender knows what the current network condition is and whether there is network overuse or underuse. In both the GCC and NADA, the sender determines the sending rate based on the information in the report and tells the encoder to encode the video frame at that rate.

The encoder rate is not the final sending rate. In NADA, the project adds a rate-shaping buffer to change the encoding rate to the final sending rate. Using the buffer, the application changes the time interval between two packets to change the final sending rate. The time interval is influenced by the current network condition. For example: Although the application prompts the encoder to encode the video frame at a specific rate, if there is a sending time interval in the buffer, the application will use different final sending rates. In the GCC and NADA, the sender matches the sending rate to the current bandwidth precisely.

In [28], the authors present an adaptation algorithm for the interactive video streaming. The algorithm uses RTCP feedbacks and FEC to help applications adapt the video quality to current network conditions. The algorithm uses FEC to not only reduce the packet loss, but also detect the available bandwidth of the network. There are two kinds of FEC packets used in the algorithm:

- The FEC probe packet: It can probe the available network bandwidth. For example, when applications send more and more FEC probe packets to the network and they do not detect the network congestion, it means that the available network bandwidth is large. Then applications can decrease the sending rate of FEC probe packets and increase the sending rate of video packets to improve the video quality.
- The FEC protect packet: It is used to reduce packet loss. It means that applications add some residual data (FEC protect packets) to video frames to reduce the packet loss of video frames during transmission.

The algorithm has five states. It changes its states based on feedbacks (both RTCP and FEC). Figure 2.7 shows the state transition diagram.

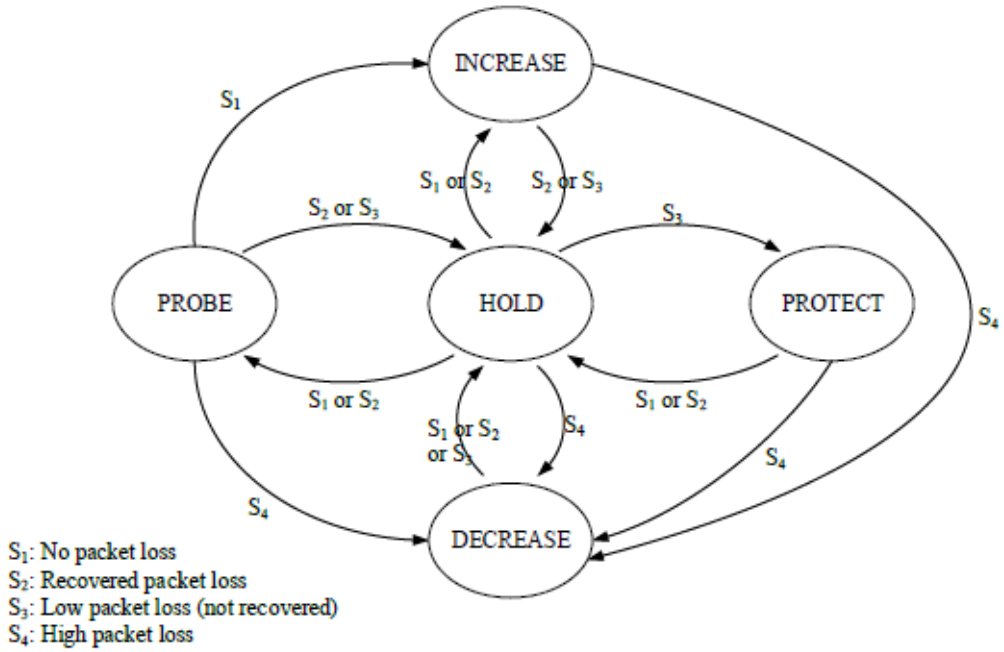


Figure 2.7: State Diagram of the algorithm in [28]

The algorithm classify network conditions to four types:

S1: In this condition, the packet loss gathered by RTCP report is 0, so the network quality is good.

S2: In this condition, we detect some packet losses, but they are recovered by FEC protect packets. The packet loss gathered by RTCP report is not 0, but the residual packet loss gathered by FEC feedback is 0.

S3: We detect packet losses and they are not recovered by FEC. The packet loss gathered by RTCP report is not 0, and the residual packet loss gathered by FEC feedback is also not 0.

S4: The packet loss gathered by RTCP is very high, so the network quality is very bad in this condition.

There are five states of the algorithm shown in Figure 2.7:
HOLD state is the main state. Every time we do not gather enough feedback information to determine current network conditions, the algorithm stay in this state. After gathering feedback information, if the network condition is S1 (There is no packet loss) or S2 (There are some packet losses but they are recovered by FEC protect packets), the state moves to the PROBE state. In the PROBE state, the algorithm increases the sending rate of FEC probe packets to detect the available bandwidth of the network. If the network condition is S3 (There are packet losses and they are not recovered by FEC), the state moves to the PROTECT state. In the PROTECT state, the algorithm increases the sending rate of FEC protect packets to help applications reduce the packet loss. If the network condition is S4 (too many packet losses), the state moves to the DECREASE state. Applications decrease the sending rate of the video data because the network quality is too bad now.

In the PROBE state, after the algorithm increases the sending rate of FEC probe packets, if the network condition is still S1, the state moves to the INCREASE state. In the INCREASE state, the algorithm decreases the sending rate of FEC probe packets and increases the sending rate of video packets to improve the video quality. If the network condition is S2 or S3, the network quality is not good after sending FEC probe packets. Therefore applications cannot increase the video sending rate, and the state moves back

to the HOLD state. If the network condition is S4, the network quality is too bad after sending FEC probe packets. The state moves to the DECREASE state. Applications decrease the sending rate of the video data.

In the PROTECT state, after the algorithm increases the sending rate of FEC protect packets, if the network condition is S1 or S2, it means that FEC protect packets help applications reduce the packet loss of video data. Then the state moves back to the HOLD state. If the network condition is S4, the state moves to the DEREASE state. Applications decrease the sending rate of the video data.

In summary, the authors provide detailed a comprehensive algorithm in [28] to adapt the video sending rate to current network conditions. The highlight is that the algorithm uses FEC mechanisms to both reduce the packet loss rate and detect the available bandwidth dynamically. The algorithm does not use any bandwidth estimation tools to measure the available network bandwidth, which often leads to an extra cost of network resources. The algorithm determines network conditions by sending FEC probe packets and getting feedbacks from RTCP and FEC. Based on feedbacks, the algorithm dynamically adapts the video rate to current network conditions. As the algorithm uses FEC, it reduces the packet loss rate without data retransmission.

2.5 Summary of Related Work

This thesis focuses on rate-control algorithms of real-time video applications. After reviewing related video applications, I have found the following characteristics:

- The applications use several kinds of feedback information to detect the network condition. Using just packet loss to determine the current network condition is not enough. There are many new strategies, and the packet delay is a very useful feedback. All rate-control algorithms of the aforementioned related applications use packet delay to detect the network condition. When the network is congested, the delay variation of receiving packets increases. By collecting information about the delay of packets, applications will know whether a network is overused or underused.
- Applications try to accurately and rapidly control the sending rate based on the network condition. First, applications need to match their sending rate to the current network's available bandwidth accurately to ensure that network resources are neither overused nor underused. The process of adapting sending rate to current network conditions should be as fast as possible, so that users do wait for a long time to receive appropriate video streams when the network condition changes.
- Rate-control algorithms all care about whether they are TCP-friendly.
  After implementing rate-control algorithms, applications should be able to compete for network bandwidth with other TCP flows running concurrently in the same network.
- GCC is a "state-of-the-art" congestion control algorithm for real-time video streaming. It uses both packet loss and packet delay network performance metrics to determine the current network condition. A packet-loss-based rate controller at the sender side and a delay-based controller at the receiver side work together to

adapt the video sending rate to current network conditions. By implementing GCC, the WebRTC can choose appropriate sending rate in different network conditions. I implement a new rate control algorithm in this thesis. Similar to GCC, The new algorithm uses packet loss and packet delay to determine network conditions. The unique approach is that the new algorithm implements PCT and PDT test to measure the trend of one-way delay variation of packets. Based on measurement results, two tests can determine the trend of one-way delay, which can finally determine current network conditions. The PCT and PDT test are neither used in GCC nor in other real-time video transmission applications before. It is only used in some bandwidth estimation tools. Therefore, the thesis proposes a new approach for using packet delay metrics to determine the network conditions in real-time video transmission applications.

3 Chapter: The Test Bed

In this chapter, I will introduce the test bed. I ran all my experiments and collected test results on this test bed. In Section 3.1, I discuss the structure of the test bed. Section 3.2 discusses a software called NETEM. NETEM helps us to change the metrics of the network so that we can emulate different kinds of network conditions on the test bed. In Section 3.3, I discuss a program called IPERF, which generates TCP flows through two clients. IPERF enables us to simulate a true network environment with TCP flows.

3.1 Test Bed Structure

To set up the video conferencing test bed, four computers are needed. Two of the computers must be powerful enough to run the conferencing software. The other two computers can be less powerful; one computer runs the server program, the other computer acts as a emulator and runs the NETEM program (this computer runs Ubuntu 12.04). Other required components are a network switch, four Ethernet cables, and two webcams for video conferencing machines. Although OmniSHIELD running on the test bed is a video conferencing application, my thesis does not address any issues about video conferencing. I focus on adapting the sending rate of real-time video transmission to the available network bandwidth. There are only two clients sending video stream between each other. The work focuses on the unidirectional video transmission. The sender derives feedback information from the receiver. Based on these feedbacks, the sender determines current network conditions and adapts the sending rate of video stream to current network conditions.

When all the hardware has been acquired, all computers must be networked together by first connecting them all to the same local area network switch and then assigning each of them a static IP address. The IP addresses that should be used are shown in Table 3.1.

Computer	IP Address
Emulator (runs NETEM)	10.0.0.1
Server	10.0.0.2
Clients	10.0.0.3 and 10.0.0.4

Table 3.1: IP Address Assignment

For the video conferencing software (OmniSHIELD) to function properly, both the client-side and server-side applications must be configured. Both applications have their own configuration files, which can be edited in any text editor (The server-side configuration file is called Omniserver.exe.config, and the client-side configuration file is called ScsCollaborate.exe.config). These files have already been configured for the IP addresses listed in Table 3.1.

Figure 3.1 shows how the four computers are connected physically. Figure 3.2 shows roles of four computers in the test bed. The server is responsible for setting up and breaking off the connection between two clients. Two clients send and receive video streams between each other. The emulator changes the network condition of the test bed.

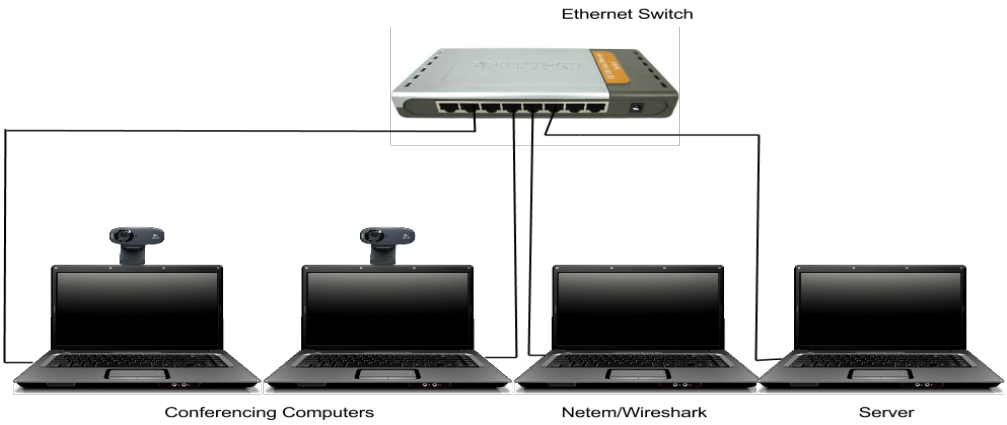


Figure 3.1: Test Bed Setup (1)

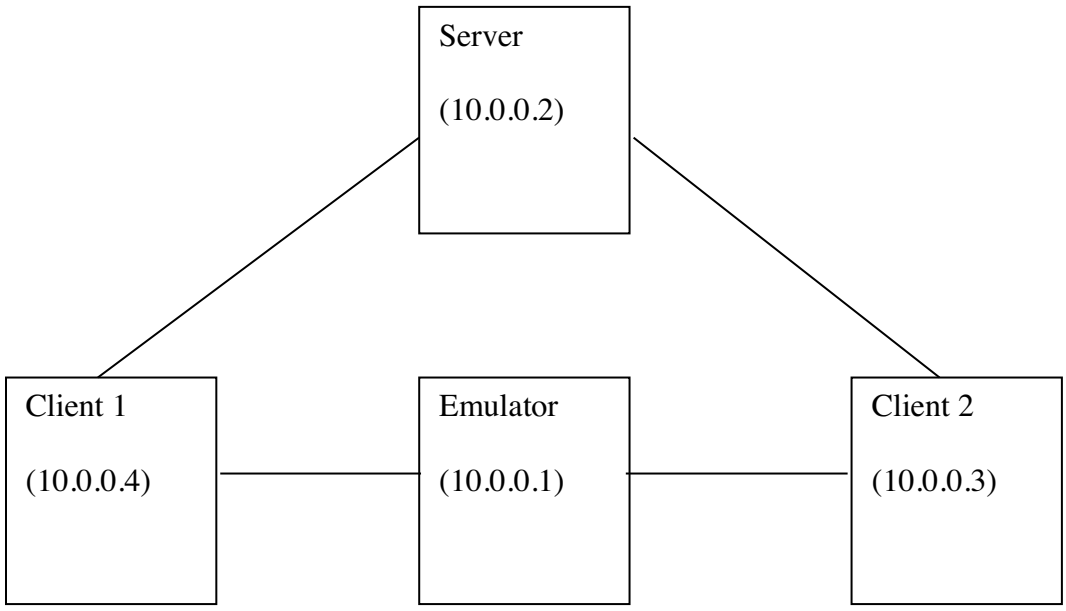


Figure 3.2: Test Bed Setup (2)

In the test bed, we need to change routing tables of two clients to make sure that video streams are transmitted from the sender to the receiver through the emulator. We can add a static route in the routing table of client 1 (10.0.0.4) using following command:
#route add 10.0.0.3 mask 255.0.0.0 10.0.0.1

All the video flows sending from client 1 (10.0.0.4) to client 2 (10.0.0.3) must transmit through the emulator (10.0.0.1). Similarly, we also add a static route in the routing table of client 2. After changing routing tables of two clients, all four computers need to disable the ICMP redirection. If we do not disable this function, the two clients still send packets directly between each other. In Windows, ICMP redirection can be disabled through the registry. In Linux, it can be disabled by changing contents of a configuration file.

After finishing all the configuration, we can use traceroute command to check the setup of the test bed. If there is one hop between two clients in traceroute test, the test bed is successfully established. The video flows transmitting between two clients should go through the emulator. Therefore, the emulator can control the network conditions between two clients.

3.2 NETEM

To control the network status through the test bed, I add an emulator computer. The emulator runs a program called NETEM. NETEM is widely used in different kinds of experiments to change the network status of a system, and has rich and strong features. I

only discuss the features that I use in my experiments. There are many studies on NETEM, but I mainly refer to [29] and [30].

- Packet Delay Modification

NETEM can change the packet delay of a network. If we want to add a 100 ms delay to the current network, we can type the following command:

#tc qdisc add dev eth0 root netem delay 100ms

- Packet Loss Modification

NETEM can also change the packet loss of a network. If we want to add 1% packet loss to the current network, we can use the following command:

#tc qdisc add dev eth0 root netem loss 1%

- Bandwidth Modification

NETEM allows us to change the available bandwidth of a network. If we want to change the bandwidth of the current network to 1 Mbit/s, we can use the following command:

#tc qdisc add dev eth0 root tbf rate 1mbit burst 10k latency 70ms

The Token Bucket Filter (TBF) is used to change the network bandwidth. Some parameters of the TBF buffer will be discussed shortly. The usage of TBF and the referred parameters are discussed in [31]. Figure 3.3 illustrates the mechanism of the TBF buffer.

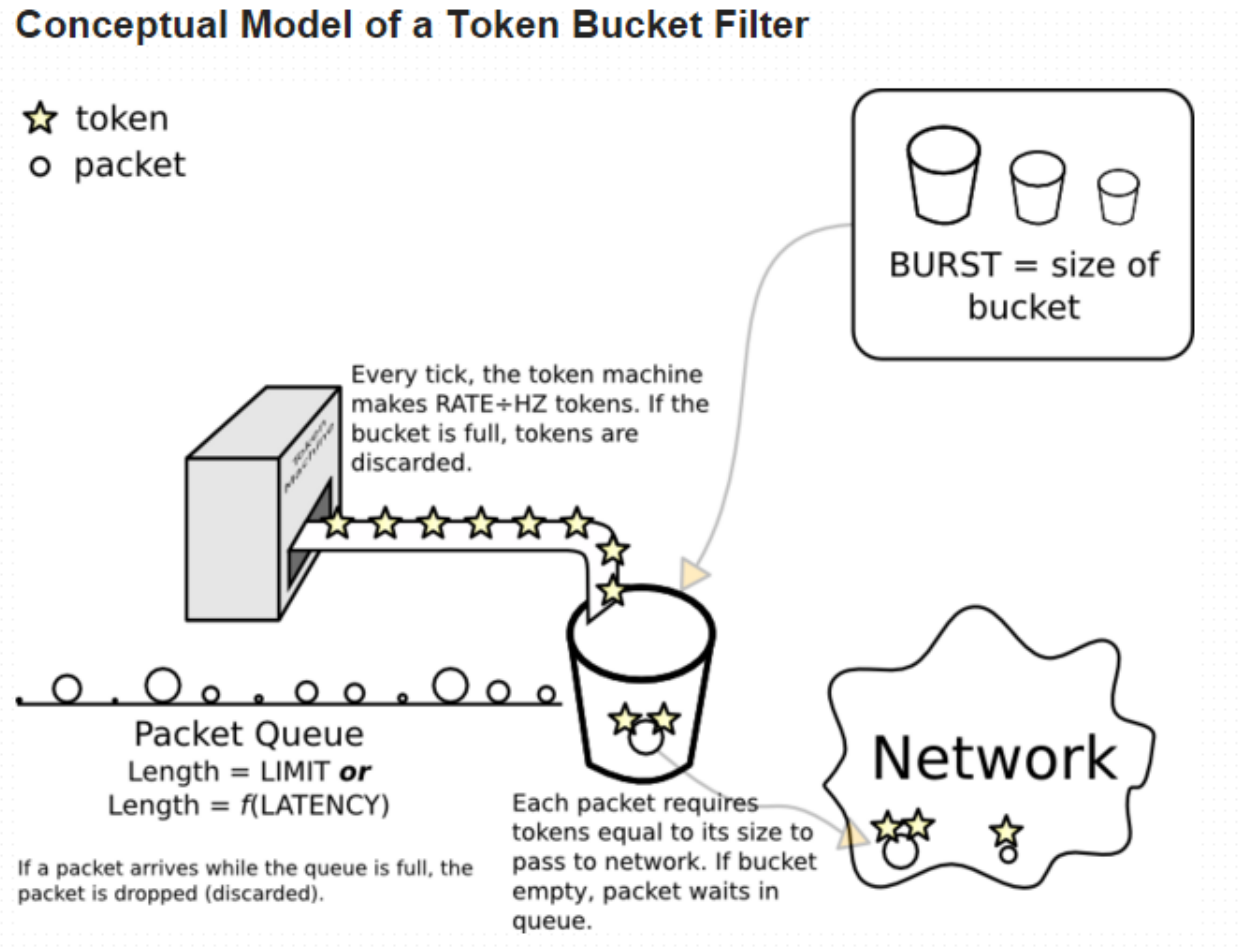


Figure 3.3: The TBF Buffer

The following parameters of the TBF need to be discussed:

1. Limit or latency

Limit is the total number of bytes that can be queued waiting for tokens to become available. The latency parameter specifies the maximum amount of time a packet can spend in the TBF. The limit or latency parameter determines how incoming packets react if there is no token. These two parameters are mutually exclusive. In my test, I set the latency parameter to 70 ms. This means that if the sending rate is higher than the current

network bandwidth, incoming packets will be delayed in the buffer. If a packet stays in the buffer for more than 70 ms, it will be dropped.

