Cross Layer Protocol Support for Live Streaming Media

Syed Hasan[1], Laurent Lefevre[2], Zhiyi Huang[1] and Paul Werstein[1]

[1] Department of Computer Science
University of Otago,
Dunedin, New Zealand,
Email: {shasan|hzy|werstein}@cs.otago.ac.nz

[2] INRIA RESO - LIP (UMR CNRS, INRIA, ENS, UCB)
Ecole Normale Superieure,
46, allee d'Italie - 69364 LYON Cedex 07 - FRANCE ,
Email: laurent.lefevre@inria.fr

Abstract

Delivering live streaming content over the Internet requires low delay and smooth packet transmission rate. TCP introduces rate oscillations and requires more buffering and bandwidth to sustain uninterrupted playback. In this paper we propose a new framework which facilitates streaming flows. Our solution provides a smoother rate control than TCP and improves streaming performance based on cross layer feedback between the transport protocol and streaming server. We present our experimental results through simulation.

1. Introduction

There is a growing trend of streamed audio-video applications on the Internet. According to media research, the amount of streamed video increased by 39% in 2006 to 24.92 billion streams [2]. Faster and cheaper access bandwidth is enabling various streamed audio-video services to the end user. However the user experience of streaming media is not satisfactory always. Recent research shows that on the Internet, about 13% of home and 40% of business streaming sessions suffer various quality degradation [8].

The smooth and flexible sending rate required for live streaming is hard to achieve on the Internet which only provides 'best-effort' delivery of packets. In times of congestion, queues build up inside the routers delaying/dropping incoming packets. In order to mitigate this problem, streaming applications use techniques like client-side playout buffering and stream switching. The client-side playout buffer essentially borrows some current bandwidth to prefetch packets for protection against future rate reduction. The buffer size has to be large enough to ensure that in times of congestion it does not run out of packets and continues smooth playout. Stream switching allows changing the streamed bit rate depending on the significance of congestion. Despite these techniques, much of the performance issues for streaming depend on the underlying transport protocol. A transport protocol which provides a smooth low delay transmission of packets is highly desirable.

The success of the Internet can be attributed to TCP's congestion control mechanisms [10]. TCP controls the sending rate of the application in order to ensure fairness and avoid congestion. This proactive rate control is at odds with streaming applications as they cannot change or sustain a particular transmission rate whenever they want to. Unlike TCP, UDP is a fast, light weight protocol without any congestion control or retransmission functionality. This makes UDP an ideal protocol for transmitting audio-video data which can tolerate a few packet losses. Applications using UDP have complete control over their sending rates and are responsible for avoiding congestion and ensuring fairness so that other flows are not starved. However UDP is a connectionless protocol and very often firewalls block UDP traffic for security reasons. In that case applications revert to TCP and sometimes use HTTP over TCP to penetrate firewalls. A study shows TCP is used by 66% to 72% of all streaming sessions [8].

Datagram Congestion Control Protocol (DCCP) is a newly proposed transport level protocol designed to overcome the drawbacks of TCP [12]. DCCP is connection oriented, unreliable and incorporates TCP-Friendly Rate Control (TFRC) algorithm as an optional congestion control mechanism [9]. TFRC has been designed to provide smoother transmission rate and ensure fairness with other

TCP flows. TFRC focuses on smooth sending rate while avoiding any sharp rate change which is often required by streaming applications. DCCP is still under development and needs more testing. In an experimental study on audio streaming, it was reported that quality is not improved when TFRC is used for rate control [3].

In this paper, we study the interaction of streaming applications with TFRC and propose a new framework which facilitates streaming flows. In TCP based streaming, an application's control loop is decoupled from TCP's control loop. The transport layer is not aware of the application's requirement and tries to control the application's sending rate only to avoid congestion and ensure fairness. Our solution couples the application's control loop with transport protocol's control loop through cross layer information exchange. Cross layer information exchange breaks away from traditional network design paradigm where each layer of the protocol stack operates independently. In our case, information is exchanged between the application layer and transport layer. The application informs the transport layer about its required average transmission rate and the transport layer provides early congestion feedback to the streaming application based upon send buffer queue size. Experimental results show that this framework reduces packet loss by avoiding unnecessary overshooting of bandwidth, reduces the end-to-end streaming latency by keeping the send buffer queue size at an optimum level and enables the streaming application to adapt the sending rate fast enough to avoid a re-buffering event. We named the framework Dynamic Buffer Active Tuning (DBAT).

