Non bandwidth-intrusive video streaming over TCP

Hiroyuki Hisamatsu
Department of Computer Science
Osaka Electro–Communication University
1130-70 Kiyotaki, Shijonawate
Osaka 575–0063, Japan
Email: hisamatu@isc.osakac.ac.jp

Go Hasegawa
Cybermedia Center, Osaka University
1-32 Machikaneyama, Toyonaka
Osaka 560–0043, Japan
Email: hasegawa@cmc.osaka-u.ac.jp

Masayuki Murata
Graduate school of Information Science
and Technology, Osaka University
1-5 Yamadaoka, Suita
Osaka 565–0871, Japan
Email: murata@ist.osaka-u.ac.jp

***Abstract*—Video streaming services using TCP as a transport layer protocol—represented by YouTube—are becoming increasingly popular and, accordingly, have come to account for a significant portion of Internet traffic. TCP is greedy; that is, it tries to exhaust the entire bandwidth. Thus, video streaming over TCP tends to unnecessarily take bandwidth from competing traffic.**

In this paper, we first investigate the data transfer mechanisms of the current video streaming services using TCP and show that they perform data transfer at much higher rates than the video playback rate. We then propose a new transfer mechanism for video streaming over TCP, one that controls the data transfer rate based on the network congestion level and the amount of buffered video data at the receiver. Simulation results show that the proposed mechanism has two characteristics lacked by current video streaming over TCP, specifically a low frequency of buffer underflow at the receiver and a lack of excessive bandwidth "stealing" from competing traffic.

***Index Terms*—TCP (Transmission Control Protocol), Congestion Control, Video Streaming, YouTube**

I. Introduction

Video streaming services have gained popularity in recent years, supported by a rapid increase in network bandwidth. A good number of video streaming services utilize UDP as a transport layer protocol. Also, such leading video players as Windows Media Player and Real Player utilize UDP if it is available. UDP-based applications are able to adjust their data transfer rate since UDP does not conduct congestion control and does not retransmit packets discarded in the network. However, because UDP-based communications are often intercepted by firewalls and/or NATs, there are environments where UDP-based video streaming services cannot be offered. TCP, on the other hand, can easily bypass firewalls and NATs. For this reason, most of the current video streaming services, such as YouTube [1] and nicovideo [2], support TCP-based streaming.

However, it has been pointed out that TCP is not suitable for video streaming because of its congestion control [3]. When TCP detects a packet loss, it markedly shrinks the congestion window size. Consequently, the TCP transfer rate decreases greatly. Since constant-rate data transmission is desirable for video streaming, TCP is generally not suitable for that service. Another issue with TCP is that when the network bandwidth is larger than the video playback rate, TCP, because of its greedy, tries to exhaust the entire bandwidth. Thus, video streaming over TCP has the problem that TCP increases its transfer rate regardless of the video playback rate and unnecessarily takes the bandwidth from other competing traffic.

Much research has been conducted on a new transport layer protocols for video streaming [4]–[9]. For instance, proposed in [4]–[6] are the transport layer protocols that, relative to TCP, do not rapidly increase/decrease the congestion window size, yet remain TCP-friendly. However, these protocols do not avoid the above-mentioned issue of having too much throughput relative to the video playback rate.

Moreover, in [7]–[9], the authors propose TCP modifications to keep the data transfer rate as requested from upper-layer applications. For instance, in [8], [9], the authors propose mechanisms to stabilize TCP throughput by concealing packet loss from TCP by installing a Forward Error Correction (FEC) layer in the lower part of a transport layer at both the sender and receiver. However, these mechanisms transmit excessive traffic to the network due to the redundancy of FEC.

Also presented are application layer protocols for media streaming [10], [11]. Real Time Streaming Protocol (RTSP) [10] is used by the Windows Media Player and Real Player. Real Time Messaging Protocol (RTMP [11]) is a proprietary protocol of Adobe Systems. Currently, both RTSP and RTMP only offers operations for viewing and listening to streamed media. They request data transfer to their lower protocols and do not conduct transfer control. Thus, they do not resolve the problem of streaming traffic unnecessarily taking away bandwidth from competing traffic.