2. Burst

This is the maximum amount of bytes that tokens can be instantaneously available. The burst parameter determines the size of bucket. If it is too small, the rate will not reach the desired bandwidth because there is not enough space for the token. In my test, I set the burst parameter to 10k.

3. Rate

This parameter determines the available bandwidth of the network. In the experiment, we need to only change the network bandwidth in one direction. This means that we only change the bandwidth from Client 1 to Client 2. Meanwhile, the bandwidth from Client 2 to Client 1 is not changed. The aforementioned TBF command directly changes the whole bandwidth of the network. To change the one-direction bandwidth, we need to add a filter command. In the test, I change the bandwidth only from Client 1 to Client 2. The following TBF commands are used to change the bandwidth from 10.0.0.4 to 10.0.0.3 through the emulator. A filter is added to complete the above task, and the third command is a filter command; it tells NETEM to only emulate the bandwidth from 10.0.0.4 to 10.0.0.3.

```
# tc qdisc add dev eth0 root handle 1: prio
# tc qdisc add dev eth0 parent 1:1 handle 10: tbf rate XXkbit burst 10k latency 70ms
# tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 match ip dst 10.0.0.3/32 flowid
1:1
```

3.3 IPERF

I use IPERF to generate the TCP flow in the test. This software can run in both Windows and Linux. Its features are described on a web page [32]. IPERF normally works by repeatedly sending an array of ***len*** bytes for ***time*** seconds. In my test, I use it to generate TCP flow from the sender to the receiver. This uses some fundamental functions of IPERF.

To generate one TCP flow from the sender (Client 10.0.0.4 in the test bed) to the receiver (Client 10.0.0.3 in the test bed), we need to first run IPERF in server mode on the receiver. On Client 10.0.0.3, the following command prompts IPERF to run in server mode:

iperf –s

Then, we need to run IPERF in client mode on the sender and establish a TCP connection between the sender and the receiver. On Client 10.0.0.4, the following command prompts IPERF to run in client mode:

iperf –c 10.0.0.3 –t 100

The parameter t determines the time of the TCP flow that sends a packet between the sender and the receiver. By using the recently mentioned commands, Client 10.0.0.4 will send a TCP flow to Client 10.0.0.3 for 100 seconds.

4 Chapter: Project Description

In this chapter, I discuss features of our application (OmniSHIELD). In Section 4.1, I discuss how the application works to transmit video and audio frames between two clients. While describing the source code of OmniSHIELD, I explain how frames are generated, transmitted and controlled. In Section 4.2, I run the application on the test bed under different network conditions. Through the test results, I discuss drawbacks of the current application.

4.1 Project and Source Code Description

OmniSHIELD is composed of the transport layer and the upper layer. The transport layer only focuses on sending packets (frames) through different channels. The upper layer deals with the codec and controls the whole video transmission.

I discuss in detail how the transport layer (channel) works, but I do not discuss the upper layer much in this thesis because it is related to H.264 encoding. I do not change the encoding algorithms in my work. The thesis deals with the packet-sending algorithm in the video channel of the OmniSHIELD application at the transport layer.

Three different kinds of channels in the application are shown in Figure 4.1.

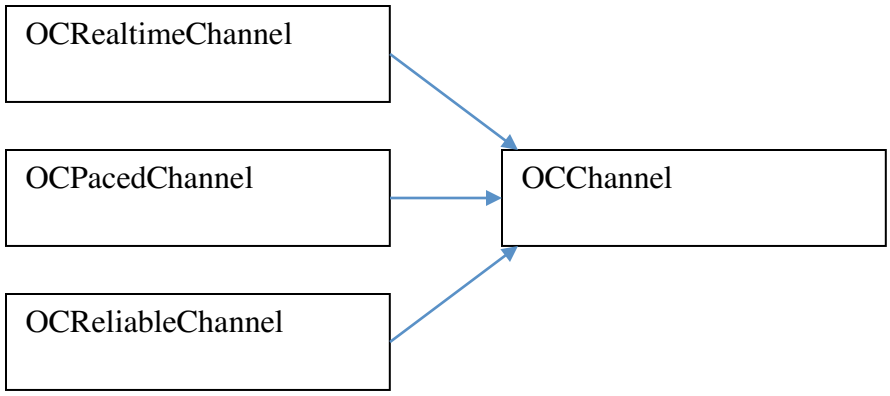


Figure 4.1: Three Channels

OCRealtimeChannel is the audio channel that is used by the application to send audio frames. Every time the channel gets an audio packet, the packet is sent immediately with no additional delay. The length of audio packets is too short compared to video packets. Sending audio packets does not consume a lot of network bandwidth, so there is no need for the application to send audio packets one by one with some delay.

OCPacedChannel is the video channel, and is the focus of this thesis. There is always a delay between two packets in this channel. The delay varies based on different network conditions. This influences the video frame rate. Controlling this time interval in different network conditions is the main focus of the congestion-control algorithm of the OmniSHIELD.

OCReliableChannel is the message channel. This channel has acknowledgement and retransmission mechanisms that ensure that there is no mistake during the message transmission.

We focus on the OCPacedChannel as it transmits video packets. The following code defines typical packet stats:

```
struct OCPacketStats
{
	UINT32 deltaPacketLoss;
	UINT32 deltaPacketOutOfOrder;
	NetworkStatus_t networkStatus;
	UINT32 dataPacketsReceived;
	UINT32 dataBytesReceived;
	UINT32 dataPacketsLost;
	UINT32 dataPacketsOutOfOrder;
};
```

After receiving 50 packets, the receiver generates a report. Different statistics are included in the report, and two of them are network feedback statistics: PacketLoss and PacketOutOfOrder. The two statistics record the PacketLoss and PacketOutOfOrder statistics during transmission. The following code describes how these two statistics are generated:

```
void OCConnection::checkFrameId(OCPacket* pFrame)
{
	UINT32 frameId = pFrame->frame.head.frameId;
	INT32 frameDelta = frameId - _expectedDataFrameId;
	// compare frameId with expect frameId
	if (frameDelta < 0)  // an old frame
	{
		if (frameDelta > -MAX_OUT_OF_ORDER_DELTA)
		// ignore REALLY OLD packets
		{
			dataStats.deltaPacketOutOfOrder++;
			if (dataStats.deltaPacketLoss > 0)
				dataStats.deltaPacketLoss--;

			dataStats.dataPacketsOutOfOrder++;
			dataStats.dataPacketsLost--;
			// a packet out of order detected
		}
		return;
	}
	else
	{
```

```
                if (frameDelta > 0)
                {
                      dataStats.dataPacketsLost += frameDelta;
                      dataStats.deltaPacketLoss += frameDelta;
                      // a packet out of order detected

                }
                _expectedDataFrameId = frameId + 1;
          }
          if (++_receiveDataCount >= DEFAULT_MAX_IDS_IN_BUF)
          //every 50 frames, we will look into the buffer
          {
                send(OCPacket::createFrame(this,
                      SIG_SUPPORT_STATISTICS,
                      0, sizeof(dataStats), &dataStats));
                dataStats.deltaPacketLoss = 0;
                dataStats.deltaPacketOutOfOrder = 0;
                _receiveDataCount = 0;
          }
    }
```

Every time the sender sends a packet, it adds a unique `frameId` to the packet. At the receiver's side, if the `frameId` is not the expected one, the receiver knows the error is in transmission. By checking the real data `frameId` and comparing it with the expected `frameId`, the receiver collects PacketLoss and PacketOutOfOrder statistics. At the end of the foregoing code, the feedback report is generated every 50 packets.

The following code defines the network status using the PacketLoss and PacketOutOfOrder statistics:

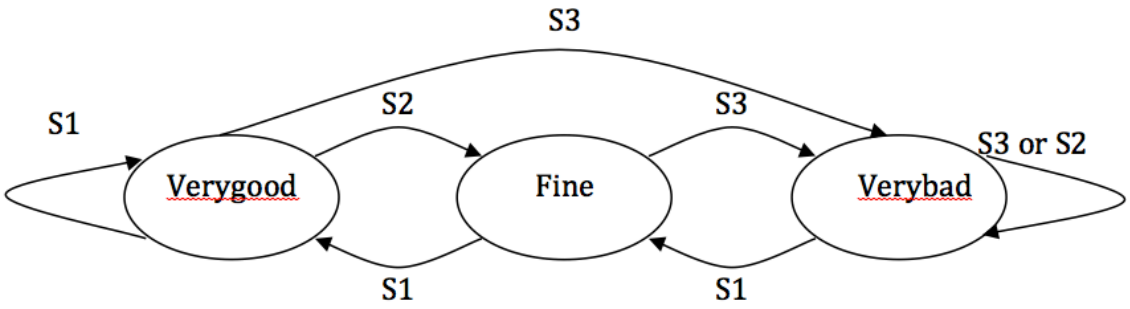
```
void OCConnection::handleFCStatistics(OCPacket* pFrame)
{
      int copyBytes =
      pFrame->frameLength - sizeof(pFrame->frame.head);
      if (copyBytes > sizeof(remoteDataStats))
            copyBytes = sizeof(remoteDataStats);
      memcpy(&remoteDataStats, pFrame->frame.frameData,
      copyBytes);

      int threshold = remoteDataStats.deltaPacketLoss +
      (int)(remoteDataStats.deltaPacketOutOfOrder * 0.6);
```

```
        NetworkStatus_t status = networkStatus();
        if (threshold >= 2)  // 4% packet loss
        {
              _upgradeStatusCounter = UPGRADE_STATUS_CNT;
              status = VERYBAD;
        }
        else if (threshold > 0)  // minor packet loss
        {
              _upgradeStatusCounter = UPGRADE_STATUS_CNT;
              status = (networkStatus() == VERYBAD) ? VERYBAD :
              FINE;
        }
        else  // no packet loss
        {
              if (--_upgradeStatusCounter <= 0)
              {
                    _upgradeStatusCounter = UPGRADE_STATUS_CNT;
                    status = (networkStatus() == VERYBAD) ? FINE :
                    VERYGOOD;
              }
        }

        if (networkStatus() != status)
        {
              dataStats.networkStatus = status;
              //notify upperlevel on a change
              onNetworkStatus(status);
        }

        // notify channels every time, they might adjust further.
        for (int i = 0; i < MAX_CHANNELS; i++)
        {
              if (_channels[i] != NULL)
                    _channels[i]->onNetworkStatus();
        }
  }
```

It defines a threshold that is equal to PacketLoss + PacketOutOfOrder * 0.6. This formula is based on experience. Our test bed is small, so all the packets transmit from the sender to the receiver in one hop. The packets are queued and sent by the emulator in order. Therefore, no packets are out of order in our test bed. In my thesis, I just discuss the influence of the packet loss.

If the threshold is greater than 2, which also means that the packet loss is larger than 4% (we receive one report every 50 packets), the project defines the network state as

VERYBAD. If the threshold is 1, which means 2% packet loss, when the preceding state is VERYBAD, then the current state is also VERYBAD. When the preceding state is not VERYBAD (FINE or VERYGOOD), the current state is FINE. If the threshold is equal to 0, the sender needs to get four consecutive reports. If all four reports say that there is no packet loss, then the application changes the network state. When the preceding state is VERYBAD, the current state is FINE. When the preceding state is not VERYBAD (FINE or VERYGOOD), then the current state is VERYGOOD. The following diagram shows the transformation process of the network state of the OmniSHIELD application:



S1: threshold == 0 (0% packet loss), sender receives 4 continuous 0-packet-loss reports
S2: threshold == 1 (2% packet loss)
S3: threshold >=2 (>4% packet loss)
Threshold = packet loss + 0.6 * packet out of order (in every 50 packets)

Figure 4.2: Transformation of Network States

Every time the sender gets a report showing that the threshold is greater than 2, the application immediately changes the network state to VERYBAD immediately. When the sender gets a report showing that the threshold is equal to 0, the application needs four consecutive reports to verify that there is no packet loss in the network.

After receiving the network state, the application uses this information to control the video sending rate.

```
void OCPacedChannel::onNetworkStatus()
{
NetworkStatus_t status = connection()->networkStatus();

      //varies between 5ms ~ 80ms
      int sendTime = _sendTime;
      switch (status)
      {
      case VERYBAD:
            sendTime += 20;
            if (sendTime > 80)
                  sendTime = 80;
            break;
      case FINE:
            if (sendTime > 40)
                  sendTime -= 10;
            else if (sendTime < 10)
                  sendTime += 5;
            break;
      case VERYGOOD:
            sendTime -= (sendTime > 20) ? 10 : 5;
            if (sendTime < 5)
                  sendTime = 5;
            break;
      }
      _sendTime = sendTime;
}
```

The sendTime variable defines the time interval between two packets. It varies between 5 ms and 80 ms. Every time the receiver receives a VERYBAD report, the sendTime is increased by 20. When the network is FINE, if the sendTime is larger than 40, it is decreased by 10. If it is smaller than 10, it is increased by 5. When the network is VERYGOOD, if the sendTime is larger than 20, it is decreased by 10; otherwise, it is decreased by 5.

The network state is also sent to the upper layer, and is used by the H.264 codec to encode the video frame in proper quality.

```
public void UpdateStatus(PeerConnection.NetworkStatus
networkStatus)
            {
                switch (networkStatus)
                {
                    case NetworkStatus.VERYBAD:
                        _bytesPerSecond = 20000;
                        break;
                    default:
                    case NetworkStatus.FINE:
                        _bytesPerSecond = 40000;
                        break;
                    case NetworkStatus.VERYGOOD:
                        _bytesPerSecond = 80000;
                        break;
                }
            }
```

In the upper level, the `bytesPerSecond` variable changes based on different network states. This parameter prompts the H.264 codec to encode the video frame in different qualities at different encoding rates. The H.264 codec changes the quality of every frame, but the channel changes the frame rate. They all influence the quality of the video from the user's side.

In conclusion, the transport layer changes the frame rate based on the observed network state. The upper level (H.264 adaptive codec) changes the quality of every frame encoded. The structure of the sender model is illustrated in Figure 4.3.

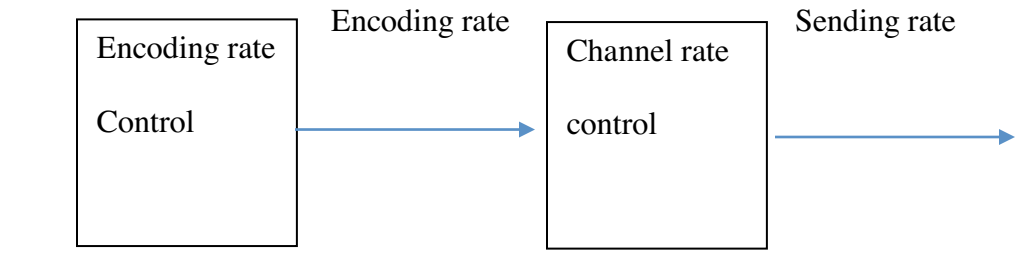


Figure 4.3: Two Steps of Rate Control

Based on the network state (VERYGOOD, FINE or VERYBAD), the H.264 codec encodes video frames at three different rates (80,000, 40,000 or 20,000 byte/s). After their encoding at a specific rate, video packets are sent to an I/O buffer. In the buffer, packets are sent one by one, and each packet is separated from a preceding one by a specific time interval. A large time interval results in a low final sending rate. The time interval is defined by the sendTime variable. Every time the application updates the network state, it also changes the sendTime variable based on the current network state.

The aforementioned two steps determine the final sending rate. This is the congestion-control algorithm of the OmniSHIELD application. It does not precisely match the sending rate with the network available bandwidth as it has only three encoding rate levels (not a dynamically changing encoding rate based on network bandwidth, such as that of WebRTC or Skype). I will discuss the performance of the algorithm in the following section.

4.2 Test Bed Evaluation

In Section 4.2.1, I present the test results for when the project runs in a stable state for some time, which is called a static test. Section 4.2.2 presents the test results for when the project runs from one state to another state, which is called a dynamic test. In Section 4.2.3, I present the test results for when the project runs in the test bed with a TCP flow.

4.2.1 Static Test

I evaluate the relationship between the sendTime variable, network state, frame rate and sending rate by changing the sendTime variable and the network state in the program. All the referred parameters are collected through the statistic function of OmniSHIELD. Figures 4.4 and 4.5 show the test results.

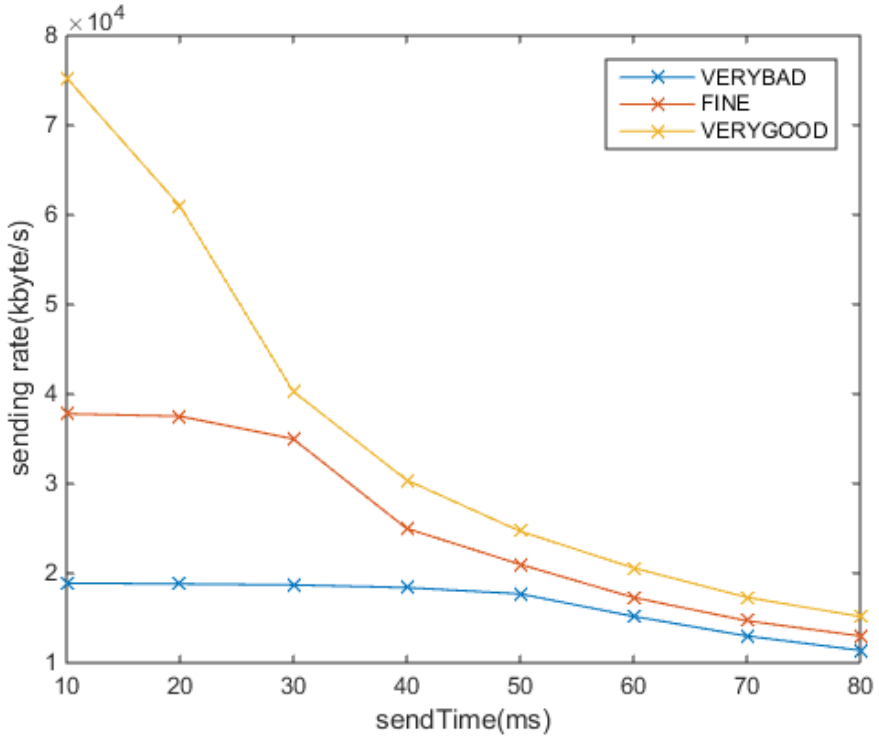


Figure 4.4: Static Test Results (1)

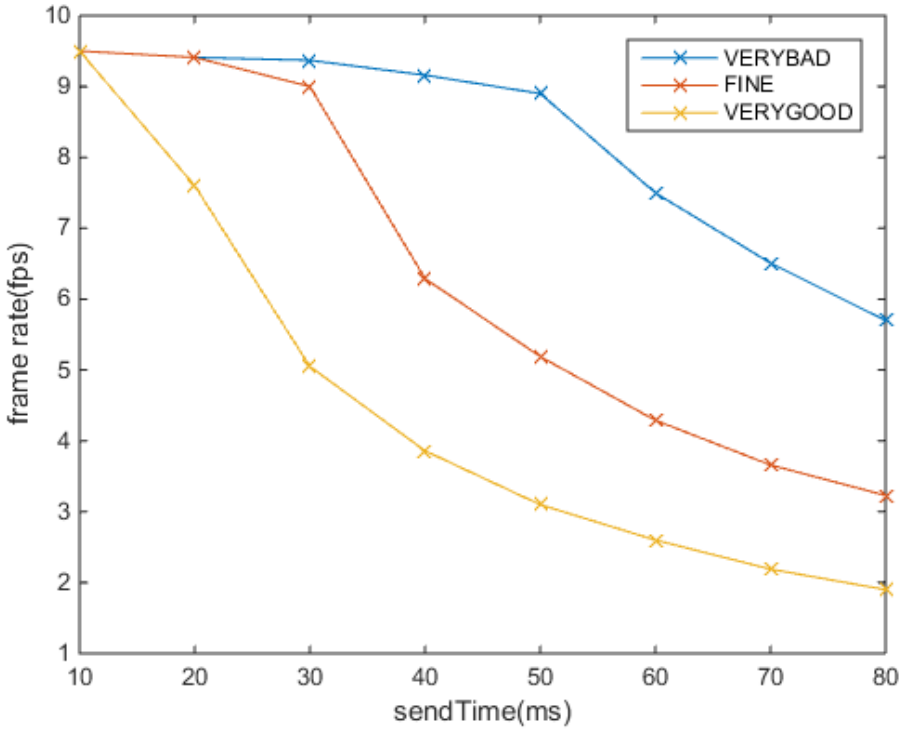


Figure 4.5: Static Test Results (2)

As shown in Figures 4.4 and 4.5, when the time interval increases, both the final sending rate and the frame rate decrease. The encoding rate determined by the network status is the upper bound of the final sending rate. When sendTime (time interval) increases at the start, the rate decreases rapidly.