The paper is organized as follows. In Section 2, some background on streaming techniques and the underlying transport protocols are discussed. The proposed framework is presented in Section 3. In Section 4, the experimental results are illustrated. Related work is discussed in Section 5 and Section 6 contains the conclusions and future work.

2. Background

Classical streaming applications support several streamed bit rates in order to match the available bandwidth with the streamed bit rate. In addition, streaming incorporates several quality adaptive techniques. We begin this section with a brief discussion of the application model for streaming, its key performance indicators and discuss two protocols for the transmission of streaming packets.

2.1 Streaming Application Model

In traditional streaming solutions, the client and server exchange control packets to negotiate appropriate sending rates. At the beginning of the session, the server uses a packet pair based bandwidth probing technique to determine the available bandwidth and chooses the streaming bit rate accordingly. A playout buffer is used at the client side to reduce the effects of inter-packet jitter. Playback starts as soon as the buffer is fulled to a certain threshold. A streaming system goes through three phases:

- *Buffering:* If the size of the playout buffer is large, the initial buffering period is longer, but it protects against playback interruptions when the available bandwidth briefly drops below streamed bit rate. For live streaming, this delay in buffering should be low.
- *Playback:* As long as there are packets in the playout buffer, the client keeps playing at the encoding rate.
- *Re-buffering:* If the buffer gets empty, playback stops until the buffer is filled to the threshold level.

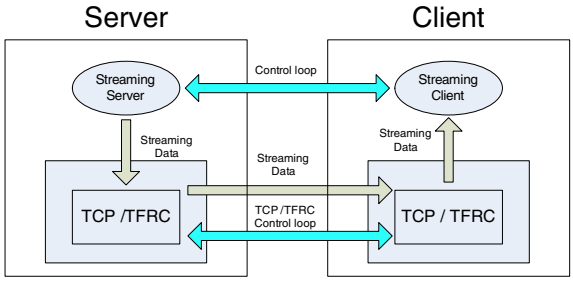

Figure 1. Classical streaming architecture based on TCP / TFRC

The streaming client and server use a control loop to monitor the packet loss rate and client side buffer's status. Whenever the packet loss rate crosses a pre-defined threshold or a re-buffering event occurs, depending on the available bandwidth, the streaming server might change the streamed bit rate and start streaming at the new rate. As shown in Figure 1, this loop is decoupled from the rate control loop of the transport protocol, i.e there is no exchange of cross layer information. TCP/TFRC exchange their own set of control packets in order to regulate the application's sending rate, whereas the application uses another set of control messages to determine its appropriate sending rate. There is no cooperation between these two loops which try to determine the optimum sending rate independently.

2.2 Streaming Performance Metrics

Streaming performance can be evaluated in terms of re-buffering events, packet loss rate, and smoothness in achieved throughput of streamed flows.

- *Number of Packet lost in Burst:* Streaming audio-video applications are able to tolerate a few packet losses but streaming performance degrades if packets are lost in burst.

- *Number of Re-buffering Events:* Every time the play-out buffer gets drained below the threshold, playback is paused while the buffer becomes full again. These abrupt interruptions in playback drastically impacts the quality of the streaming session. The number of re-buffering events and the percentage of time spent for buffering can be a good performance indication.

- *Average Service Rate:* If the application is able to sustain to a high streamed bit rate for long time, the streamed content is of high quality and this improves the users' perception of streaming.

A streaming application is able to reduce the amount of packet loss and/or the possibility of re-buffering events by quickly adjusting the sending rate. The role of the underlying transport protocol is very important for such an application. A send buffer is employed to deal with the rate mismatch between the application's sending rate and the transport protocol's allowed transmission rate. This buffering adds end-to-end delay and may become an obstacle for achieving the new streamed bit rate when stream switching occurs. We call this stream switch response time.