In this paper, we propose an application-based data transfer mechanism that transfers video data at the minimum required rate for video streaming over TCP. First, we investigate the characteristics of the data transfer mechanisms of current video streaming services using TCP and show that they perform data transfer at a much higher rate than the video playback rate. We then propose a new data transfer mechanism that resolves this issue. Our proposed mechanism acquires TCP state variables to estimate the level of network congestion. It then controls data transfer at an application layer according to the network congestion level and the buffered video data size at the receiver. By simulation experiments, we show that the proposed mechanism (1) decreases the frequency of buffer underflow at the receiver and (2) avoids excessively taking bandwidth from competing traffic. Note that the current mechanism of TCP video streaming fails to achieve either.

The reminder of this paper is organized as follows. First, in

978-0-7695-4367-3/11 $26.00 © 2011 IEEE
DOI 10.1109/ITNG.2011.21

IEEE computer society

Section II, we investigate the characteristics of the data transfer mechanisms of current video streaming services using TCP. In Section III, we explain our data transfer control mechanism. We then evaluate the performance of our proposed mechanism in Section IV. Finally, in Section V, we present a conclusion and discuss future works.

II. Investigation of current Video Streaming Mechanisms using TCP

In this section, we investigate the currently utilized video streaming services using TCP and clarify the features and problems of their data transfer mechanisms. Specifically, we examined YouTube and nicovideo. Due to space limitations, we only explain the investigation results of YouTube. Note that the transfer mechanism of nicovideo has the same kind of problem as that of YouTube.

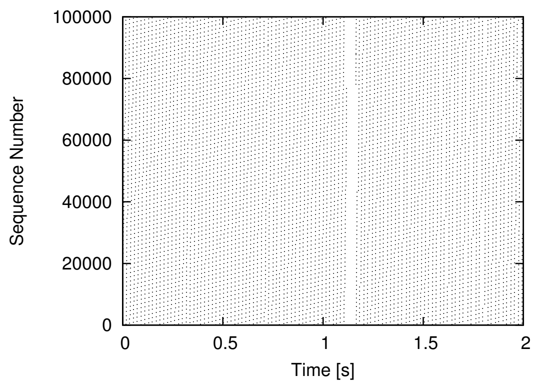
In the investigation, we observed data transfer in video streaming at a packet level using `tcpdump` at a receiver. The receiver is located at Osaka Electro-Communication University in Shijonawate-shi, Osaka, Japan. We acquired video sequences from servers thought to be in Japan (although YouTube dose not disclose the location of their servers, we conclude that the servers investigated here are located in Japan from the results of `traceroute` commands). We measured 10 video sequences on YouTube, and ran five measurements for each video sequence. The playback time of the YouTube video sequences was 10 [m] and the quality was 1080p (the playback rate was about 3.60 [Mbit/s]).

We disabled the TCP delayed ACK option since we wanted to observe typical TCP behaviors. Moreover, we configured the advertising window size, which is the buffer size of the receiver TCP, to 224 [KByte], a size sufficiently large to avoid throughput limitations.

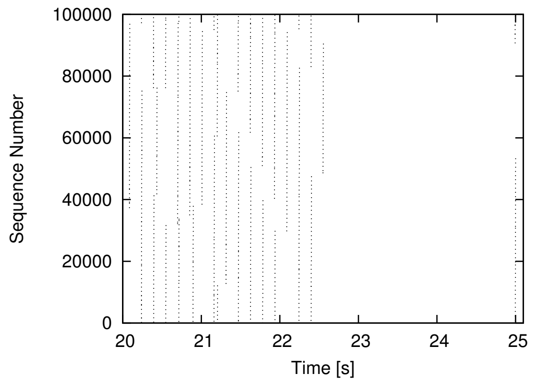
From the observation results, we found that YouTube utilizes two mechanisms for video data transfer. Moreover, by conducting five time experiments against the same video sequence from the same server, we could not determine the detailed conditions for selecting one of those two mechanisms. We found that the data transfer behavior of YouTube is independent of available bandwidth between the server and the receiver. We confirmed these characteristics by experiments under various bandwidth limitations on the link connected to the receiver. We refer to these two mechanisms below as mechanism(i) and mechanism(ii), respectively.