In Figure 4.5, the frame rate decreases rapidly in the VERYGOOD network state. When the project sends packets with a long time interval, final sending rates are nearly the same between the three network states. But in the VERYGOOD state, the codec encodes the frame in a higher quality. So the application needs to sacrifice the frame rate to maintain the high quality of every frame.

The user experience of the video is determined by the frame rate and the quality of frames [33]. For example: In my test, if the application encodes video frames of high quality (VERYGOOD state) at nearly equal low sending rates, it gets a low frame rate. From the user's side, they get high-quality images of a video, but the smoothness of the video (dynamic performance) is reduced by a low frame rate. So if we just send static images, high quality is important, but if we send a dynamic video, a low frame rate causes poor performance.

In our application, we can change the frame rate by changing the time interval between the packets. The video quality is determined by the codec, and the application does not precisely match the frame rate to the network bandwidth. But by changing packets' time

interval, the application can change the frame rate (the encoding rate is not changed) and then finally change the sending rate.

Next, I test the measurement accuracy of the network parameters (PacketLoss) in the application. In the test bed, I first set the packet loss to different values to see what happens in the application. I find that when I set the packet loss of the network to 2%, the application goes to the VERYBAD state and remains in that state. But the VERYBAD state means 4% or higher packet loss in our application. So the project always overestimates the network packet loss. When I change the packet loss to about 1.5%, the state varies between VERYBAD and FINE. When I change the packet loss to 1%, the state is mostly in FINE and sometimes VERYBAD.

So the application can detect the packet loss of the network. When there is a packet loss, the state will no longer be VERYGOOD. The application cannot accurately measure how much packet loss is in the network. So the boundary between VERYBAD (4% or larger packet loss) and FINE (2% packet loss) states is not clear. In the application, the feedback report is generated after every 50 packets. The sample number (50) is not large enough to accurately measure the packet loss rate.

For example: Setting the network to 2% packet loss means that there is ideally 1 packet loss for every 50 packets. If there is another loss in the 50-packet sample, the loss rate directly turns to 4% (2 out of 50). Due to the small sample size, the boundary between 2% and 4%, which means FINE and VERYBAD, just depends on whether there is one

more packet loss. Therefore, for a small sample size, the measurement is not accurate because a small difference in the packet loss (between 1 and 2 losses) leads to a different state (FINE or VERYBAD), which finally causes a large difference in the final sending rate. So the application can detect the packet loss, but the accuracy of the measurement is low.

One way of solving this problem is to increase the sample size, but a larger sample size also means that the report is sent at larger time intervals, i.e., less frequently. In our application, if we send a video at a rate of 40 kbyte/s and the average packet size is 1,000 bytes (in the application, the largest packet payload size is 1,456 bytes), then the packet rate will be 40 packets/s. With a 50-packet sample size, the report is sent every 1.25 s. If we use 100 as the sample size, the measurement of the packet loss is more accurate, but the report is sent every 2.5 s (it should be longer in a real network). The project cannot react to changes in network conditions in time.

So adding a delay-based feedback parameter in our application is a good idea, as this addition allows the project to estimate the network state more accurately. We also need to change the threshold formula to include new parameters. Add delay-based parameters ensure that the application no longer depends only on packet loss.

Thus far, we have focused on the static behavior of the application. All results are obtained after the application goes to and stays in one state for some time. The dynamic behavior of the application is also important. For example: When the network condition

changes, how does the project dynamically change the sending rate? When the application starts, how does the sending rate change from the initial value to the appropriate value? These are dynamic behaviors I will discuss in the following section.

4.2.2 Dynamic Test

I use Wireshark to track the client and use its I/O graph function to collect the dynamic statistics related to the sending rate. Figure 4.6 shows test results when the network bandwidth decreases from 1 mbit/s to 400 kbit/s.

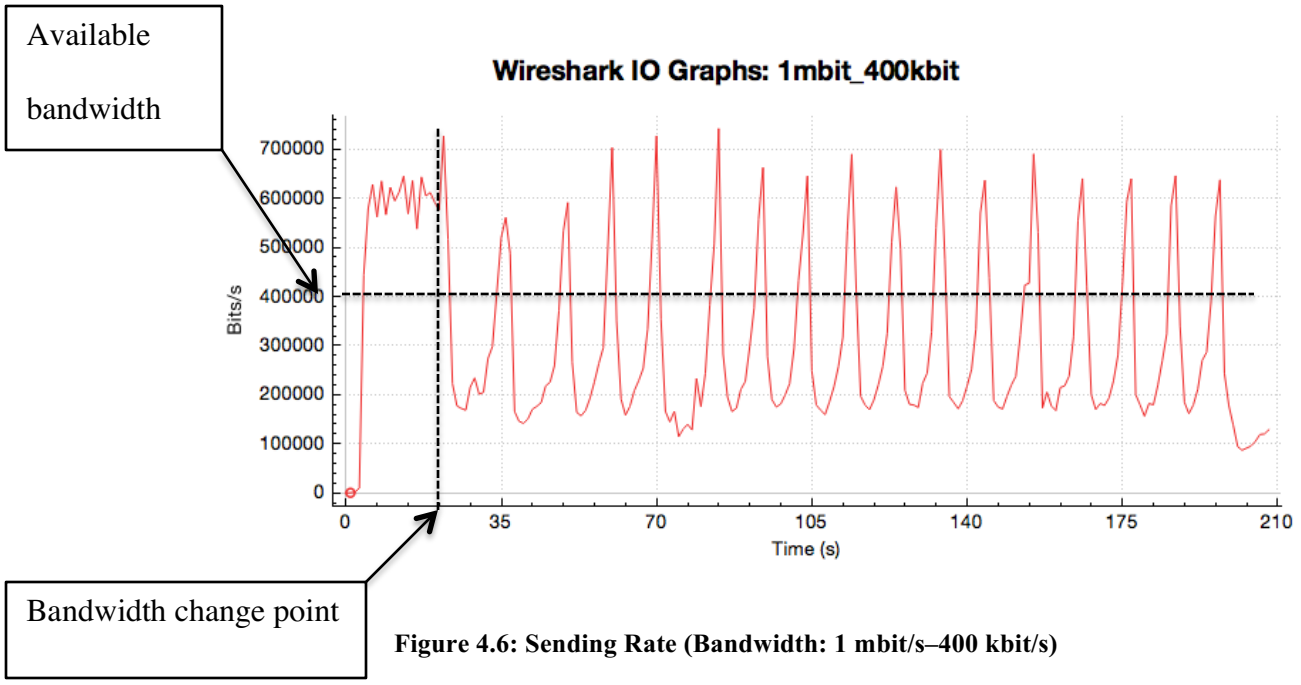


Figure 4.6: Sending Rate (Bandwidth: 1 mbit/s–400 kbit/s)

We can clearly see the dynamic behavior in this figure. The unit of the y-axis is bit/s. I start the application with 1 mbit/s bandwidth, and it takes about 5 s for the sending rate to reach the maximum value (about 600 kbit/s). As there is no packet loss, the state goes to VERYGOOD and sendTime decreases. The sending rate increases very fast to the maximum point. Then at around 20 s, I limit the bandwidth to 400 kbit/s. The sending

rate (around 600 kbit/s) is now higher than the available bandwidth. There is > 4% packet loss, so the state changes to VERYBAD, and sendTime increases by 20 ms per report. The sending rate then decreases to the minimum value in about 5 s.

After the sending rate reaches the minimum value (about 100 kbit/s), the bandwidth is 400 kbit/s, and there is no packet loss. The sender gets 4 continues no packet loss reports. The network state then changes to FINE, and sendTime decreases by 10 ms per report. The rate increases slowly after the 25 s, but the sending rate is still lower than 400 kbit/s. There is still no packet loss. The state changes from FINE to VERYGOOD, and sendTime decreases by 10 ms per report. The rate increases faster and faster until it is higher than 400 kbit/s. Then there is packet loss. The state changes from VERYGOOD to VERYBAD, and the rate goes to the minimum point again after 35 s.

In summary, when the application faces limited bandwidth (smaller than the current sending rate), the sending rate will first go to the minimum point. After the rate reaches the minimum point, it will increase until it reaches current available bandwidth. When the rate reaches the available bandwidth, it then decreases to the minimum point again. The sending rate changes in this manner periodically until there is another bandwidth change.

So our application has the following drawback: the sending rate cannot remain in the small range in a limited-bandwidth network. The sending rate increases and decreases periodically, although the bandwidth is fixed at a specific value.

In the application, when the sending rate is higher than the bandwidth, the state changes from VERYGOOD to VERYBAD without first changing to FINE. If the receiver receives 50 packets during one sampling period, the state transition will be S1 as shown in Figure 4.3; if it receives 48 or less packets, the state transition will be S3. Only when the receiver receives 49 packets will the state transition be S2. So the state change from VERYGOOD to FINE is very rare (it needs the condition of S2). The state changes from VERYGOOD to VERYBAD almost every time we face a bandwidth limitation.

4.2.3 Project Runs with TCP Flow

The OmniSHIELD application performs poorly when it sends video frames concurrently with TCP flows in the network. I test our application's behavior when it runs with one TCP flow in the same network. The results of this test are also captured using Wireshark. When the application runs concurrently with a TCP flow (generated by IPERF) under a 500 kbit/s bandwidth network, I get the results shown in Figure 4.7.

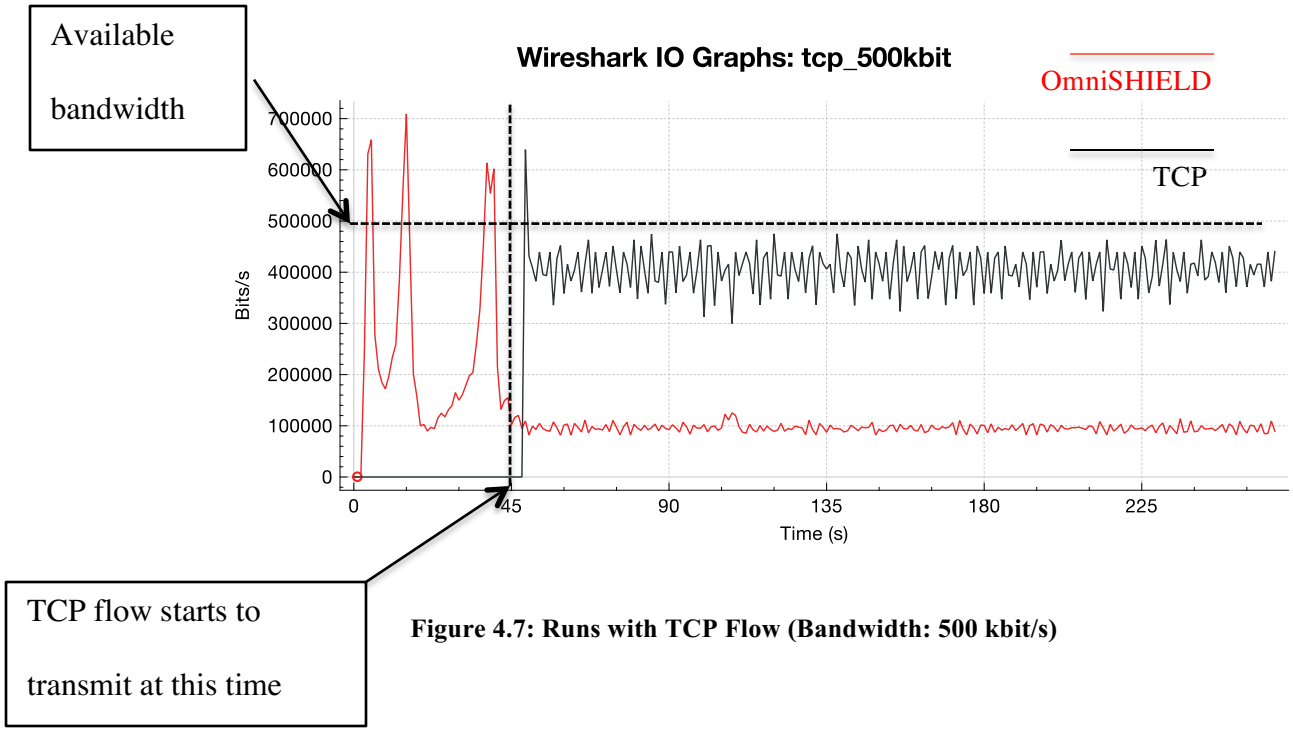


Figure 4.7: Runs with TCP Flow (Bandwidth: 500 kbit/s)

The red line is the UDP flow (video flow), and the black line is the TCP flow. I start by sending the UDP flow under the 500 kbit/s bandwidth network. At about 45 s, I start the IPERF to send the TCP stream, and I notice that the TCP rate is frequently higher than the UDP rate, and the total bandwidth usage is near 500 kbit/s. The video flow runs at the minimum rate most of the time. Sometimes the UDP flow rate tends to increase while the TCP flow rate decreases. But then the rate of the UDP flow rapidly decreases to its minimum point. In this scenario, the TCP flow uses more bandwidth than the UDP flow. This is because the minimum rate of the video flow is about 100 kbit/s. In the 500 kbit/s bandwidth network, the video flow shares bandwidth with the TCP flow. But when I run the test with a higher bandwidth, the superiority is obvious. Figure 4.8 shows the test results when I set the bandwidth to 1 mbit/s.

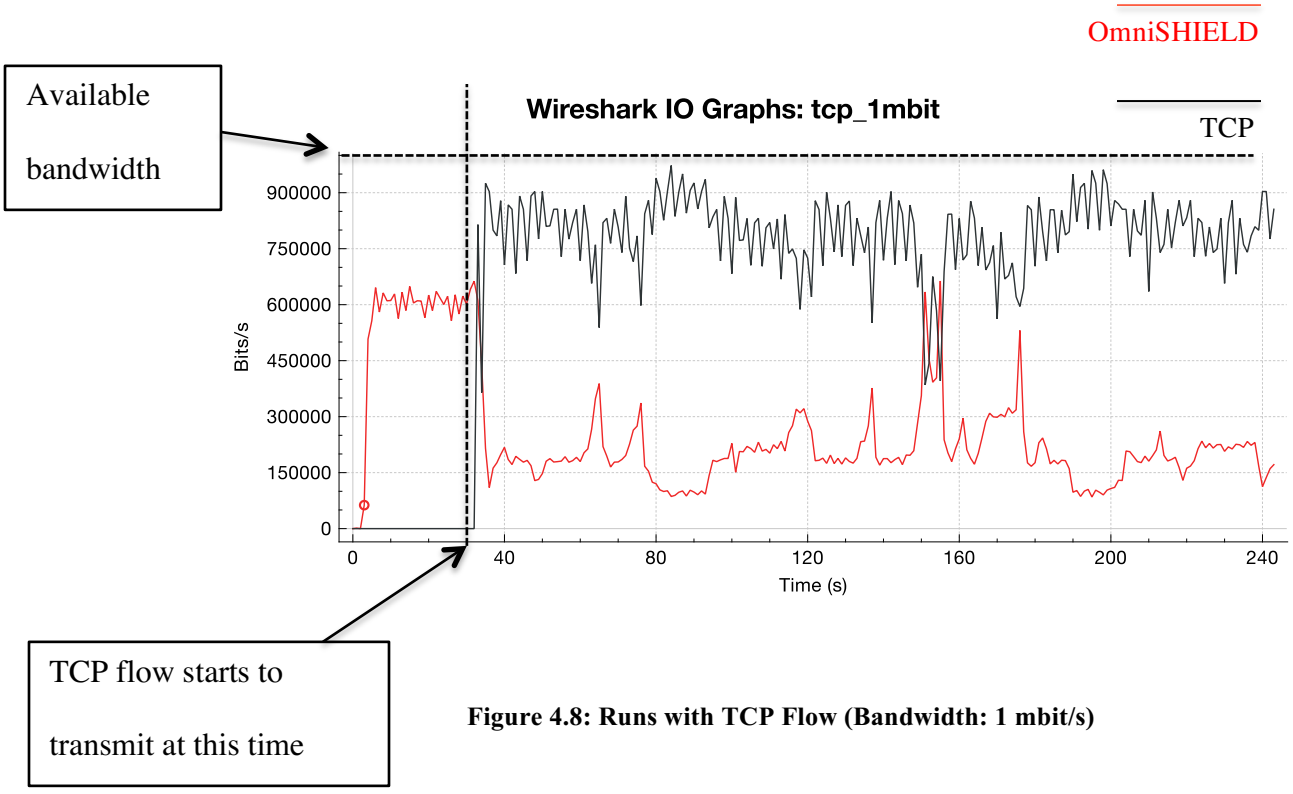


Figure 4.8: Runs with TCP Flow (Bandwidth: 1 mbit/s)

Again, the black line represents the TCP flow, and the red line represents the UDP video stream. In this scenario, the video flow has a significantly lower sending rate when

running concurrently with the TCP flow. When the UDP flow runs at the minimum rate for some time, the sender detects no packet loss. The UDP flow tries to increase the sending rate, which is then decreased by the TCP flow. But when the sender of the project receives a packet loss report, the sending rate of the UDP flow then immediately decreases to the minimum value. Concurrently, TCP also detects packet loss and reduces the flow rate, but the TCP rate recovers very fast. It is evident that the TCP congestion-control algorithm is much more aggressive than our project's algorithm. When facing packet loss, TCP stream also reduces the sending rate, but it does not reduce rate to the minimum value. If it detects no packet loss, the rate recovers rapidly. The algorithm of our application is too conservative because every time the sender receives a packet loss report, the network state changes to VERYBAD, and the sending rate rapidly decreases to the minimum point. In this scenario, the sending rate of the video flow is much slower than the rate of the TCP flow. At the congestion-control side, the TCP and the UDP flow do not fairly share the network bandwidth.

One-way of improving the application performance is adding a delay-based congestion-control algorithm to the application. The other way is modifying the rate-adjustment algorithm. Based on the detected network status, the sender changes the sending rate with a better algorithm so that it can fairly compete for network resources with other flows (such as TCP flows).

There are other ways to improve the application performance. For example, we can add a playout buffer at the receiver side to improve the performance. When streaming packets

arrive at the receiver, they need to be broadcast on the receiver computer at right time. Because the transmission time of each packet on the network is different, if packets are sent in a time interval at the sender side, this interval will be different at the receiver side. This is called jitter caused by the delay of network. The solution is to buffer packets at the receiver side in a playout buffer. Then packets are broadcast in sequence at right time based on timestamps calculated at the sender side. These timestamps indicate the time interval between the current packet and the first packet. As I only focus on the rate control algorithm, I implement my new algorithm to address issues of the application. After some related work reviewing, GCC talks about adding delay metrics to determine the network conditions. PathLoad, a bandwidth estimation tool, provides two tests (PCT and PDT tests) to measure the trend of one-way delay variation of packets. I decide to combine these two methods to implement my new delay-based rate control algorithm on OmniSHIELD. I will describe my new algorithm in the next chapter.

5 Chapter: New Algorithm's Description, Implementation, and Test Results

In this chapter, I discuss my new algorithm of our application in detail. In Section 5.1, I discuss my implementation plan of the new algorithm in the application. In Section 5.2, I add time stamps to every packet sent by the application. Section 5.3 focuses on the addition of the PCT and PDT tests to the application. PCT and PDT tests are two delay-based parameters that help the sender to determine the network state. Finally, I present test results of the new algorithm in Section 4.5. The test results show that the new algorithm improves the project performance compared with the original one.

5.1 The New Algorithm

After testing the program performance in the test bed, I observed both the dynamic and the static behaviors of the program in the preceding chapter. Based on the lessons I learned, I designed and implemented a new rate-control algorithm in the program, and tested the results. The new algorithm aims to improve the program performance. The algorithm has two parts:

1. Adding timestamps to packets and measuring the one-way delay variation of packet pairs.
2. Modifying the rate-control algorithm of the program.

As I mentioned in Chapter 4, the packet loss calculation is not accurate in our application. Observing only PacketLoss statistics cannot accurately reflect the network state. For example: The wireless network always has packet loss, but that packet loss does not

mean the network is congested. In Chapter 2, I discussed some papers talking about different congestion-control algorithms, and some of the algorithms (such as the GCC) use a delay-based algorithm concurrently working with packet loss to control the sending rate of packets.