A canonical streaming application emits packets at a constant rate. The transport protocol is responsible for sending the packets from the send buffer to the network interface. Packets are queued at the send buffer when the application's packet generation rate is higher than the transmission rate of the underlying transport protocol. As the feedback delay between the client and the server increases, the client's feedback becomes outdated, and the application level rate adaptation mechanism fails to react soon enough to reduce packet loss and re-buffering events. Some mechanism for reducing this feedback delay will be hugely beneficial.

2.3 Transport Protocol for Streaming

2.3.1 TCP

It is well known that TCP's congestion control mechanism is vital for the scalability of the Internet [6]. In order to avoid congestion and ensure fairness among competing flows, TCP controls the sending rate of the application using an Additive-Increase-Multiplicative-Decrease (AIMD) algorithm. TCP keeps an estimate of the available bandwidth for the next RTT (round trip time) using a variable known as congestion window.

Although UDP is preferable to most streaming applications, TCP is used more often. However the reliable, inorder and congestion controlled service model of TCP is inappropriate for streaming flows which require more control and flexibility over its flows. The main obstacles for streaming using TCP are:

- *Information Hiding:* TCP hides the loss rate and RTT information from the application.

- *Buffering Delay:* TCP's window based congestion control mechanism requires a send buffer at the application to transport layer interface for briefly storing the in-flight packets as well as enough new packets to saturate the congestion window in the next flight.

- *Abrupt Rate Controlling:* TCP's AIMD reduces the application's sending rate by half on a single packet loss. The application does not get enough time to adapt the sending rate. As a result, a large number of packets are buffered at the send buffer.

- *Head-of-line Blocking:* Whenever a packet loss is detected, TCP's in order delivery mechanism blocks the delivery of received packets to the client until the lost packet is delivered through retransmission.

The effects of TCP's rate fluctuation due to congestion control can be reduced using the client side playout buffer. However, as the link delay increases, the buffering becomes insufficient to reduce the effects of rate variation. For live streaming, it is challenging to stream on TCP if the link-delay and/or loss rate is comparatively significant.

2.3.2 TFRC

TCP-Friendly Rate Control(TFRC) is a rate control algorithm which provides smoother throughput by reacting slowly on packet loss rate while being friendly to other TCP flows [9]. Since most applications on the Internet are TCP based, in order to be a good network citizen, a deployable congestion control algorithm should be friendly to TCP flows. A flow is TCP-friendly if its average sending rate is no more than a TCP flow running between the same links. A TFRC sender calculates TCP throughput using an equation based on receiver's feedback on loss event rate, received packet rate and the RTT information [16].

TFRC has been incorporated as an alternative congestion control algorithm for the newly standardized Datagram Congestion Control Protocol (DCCP) [12]. DCCP provides an unreliable service with reliable connection establishment and option negotiation states. Applications using DCCP have the option to choose different congestion control mechanism for each direction. Right now only two types of congestion control has been standardized, TCP-like and TFRC.

Due to the smoother rate control, TFRC requires less playout buffer space than TCP. However various studies have reported poor performance of TFRC based audio-video transmission. For streaming applications even though TFRC reacts slowly on congestion events, the sender only reacts based on receivers feedback. An early feedback on congestion events would give more time to the sender for rate adaptation.

3 Dynamic Buffer Active Tuning (DBAT)

In this section, we discuss our active queue management mechanism named Dynamic Buffer Active Tuning (DBAT). The goals of DBAT are as follows:

- *Reduce buffering delay:* DBAT keeps the send buffer queue at a minimum level.
- *Provide feedback to application:* DBAT sends feedback to the application depending on the send buffer queue size.
- *Preferential treatment of marked packets:* When the send buffer queue size increases beyond a certain threshold, DBAT only sends marked packets.