Mechanism(i) has two phases: a first and a second phase. Figure 1(a)(b) shows the receiving timing of data packets in the first phase, and in the second phase, respectively. The y axis of the graphs represents the byte-count sequence number (mod 100,000) of received TCP data packets. The x axis represents the time, which is zero when the first data packet is received.

In mechanism(i), a server transmits roughly 80 [MByte] data in the first phase, whereupon it interrupts data transfer. The average transfer rate in the first phase is about 43.4 [Mbit/s], which is excessively high compared to the video playback rate. Some of the short interruptions in packet transmission apparent in Fig. 1(a) are not caused by packet losses; instead, the server may stop packet generation for some reasons. We believe that Youtube conducts such a greedy, high-rate data transmission at the beginning of the transfer to buffer sufficient amount of video data at the receiver.



(a) First phase



(b) Second phase

Fig. 1. YouTube data packet receiving: mechanism(i)

Several seconds after the first phase has finished, YouTube shifts to the second phase. In the second phase, YouTube transmits data and then switches over to discontinuous data spurts that continue until finishing the data transfer. The server sends between 32 [KByte] and 128 [KByte] of data within one round-trip time duration. Then, once the server sends a total of about 2.20 [MByte] data, it interrupts the data transfer for several seconds. The average transmission rate at the time of data transfer in the second phase was about 6.13 [Mbit/s], which is also larger than the video playback rate.

With regards to mechanism(i), we find that YouTube transmits data at a higher rate than the video playback rate in both the first and the second phase. The reason for the decreased transfer rate in the second phase, we presume, is that the receiver is thought to have buffered enough video data in the first phase. Moreover, throughout experiments at different bandwidth limitations, we found that the data transfer size and rate are largely independent of the network bandwidth. We believe that the server dose not "care" the network status but rather the such differences in transfer sizes/rates are attributable to server conditions, such as CPU load, and number of TCP connections to connect.

Next, we explain mechanism(ii) of YouTube. In mechanism(ii) of Youtube, unlike in mechanism(i) of Youtube, we found that there is no special control. The server sends video data at an extremely high rate from the beginning to the end of the video sequence. The average data transfer rate is about 45.1 [Mbit/s], which is much higher than the video playback rate. From these observations, it is apparent that the data transfer mechanism for transmits video data at a rate far beyond what is necessary.

We found that YouTube transfers video data at extremely high rate relative to the video playback rate. This is good from the viewpoint of enabling continuous playback. However, it is a serious problem from the viewpoint of needless competition with other applications.

It is more desirable to transmit a small chunk of data for every interval. However, optimal chunk size and chunk transmitting interval are strongly dependent on the network. Video streaming with such fixed parameters may overflow/underflow network capacity and produce failures in video playback at the receiver.

III. Non bandwidth intrusive video streaming over TCP

In this section, we propose a new data transfer mechanism for video streaming over TCP that transfers video data at the minimum required rate and dose not unnecessarily deprive the other competing traffic of bandwidth. It does this by transmitting a sufficient amount of data such that buffer underflows do not occur at the receiver.

A. Outline of Proposal Mechanism

The proposed mechanism is assumed to be installed as an application program at the sender and receiver. Moreover, the sender-side application requires the mechanism to acquire TCP state variables from a TCP connection transmitting video data. It is easily possible to acquire TCP state variables, for instance in the case of Linux, by using web 100 kernel [12]. Note that we do not modify any TCP operations, including the congestion control mechanism.

The receiver-side application notifies the sender-side application of the amount of buffered video data (b_{dst}) per one round-trip time (RTT) using the TCP connection for data transfer. The sender-side application, while considering the network congestion level and buffered video data size at the receiver, calculates the application-level window size and the amount of video data to send in the next RTT to avoid buffer underflow and playback interruption. Control of the proposed mechanism operates in the unit of one RTT.

