MTP: A Streaming-Friendly Transport Protocol

Jae Chung, Mark Claypool and Robert Kinicki
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA, 01609, USA
{goos|claypool|rek}@cs.wpi.edu

Abstract— **Today's streaming applications typically use TCP or UDP to transmit media over the Internet. However, streaming over UDP is problematic due to firewalls that restrict UDP penetration and because of the potential for excessive congestion induced by unresponsive UDP flows. Streaming over TCP is also troublesome because media scaling is difficult given current TCP's application programming interface (API) and because the TCP reliable in-order delivery requirement yields damaging media frame reception jitter. Recently proposed TCP-Friendly protocols have faced deployment resistance due to the firewall situation and the inability to demonstrate stability equivalent to that of TCP. To enhance TCP support for delay-sensitive streaming media, this paper proposes the Multimedia Transport Protocol (MTP), a unreliable data transport mode for TCP. MTP gracefully disables TCP retransmissions, uses delay-aware sender buffer queue management, and provides a transparent API that enables streaming media applications to make informed media scaling decisions. MTP is simple to implement, has the proven stability of TCP, and inherits the full benefit of network support for TCP. Through a simulation study that includes implementation and validation of a streaming media client and server, this paper shows that MTP offers streaming performance comparable to that provided by UDP, while doing so under a TCP-Friendly rate.**

I. Introduction

Streaming applications today choose either TCP or UDP as their transport protocol, depending on individual needs. It has been a myth that UDP is the dominant choice for streaming media on the Internet, believing that it is difficult to achieve acceptable streaming performance over TCP. However, Merwe, Sen and Kalmanek [1] report that video on demand (VoD) and live broadcasting applications predominantly use TCP over UDP, with TCP used for about 72% to 75% of all bytes transferred.

Streaming over UDP is undesirable when firewalls block UDP to limit the penetration of streaming traffic. This restriction for UDP also occurs when network address translation (NAT) is employed at end-user routers in home and small corporation networks. Furthermore, the unresponsiveness of UDP streams can lead to excessive congestion at the bottlenecked router. While media scaling can be effectively employed by streaming applications to respond to network congestion, it is often not deployed in an appropriate congestion avoidance fashion [2], [3]. Using streaming repair techniques [4], [5], [6], [7], can partially or fully conceal UDP packet losses thus reducing the incentive for UDP flows to be responsive to congestion.

Recent research has proposed TCP-Friendly streaming transport protocols [8], [9], [10], [11] in the hope they will be used by streaming media applications. However, most TCP-Friendly protocols focus on achieving smooth and TCP-Friendly transmission rates, but with little concern for an application programming interface (API) to meet media scaling performance requirements. Thus, most proposed TCP-Friendly protocols are not streaming-friendly in that they make it difficult to acquire network information and effectively perform media scaling over the protocols. Additionally, very little progress has been made on deployment of TCP-Friendly protocols because they do not have stability equivalent to that of TCP and cannot obtain support from current firewalls.

Taking into account the problems with both UDP and TCP-Friendly protocols for streaming applications, recent research analyzes cases where TCP streaming provides satisfactory performance [12], and suggests ways to improve streaming performance over TCP [13], [14], [15]. Nevertheless, TCP streaming remains problematic due to following reasons: 1) Performing media scaling over TCP is difficult, since TCP's API hides network information such as packet loss rate and round-trip times that are essential for making efficient media scaling decisions [2]; 2) Overestimating the available TCP data rate leads to stream quality degradation due to large queuing delays at the TCP sender [2], [16]; 3) TCP's reliable in-order packet delivery often induces large frame reception jitter that can interrupt streaming media playout.