So I have decided to add delay-based feedback to our application. To implement this, I add a time stamp to each packet at the sender's side. The time stamp records the time the packet is sent from the sender. At the receiver's side, after one packet has been received, the receiver also records the time it receives the packet from the sender. The receiver will then compare the receiving time with the sending time. This allows it to collect one-way delays (the sending time – the receiving time) of each packet.

The program will then determine if the network is congested based on the one-way delay of packets. As we do not have synchronized clocks between the sender and the receiver, one-way delay of a single packet is not useful. As shown in [26] and [27], delay variations of every two consecutive packets (packet pairs) can accurately reflect the network state. If delay variations of packet pairs have an increasing trend for some time, we can say that the sending rate is larger than the available bandwidth, and the network is congested. The sender should, therefore, reduce the sending rate. If delay variations show a non-increasing trend, the network is not overused, and the sender can increase the sending rate.

We can use the following two formulas to determine the trend of one-way delay variations. These formulas are used in a bandwidth estimation tool called Pathload [8].

1) Pairwise Comparison Test (PCT)

The formula of the PCT is

$$R_{pct} = \frac{\sum_{j=2}^{K} I(D_j > D_{j-1})}{K-1}$$

$$0 \leq R_{pct} \leq 1$$

D is the one-way delay of one packet, and K is the packet number (sample space) we use to determine the trend. If the following packet's one-way delay is larger than that of the preceding packet ($D_j > D_{j-1}$), the parameter I ($D_j > D_{j-1}$) = 1. Otherwise, I ($D_j > D_{j-1}$) = 0. The PCT then calculates the average number of I ($D_j > D_{j-1}$) in K continuous packets. If all the following packets' one-way delays are greater than those of the preceding packets, R_{pct} = 1. If all the following packets' one-way delays are smaller than those of the preceding ones, then R_{pct} = zero. By setting a threshold for PCT, we can determine the trend of one-way delay variations. For example: Let us set the PCT threshold to 0.8. A PCT of a packet train (K consecutive packets) that is larger than 0.8 it means that over 80% of the following packets' one-way delay is larger than that of the preceding ones. We can then conclude that the one-way delay has an increasing trend (the sender should reduce the sending rate). If the PCT of a packet train is smaller than 0.8, then the delay has a non-increasing trend.

2) Pairwise Difference Test (PDT)

The formula of the PDT is

$$R_{pdt} = \frac{\sum_{j=2}^{K} (D_j - D_{j-1})}{\sum_{j=2}^{K} |D_j - D_{j-1}|}$$

If all the following packets' one-way delays are larger than those of the preceding packets, $R_{pdt} = 1$. If all the following packets' one-way delays are smaller than those of the preceding ones, $R_{pdt} = -1$. Therefore, the one-way delay of K packets has an increasing trend when PDT tends to 1. So we can also set a PDT threshold to determine if the delay has an increasing trend. By combining both formulas, we can finally determine if one-way delays of packets have an increasing trend.

In [8], the authors discuss why we need two formulas to detect the increasing trend of one-way delay. PCT measures the fraction of consecutive packet pairs which have an increasing one-way delay. PDT quantifies how strong the start-to-end one-way delay variation is, relative to the one-way delay absolute variations during the stream. As shown in Figure 5.1(a), the PCT can detect the increasing trend of the one-way delay of a 12-packet train; the PDT cannot detect that. Figure 5.1(b) shows that the PCT cannot detect the increasing trend of the one-way delay of a 12-packet train, but the PDT can detect the trend.

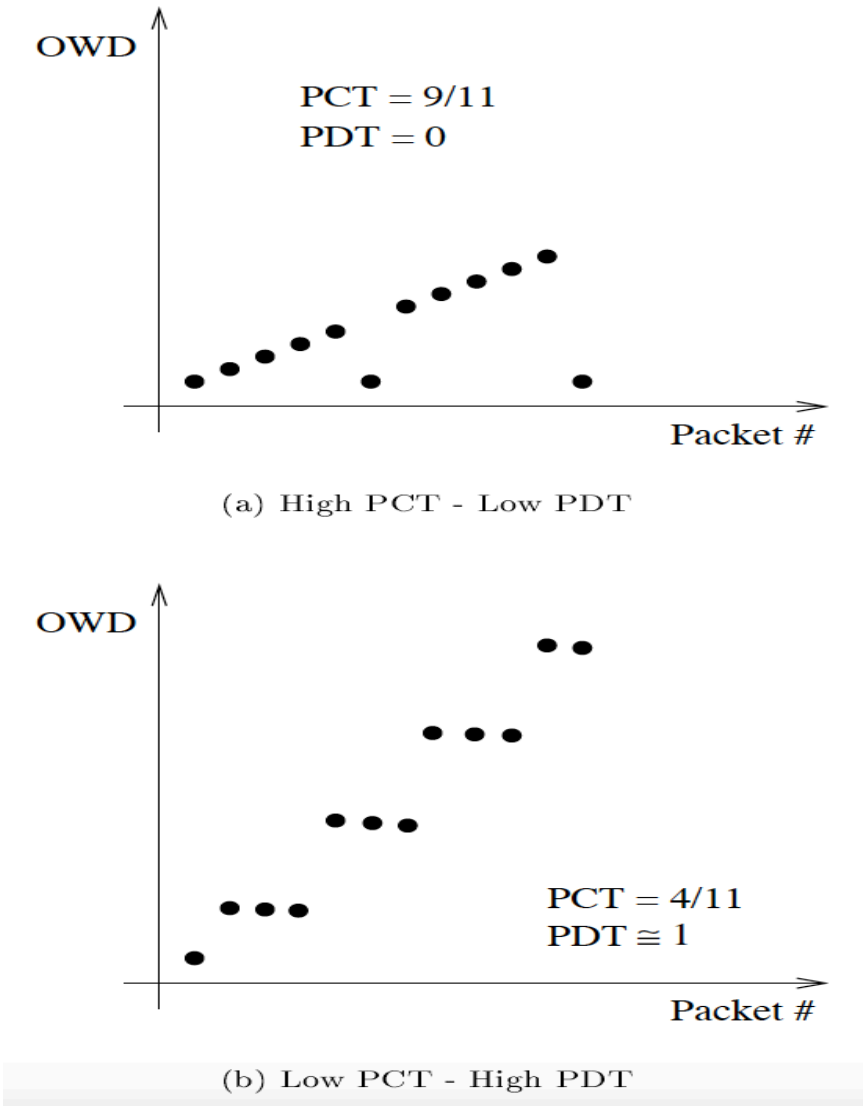


Figure 5.1: PCT and PDT Tests [8]

As I explained in Chapter 4, through testing the dynamic behavior of our application when experiencing bandwidth changes, we have found that the current congestion-control algorithm has some drawbacks. Here, I modify the current algorithm with the help of the new delay-based feedback. There are two main changes to the algorithm:

- The three states of the algorithm (VERYGOOD, FINE and VERYBAD) are equally important. In the old algorithm, the application rarely stays in the FINE

state. When the network bandwidth is not enough, the application immediately changes the state to VERYBAD. When the network bandwidth is higher than the sending rate, the project changes the state to VERYGOOD. So we can see that the sending rate varies sharply and periodically between the maximum and the minimum point in the test. In the new algorithm, the application defines the FINE state effectively. When the bandwidth is a little lower than the maximum sending rate, the sending rate no longer decreases to the minimum. The state can stay FINE, and the sending rate stays at mid-level.

- Two counters are added to control the state transition. In the old algorithm, when packet loss is detected, the state immediately turns to VERYBAD. So the application cannot compete for the bandwidth with the TCP flow. In the new algorithm, I add a state_down counter. When there are enough continuous "network overused" reports, the state_down counter will decrease to 0, and the state will change from a high level to a low level. With this strategy, the application does not decrease the sending rate immediately when the network bandwidth is not enough. Thus, the project can compete for the bandwidth with the TCP flow. But if the network is overused for some time, the state_down counter will decrease to 0, and the sending rate will decrease to an appropriate value. There is also a state_up counter that controls the state transition from a low level to a high level. In the old algorithm, the project increases the state when the sender receives four continuous "no packet loss" reports. The sender tries to increase the sending rate more frequently in new algorithm, and the state_up counter is set to a high value. So the project increases the sending rate only when

the sender gets enough "network underused" reports. The frequency of increasing the sending rate is relatively low compared with that in the old algorithm. So the project can stay at a stable state most of time. The sending rate no longer varies sharply and rapidly.

5.2 Adding Time Stamp to the Application

I successfully added a time stamp to packets in the application. I test the one-way delay variation of packets in different network conditions. Test results indicate that the delay variation could reflect the network status. We can add the delay feedback to detect the network condition.

I added time stamps to every data packet of the network. I use the Windows function timeGetTime() to retrieve the system time. The usage of timeGetTime() is described in [34], and the differences between various time functions are described in [35]. Using timeGetTime(), I obtain a time stamp with a millisecond precision. In the application, the function getTimeTicks() replaces timeGetTime(). The getTimeTicks() function returns a 4-byte time stamp, which is platform-independent. So I use getTimeTicks() to measure the time stamp.

```
typedef unsigned int UINT32;

UINT32 getTimeTicks()
{
      return (UINT32)timeGetTime();
}
```

I add invocations of the getTimeTicks() function in two positions in the application. Before the packet is sent, I use getTimeTicks() to get the sending time of packets.

```
void OCConnection::sendPacket(OCPacket* packet)
{
      if (_state == HS_SHUTDOWN)
            return;
if ((packet->frame.head.sigType & SIG_TC_DATA) == SIG_TC_DATA)
      {
            packet->frame.head.frameId = ++_dataFrameId;
      packet->frame.head.timestamp = getTimeTicks();
// get sending timestamp of packets

      }
}
```

I also add the variable time stamp to record the sending time of packets.

```
union {
            struct {
                  struct {
                        UINT8  sigType;
                        UINT8  sigFlag;
                        UINT16 seqId;
                        UINT32 frameId;
                        UINT32 timestamp;
// sending timestamp
                        } head;
                  BYTE   frameData[MAX_FRAME_DATALEN];
            } frame;
```

After the receiver has received an incoming packet, I invoke getTimeTicks() to obtain the receiving time of packets.

```
void OCConnection::onIncomingPacket(OCPacket* packet)
{
      stats.totalPacketsReceived++;
      stats.totalBytesReceived += packet->frameLength;

      packet->recvTimestamp = getTimeTicks();
// get receiving timestamp of packets
```

The variable `recvTimestamp` is also a new variable that records the receiving time of packets. I also added it to the OCPacket class. I thus collect both the sending time and the receiving time of every packet.

As described in Section 4.1, video packets are encoded with a specific encoding rate at the sender side. Then packets are sent to an I/O buffer at the sender side. As the paced channel of the OmniSHIELD sends packets one by one with a time interval, packets are delayed in the buffer for some time. Then packets are sent to the receiver. During transmission, packets should be delayed in routers if the network is congested. If the packet is not lost during transmission, it will transmit to the receiver. During packets transmission, delay occurs at the sender side and in middle routers. In my thesis, I measure the sending time stamp when packets are sent to the network. I measure the receiving time stamp when packets arrive at the receiver. Therefore, I focus on the delay of packet transmission in the network. I do not care about the delay of packets in the buffer at the sender side. I also do not include the delay of packets in the playout buffer at the receiver side. The delay I calculated mainly consists of packet transmitting time in the network and packet queuing time in network routers. The queuing delay in routers can reflect the network conditions, so packet delay I measured can be used by applications to determine current network conditions.

5.3 Adding PCT and PDT Tests to the Project

In this section I describe the implementation of adding PCT and PDT formulas in the application. The test results show that adding PCT and PDT formulas is useful in helping

us determine the network condition. Later on, I will use these two new parameters together with the existing PacketLoss parameters to determine the network condition.

I modify the codes in the project to calculate the PCT and PDT for every 50 packets received at the receiver's side and then sends a feedback report to the sender. For every 50 packets received at the receiver's side, the receiver calculates PCT and PDT over the preceding 50 packets. The receiver then sends a report to the sender. The report also includes the already existing PacketLoss statistic.

To analyze test results, I printed some logs with the new PCT and PDT parameters that I tested in a 600 kbit/s bandwidth network. I fixed the sending rate to the highest level (VERYGOOD and sendTime = 5 ms). So the sending rate is 80 kbyte/s. In this condition, the bandwidth is slightly lower than the sending rate. I attach the log near the bandwidth-changing point.

```
ERROR: Omni.Comms: received data packet 4398 @ 4674935, sent 4689651, delay -14716, delayvar 1, type 193, count 47
ERROR: Omni.Comms: received data packet 4399 @ 4674935, sent 4689651, delay -14716, delayvar 0, type 194, count 48
2002/03/11 00:33:33.924
ERROR: Omni.Comms: received data packet 4400 @ 4674940, sent 4689656, delay -14716, delayvar 0, type 193, count 49
ERROR: Omni.Comms: packetloss = 0, packetoutoforder = 0, PCT = 0.244898, PDT = -0.057971, count = 50
2002/03/11 00:33:33.939
ERROR: Omni.Comms: received data packet 4401 @ 4674955, sent 4689671, delay -14716, type 194, count 0
2002/03/11 00:33:34.014
ERROR: Omni.Comms: received data packet 4402 @ 4675030, sent 4689732, delay -14702, delayvar 14, type 194, count 1
ERROR: Omni.Comms: received data packet 4403 @ 4675030, sent 4689735, delay -14705, delayvar -3, type 193, count 2
ERROR: Omni.Comms: received data packet 4404 @ 4675030, sent 4689740, delay -14710, delayvar -5, type 193, count 3
ERROR: Omni.Comms: received data packet 4405 @ 4675030, sent 4689745, delay -14715, delayvar -5, type 193, count 4
```

Figure 5.2: Logs of the PCT and PDT Tests (1)

In Figure 5.2, the blue line is the last report before the bandwidth changes from 1 mbit/s to 600 kbit/s. As we can see, there is no PacketLoss in the report. PCT is 0.24, and PDT is –0.05. As discussed in Section 5.1, when the PCT reaches 1, the one-way delay has an

increasing trend; when the PCT is 1, every packet has higher delay than the preceding one. When PCT reaches 0, the one-way delay has a decreasing trend; when PCT is 0, every packet has lower or equal delay than the preceding one. In this case, PCT is almost 0, so we conclude that the one-way delay does not have an increasing trend. Our sending rate is adequate under the current network condition.

For the PDT test, when PDT reaches 1, the delay has an increasing trend. When PDT reaches −1, the delay has a decreasing trend. In this case, the PDT is almost 0, we conclude that the one-way delay has neither an increasing trend nor a decreasing trend.

```
ERROR: Omni.Comms: received data packet 4438 @ 4675355, sent 4690067, delay -14712, delayvar 1, type 194, count 37
2002/03/11 00:33:34.359
ERROR: Omni.Comms: received data packet 4439 @ 4675375, sent 4690068, delay -14693, delayvar 19, type 193, count 38
2002/03/11 00:33:34.378
ERROR: Omni.Comms: received data packet 4440 @ 4675394, sent 4690073, delay -14679, delayvar 14, type 193, count 39
2002/03/11 00:33:34.387
ERROR: Omni.Comms: remotepacketloss = 0, remotepacketoutoforder = 0, pct = 0.163265, pdt = -0.008576, threshold = 0
2002/03/11 00:33:34.400
ERROR: Omni.Comms: received data packet 4441 @ 4675416, sent 4690078, delay -14662, delayvar 17, type 193, count 40
2002/03/11 00:33:34.434
ERROR: Omni.Comms: received data packet 4442 @ 4675450, sent 4690083, delay -14633, delayvar 29, type 193, count 41
ERROR: Omni.Comms: received data packet 4443 @ 4675450, sent 4690087, delay -14637, delayvar -4, type 194, count 42
2002/03/11 00:33:34.447
ERROR: Omni.Comms: received data packet 4444 @ 4675463, sent 4690088, delay -14625, delayvar 12, type 193, count 43
ERROR: Omni.Comms: received data packet 4445 @ 4675463, sent 4690093, delay -14630, delayvar -5, type 193, count 44
2002/03/11 00:33:34.452
ERROR: Omni.Comms: received data packet 4446 @ 4675468, sent 4690109, delay -14641, delayvar -11, type 194, count 45
2002/03/11 00:33:34.466
ERROR: Omni.Comms: received data packet 4447 @ 4675482, sent 4690135, delay -14653, delayvar -12, type 194, count 46
ERROR: Omni.Comms: received data packet 4448 @ 4675482, sent 4690148, delay -14666, delayvar -13, type 194, count 47
ERROR: Omni.Comms: received data packet 4449 @ 4675482, sent 4690169, delay -14687, delayvar -21, type 194, count 48
2002/03/11 00:33:34.486
ERROR: Omni.Comms: received data packet 4450 @ 4675502, sent 4690171, delay -14669, delayvar 18, type 193, count 49
ERROR: Omni.Comms: packetloss = 0, packetoutoforder = 0, PCT = 0.448980, PDT = 0.120823, count = 50
2002/03/11 00:33:34.517
ERROR: Omni.Comms: received data packet 4451 @ 4675533, sent 4690176, delay -14643, type 193, count 0
2002/03/11 00:33:34.538
ERROR: Omni.Comms: received data packet 4452 @ 4675554, sent 4690181, delay -14627, delayvar 16, type 193, count 1
2002/03/11 00:33:34.545
ERROR: Omni.Comms: received data packet 4453 @ 4675561, sent 4690190, delay -14629, delayvar -2, type 193, count 2
2002/03/11 00:33:34.587
ERROR: Omni.Comms: received data packet 4454 @ 4675603, sent 4690192, delay -14589, delayvar 40, type 194, count 3
```

Figure 5.3: Logs of the PCT and PDT Tests (2)

The logs in Figure 5.3 show the test results when the bandwidth changes to 600 kbit/s. Based on the figure, the one-way delay begins to increase because we send packets at the higher rate than the available bandwidth. The application does not detect the packet loss

although the network is already congested. But both PDT and PCT increase. The packet loss detection is less sensitive than the delay detection when the network condition changes. Since the bandwidth is slightly lower than the sending rate, the change is not too large.

```
2002/03/11 00:33:35.087
ERROR: Omni.Comms: received data packet 4497 @ 4676102, sent 4690624, delay -14522, delayvar 14, type 193, count 44
2002/03/11 00:33:35.111
ERROR: Omni.Comms: received data packet 4498 @ 4676127, sent 4690629, delay -14502, delayvar 20, type 193, count 45
TRACE: Omni.Comms: MasterUser 10.0.0.4:53088 (op4) [chan 1]: Discarding old incomplete message (seqid:2842) after m
ERROR: Omni.Comms: received data packet 4501 @ 4676127, sent 4690644, delay -14517, delayvar -15, type 193, count 4
2002/03/11 00:33:35.132
ERROR: Omni.Comms: received data packet 4502 @ 4676148, sent 4690646, delay -14498, delayvar 19, type 194, count 47
ERROR: Omni.Comms: received data packet 4503 @ 4676148, sent 4690666, delay -14518, delayvar -20, type 194, count 4
ERROR: Omni.Comms: received data packet 4504 @ 4676148, sent 4690687, delay -14539, delayvar -21, type 194, count 4
ERROR: Omni.Comms: packetloss = 4, packetoutoforder = 0, PCT = 0.571429, PDT = 0.155689, count = 50
2002/03/11 00:33:35.134
ERROR: Omni.Comms: received data packet 4505 @ 4676150, sent 4690734, delay -14584, type 193, count 0
2002/03/11 00:33:35.145
ERROR: Omni.Comms: received data packet 4506 @ 4676161, sent 4690735, delay -14574, delayvar 10, type 194, count 1
2002/03/11 00:33:35.146
ERROR: Omni.Comms: received data packet 4507 @ 4676162, sent 4690745, delay -14583, delayvar -9, type 194, count 2
```

Figure 5.4: Logs of the PCT and the PDT Tests (3)

The logs in Figure 5.4 show the next report. The packet loss is now 4. In our application, the sender will receive the report and change the state to VERYBAD, although we set the bandwidth just a little lower than the sending rate. PCT is 0.57, which is relatively high. More than half of packets have a higher delay than preceding packets. This also reflects that the network is congested.