3.1 Motivation

The motivation for designing DBAT is to make the streaming application more adaptive and reactive. The traditional method of changing streamed bit rate based upon packet loss rate and client side buffer underflow is inefficient. By the time the application reacts to the changing available bandwidth, it might be too late due to the delay in the feedback loop and send buffer. The idea behind

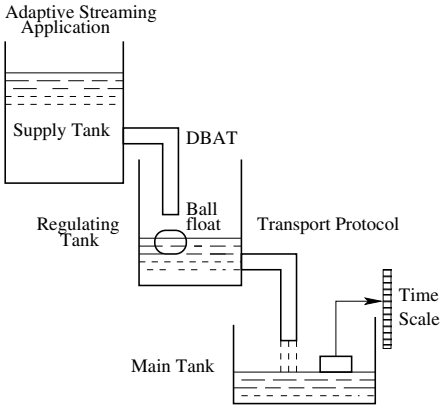

Figure 2. Water Clock Model

DBAT can be easily understood by looking at the water clock model shown in the Figure 2 [15]. In order to maintain a constant flow rate into the main tank of the clock, the water level at the regulating tank is held nearly constant. This constant level is achieved through a float valve, which is essentially a feedback mechanism. Water from an external supply enters the regulating tank through a pipe. When the water level at the regulating tank rises, it forces the floating ball to close against the pipe opening, reducing the input supply rate. When the level drops, the input rate increases. In our streaming architecture, DBAT plays the role of the regulating tank to keep the packet transmission rate at a constant level.

3.2 DBAT Architecture

As shown in Figure 3, DBAT couples the application's control loop with the transport protocol's control loop. Upon connection establishment, the application informs the transport protocol about its streamed bit rate, and the transport protocol tries its best to sustain that rate. It is noted that streaming applications are data limited and as such cannot grab the available capacity. Knowing the applications' desired rate, the transport protocol keeps its sending rate within a certain range. DBAT monitors the send

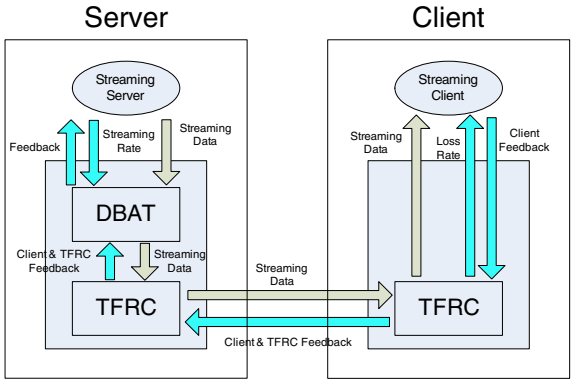

Figure 3. DBAT Architecture

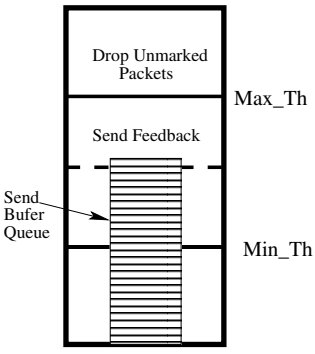

Figure 4. DBAT Queue

```
Min_Th = streamed_bit rate*delay
Max_Th = 2*Min_Th

On each packet arrival, compute the
    weighted avg queue length, Qavg
If Qavg > Max_Th
    preferentially drop unmarked packets
else if ((Qavg > (Max_Th + Min_Th)/2) &&
    (last_feedback_time > RTT ))
    send feedback
```

Figure 5. DBAT Algorithm

buffer queue size and sends feedback to the sender application when the queue size is increased beyond a threshold. As shown in Figure 4, DBAT keeps a minimum threshold (Min_Th) and a maximum threshold (Max_Th) for controlling the send buffer queue size. Minimum threshold is calculated by multiplying the streamed bit rate with the delay. The maximum threshold is set to twice the minimum threshold in order to accommodate some packets while the application takes time to react on sending rate. On each packet arrival, the weighted queue length, *Qavg* is calculated. If the average queue length increases beyond the mid point of minimum and maximum threshold, feedback is sent to the sender. The amount of feedback is limited to at most one per RTT. If *Qavg* grows beyond the maximum threshold, only marked packets are transmitted. The algorithm is given in Figure 5.

4. Experimental Evaluation

In this section, we present the experimental results. We use network simulator, *ns-2* as our preferred vehicle for simulation [1]. Currently the standard *ns-2* distribution does not have any streaming module included. However we have found a streaming module named *Goddard* [5] which is suitable for our experiments. We integrated *Goddard* in *ns-2* and conducted experiments using it. *Goddard* is based on the studied behaviors of *Real Networks* and *Windows Streaming Media* [4]. During streaming the *Goddard* client and server re-select the streamed bit rate in response to network packet loss or re-buffering events that occur when the client's playout buffer gets emptied. The *Goddard* server supports multiple bit rate streaming. For ease of simulation, we only vary the inter-packet gap to stream at the rate of 80, 120, 240, 320, 640, 960 and 1920 kbps.