When a TCP sender's transmission rate is less than the streaming bitrate, media frames are queued and delayed at the sender's protocol buffer. Since high-quality media frames can block transmission of lower-quality media frames in the sender buffer, the delay added by the protocols sender buffer can significantly degrade stream quality when the streaming system is downscaling the media quality. While delay-aware TCP input queue adaptation, such as in [16], can reduce media frame reception jitter, the queue length adaptation alone does not resolve the difficulties in obtaining network state information over TCP nor reduce retransmission-induced jitter. Explicit Congestion Notification (ECN) does avoid TCP packet losses that cause media frame reception jitter, but ECN deployment has been severely hampered by the requirement for router support and serious network security concerns [17], [18].

This paper presents Multimedia Transport Protocol (MTP), a modified form of TCP for streaming that favors prompt and timely datagram delivery service over reliable in-order transmission service. By removing retransmissions from the TCP protocol, MTP instead sends the packet with the highest sequence number in place of a retransmission. By removing both retransmissions and ordered packet delivery from TCP, MTP reduces the high delay and jitter characteristics that make TCP impractical for interactive applications, and makes network information such as packet loss and round-trip time transparent for making media scaling decisions.

MTP offers two modes of transmission at the API: 1) non-blocking transmission mode offering UDP API semantics, and 2) the block-on-full-queue mode of default TCP. It is a common practice that streaming media servers, particularly ones over UDP, operate in a rate-based frame transmission mode, a so called "fire-and-forget" mode, receiving no indications of packet delivery [19]. The non-blocking transmission mode of MTP supports the contemporary rate-based streaming applications by requiring little modification to switch to MTP. Offering UDP transmission semantics requires MTP senders to use a best-effort queue management mechanism to drop packets from applications when the sender buffer is full. For implementation of the non-blocking transmission API, MTP uses a simple drop-front queue management that works well with streaming media. On the other hand, the block-on-full-queue mode of MTP offers prospective streaming applications an advanced control over frame transmissions. For both transmission modes, MTP may additionally use a dynamic queue length adaptation mechanism introduced in [16] to support interactive streaming applications.

MTP has advantages over other TCP-Friendly transport protocols: 1) MTP has the proven stability of TCP since it has the exact congestion avoidance mechanism of TCP; 2) MTP can be implemented as a mode for TCP in the deployment phase as well as a separate protocol to get full support from the existing firewalls; 3) MTP can be easily made available for all operating system distributions, since an MTP implementation can reuse most of an existing TCP implementation; 4) Existing UDP streaming applications can easily switch to MTP with a minimum change using the non-blocking transfer mode, while new streaming applications may use block-on-full-queue option as well. In addition, the Internet community proposes to build an unreliable transport protocol incorporating end-to-end congestion control, called Datagram Congestion Control Protocol (DCCP) [20]. DCCP proposes to support a TCP-like window-based congestion control mechanism (Congestion Control ID 2) similar to MTP and to support TFRC [8] (Congestion Control ID 3), a rate-based end-to-end mechanism. The design and evaluation of MTP for streaming media is a valuable contribution toward the design and evaluation of the DCCP ID 2 congestion control mechanism.

MTP evaluation requires a realistic streaming application that performs media scaling and simulates media buffering and playout. While implementing and evaluating MTP on Linux was considered, this approach was dropped when we were unable to find a reasonable open-source streaming application that could be modified to use MTP. Thus the alternative path of evaluating MTP using NS [21] simulations was selected. This required building MTP into NS based on the existing TCP Reno implementation and designing and implementing a video streaming system, called *Goddard*, into NS based on streaming application behavior observed in [2], [3].

The simulations show that MTP video streams adapt as quickly as UDP streams to available bandwidth and significantly reduce rebuffering events that are common in TCP streams while maintaining other media qualities such as frame rate and picture resolution at TCP stream levels. The results also show that existing UDP streaming application can use MTP with little modification to their media scaling mechanisms to achieve better quality than TCP streams.

The paper is organized as follows: Section II presents the design of MTP; Section III describes the design of the Goddard streaming server and client; Section IV evaluate MTP in comparison with TCP and UDP using Goddard; and Section V summarizes our conclusions and lists possible future work.