I have summarized the test results in Table 5.1. These results show changes in packet loss, PCT and PDT in six network conditions. I fix the sending rate to the highest level (VERYGOOD and sendTime = 5 ms). So the sending rate is 80 kbyte/s.

- The bandwidth changes from 1 mbit/s to 600 kbit/s

	loss	PCT	PDT
Report before change	0	0.24	-0.05
Report after change	0	0.44	0.12
Next report	4	0.57	0.15
	6	0.57	0.06
	18	0.55	0.05
	12	0.53	-0.04

- The bandwidth changes from 600 kbit/s to 1 mbit/s

	loss	PCT	PDT
Before resume	10	0.53	-0.01
After resume	11	0.38	-0.18
	0	0.46	-0.1
	0	0.24	0.00

- The bandwidth changes from 1 mbit/s to 400 kbit/s

	loss	PCT	PDT
Before change	0	0.28	0.06
After change	11	0.57	0.38
	48	0.53	-0.03
	39	0.57	-0.01

- The bandwidth changes from 400 kbit/s to 1 mbit/s

	loss	PCT	PDT
Before resume	35	0.44	0.04
After resume	22	0.22	-0.45
	0	0.36	0.01
	0	0.32	0.00

- The bandwidth changes from 1 mbit/s to 200 kbit/s

	loss	PCT	PDT
	0	0.3	-0.10
Before change	0	0.32	0.01
After change	49	0.57	0.41
	67	0.38	-0.01

- The bandwidth resumes from 200 kbit/s to 1 mbit/s

	loss	PCT	PDT
Before resume	65	0.48	0.002
After resume	53	0.16	-0.76
	0	0.3	0.00
	0	0.32	-0.05

Table 5.1: PCT and PDT Test Results

I use red to mark high values (PCT > 0.5, PDT > 0.3). These values can help us determine if the one-way delay has an increasing trend, and when the sender must decrease the sending rate. I use blue to mark low values (PCT < 0.2, PDT < –0.3). These values can help us determine whether the one-way delay has a decreasing trend, so that the sender could increase the sending rate.

From the preceding analysis, we can see that PCT and PDT can determine the trend of one-way delay variation. Based on PCT and PDT, the project can detect the current network condition.

I choose the threshold value of the PCT and PDT tests based on these test results:

- If PCT > 0.52 or PDT > 0.20, the delay variation has an increasing trend.
- If the PDT < –0.25, the delay variation has a decreasing trend.
- If the delay variation has neither an increasing nor a decreasing trend, we say that the one-way delay has an ambiguous trend.

While adding these new feedback parameters, I do not discard the old ones. When PDT and the PCT measurements determine an ambiguous trend, the application determines the

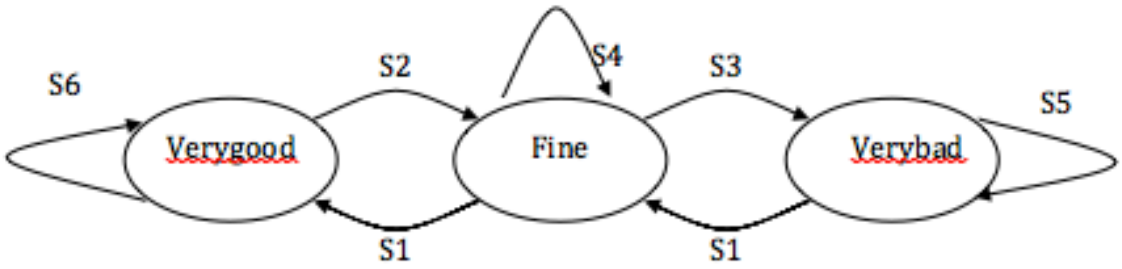
current network status based on PacketLoss parameters. For every 50 packets, the receiver sends a report including PCT, PDT and PacketLoss.

5.4 New Rate Control Algorithm

In this section I design a new sending rate control algorithm by using delay-based feedback parameters (the PCT and PDT tests I described in Section 5.1). I implement the new algorithm within our application. Based on test results, the new algorithm improves the application's performance compared with the original algorithm.

5.4.1 The New Algorithm

I designed a new algorithm using both delay-based and packet-loss-based feedback parameters. Here, I first describe the new algorithm and then provide test results. I then modify the state diagram in the application. The new state diagram is described in Figure 5.5.



Network is under-used: the one-way delay has decreasing trend
or (the one-way delay is ambiguous and threshold is equal to zero)

Network is over-used: the one-way delay has increasing trend
or (the one-way delay is ambiguous and threshold is greater than zero)

Threshold=packet loss+ 0.6 *packet out of order (in every 50 packets)

S1: network is under-used and the state_up counter is equal to zero

S2: network is over-used

S3: network is over-used and the state_down counter is equal to zero

S4: if network is over-used and the state_down counter is not equal to zero, the state_down counter decrements by one. The state_up counter is reset to 40.

If network is under-used and the state_up counter is not equal to zero, the state_up counter decrements by 5 when the one-way delay has decreasing trend, or it decrements by one when the threshold is equal to zero. The state_down counter is reset to 10.

S5: if network is over-used, the state_down counter is reset to 10. The state_up counter is reset to 40.

If network is under-used and the state_up counter is not equal to zero, the state_up counter decrements by 5 when the one-way delay has decreasing trend, or it decrements by one when the threshold is equal to zero. The state_down counter is reset to 10.

S6: if network is under-used, the state_down counter is reset to 10. The state_up counter is reset to 40.

Figure 5.5: The New State Diagram

I use two counters (state_up counter and state_down counter) to control state changes. The default value of the state_up counter is 40, whereas tthat of the state_down counter is 10. When the current state is VERYGOOD, if the sender gets a "network underused" report, the next state is also VERYGOOD. The program then resets both counters to their default values. If the sender gets a "network overused" report, the network state changes to FINE. Subsequently, the program resets both counters to their default values. The process is shown in Figure 5.6.

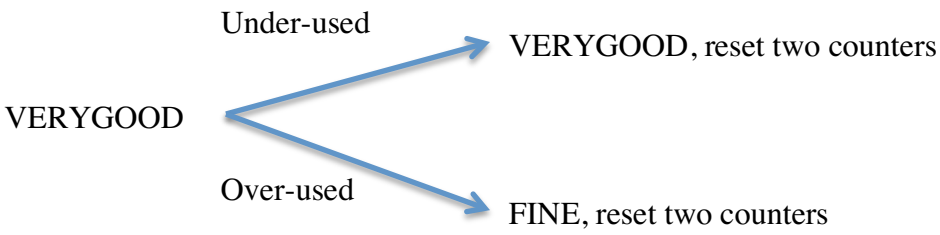


Figure 5.6: New State Transition of the VERYGOOD State

The current state is FINE. If the sender gets the "network underused" report, the program does not change the state to VERYGOOD directly. If the one-way delay has a decreasing trend, the state_up counter decreases by 5. If the one-way delay is ambiguous and there is no loss, the state_up counter decreases by 1. Then, if the state_up ≤ 0, the next state is VERYGOOD, and both counters are reset to their default values. If the state_up counter is larger than 0, the state remains FINE, and the state_down counter is reset to its default value. The process is shown in Figure 5.7.

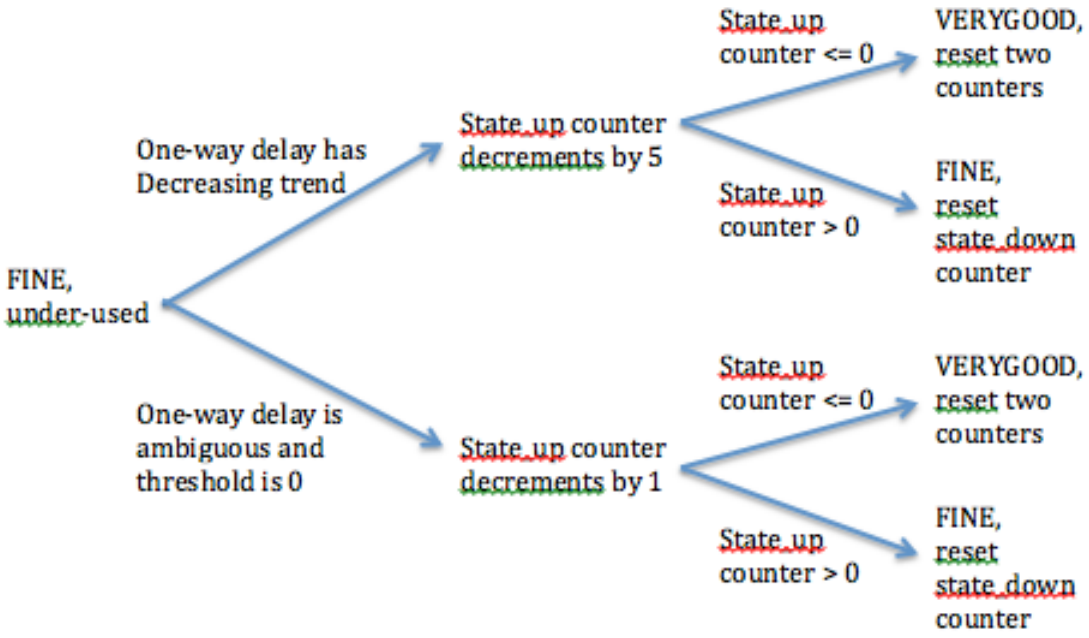


Figure 5.7: New State Transition of the FINE State (1)

The current network state is still FINE. If the sender gets a "network overerused" report, the state_down counter decreases by 1. Then if the state_down counter ≤ 0, the next state is VERYBAD, and the program resets both counters to their default values. If the state_down counter > 0, the next state stays FINE, and the state_up counter is reset to its default value. The process is shown in Figure 5.8.

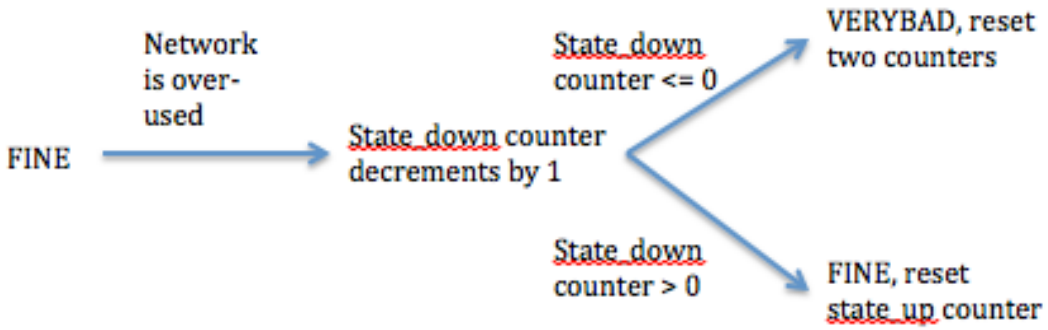


Figure 5.8: New State Transition of the FINE State (2)

The current network state is VERYBAD. If the sender gets a "network overused" report, the next state is VERYBAD. Both counters are reset to their default values. If the sender gets a "network underused" report, the program does not directly change the state to FINE. If the one-way delay has a decreasing trend, the state_up counter decreases by 5. If the one-way delay is ambiguous and there is no loss, the state_up counter decreases by 1.

Then, if the state_up counter is ≤ 0, the next state is FINE, and both counters are reset to their default values. If the state_up counter > 0, the next state is VERYBAD, and the state_down counter is reset to its default value. The process is shown in Figure 5.9.

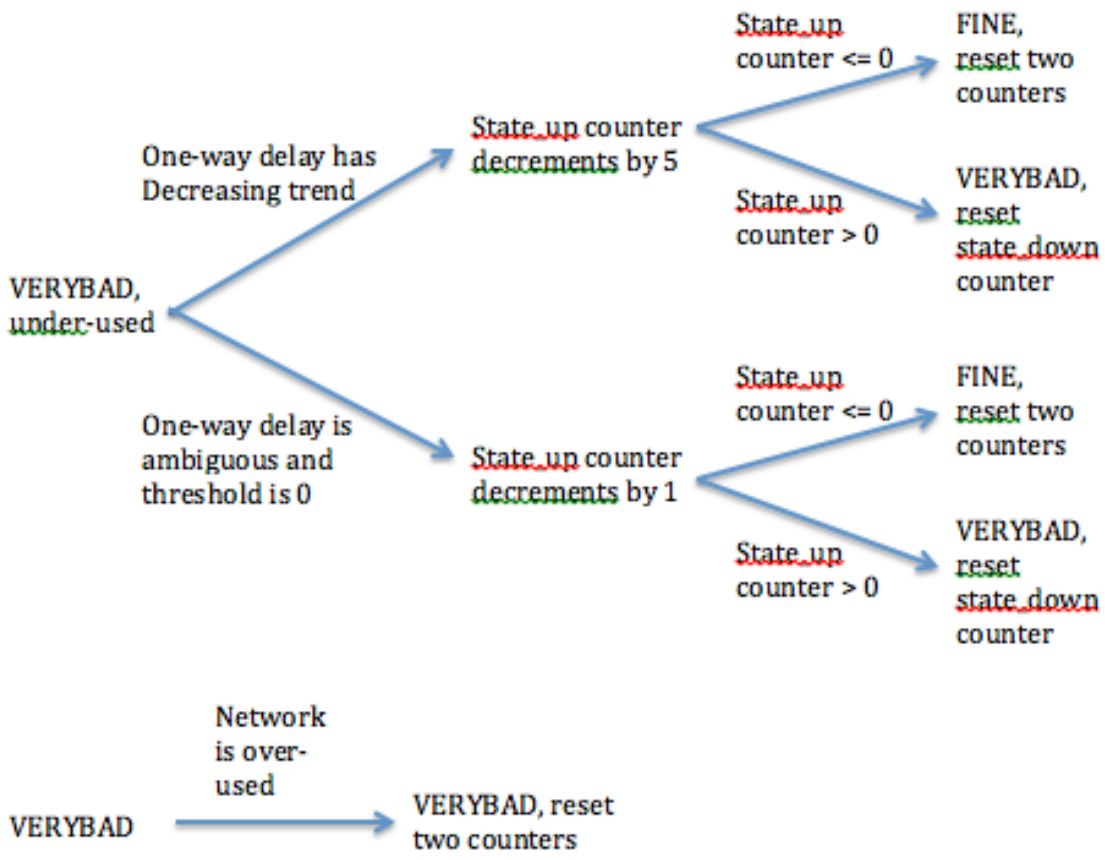


Figure 5.9: New State Transition of the VERYBAD State

I have added and modified codes that implement the recently described algorithm in the handleFCStatistics member function of the OCConnection class. The modified algorithm has several advantages:

- The new algorithm can use the FINE state more often. The original one rarely stays in the FINE state.
- When the current state is FINE or VERYBAD, if the sender gets a "network underused" report, the program does not immediately increases the sending rate (change the state). The program increases the sending rate only when the sender

gets enough continuous “network underused” reports. Then the program tries to increase the rate. So the sending rate will not fluctuate as sharply as in the original algorithm; it stays in one stable state for some time.

- The weights of one-way delay-decreasing trend and no packet loss are different. If the sender gets a “one-way delay-decreasing trend” report, the state_up counter decreases by 5. If the sender gets a “threshold is equal to zero” report, the state_up counter decreases by 1. So one “one-way delay-decreasing trend” report is equal to five “threshold is equal to zero” reports. If there is one-way delay-decreasing trend, the program tries to change the state in smaller increments.
- When the current state is VERYGOOD, if the sender gets a “network overused” report, the sender immediately changes the state to FINE. There are two benefits of this strategy. First, the VERYGOOD and FINE states have a great difference in sending rates (the sending rate in the VERYGOOD state is around 75 kbyte/s, whereas the sending rate in the FINE state is around 35 kbyte/s). When the current network state is VERYGOOD, if the network is overused, the program rapidly decreases the sending rate.
- When the current state is FINE, if the sender gets a “network overused” report, the sender does not immediately change the state to VERYBAD. This strategy can help our application to compete for bandwidth with a TCP flow. When there is a TCP flow, the program can stay in the FINE state and get more bandwidth than the original algorithm.
- The default values of state_up and state_down counters are different (40 and 10, respectively). The state_up value is 40 because the program tries to increase the

state over a long time period to make the sending rate stable. The state_down value is 10 to help the program to compete for bandwidth with TCP flows. I used different values of two counters to run the experiments. These two values achieved the best results.

When the program starts to run, the state is FINE. The values of both counters are set to 10. So the sending rate can increase to the maximum value quickly (get 10 "network underused" reports, or less when there are "one-way delay-decreasing trend" reports) when the program starts to run in networks with enough bandwidth.

5.4.2 Test Results

1) Bandwidth changes from 1 mbit/s to 400 kbit/s

First, I run the program when the bandwidth decreases from 1 mbit/s to 400 kbit/s. Figures 5.10 and 5.11 show the test results of the original algorithm and the new algorithm, respectively. In the original algorithm test, I change the bandwidth at approximately 20 s. In the new algorithm test, I change the bandwidth to approximately 60 s. I did not change the bandwidth immediately after the application started in the two tests because the algorithms are in a steady state before and after the bandwidth changes.

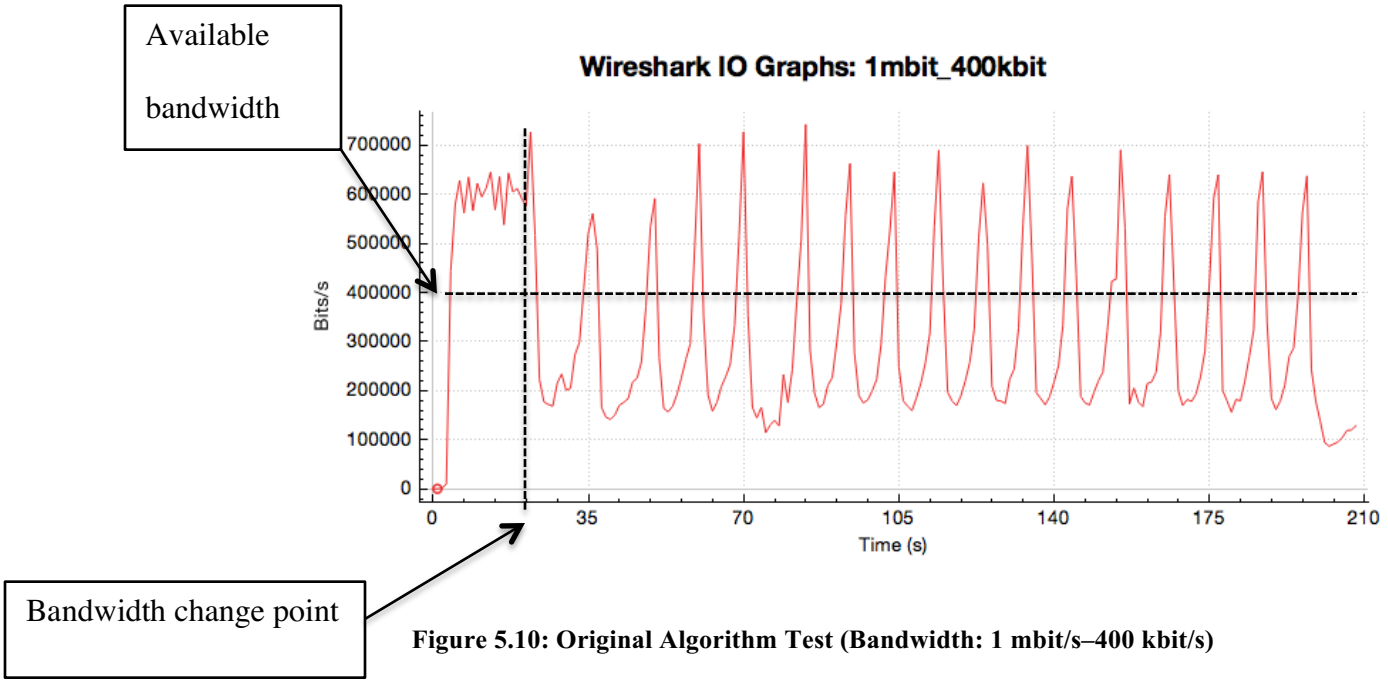


Figure 5.10: Original Algorithm Test (Bandwidth: 1 mbit/s–400 kbit/s)

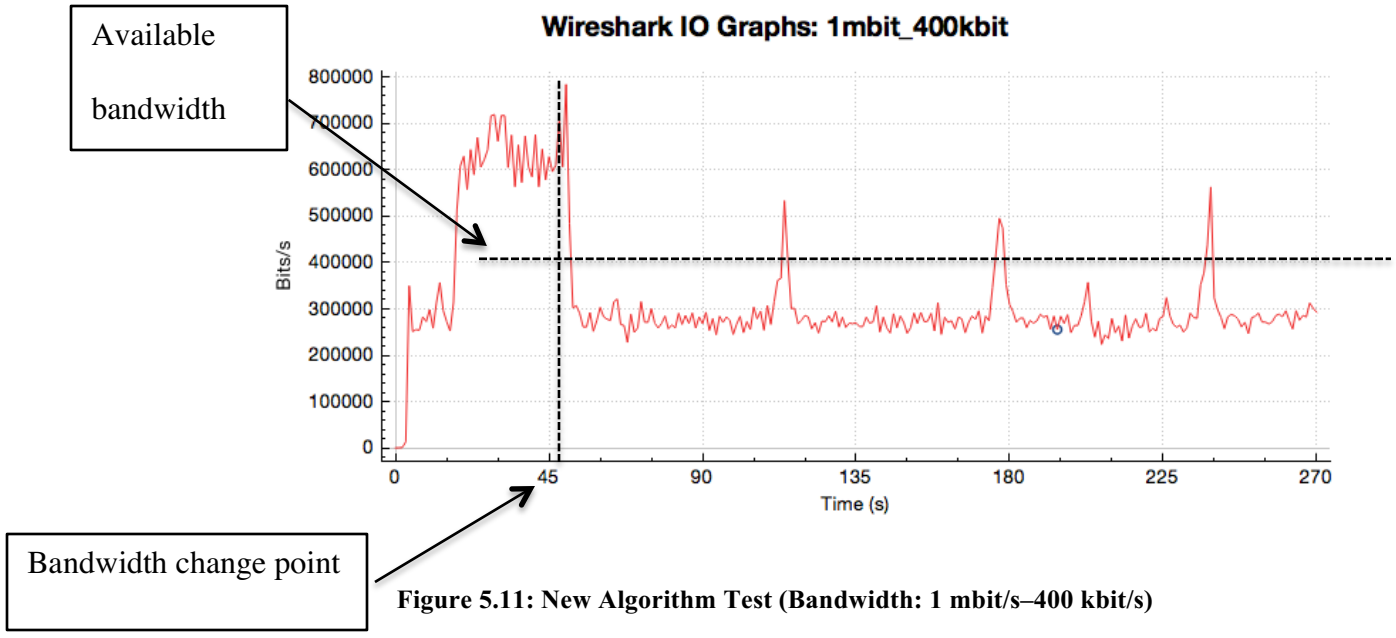


Figure 5.11: New Algorithm Test (Bandwidth: 1 mbit/s–400 kbit/s)

In the original algorithm, the sending rate fluctuates sharply and periodically between 100 kbit/s and 700 kbit/s. The sending rate often goes above the available bandwidth (400 kbit/s). This causes network congestion and many packet losses. From the user's side, the video frames are unstable, frozen and choppy.