B. TCP State Variables to Acquire

In the proposed mechanism, the sender-side application acquires the TCP state variables to estimate the network congestion level. Specifically, the sender application acquires the current congestion window size (cwnd), smoothed RTT (srtt), retransmission timer value (RTO), slow-start threshold (ssthreh), and maximum segment size (MSS). These variables are obtained at every RTT. In addition, when a packet loss is detected at TCP, the sender-side application is immediately informed of the occurrence of the packet loss and its detection method: reception of three duplicate ACKs (TD-ACKs) and occurrence of the timeout (TO).

C. Congestion Level Estimation

We estimate the number of packets queued throughout the network as an index of the network congestion level. Henceforth, we use the network congestion level in the sense of the number of packets queued over the entire network. Based on the mechanism in TCP Vegas [13], the network congestion level $cl(i)$ in i_{th} RTT is calculated as

$$cl(i) = \frac{baseRTT(i-1)}{MSS} \times \left(\frac{apwnd(i-1)}{baseRTT(i-1)} - \frac{apwnd(i-1)}{RTT(i-1)} \right)$$

where $apwnd(i)$, $baseRTT(i)$, $RTT(i)$ and MSS are the application-level window size of the proposed mechanism, the minimum srtt, the srtt, and the maximum TCP segment size, respectively.

The proposed mechanism determines what packet loss occurs in what congestion level from the packet loss detection method and the congestion level when the packet loss occurred. When the sender-side application is notified of an occurrence of a packet loss by TCP, it calculates the current number of packets queued throughout the network cl, and records cl and the packet loss detection method. From cls and the detection methods that were recorded, the sender-side application calculates dup_{min} and TO_{min}, with dup_{min} and TO_{min} being the lower bound of 95% confidence interval of the congestion level when packet losses are detected by TD-ACKs, and that when packet losses are detected by TO, respectively. We calculate dup_{min} and TO_{min} from cl of the last α [h].

D. Target Value of Video Data to be Buffered

We first introduce $b_{ret}(i)$, $b_{dup}(i)$ and b_{TO}. Here, $b_{ret}(i)$, $b_{dup}(i)$ and $b_{TO}(i)$ are the amount of video data to be buffered to avoid buffer underflow when one packet loss is detected by TD-ACKs, that when two or more packet losses are detected by TD-ACKs, and that when packet losses are detected by TO, respectively.

$b_{ret}(i)$ is the amount of video data buffered at the receiver to avoid buffer underflow when a packet loss occurs, it is detected by TD-ACKs, and no additional packet loss occurs until TCP cwnd recovers to catch up with the video rate. Using the current TCP congestion window size $cwnd(i)$ and the video playback rate $rate$, $b_{ret}(i)$ in i_{th} RTT is given by

$$b_{ret}(i) = (3 + \max(nwnd(i-1) - \lfloor cwnd(i-1)/2 \rfloor), 0)) \times RTT_{max}(i-1) \cdot rate - MSS \sum_{k=\lfloor cwnd(i-1)/2 \rfloor}^{nwnd(i-1)} k$$

where $nwnd(i)$ is the congestion window size to achieve the video playback rate. $nwnd(i)$ is given by

$$nwnd(i) = RTT_{max}(i) \cdot rate/MSS$$

$RTT_{max}(i)$ is given by

$$RTT_{max}(i) = \overline{RTT}(i) + \gamma RTT_{std}(i) \tag{1}$$

where $\overline{RTT}(i)$ is the exponential weighted moving average of the RTT with weight β, and $RTT_{std}(i)$ is the standard deviation of the RTT. The reason for not using the current RTT is to take account of RTT fluctuation.