Goddard does not have any support for TFRC. We modified the code so that we can use TFRC as a transport protocol for streaming. We found that the TFRC implementation in *ns-2* does not have any real data transmission capability which is required by the streaming module. We changed the interface of this implementation so that data can be transmitted with each packet enabling the *Goddard* client and server to exchange media frames. By default the *ns-2* implementation of TFRC has an infinite send buffer. We introduced a send buffer with adjustable size into TFRC. As for TCP we modified the full-TCP implementation of *ns-2* to support adjustable send buffer size. To the best of our knowledge we are the first to conduct experiments involving the interaction of streaming applications with TFRC in *ns-2*. We use the dumbbell topology for simulating (Figure 6) streaming client server communication. The bottleneck link is 1 Mbps. In all cases, one streaming flow is competing with a background FTP flow and some HTTP traffic. The HTTP traffic is generated using empirical data provided by *ns-2*. The background FTP and HTTP traffic simulates a real world scenario where most streaming flows compete with web and FTP flows. The FTP application starts at 0.1 seconds and stops at 200 seconds. The streaming flow starts at 30 seconds and stops at 240 seconds. The bottleneck router queue size is set to twice the bandwidth and delay product of the link. Due to the randomness of the background HTTP traffic, the loss rate of the bottleneck link may vary. Therefore for each scenario, we run the experiment several times and plot the average values only. The client side playout buffer holds 1 second of media data before its playback. The server side send buffer size for TCP and TFRC based streaming is set to a fixed value of 64 packets. This is in line with the deployed Linux and Windows systems where the default send buffer is set to 64KB (64 1KB packet). We assume that for live streaming applications, blocking socket mode is not feasible. Instead in order to reduce the latency, non-blocking mode socket is used and arriving packets at the send buffer are dropped when the send buffer is full.

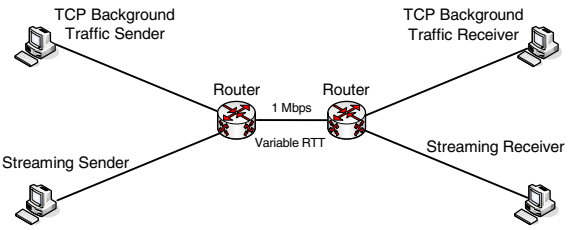

Figure 6. Live streaming simulation topology

We illustrate the throughput, end-to-end application level latency, packet loss and time spend in buffering phase. We set the RTT of the link as 150ms for measuring throughput, latency and packet loss events. We demonstrate the total buffering times in links with 20ms, 150ms and 450ms

RTTs. The throughput is calculated with one second granularity. The latency is calculated by measuring the delay between the time when the streaming sender sends a packet to the transport layer and the time when that packet arrives at the streaming receiver for playback. In all cases, DBAT is used on top of TFRC.

4.1 Throughput

Figure 7 shows the streaming throughput when TCP is used. In this case, the streaming flow gets less than its fair share. The streaming flow is continuously beaten by the competing FTP and background web traffic. Whenever a packet is lost other packets from the TCP receive buffer are not delivered to the client side playout buffer. As the playout buffer size is set to 1 seconds, it runs out when packets are lost in burst. This temporarily pauses media playback. Packet transmission is stopped until a new streamed bit rate is decided through bandwidth probing technique. During this brief pause, other competing TCP flows grabs the available bandwidth and the streaming flow loses its share. Moreover since TCP hides the packet loss information, the streaming application has no way of adjusting the sending rate before the playout buffer runs out. It is noted that due to the background web traffic the FTP flow could not grab the available bandwidth during the first 30 seconds.