In the new algorithm, the sending rate stays at 300 kbit/s after the bandwidth is decreased to 400kbit/s. The program tries to increase the sending rate over a long time period (about 60s in Figure 5.11). This is because the program changes the state from FINE to VERYGOOD only when it gets enough "network underused" reports. When it tries to increase the sending rate, the program faces the bandwidth limitation (400 kbit/s). The sending rate rapidly changes to 300 kbit/s, as shown by a small spike in Figure 5.11. The sending rate resumes rapidly because the state changes from VERYGOOD to FINE when one "network underused" report appears. So the new algorithm has less packet losses compared with the original algorithm, and the network is not congested for a long time. From the user's side, the video frames are stable and smooth, so the video not get as frozen and choppy as that from the original algorithm.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.24	-0.07	Ambiguous	FINE	9	10
2	0	0.30	0.00	Ambiguous	FINE	8	10
3	0	0.32	0.00	Ambiguous	FINE	7	10
4	0	0.32	0.00	Ambiguous	FINE	6	10
5	0	0.30	-0.02	Ambiguous	FINE	5	10
6	0	0.30	0.00	Ambiguous	FINE	4	10
7	0	0.38	-0.01	Ambiguous	FINE	3	10
8	0	0.34	0.05	Ambiguous	FINE	2	10
9	0	0.26	-0.03	Ambiguous	FINE	1	10
10 (state change)	0	0.40	0.02	Ambiguous	FINE	0	10
11	0	0.40	0.02	Ambiguous	VERYGOOD	40	10

Table 5.2: Test Log of the New Algorithm (1)

The test log summarized in Table 5.2 shows the starting process. The program changes the state from FINE to VERYGOOD after getting 10 “network underused” reports. The initial value of the state_up counter is set to 10 so that it takes less time for the program to get to the VERYGOOD state.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.28	-0.01	Ambiguous	VERYGOOD	40	10
2	0	0.71	0.45	Increase	VERYGOOD	40	10
3	12	0.55	0.09	Increase	FINE	40	9

Table 5.3: Test Log of the New Algorithm (2)

The test log summarized in Table 5.3 shows the bandwidth-changing process. When the network bandwidth changes from 1 mbit/s to 400 kbit/s, the receiver first detects the one-way delay-increasing trend. The second report in Table 5.3 shows a one-way delay-increasing trend. PCT is 0.71, and PDT is 0.45 (higher than 0.52 and 0.20, respectively). The third report shows both a one-way delay-increasing trend and packet loss. The one-way delay-increasing trend is detected earlier than the packet loss when the network is congested. This is one benefit of using delay-based feedback. When there is a “network overused” report, the state immediately changes from VERYGOOD to FINE.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	12	0.55	0.09	Increase	FINE	40	9
2	0	0.55	-0.16	Increase	FINE	40	8
3	0	0.36	-0.5	Decrease	FINE	35	10
4	0	0.34	0.01	Ambiguous	FINE	34	10
5	0	0.34	-0.01	Ambiguous	FINE	33	10
6	0	0.32	0.02	Ambiguous	FINE	32	10

Table 5.4: Test Log of the New Algorithm (3)

The test log summarized in Table 5.4 shows the process after the state changes from VERYGOOD to FINE. After the state changes to FINE, the sending rate is less than the available bandwidth (400 kbit/s). The third report in Table 5.4 shows that the one-way delay has a decreasing trend (PDT is −0.5). So the state_up counter decreases by 5. The report after that shows that the one-way delay has neither an increasing nor a decreasing trend (ambiguous). There is no packet loss. So the state_up counter decreases by 1. The default value of the state_up counter is 40.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.34	0.00	Ambiguous	FINE	1	10
2	0	0.38	-0.01	Ambiguous	FINE	0	10
3 (state change)	0	0.34	0.02	Ambiguous	VERYGOOD	40	10
4	0	0.38	-0.02	Ambiguous	VERYGOOD	40	10
5	0	0.63	0.51	Increase	VERYGOOD	40	10
6 (state change)	0	0.51	-0.10	Ambiguous	FINE	39	10

Table 5.5: Test Log of the New Algorithm (4)

The test log summarized in Table 5.5 shows the process when the program tries to increase the sending rate. The up_state counter goes to zero in the second report in Table 5.5. This means that the receiver has already received enough continuous "network underused" reports. So the program changes the state from FINE to VERYGOOD to increase the sending rate, but the available bandwidth is 400 kbit/s. The sending rate faces the bandwidth limitation. In the 5^{th} report, a one-way delay-increasing trend is detected. The state changes to FINE although there is no packet loss detected. The state stays VERYGOOD only during three reports, so the network is not congested. There are no packet losses during the increasing process. Our user does not get frozen images of the video. This is a major improvement compared with the original algorithm.

2) Bandwidth changes from 400 kbit/s to 200 kbit/s

s 5.12 and 5.13 show test results when the bandwidth changes from 400 kbit/s to 200 kbit/s. In the original algorithm test, I change the bandwidth at approximately 28 s; in the new algorithm test, I change the bandwidth at 40 s.

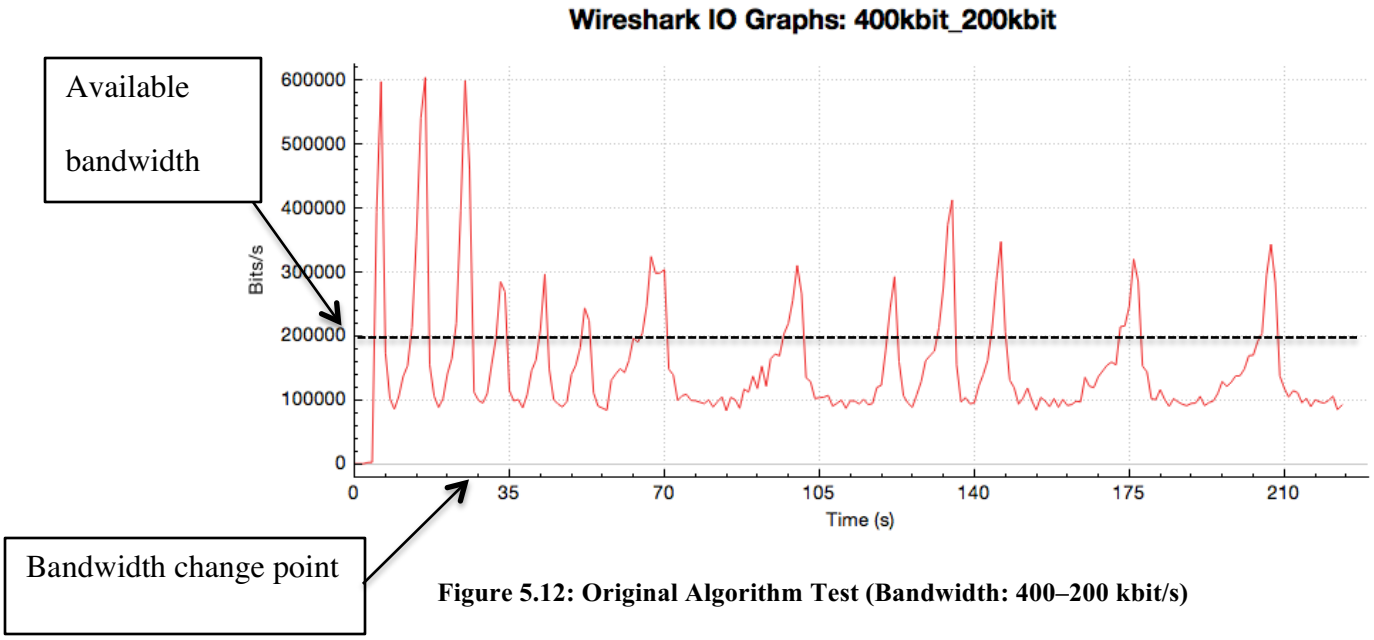


Figure 5.12: Original Algorithm Test (Bandwidth: 400–200 kbit/s)

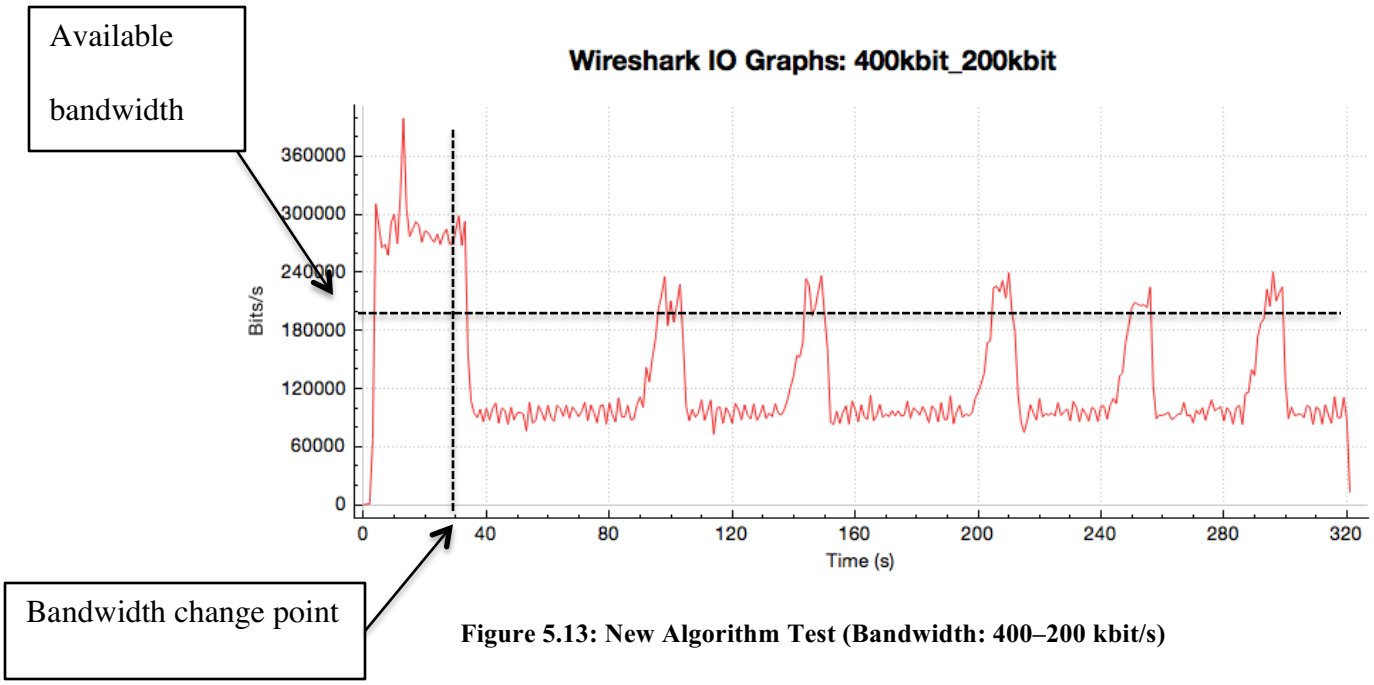


Figure 5.13: New Algorithm Test (Bandwidth: 400–200 kbit/s)

In the original algorithm, when the bandwidth is 400 kbit/s, the sending rate fluctuates sharply between 150 kbit/s and 650 kbit/s. When the bandwidth reduces to 200 kbit/s, the sending rate fluctuates between 100 kbit/s and 500 kbit/s. The user experiences an unstable and choppy video. Using our new algorithm, the program has a more stable sending rate. After the network bandwidth reduces to 200 kbit/s, the sending rate stays between 100 kbit/s and 200 kbit/s. The program tries to increase the sending rate over long time intervals. In Figure 5.13, sometimes the time interval is around 60 s, and sometimes it is around 40 s because reports that show a one-way delay-decreasing trend speed up the counter decrement. The program tries to increase the sending rate, which is higher than the bandwidth. But the sending rate stays higher than the bandwidth for some time because the receiver needs to get 10 ”network overused” reports to change the state from FINE to VERYBAD. The program needs this strategy to compete for bandwidth with a TCP flow. But this also causes some packet losses. Therefore, there is a trade-off between the sending rate and packet losses.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.36	0.46	Increase	FINE	40	9
2	2	0.55	0.55	Increase	FINE	40	8
3	23	0.40	0.03	Ambiguous	FINE	40	7
4	20	0.46	0.04	Ambiguous	FINE	40	6
5	18	0.34	0.09	Ambiguous	FINE	40	5
6	25	0.51	0.00	Ambiguous	FINE	40	4
7	15	0.53	-0.05	Increase	FINE	40	3
8	17	0.36	0.07	Ambiguous	FINE	40	2
9	19	0.44	0.00	Ambiguous	FINE	40	1
10	15	0.38	-0.01	Ambiguous	FINE	40	0
11 (state change)	20	0.28	-0.10	Ambiguous	VERYBAD	40	10

Table 5.6: Test Log of the New Algorithm (5)

The test log summarized in Table 5.6 shows the process when the bandwidth is reduced from 400 kbit/s to 200 kbit/s. The first report in Table 5.6 indicates that the one-way delay has an increasing trend. The following 10 reports all show that the network is overused. When the sender receives one ”network overused” report, the state_down

counter decreases by one. The state_down counter decreases to 0 in the 10th report in Table 5.6, and then the state changes from FINE to VERYBAD. The default value of the state_down counter is 10.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.32	-0.02	Ambiguous	VERYBAD	2	10
2	0	0.24	0.00	Ambiguous	VERYBAD	1	10
3	0	0.26	0.00	Ambiguous	VERYBAD	0	10
4 (state change)	0	0.26	0.00	Ambiguous	FINE	39	10
5	0	0.22	0.03	Ambiguous	FINE	38	10

Table 5.7: Test Log of the New Algorithm (6)

The test log summarized in Table 5.7 shows the process when the program tries to increase the sending rate. After the state changes from FINE to VERYBAD, the sending rate is less than the available bandwidth. The sender then receives ”network underused” reports. After the sender gets enough ”network underused” reports, the state_up counter goes to 0. So the program changes the state from VERYBAD to FINE shown in the 4th report in Table 5.7.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.40	0.12	Ambiguous	FINE	34	10
2	0	0.38	0.20	Increase	FINE	40	9
3	0	0.48	0.48	Increase	FINE	40	8
4	7	0.46	0.03	Ambiguous	FINE	40	7
5	12	0.53	0.08	Increase	FINE	40	6
6	13	0.36	-0.02	Ambiguous	FINE	40	5
7	5	0.46	0.01	Ambiguous	FINE	40	4
8	8	0.40	0.04	Ambiguous	FINE	40	3
9	15	0.53	0.00	Increase	FINE	40	2
10	8	0.38	0.00	Ambiguous	FINE	40	1
11	13	0.40	0.02	Ambiguous	FINE	40	0
12 (state change)	13	0.35	0.04	Ambiguous	VERYBAD	40	10

Table 5.8: Test Log of the New Algorithm (7)

The test log summarized in Table 5.8 shows the process after the state changes from VERYBAD to FINE. After the state changes to FINE, the sending rate is higher than the bandwidth (200 kbit/s). So the sender gets ”network overused” reports. Unlike the

process that occurred in the VERYGOOD state, the state does not change to VERYBAD until the sender gets 10 ”network overused” reports. This strategy helps the program to compete for bandwidth with a TCP flow, although the strategy also causes some packet losses.

3) Bandwidth changes from 200 kbit/s to 400 kbit/s

Figures 5.14 and 5.15 show test results when the bandwidth changes from 200 kbit/s to 400 kbit/s. In the original algorithm test, I change the bandwidth at around 60 s; in the new algorithm test, I change the bandwidth at 50 s.