II. Multimedia Transport Protocol

Multimedia Transport Protocol (MTP) is a TCP modification that disables retransmissions, while preserving the transmission timings and congestion responsiveness characteristics of TCP. MTP performs slow start, congestion avoidance, fast retransmission and fast recovery as does TCP, yet offers a UDP-like transparent API and provides UDP packet delivery semantics. Namely, MTP does not offer guaranteed or in-order packet delivery.

MTP retains the same loss detection and recovery mechanisms of TCP. Instead of a retransmission, the MTP sender temporally inflates its transmission window and sends a new packet.[1] The inflated transmission window is deflated back when an acknowledgment for the retransmission-replacement packet is received. This temporary transmission window inflation has the effect of not counting the retransmission-replacement packet sent as a new transmission when making the next new packet transmission decision and is required for MTP to have the same transmission behavior at the network layer as that of TCP.

In MTP, a retransmission replacement packet is marked in the TCP header by the MTP sender to distinguish it from a normal packet at the MTP receiver. On reception of a retransmission-replacement packet, the MTP receiver marks as received the oldest outstanding packet in its packet reception window and advances the receiver window in the same manner as a TCP receiver would upon reception of a retransmitted packet. Additionally, the MTP receiver records the sequence number of the replaced new packet, updates the receiver window again and sends an acknowledgment for the highest consecutively received packet sequence number. An MTP receiver does not hold any received packets in the receiver window for in-order delivery, but provides packets to the application as soon as they are received. Thus, the receiver window contains only sequence numbers of received packets for management of acknowledgment packets.

In summary, an MTP implementation requires a mechanism to keep track of replacement packets in the transmission

[1]Note, for this discussion the terms segment and packet are used interchangeably.

starting $ssthresh$=6		Sender State				Receiver State
		Ack	Seq	Wnd	Note	Recv Window
TCP		10	16	6		10
		10	17	6+1	dup1	__ 12
		10	18	6+2	dup2	__ 12 13
	fast-retransmit	10	11	3+3	dup3	__ 12 13 14
	fast-recovery	10		3+4	dup4	__ 12 ... 15
		10		3+5	dup5	__ 12 ... 16
		10	19	3+6	dup6	__ 12 ... 17
		10	20	3+7	dup7	__ 12 ... 18
		18	21	3		18
MTP		10	16	6		10
		10	17	6+1	dup1	__ 12
		10	18	6+2	dup2	__ 12 13
	fast-retransmit	10	19^*	3+3+1	dup3	__ 12 13 14
	fast-recovery	10		3+4+1	dup4	__ 12 ... 15
		10		3+5+1	dup5	__ 12 ... 16
		10	20	3+6+1	dup6	__ 12 ... 17
		10	21	3+7+1	dup7	__ 12 ... 18
		19	22	3		19

Fig. 1. An Example Duplicate Acknowledgment Management of TCP (Reno) and MTP: Wnd = congestion window + the number of duplicate acknowledgments (+ the retransmission replacement inflation for MTP), and the notation n^* for MTP represents packet n with the retransmission replacement bit set in the TCP header.

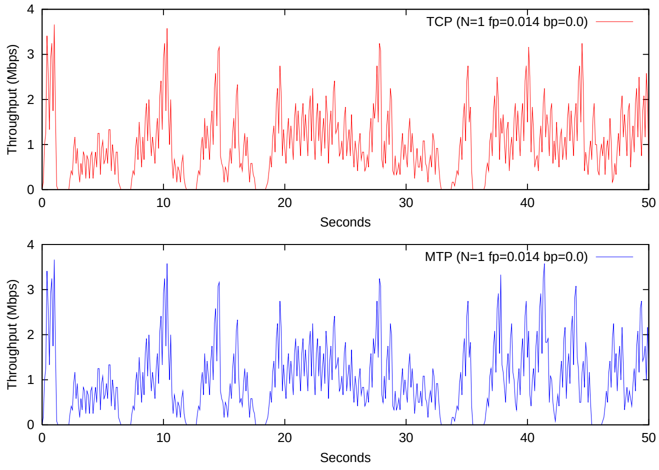

Fig. 2. TCP versus MTP Throughput: The forward network packet loss rate $p_f = 0.014$, the backward packet loss rate $p_b = 0.0$, the round-trip time $RTT = 60$ ms and the bottleneck capacity $C = 100$ Mbps.

window at the MTP sender and a bit in the TCP header, referred to as the *replacement bit*, to distinguish retransmission replacement packets from new packets. We implemented MTP in NS by extending the built-in TCP Reno code. Thus, the subsequent discussion describes the behavior of MTP built upon TCP Reno.