Figure 8 illustrates the streaming throughput when TFRC

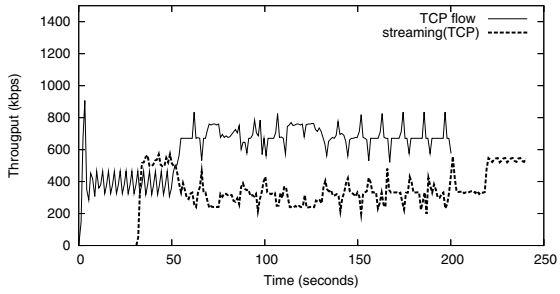

Figure 7. Streaming (TCP) throughput

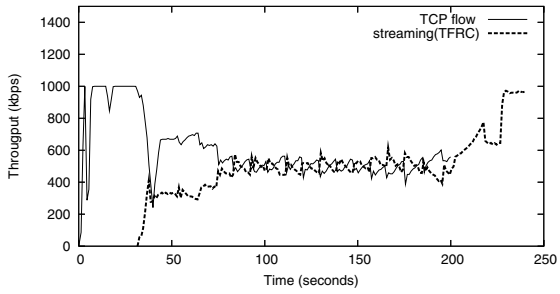

Figure 8. Streaming (TFRC) throughput

is used as a rate control protocol. As soon as the streaming flow starts, the FTP flow slows down and starts to share the bottleneck link equally with the TFRC based streaming flow. Since TFRC does not introduce head-of-line blocking, packet losses does not cause re-buffering events and transmission never pauses. Figure 9 shows the throughput for

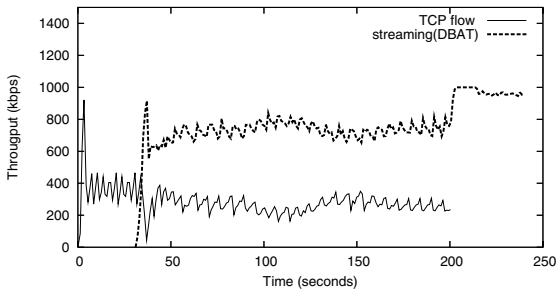

Figure 9. Streaming (DBAT) throughput

DBAT enabled streaming. In this case, streaming flow gets higher service rate. DBAT keeps the send buffer queue size at an optimal level and provides feedback before the send buffer overflows. This reduces the number of lost packets due to send buffer overflow and improves overall throughput.

The above results show that TCP performs poorly for streaming applications. One way to improve the performance of TCP based streaming is to use more client side buffer so that it does not runs out frequently. But for live streaming, the buffering delay should be kept minimum.

4.2 Latency

Figures 10, 11 and 12 show the application level one way latency for TCP, TFRC and DBAT based streaming respectively. In case of TCP based streaming (Figure 10), the

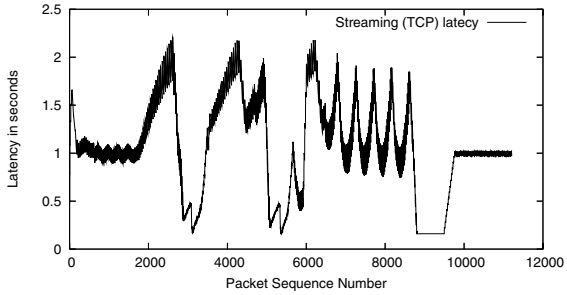

Figure 10. Streaming (TCP) latency

latency between consecutive packets sometimes varies almost by two seconds. This is mostly due to the head-of-line blocking introduced by TCP's in order delivery mechanism.

Although there is some oscillations in TFRC based streaming (Figure 11), the pattern of variation is much smaller than TCP based one. Unlike TCP which cuts the sending rate by half on a single packet loss, TFRC reacts more slowly based on average packet loss event. The 64 KB server side send buffer adds additional delay on top of the one second buffering delay at the client side. For DBAT based streaming