$b_{dup}(i)$ is the amount of video data buffered at the receiver to avoid the buffer underflow when a packet loss occurs, it is detected by TD-ACKs, and additional packet losses occurs until cwnd recovers to $nwnd(i)$. Taking into account the worst case, we consider a situation whereby the window size is reduced to one by occurring two or more packet losses. $b_{dup}(i)$ is given by

$$b_{dup}(i) = (3 + nwnd(i-1) - 1)\, RTT_{max}(i-1) \cdot rate - MSS \sum_{k=1}^{nwnd(i-1)} k.$$

$b_{TO}(i)$ is the amount of video data buffered at the receiver to avoid buffer underflow when packet losses are detected by TO. $b_{TO}(i)$ is given by

$$b_{TO}(i) = \Big\{ RTO(i-1) + \big(4 + \lfloor log_2 ssthresh(i-1) \rfloor + nwnd(i-1) - 2^{\lfloor log_2 ssthresh(i-1)\rfloor}\big) RTT_{max}(i-1) \Big\} rate - \Big(\sum_{k=0}^{\lfloor log_2 ssthresh(i-1)\rfloor} 2^k + \sum_{k=2^{\lfloor log_2 ssthresh(i-1)\rfloor}}^{nwnd(i-1)} k \Big) MSS.$$

where $RTO(i)$ is the retransmission timer value and $ssthresh$ is the threshold value for TCP slow-start.

A sender-side application calculates the amount of video data should be buffered to avoid receiver-side buffer underflow. In i_{th} RTT, the amount of data that a receiver-side application should buffer in order to play back video continuously is given by the following equation.

$$b_{tgt}(i) = \begin{cases} \frac{b_{dup}(i) - b_{ret}(i)}{dup_{min}} cl(i) + b_{ret} & (0 \leq cl(i) < dup_{min}) \\ \frac{b_{TO}(i) - b_{dup}(i)}{TO_{min} - dup_{min}} (cl(i) - dup_{min}) + b_{dup}(i) & (dup_{min} \leq cl(i) < TO_{min}) \\ b_{TO} & (TO_{min} \leq cl(i)) \end{cases} \tag{2}$$

When $cl(i)$ satisfies $0 \leq cl(i) < dup_{min}$, we assume that network packet losses do not happen very often. In this case, in order to avoid buffer underflow, the proposed mechanism buffers video data in preparation for a packet loss. As $cl(i)$ increases, the network is considered to be in an increasingly dangerous state; packet losses more serious than a single packet loss are considered likely to occur. Here, we simply increase the amount of video data to be buffered linearly to $cl(i)$.

When $cl(i)$ satisfies $dup_{min} \leq cl(i) < TO_{min}$, the network is considered to be in a congestion level at which packet loss detection by TD-ACKs may occur. In this case, the proposed mechanism buffers video data in preparation for a packet loss detected by TD-ACKs and additional packet losses until cwnd recovers to $nwnd(i)$. Moreover, the amount of video data to be buffered increases linearly to $cl(i)$, similarly to the case in which $0 \leq cl(i) < dup_{min}$ is satisfied.

When $cl(i)$ satisfies $TO_{min} \leq cl(i)$, we assume that the network is at a serious congestion level where a TO may happen. Here, we believe that enough video data should be buffered to avoid a buffer underflow even should a TO occur. However, we do not target a network with a congestion status so severe that packet losses occur repeatedly after a TO until cwnd recovers $nwnd(i)$. Here, we resign ourselves to the observation that video streaming of the video playback rate within such a network simply cannot be performed at a rate specified by the application.

Since we cannot estimate the network congestion level until the first packet loss occurs, the target value $b_{tgt}(i)$ is set equal to $b_{TO}(i)$ until the first packet loss occurs. We also configure the initial value of dup_{min} and TO_{min} to be sufficiently large.

E. Controlling Transfer Rate

The proposed mechanism controls data transfer in a sender-side application. This is performed by adjusting the amount of video data $apwnd(i)$ passed from an application to TCP in one RTT. By using $b_{dst}(i)$ and Eqs. (1)(2), $apwnd(i)$ is given by

$$apwnd(i) = \min\left(\max(RTT_{max}(i-1) \cdot rate + b_{tgt}(i) - b_{dst}(i), MSS), 2apwnd(i-1)\right).$$

The proposed mechanism determines the amount of data passed to TCP in one RTT based on the difference between $b_{tgt}(i)$, the target value of video data to be buffered, and $b_{dst}(i)$, the amount of buffered video data in the receiver. The minimum of $apwnd(i)$ accords with TCP minimum window size. It also acts to prevent resetting of the TCP congestion window. Furthermore, in order to keep from increasing the transfer rate too rapidly, the maximum value is set at the twice of $apwnd(i-1)$, which is based on TCP slow-start phase.