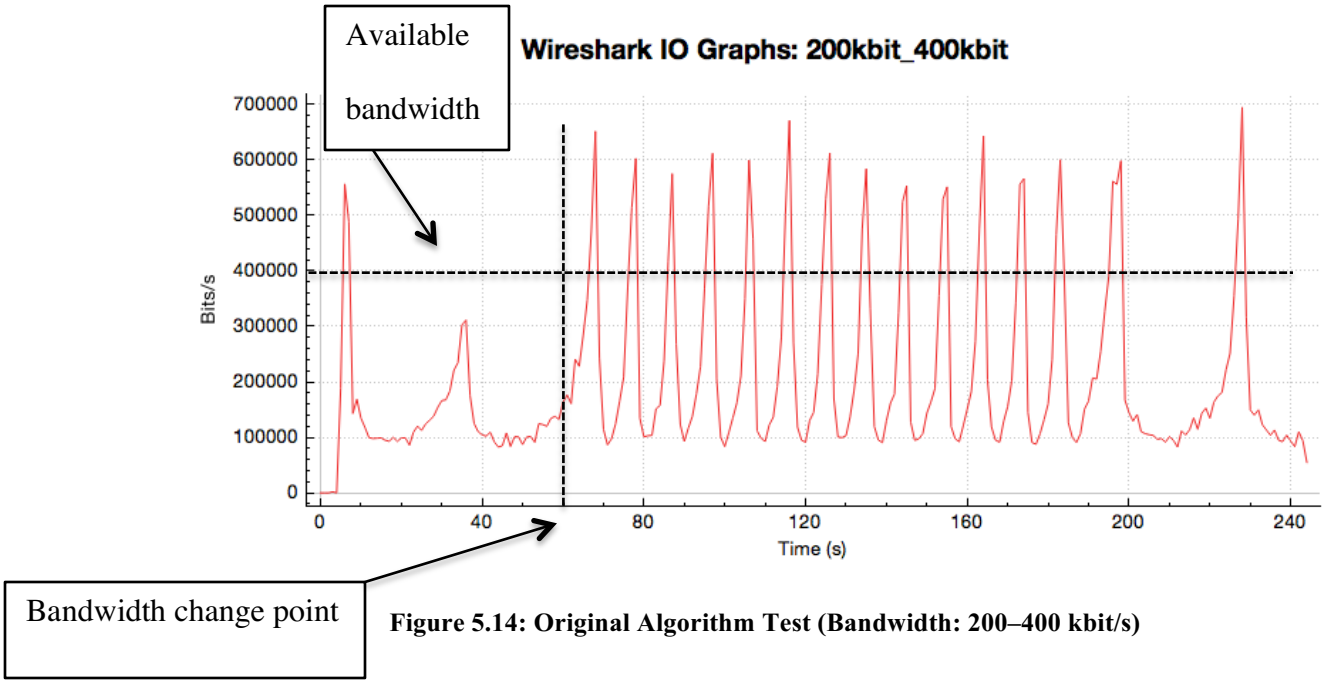


Figure 5.14: Original Algorithm Test (Bandwidth: 200–400 kbit/s)

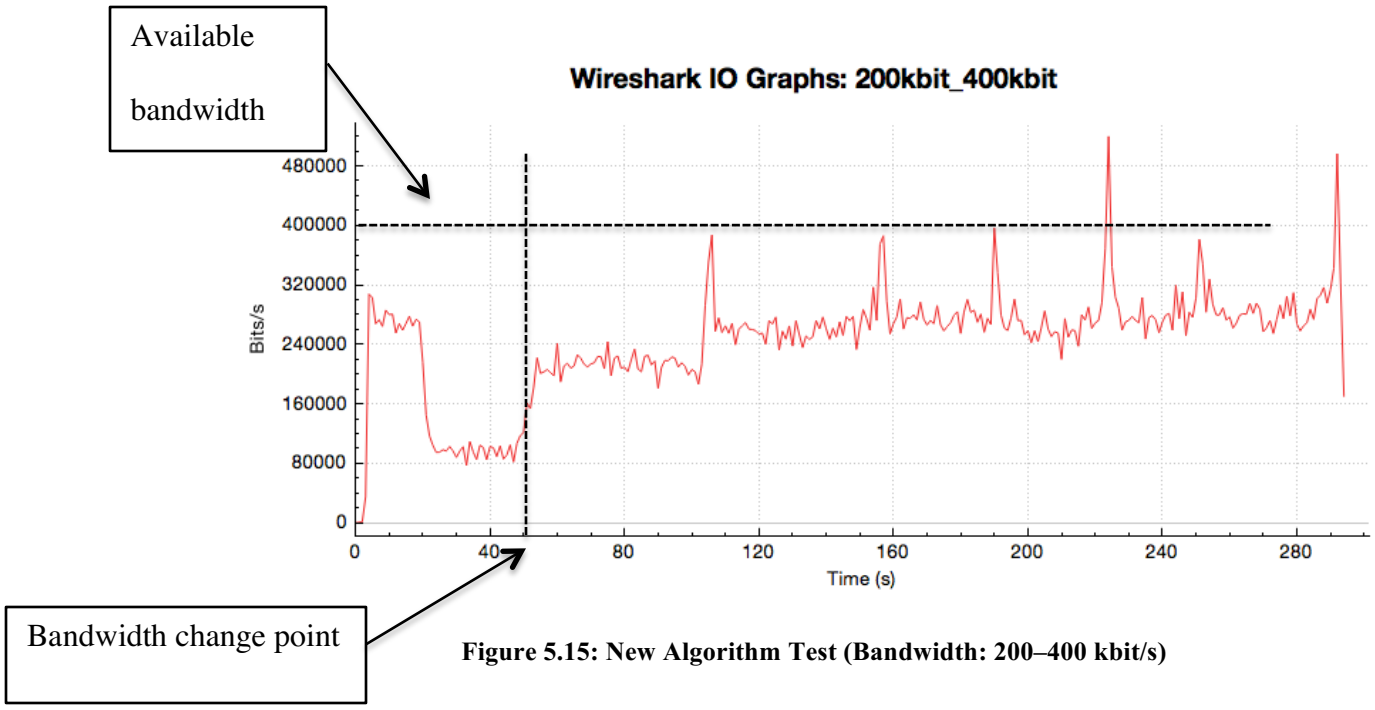


Figure 5.15: New Algorithm Test (Bandwidth: 200–400 kbit/s)

The sending rate fluctuates sharply and periodically in the original algorithm. The sending rate changes between 100 kbit/s and 600 kbit/s (the minimum point and the maximum point of the sending rate, respectively. The application's maximum rate is 80 kbyte/s. In the test results, the maximum rate stays around 600–700 kbit/s). When the program runs with the new algorithm, the sending rate is stable between 300 kbit/s and 400 kbit/s (the bandwidth is 400 kbit/s).

4) Bandwidth changes from 400 kbit/s to 1 mbit/s

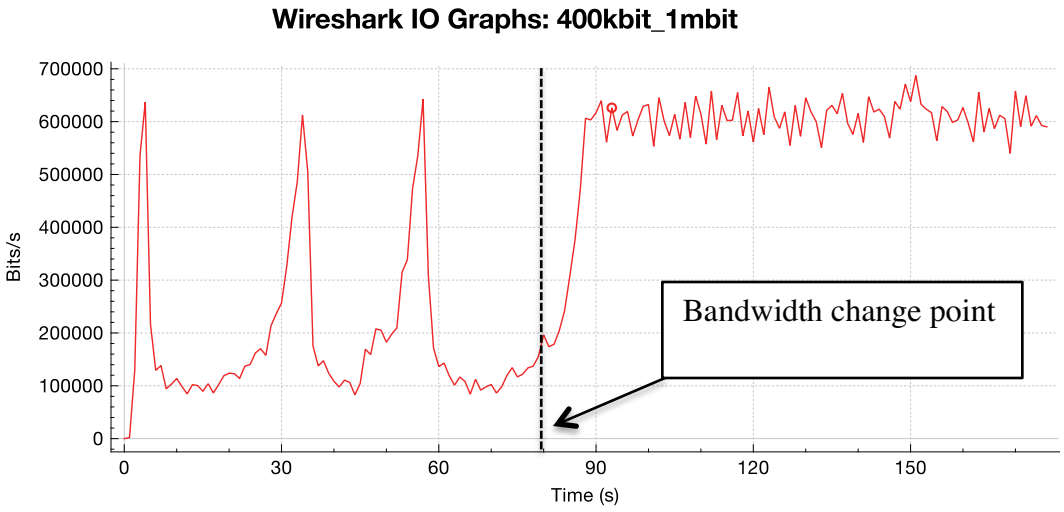


Figure 5.16: Original Algorithm Test (Bandwidth: 400 kbit/s–1 mbit/s)

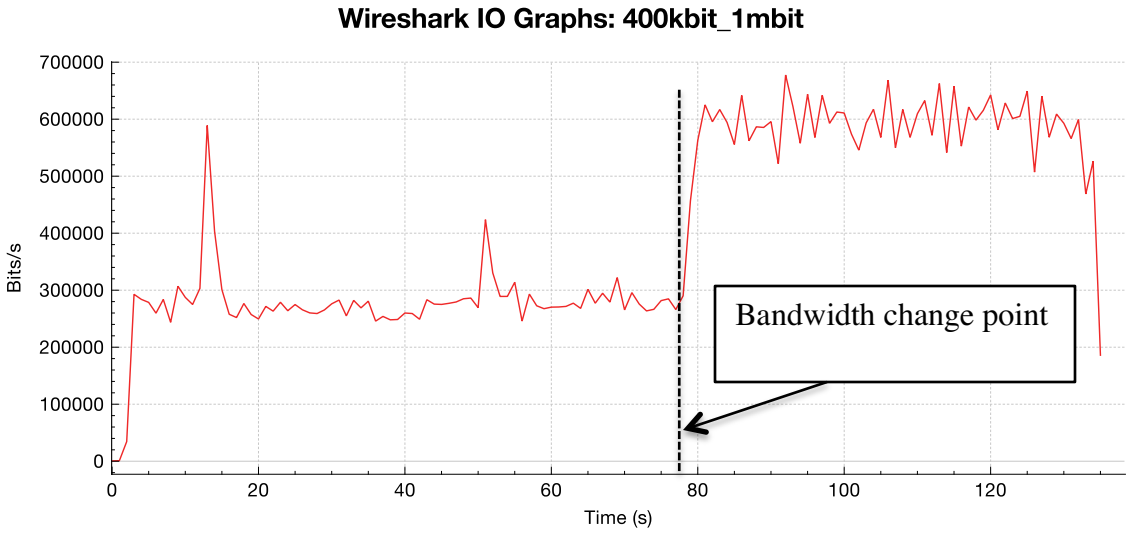


Figure 5.17: New Algorithm Test (Bandwidth: 400 kbit/s–1 mbit/s)

After the bandwidth changes from 400 kbit/s to 1 mbit/s, the sending rates of both the original and the new algorithms change to around 600 kbit/s (the maximum value of the sending rate). Before the bandwidth changes, the sending rate of the original algorithm fluctuates sharply, but the sending rate of the new algorithm remains stable.

5) Bandwidth changes from 1 mbit/s to 200 kbit/s

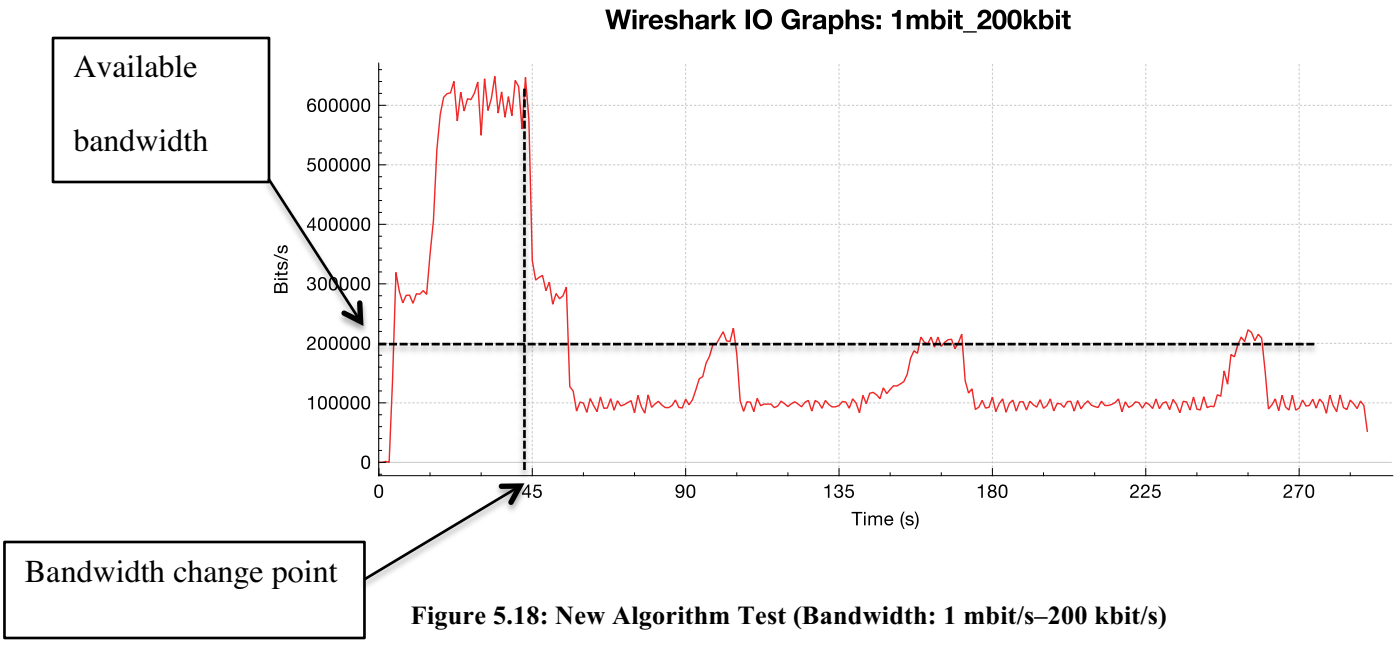


Figure 5.18: New Algorithm Test (Bandwidth: 1 mbit/s–200 kbit/s)

In the new algorithm, when the program starts to run, the initial state is FINE. The initial value of state_up counter is 10. So the sending rate stays at 300 kbit/s for some time. When the sender gets 10 ”network underused” reports, the state is changed to VERYGOOD. The sending rate increases to the maximum value.

When the bandwidth decreases to 200 kbit/s at 45 s, the state changes from VERYGOOD to FINE after the sender gets a ”network overused” report. The state then remains in FINE. After the sender gets 10 continuous ”network overused” reports, the state changes from FINE to VERYBAD, and the sending rate remains at 300 kbit/s for some time during the decreasing process. After the state changes to VERYBAD, the network is

underused. The sender tries to increase the sending rate after it gets enough ”network underused” reports.

6) The program competes for bandwidth with a TCP flow over a 1 mbit/s network.

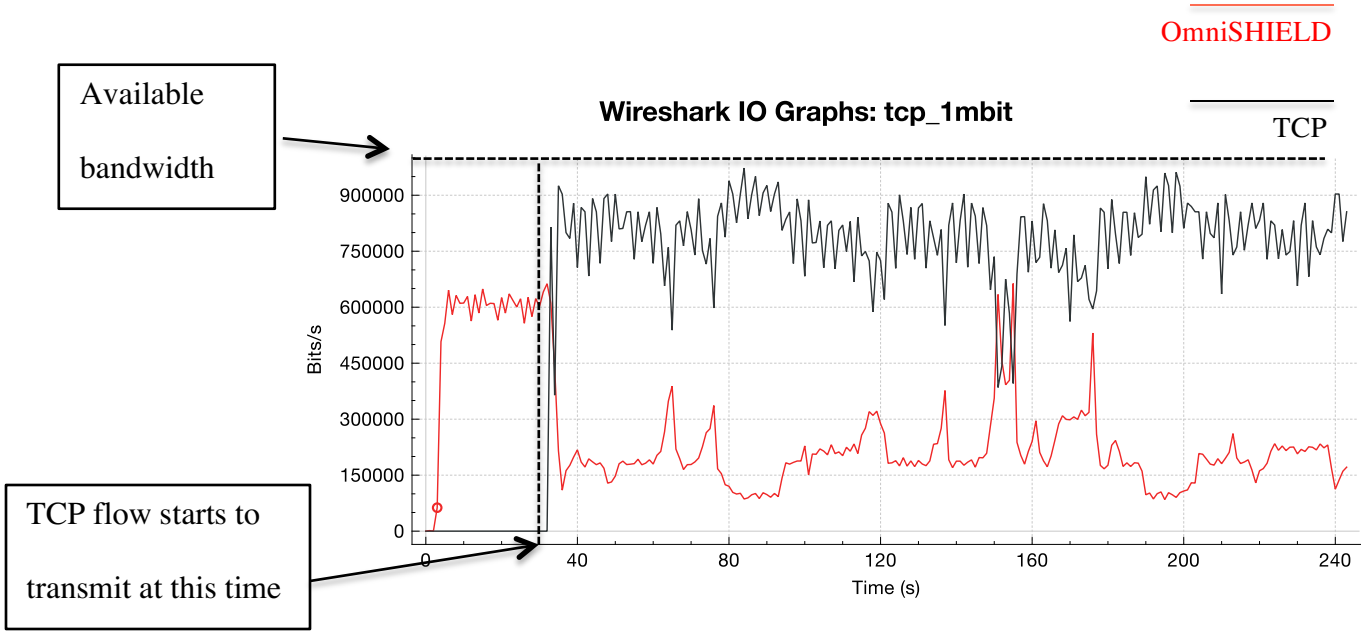


Figure 5.19: Original Algorithm Test (Runs with the TCP Flow in a 1 mbit/s Network)

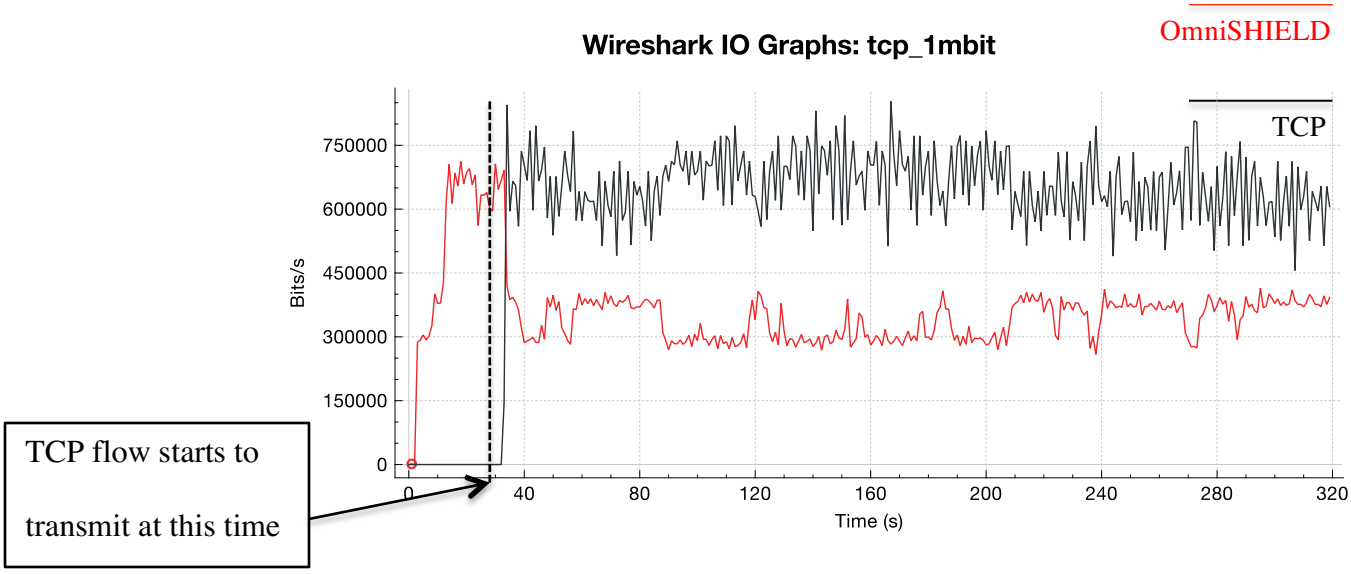


Figure 5.20: New Algorithm Test (Runs with the TCP Flow in a 1 mbit/s Network)

When a video flow and a TCP flow are in the same network, the program that uses the original algorithm does not compete for bandwidth at all. In Figure 5.19 and 5.20, the black line is TCP flow, and the red line is UDP flow. With the original algorithm, the video flow gets much less bandwidth than the TCP flow. After the video flow runs at the minimum point for some time, the sender detects no packet loss and then tries to increase the sending rate. The TCP flow decreases the rate. But then the sender gets a "packet loss" report. The sending rate decreases to the minimum value again. Concurrently, TCP also detects the packet loss and reduces the rate. But the TCP flow recovers rapidly (because the video rate decreases, there is some spare network bandwidth). Therefore, the TCP congestion-control algorithm is much more aggressive than our application's original algorithm. When facing a packet loss, the TCP flow also reduces the sending rate, but the rate does not reach the minimum value. The TCP sending rate recovers rapidly if there is no packet loss. Our application's original algorithm is too conservative. Every time the sender receives a packet loss report, the state turns to VERYBAD, and the sending rate immediately decreases to the minimum value. After the sender runs at this minimum rate and gets four "no packet loss" reports, the sender turns the state to FINE and tries to increase the rate. But if the sender gets a "packet loss" report when increasing the rate (the rate is above the available bandwidth), the rate goes to the minimum again.