A. Duplicate Acknowledgment Management

When encountering duplicate acknowledgments, the MTP source performs congestion avoidance, fast retransmission and fast recovery as does TCP, and yields identical congestion window movement and packet transmission timings. However, unlike TCP, on reception of a triple duplicate acknowledgment, MTP inflates the transmission window size by the number of not-yet-acknowledged retransmission-replacement packets including the current one, advances the highest sequence number sent thus far, and transmits a new data packet in the place of the retransmission with the highest sequence number and the replacement bit set in the TCP header. In the case there is no new data available in the input buffer on receiving a triple duplicate acknowledgment, the MTP sender transmits the acknowledged lost packet in the transmission window. Retransmission-replacement packets received by a MTP receiver are handled as described above.

Figure 1 provides an example of MTP duplicate acknowledgment management behavior as compared to TCP Reno. In this example, both the TCP and MTP senders detect a single packet loss while they have the congestion window size ($cwnd$) of 6 and are in congestion avoidance mode indicated by $cwnd$ equal to the slow start threshold ($ssthresh$). TCP retransmits the lost packet 11 when getting triple duplicate acknowledgment for packet 10 and halves $cwnd$ to 3. On the other hand, MTP transmits the retransmission replacement packet 19 when detecting loss of packet 11 and halves $cwnd$ to 3. Thus, although the packet sequence number advancements are different, the congestion window movement and packet transmission timings at the network packet level of TCP and MTP are identical for a single packet loss in a window. This is also true for multiple packet losses in a window and for acknowledgment packet losses as long as the sender does not timeout due to lack of duplicate acknowledgments or loss of a retransmitted (or retransmission-replacement) packet during fast recovery.

Figure 2 compares the throughput (measured in 100 ms intervals) of a single TCP and a single MTP flow on a simulated network with capacity of 100 Mbps, round-trip link delay of 60 ms and forward direction packet loss rate $p_f = 0.014$, where the same simulation was run once with a TCP flow and then again with an MTP flow with packet drops generated by the same random seed. The two lines on top of each other before the timeout around 37 seconds show that MTP packet transmission characteristics are identical to that of TCP as long as the senders can effectively detect lost packets via the triple duplicate acknowledgment mechanism. In addition, the TCP and MTP throughputs are identical even for the five timeouts before 37 seconds indicating that TCP and MTP transmission characteristics are equivalent even for some timeout recovery situations. However, the timeout recovery behavior of MTP can be slightly different from that of TCP under some conditions, as discussed in detail in the next section.

B. Retransmission Timeout Recovery

Retransmission timeout in TCP and MTP occurs in two cases: 1) When there are not enough outstanding packets to generates three duplicate acknowledgments for all the lost packets in a window, or there is a failure to deliver the acknowledgments to the sender due to lost acknowledgment packets; 2) When a retransmitted or retransmission-replacement packet generated during fast retransmit is also lost, making the sender unable to return to the congestion avoidance mode after fast recovery.