IV. Performance Evaluation by Simulation Experiments

Using ns-2 simulator, we ran extensive simulation experiments at the packet level to confirm the effectiveness of our proposed mechanism. Figure 2 shows the network model for the experiments, where five TCP connections for video streaming, TCP connections for FTP, and one UDP flow share the single bottleneck link having a capacity of 100 [Mbit/s]. The arrival of UDP packets follows a Poisson process, with an average arrival rate of 10 [Mbit/s]. For comparison purposes, we utilized two kinds of YouTube-like transfer mechanisms, to which we refer below as YouTube-like(i) and YouTube-like(ii).

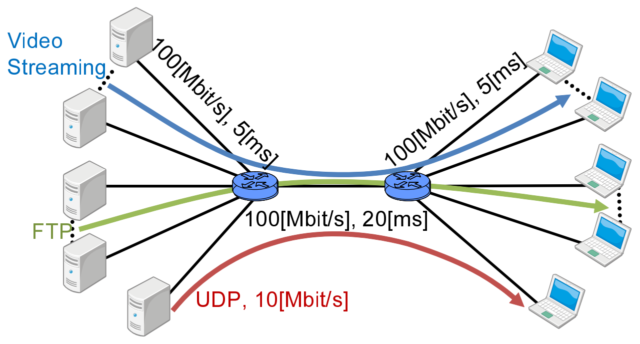


Fig. 2: Simulation model

Both of YouTube-like(i) and YouTube-like(ii) operate based on measurement results like those described in Section II. YouTube-like(i) has two phases such as in mechanism(i) in Subsection II. In the first phase, 80.0 [MByte] of video data is transferred. Then, YouTube-like(i) stops sending for 2.0 [s], after which it enters the second phase. It then sends video data in on-off fashion, where transmission of 2.2 [MByte] data (it is divided into chunks of 128 [KByte] and one chunk is transmitted in one RTT) followed by a 1.5 [s] pause are repeated. In contrast, YouTube-like(ii) transfers video data without any control in an application layer. In simulation experiments, we use the following parameters for the proposed mechanism: $\alpha = 1$ [h], $\beta = 0.2$, and $\gamma = 1$.

In simulation experiments, the UDP flow and the TCP connection(s) for FTP begin data transfer when a simulation starts to create background traffic and competing traffic in the network. At 30 [s], TCP video streaming connections start data transfer. The playout delay of video streaming at the receiver is 5 [s] and the packet size is 1500 [Byte]. The simulation finishes when the transfer of all video data is completed. We use the simulation result (excluding the first 30 [s] of the simulation time) to calculate such performance metrics as average throughput, average packet loss probability and average underflow time experienced by the receiver-side application. The video sequence has 270 [MByte], the playback rate is 3.6 [Mbit/s], and the playback time is 600 [s], which is equivalent to YouTube's 1080p video.

Figure 3 exhibits simulation results as a function of the number of TCP connections. Figures 3(a)–(d) show (a) the average throughput of TCP connections for video streaming, (b) the average total buffer underflow time per video streaming connection, (c) the average packet loss probability in the network during the experiments, and (d) the average total throughput of TCP connections for FTP, respectively. In Fig. 3(a), we see that the YouTube-like(ii) transfer rate is much higher than the video playback rate, while the transfer rates under the proposed mechanism and under YouTube-like(i) are almost equal rate to the video playback rate. Note that it is not necessary to transfer video data at a much higher rate than the video playback rate (as in the case with YouTube-like(ii)) to maintain continuous video playback.