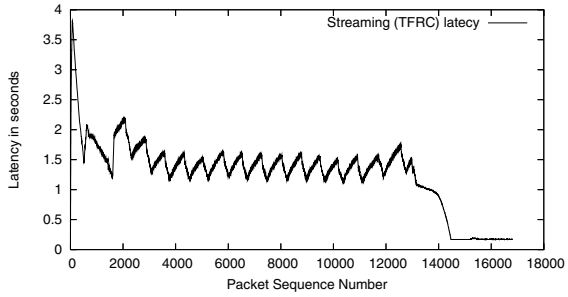

Figure 11. Streaming (TFRC) latency

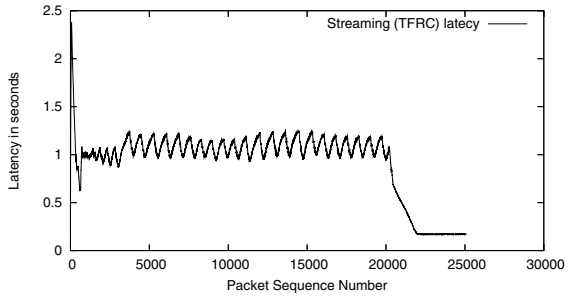

Figure 12. Streaming (DBAT) latency

jitter is the minimum. DBAT reduces latency by keeping the send buffer queue size at an optimal level. The above figures also show that more packets are transmitted with DBAT based streaming since DBAT attains a higher packet rate than TCP or TFRC based streaming.

4.3 Packet Loss

Although streaming audio-video applications may tolerate a few packet losses, one of the motivations behind introducing DBAT is that it should reduce the packet loss in bursts. Figures 13, 14 and 15 show the amount of lost packets for TCP, TFRC and DBAT based streaming. In case of TCP, a large number of packets are lost in bursts. This is due to the inability of the streaming application to adjust sending rate quickly when TCP is used. TFRC improves this situation significantly (Figure 14). For DBAT based streaming, the amount of lost packets is the minimum. Since DBAT provides feedback to the application before the send buffer queue overflows, the application gets some time

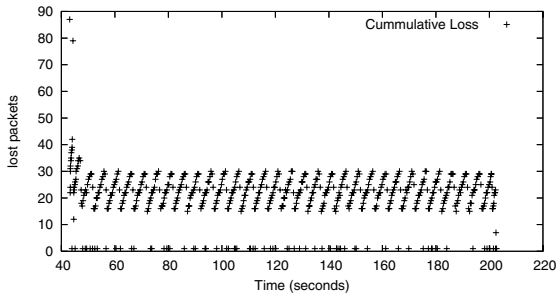

Figure 13. Streaming (TCP) lost packets

to adjust its sending rate. This reduces the number of packets lost due to send buffer overflow.

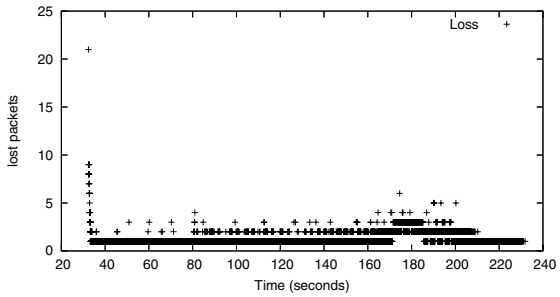

Figure 14. Streaming (TFRC) lost packets

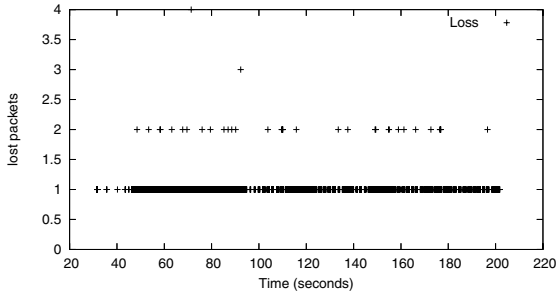

Figure 15. Streaming (DBAT) lost packets

4.4 Buffering Time

The amount of time spent in the buffering state is the most important performance indicator for streaming sessions. The playback remains paused during this period and this is unacceptable for users. Table 1 shows the amount of time spent for buffering with different protocols in three representative scenarios having 20ms, 150ms and 450ms RTTs. We choose these three RTTs to represent national (20ms RTT), international (150ms) and satellite links (450ms) for streaming. For the scenario with 450ms

RTT, TCP based streaming flow spends 37.80% of its total streaming time in the re-buffering state. In the scenario with 150ms RTT, 10.46% is spent in buffering state. TCP shows good performance in the scenario where the link delay is only 20ms. This is because in low delay links TCPs acknowledgment based feedback mechanism is effective. TFRC and DBAT reduces the re-buffering period significantly.