In contrast, the new algorithm competes more aggressively for bandwidth with the TCP flow. In 5.27, the state remains FINE when there is a TCP flow in the 1 mbit/s bandwidth network. So the sending rate of the video flow remains between 300 kbit/s and 400 kbit/s. The rate does not decrease to the minimum point as in the original algorithm.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.34	0.26	Increase	VERYGOOD	40	10
2	1	0.46	0.32	Increase	FINE	40	9

Table 5.9: Test Log of the New Algorithm (8)

The test log summarized in Table 5.9 shows the process when I add the TCP flow to the network using IPERF. The first report in Table 5.9 shows an increasing trend of delay variation because PDT is 0.26 (above 0.20). Therefore, the sender changes the state from VERYGOOD to FINE.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	0	0.46	0.02	Ambiguous	FINE	38	10
2	1	0.36	0.10	Ambiguous	FINE	40	9
3	2	0.42	-0.08	Ambiguous	FINE	40	8
4	0	0.32	-0.15	Ambiguous	FINE	39	10

Table 5.10: Test Log of the New Algorithm (9)

The test log summarized in Table 5.10 shows the process after the state changes to FINE. As there is a TCP flow, the sender gets some "network overused" reports. Unlike in the original algorithm, the sender does not immediately change the state from FINE to VERYBAD. The sender needs 10 continuous "network overused" reports. In Table 5.10, the sender gets two reports including packet loss. The state_down counter decreases to 8, but the 4th report shows no packet loss. The state_down counter is reset to 10. With this strategy, the state remains in FINE when the application compete the bandwidth with the TCP flow.

7) The program competes for bandwidth with a TCP flow in a 500 kbit/s network.

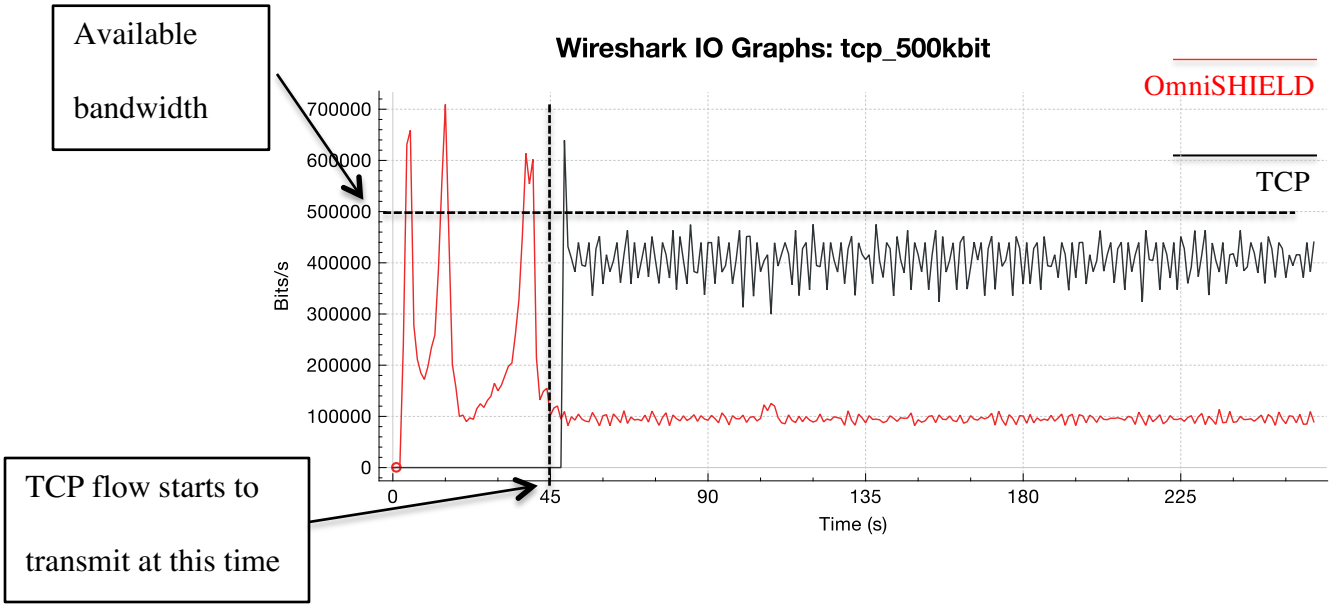


Figure 5.21: Original Algorithm Test (Runs with the TCP Flow in a 500 kbit/s Network)

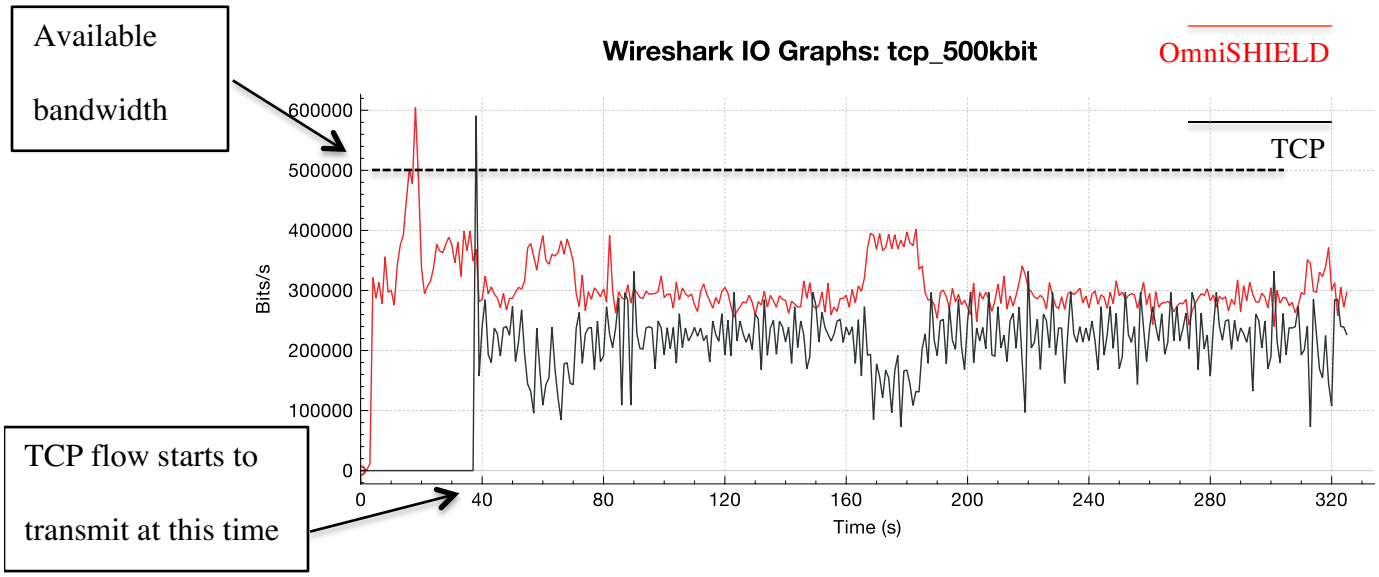


Figure 5.22: New Algorithm Test (Runs with the TCP Flow in a 500 kbit/s Network)

When a TCP and a video flow are in the same network with a 500kbit/s bandwidth, the sending rate of the original algorithm is very low. The sending rate stays around 100 kbit/s (the minimum value). The TCP flow occupies the 400 kbit/s bandwidth.

With the new algorithm, the sending rate remains around 300 kbit/s. The state remains FINE, so the program competes for bandwidth with the TCP flow. Figure 5.21 and 5.22 show the test results of the old and new algorithm performances.

Report packet	Packet loss	PCT	PDT	Trend of delay variation	State	State_up counter	State_down counter
1	5	0.46	0.04	Ambiguous	FINE	40	9
2	3	0.44	-0.07	Ambiguous	FINE	40	8
3	1	0.32	0.07	Ambiguous	FINE	40	7
4	1	0.30	-0.12	Ambiguous	FINE	40	6
5	2	0.30	0.11	Ambiguous	FINE	40	5
6	0	0.22	-0.32	Decrease	FINE	35	10

Table 5.11: Test Log of the New Algorithm (10)

When the total bandwidth is 500 kbit/s, the sender gets more "network overused" reports than in the preceding scenario (where the total bandwidth is 1 mbit/s). The test log summarized in Table 5.11 shows that the sender receives five continuous reports that indicate there are packet losses in the network. In the sixth report in Table 5.11, the delay variation shows a decreasing trend, so the state_down counter is reset to 10. With this strategy, the state stays in FINE and the sending rate remains around 300 kbit/s when there is another TCP flow in the network.

8) The program competes for bandwidth with a TCP flow in a 300 kbit/s network.

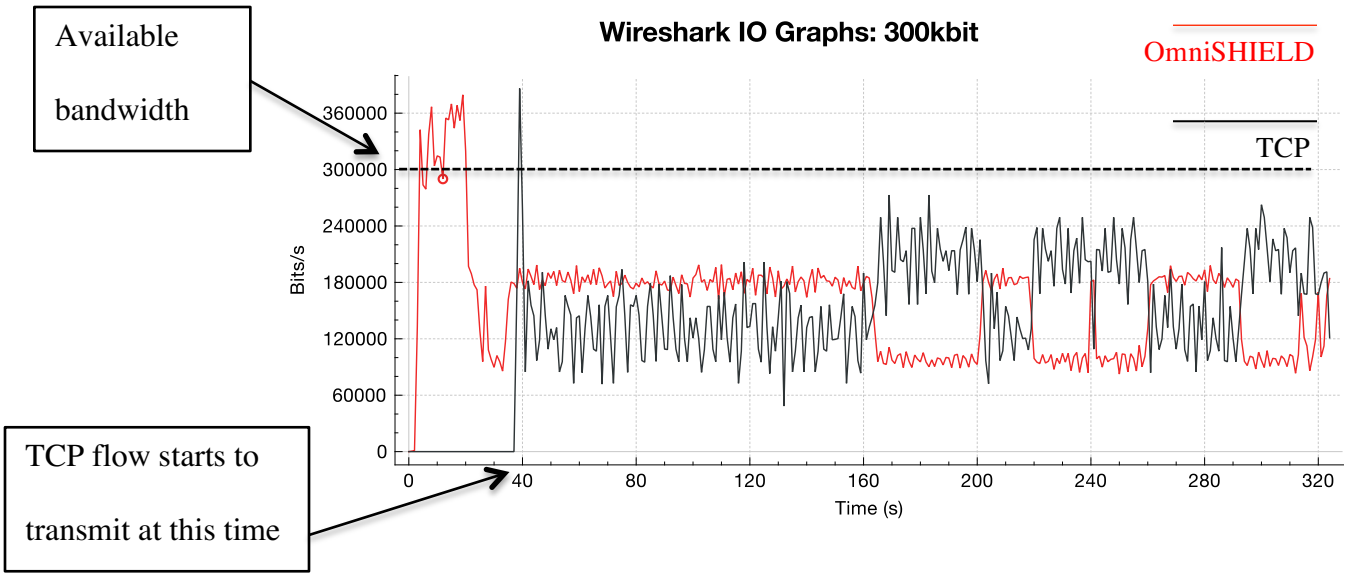


Figure 5.23: New Algorithm Test (Runs with the TCP Flow in a 300 kbit/s Network)

When the total bandwidth is 300kbit/s, the state of the program changes to VERYBAD. As shown in Figure 5.23, the TCP flow and the video flow share the bandwidth. The sending rate of the program stays between 100 kbit/s and 180 kbit/s. If the sender gets enough “network overused” reports, it changes the state from FINE to VERYBAD. In this scenario, the sender sends packets at the minimum rate to avoid network congestion.

6 Chapter: Conclusions and Future Work

6.1 Conclusions

In this thesis, a new delay-based rate control algorithm is proposed. I have implemented the new algorithm on the project. After the implementation, I have tested the performance of the new algorithm in a test bed. By analyzing and comparing the test results of the new and old algorithms, I have seen that the performance of the new algorithm improves on the old one.

There are two main drawbacks of the old algorithm:

- The sending rate will change periodically and fluctuate sharply if the network bandwidth is less than the highest video sending rate. From the user's perspective, when the network bandwidth is not enough, the user will get unstable and choppy video streams. This is not acceptable for a video-conferencing application.

- The video flow cannot compete for bandwidth with other flows in the same network. It occupies less bandwidth resources when there is a TCP flow in the same network.

The new delay-based algorithm solves the above two issues. When the new algorithm is implemented, the sending rate of the video stream does not fluctuate sharply when the network condition changes. The sending rate converges to a small range, so the users receive relatively stable video streams when they use the new algorithm. The new algorithm can also compete for bandwidth with concurrent TCP flows. When there is a

TCP flow in the network, the new algorithm no longer sends the stream at the minimum rate as the old algorithm does. In conclusion, the new delay-based algorithm significantly improves the performance of the real-time video transmission application.

This thesis proposes a new approach for using packet delay metrics to determine the network conditions in real-time video transmission applications. The new algorithm implements PCT and PDT test to measure the trend of one-way delay variation of packets. Based on measurement results, two tests can determine the trend of one-way delay variation, which can finally determine current network conditions. Based on test results, the packet delay metrics can accurately reflect network conditions, and it is more sensitive to changes of network conditions compared with the packet loss.

6.2 Future Work

The following are some aspects on which we can focus to improve the rate-control algorithm in the future:

1. In this thesis, my algorithm discusses how to send a video stream at an appropriate sending rate under different network conditions. But sometimes the sending rate alone does not adequately determine the quality of video streams. With a better codec, the video quality can be high when the sending rate is relatively low. From the users' perspective, different users need different kinds of video streams. Someone may prefer high quality images while others like high frame rate videos. The data-sending rate is just an abstract value. Users need a

video quality suitable for their needs. Only consider the sending rate of video streams is not sufficient.

2. To finish the test, I set up a test bed in a lab. The test bed can emulate real-network conditions with the help of NETEM. But a real network is much more complex than a test bed. So implementing the new algorithm on a real network is another challenge. Our test bed is very simple. Only four devices are connected. Packets sent from the sender to the receiver just go through one router. If the test bed is more complex, the conclusions may be different.
3. In the thesis, I just improve the application's performance when there is one other TCP flow running with our project. If there are several kinds of flows competing for bandwidth with our application, we need to be sure that the new algorithm can ensures that our application occupies a fair share of bandwidth.
4. There are currently many different kinds of congestion-control algorithms. Adding the delay-based feedback is not the only way to improve the performance of the video streams. Using those new technologies such as Forward Error Correction or ECN-based algorithms may further improve the performance of our project.
5. In my thesis, the sender does not know the current network condition. So it is difficult for the sender to determine the sending rate of the stream. If the sending rate is too high, the network will be congested, and a lot of packets will be delayed or lost during transmission, which will cause a poor quality of video streams. The sender needs to receive different feedback (PacketLoss, PacketOutOfOrder, etc.) from the receiver to determine the current network

condition. If there are some tools that measure the available bandwidth of the network, the application can choose the sending rate easily based on the available bandwidth measured. I recommend this as a research direction.

6. In this thesis, I implement a new rate control algorithm on a real-time video transmission application called OmniSHIELD. Although OmniSHIELD is a video conferencing application, the work focuses on the unidirectional video transmission. Applying the new algorithm to real video conferencing applications and testing the performance are also a meaningful research direction in the future.

References

[1] A. Vakili, “QoE Management for Video Conferencing Applications,” Computer Networks, vol. 57, no. 7, pp. 1726–1738, 2013.

[2] L. Dounis, I. Politis and T. Dagiuklas, ”On the Comparison of Real-time Rate Control Schemes for H.264/AVC Video Streams over IP- based Networks Using Network Feedbacks,” Proc. IEEE ICC, pp. 1-6, 2011.

[3] Google Inc., “On2 Video Codec,” *http://www.on2.com/on2-video/*

[4] A. Morton, G. Ramachandran and G. Maguluri, “Reporting IP Network Performance Metrics: Different Points of View,” IETF, RFC 6703, 2012.

[5] P. Chimento and J. Ishac, “Defining Network Capacity,” IETF, RFC 5136, 2008.

[6] M. Allman, V. Paxson and W. Stevens, “TCP Congestion Control,” IETF, RFC 2581, 1999.

[7] R. Prasad, C. Dovrolis, M. Murray and K. Claffy, “Bandwidth estimation: metrics, measurement techniques, and tools,” IEEE Network, vol. 17, pp. 27–35, 2003.

[8] M. Jain and C. Dovrolis, “Pathload: A Measurement Tool for End-to-end Available Bandwidth,” In Proceedings of Passive and Active Measurements (PAM) Workshop, pp. 14-25, 2002.

[9] V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil and L. Cottrell, “PathChirp: Efficient Available Bandwidth Estimation for Network Paths,” PAM Workshop, 2003.

[10] A. Shriram, M. Murray, Y. Hyun, N. Brownlee, A. Broido and M. Fomenkov, “Comparison of Public End-to-end Bandwidth Estimation Tools on High-speed Links,” PAM Workshop, 2005.

[11] S. Wang and H. Hsiao, "Fast End-to-end Available Bandwidth Estimation for Real-time Multimedia Networking," IEEE Multimedia Signal Processing, pp. 415–418, 2006.

[12] Q. Wang and L. Cheng, "FEAT: Improving Accuracy in End-to-end Available Bandwidth Measurement," In Proc. of 49th IEEE GLOBECOM Conference, 2006.

[13] Z. Lai, C. Todd and M. Rio, "Pathpair: A Fast Available Bandwidth Estimation Tool with the Asymptotic One Way Delay Comparison Model," IET Communications, pp. 967–978, 2009.

[14] T. Oshiba and K. Nakajima, "Quick End-to-End Available Bandwidth Estimation for QoS of Real-Time Multimedia Communication," IEEE Computers and Communications, pp.162-167, 2010.

[15] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," IETF, RFC 3550, 2003.

[16] J. Ott, S. Wenger, N. Sato, C. Burmeister and J. Rey, "Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF)," IETF, RFC 4585, 2006.

[17] I. Johansson and M. Westerlund, "Support for Reduced-Size Real-Time Transport Control Protocol (RTCP): Opportunities and Consequences," IETF, RFC 5506, 2009.

[18] J. P. S. Floyd, M. Handley and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification," IETF, RFC 5348, 2008.

[19] E. Kohler, M. Handley and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," IETF, RFC 4340, 2006.

[20] A. Asiri and L. Sun, "Performance Analysis of Video Calls Using Skype", Advances in Communications, Computing, Networks and Security Volume 10, pp. 155-162, 2013.

[21] L. De Cicco, S. Mascolo and V. Palmisano, "Skype Video Congestion Control: An Experimental Investigation," Computer Networks, vol. 55, no. 3, pp. 558–571, 2011.

[22] X. Zhang, Y. Xu, H. Hu, Y. Liu, Z. Guo and Y. Wang, "Profiling Skype Video Calls: Rate Control and Video Quality," IEEE Proceedings on INFOCOM, pp. 621-629, 2012.

[23] L. D. Cicco, G. Carlucci and S. Mascolo, "Experimental Investigation of the Google Congestion Control for Real-Time Flows," In Proc. of ACM SIGCOMM 2013 Workshop on Future Human-Centric Multimedia Networking, 2013.

[24] H. Alvestrand, S. Holmer and H. Lundin, "A Google Congestion Control Algorithm for RealTime Communication on the World Wide Web," IETF Internet Draft, 2012.

[25] M. Nagy, V. Singh, J. Ott and L. Eggert, "Congestion Control Using FEC for Conversational Multimedia Communication," In Proceedings of the 5th ACM Multimedia Systems Conference, pp. 191–202, 2014.

[26] X. Zhu and R. Pan, "NADA: A Unified Congestion Control Scheme for Real-Time Media," IETF Internet Draft, 2013.

[27] X. Zhu and R. Pan, "NADA: A Unified Congestion Control Scheme for Real-Time Media," *https://www.ietf.org/proceedings/86/slides/slides-86-rmat-0.pdf*

[28] F. Oueslati and J. Gregoire, "An Adaptation Mechanism for Robust OTT Video Transmission," 2015 IEEE 16th International Symposium on WoWMoM, pp. 1-6, 2015.

[29] TC usage, *http://manpages.ubuntu.com/manpages/trusty/man8/tc.8.html*

[30] Anders G. Moe, "Implementing Rate Control in NetEm," Masters Thesis, University of OSLO, 2013.

[31] TBF usage, *http://manpages.ubuntu.com/manpages/trusty/man8/tc-tbf.8.html*

[32] IPERF usage, *https://iperf.fr/iperf-doc.php*

[33] J. D. McCarthy, M. A. Sasse and D. Miras, "Sharp or Smooth? Comparing the Effects of Quantization vs. Frame Rate for Streamed Video," In Proc. of the SIGCHI Conf. on Human Factors in Computing Systems, pp. 535–542, 2004.

[34] timeGetTime function,

https://msdn.microsoft.com/en-us/library/windows/desktop/dd757629(v=vs.85).aspx

[35] Time functions comparison,

https://msdn.microsoft.com/en-us/library/windows/desktop/ms725473(v=vs.85).aspx