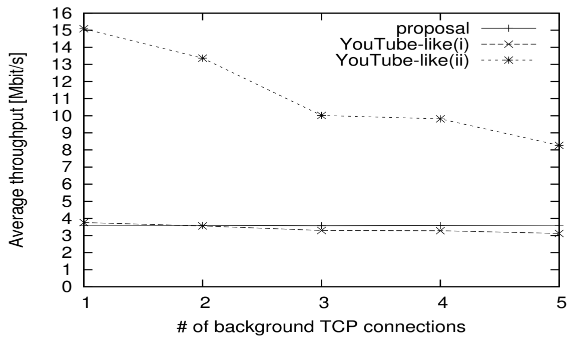
Figure 3(b) shows that buffer underflow does not occur often under the proposed mechanism and YouTube-like(ii). On the other hand, under YouTube-like(i), we found that as the number of TCP connections for FTP increases, the buffer underflow time becomes large. This is because YouTube-like(i) controls video data transfer independently of network status. In this case, YouTube-like(i) dose not obtain sufficient throughput and so buffer underflow occurs when YouTube-like(i) competes with other TCP connections.

We next focus on the packet loss probability in the network. Figure 3(c) shows that the packet loss probability of the proposed mechanism is lower than that of YouTube-like(ii). This is because YouTube-like(ii) performs video data transfer without any controls. Thus TCP tries to exhaust the entire bandwidth due to its greedy nature. Therefore, the network congestion level of YouTube-like(ii) is higher than that of the proposed mechanism, which controls the data transfer rate in accordance with congestion level and video playback status at the receiver. Conversely, the packet loss probability of the proposed mechanism is higher than that of YouTube-like(i). This can be explained as follows. When the network congestion level becomes high, the proposed mechanism increases the target value for video data to be buffered at the receiver. In other words, the propose mechanism acts to increase the data transfer rate. On the other hand and as mentioned above, YouTube-like(i) transmits packets independently of the network congestion status. Consequently, the packet loss probability of the proposed mechanism is higher than that of YouTube-like(i). Note that YouTube-like(i) experiences buffer underflow for long durations, especially when network congestion is serious, as a price to pay for low packet loss probability.

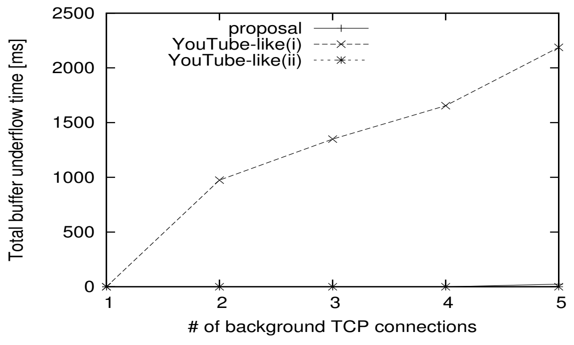
Finally, we focus on the total throughput of background traffic. From Fig. 3(d), we see that the total throughput of background traffic when using the proposed mechanism is higher than that when using YouTube-like(ii). This is because YouTube-like(ii) transfers video data at a high rate regardless of the video playback rate. As the result, the video streaming connections steal the bandwidth from background traffic. On the other hand, the total throughput of background traffic when using YouTube-like(i) is higher than that when using the proposed mechanism. As explained earlier, the proposed method increases the number of packets sent to the network when network congestion gets worse. Thus, the video transfer rate becomes large, driving down the throughput of competing traffic. Throughout the simulation experiments, we find that the proposed mechanism is effective in (1) suppressing the occurrence of the buffer underflow (2) avoiding unnecessarily diversion of bandwidth background traffic. Note that the current video streaming mechanisms do not have these two characteristics.

V. Conclusion and Future Works

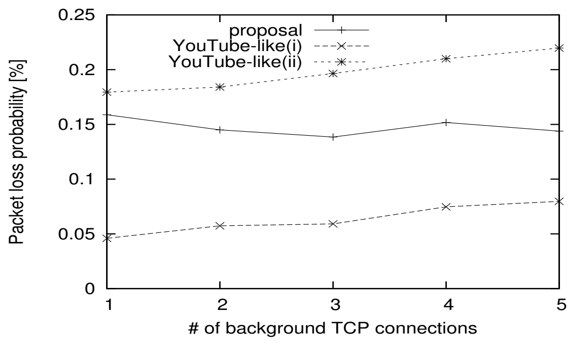
In this paper, we have proposed a non bandwidth-intrusive video streaming mechanism and have shown its effectiveness by simulation. First, we have investigated data transfer mech-