	20ms	150ms	450ms
TCP	0.39%	10.46%	37.80%
TFRC	0.84%	1.12%	4.44%
DBAT	0.43%	1.01%	4.13%

Table 1. Total buffering time

5. Related Work

Wang et al. developed an analytical model for TCP based streaming and concludes that TCP generally provides a good streaming performance when the achievable TCP throughput is roughly twice the media bit rate with only a few seconds of start up delay [17]. Luo et al. presented the result of measurement study based on large streaming media work load and shows that the median time to change to a lower bit rate stream was around 4 seconds [8]. Krasic et al. presented a framework for adaptive video streaming based on priority dropping [13]. Chung et al. developed a transport level protocol named Media Transport Protocol (MTP) which removes the burden of inorder delivery from TCP [5]. Goel et al. proposed a dynamic send buffer tuning approach where the buffer size is kept slightly larger than the TCP congestion window for TCP-based media streaming [7]. Unlike their work, we focus on media streaming on TFRC.

6. Conclusions and Future Work

In this paper we investigated the performance of live streaming. We showed that TCP performs poorly, but TFRC based streaming improves the performance. We proposed a cross layer protocol support framework named Dynamic Buffer Active Tuning (DBAT). DBAT relies on information exchanged between the application layer and the transport layer. We showed that DBAT improves the service rate of streaming, reduces jitter and packet loss. Our next steps will concern the exploration of router assisted approach to ensure fairness (like XCP[11] needs in terms of fairness with TCP[14]). In addition to that we will investigate the performance on DBAT supported streaming for variable bit rate flows. DBAT can easily be tailored to streaming in multicast scenarios by using layered media encoding format.

References

[1] ns-2 network simulator. http://www.isi.edu/nsnam/ns/.

[2] Streaming media growth and content category share:2006-2010. http://www.accustreamresearch.com.

[3] V. Balan, L. Eggert, S. Niccolini, and M. Brunner. An experimental evaluation of voice quality over the Datagram Congestion Control Protocol. In *IEEE Infocom*, Anchorage, AL, USA, May 2007.

[4] J. Chung and M. Claypool. Empirical Evaluation of the Congestion Responsiveness of RealPlayer Video Streams. *Kluwer Multimedia Tools and Applications*, 31(2), November 2006.

[5] J. Chung, M. Claypool, and R. Kinicki. MTP: A streaming-friendly transport protocol. Technical report, Oregon Graduate Institute School of Science and Engineering, 2002.

[6] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[7] A. Goel, C. Krasic, K. Li, and J. Walpole. Supporting low latency TCP-based media streams. Technical report, Worcester Polytechnic Institute, May 2002.

[8] L. Guo, E. Tan, S. Chen, Z. Xiao, O. Spatscheck, and X. Zhang. Delving into Internet streaming media delivery: A quality and resource utilization perspective. In *ACM Internet Measurement Conference(IMC)*, Rio de Janeiro, Brazil, October 2006.

[9] M. Handley, S. Floyd, J. Padhye, and J. C. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. Internet Engineering Task Force, RFC 3448, January 2003.

[10] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM*, pages 314–329, Stanford, California, United States., 1988.

[11] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *ACM SIGCOMM*, 2002.

[12] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *ACM SIGCOMM 2006*, pages 27–38, Pisa, Italy, 2006.

[13] C. Krasic, J. Walpole, and W.-c. Feng. Quality-adaptive media streaming by priority drop. In *13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 2003.

[14] D. M. Lopez Pacheco, L. Lefevre, and C.-D. Pham. Fairness issues when transferring large volume of data on high speed networks with router-assisted transport protocols. In *High Speed Networks Workshop 2007, in conjunction with IEEE INFOCOM 2007*, Anchorage, Alaska, USA, May 2007.

[15] D. Luenberger. *Introduction to Dynamic Systems*, chapter 8, page 297. Wiley New York, 1979.

[16] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *ACM SIGCOMM*, pages 30–314, 1998.

[17] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Streaming via TCP: An analytic performance study. In *ACM Multimedia*, New York City, NY, October 2004.