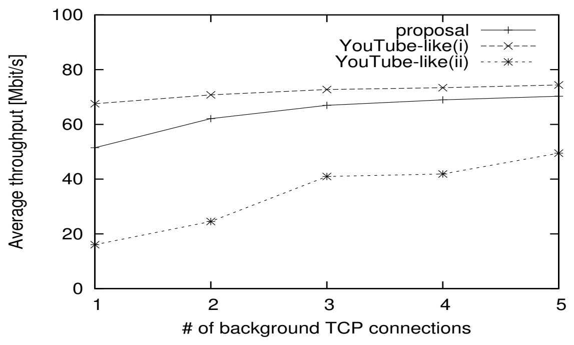
(a) Average throughput of TCP for video streaming



(b) Average total buffer underflow time



(c) Average packet loss probability



(d) Average total throughput of TCP for FTP

Fig. 3: Simulation results

anisms of the current video streaming services using TCP. Our investigation has shown that video streaming over TCP is currently performed at a much higher rate than the video playback rate. We have proposed a new data transfer mechanism to resolve this problem. The proposed mechanism transfers video data without unnecessarily taking the bandwidth from competing traffic. In simulation experiments, we have shown that proposed mechanism suppresses the occurrence of buffer underflow and dose not unnecessarily divert bandwidth from background traffic.

As future works, it is important to evaluate the performance of the proposed mechanism in a real network. We also plan to extend the proposed mechanism so that it can be operate solely by a sender-side application. Moreover, it would be interesting to consider what type of data transfer control is preferred for the network, where an increase in transfer rates for continuous video playback acts to worsens the network congestion and exacerbates video playback interruptions.

Acknowledgment

This work was partly supported by Grant-in-Aid for Scientic Research (A) 21240004 of the Japan Society for the Promotion of Science (JSPS) in Japan.

References

[1] "YouTube." available at http://www.youtube.com/.
[2] "nicovideo." available at http://www.nicovideo.jp/.
[3] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast applications," in *Proceedings of ACM SIGCOMM 2000*, pp. 43–56, Aug. 2000.
[4] L. Cai, X. Shen, J. Pan, and J. Mark, "Performance analysis of TCP-friendly AIMD algorithms for multimedia applications," *IEEE/ACM Transactions on Networking*, pp. 339–355, apr 2005.
[5] F. Yang, Q. Zhang, W. Zhu, and Y.-Q. Zhang, "End-to-end TCP-friendly streaming protocol and bit allocation for scalable video over wireless Internet," *IEEE Journal on Selected Areas in Communication*, pp. 777–790, may 2004.
[6] S. L. Bangolae, A. P. Jayasumana, and V. Chandrasekar, "TCP-friendly congestion control mechanism for an udp-based high speed radar application and characterization of fairness," in *Proceedings of the The 8th International Conference on Communication Systems*, pp. 164–168, 2002.
[7] Y. Zhu, A. Velayutham, O. Oladeji, and R. Sivakumar, "Enhancing TCP for networks with guaranteed bandwidth services," *ACM Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 51, pp. 2788–2804, July 2007.
[8] T. Tsugawa, N. Fujita, T. Hama, H. Shimonishi, and T. Murase, "TCP-AFEC: An adaptive FEC code control for end-to-end bandwidth guarantee," *in Proceedings of 16th International Packet Video Workshop (PV 2007)*, pp. 294–301, 2007.
[9] B. Libæk and O. Kure, "Protecting scalable video flows from congestion loss," in *Proceedings of the 2009 Fifth International Conference on Networking and Services*, pp. 228–234, 2009.
[10] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol (RTSP)," *Request for Comments (RFC) 2326*, Apr. 1998.
[11] "Real-time messaging protocol (RTMP) specification." available at http://www.adobe.com/devnet/rtmp.html.
[12] "The web100 project." available at http://www.web100.org/.
[13] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," in *Proceedings of ACM SIGCOMM '94*, pp. 24–35, Oct. 1994.