On a retransmission timeout, the MTP sender tries to recover from the timeout in a manner similar to that of a TCP sender. That is, MTP performs slow start until the congestion window ($cwnd$) reaches the slow start threshold

starting $ssthresh$=2	Sender State Ack	Seq	Wnd	Note	Receiver State Recv Window
TCP		11	1		___ 12 13 ___
	13	14,15	2		13 ___
	14	16	2		14
	15	17,18	3		15
MTP		15^*	1	wnd_base	___ 12 13 ___
	13	$16^* 17^*$	2+0+1	= 14	___ 15
	16	18	2	wnd_base	16
	17	19,20	3	= hi_ack	17

Fig. 3. Example Recovery of TCP and MTP from a Timeout Due to Lack of Duplicate Acknowledgments: Packets 11 and 14 are lost in the previous transmissions. Notions (Wnd and *) are the same as in Figure 1.

starting $ssthresh$=2	Sender State Ack	Seq	Wnd	Note	Receiver State Recv Window
TCP		11	1		___ 12 ___ 14
	12	13,14^1	2		___ 14
	14	15,16	2		14
	14^1	17	2+1	dup1	14
	15	18	3		15
MTP		15^*	1	wnd_base	___ 12 ___ 14
	12	$16^* 17^*$	2+0+1	= 14	___ 14 15
	16	18	2	wnd_base	16
	17	19,20	3	= hi_ack	17
	18	21	3		18

Fig. 4. Example Recovery of TCP and MTP from a Timeout Due to Lack of Duplicate Acknowledgments: Packets 11 and 13 are lost in the previous transmissions. Notions (Wnd and *) are the same as in Figure 1.

($ssthresh$), and returns to congestion avoidance mode. Yet, unlike TCP which restarts the transmission from one below the highest consecutively acknowledged packet sequence number (hi_ack), MTP restarts by transmitting a new packet[2] with the replacement bit set and advances the highest sequence number sent so far (max_seq) by one.[3].

At the beginning of timeout recovery, $cwnd$ is set to one. Thus, if the size of the sender's transmission window is computed as max_seq minus hi_ack as in TCP, MTP cannot transmit a new packet. To resolve this situation, MTP uses a new sender state variable called *transmission window base* (wnd_base) as the lower bound of the the transmission window for computing the transmission window size. When MTP is not in a timeout recovery, wnd_base is set to hi_ack whenever hi_ack is updated upon the arrival of a new acknowledgment packet. This yields the same transmission window size as in TCP. However, at the start of a timeout recovery, the MTP sender sets wnd_base to max_seq before sending out a new packet and declares timeout recovery until hi_ack is less than or equal to wnd_base.

During a timeout recovery, all packets are marked as retransmission-replacement packets by the MTP sender before transmission. However, since the sender does not know the state of the receiver's window upon restart, the sender cannot determine whether a new packet sent is indeed used as a retransmission replacement packet or not. Thus, the MTP sender does not inflate the transmission window on the transmission of an intended retransmission-replacement packet. Instead, the sender monitors acknowledgment packets during timeout recovery and inflates the transmission window when a new acknowledgment packet with the acknowledgment number less than or equal to wnd_base is received. Otherwise, since the updated hi_ack (with the new acknowledgment number) is greater than the value of wnd_base, MTP comes out of the timeout recovery, sets the wnd_base to the updated hi_ack and continues transmission in the normal transmission mode.

Figure 3 provides an examples that illustrates the timeout recovery transmission behavior of MTP in comparison with TCP. In the example, the loss of packets 11 and 14 in the previous transmission window when $cwnd = 4$ cause the retransmission timeout and $ssthresh$ before the timeout is 4. When the retransmission timer expires, TCP reduces $cwnd =$ 1 and $ssthresh = 2$ and retransmits the packet $hi\_ack + 1$ (packet 11). When the acknowledgment comes back for packet 13, TCP increases $cwnd = 2$ and retransmits packet 14 and transmits a new packet 15. Since in this example, packet 14 was also lost in the previous transmission, the retransmitted packet 14 was useful.

When the timeout occurs, MTP transmits wnd_base to max_seq, which is 14 in the example, and transmits packet 15 setting the replacement bit in the TCP header. The receiver, upon reception of the retransmission-replacement packet 15, sends acknowledgment for packet 13. On receiving the acknowledgment, the sender updates $hi\_ack = 15$ and $cwnd =$ 2. Then, the MTP sender compares hi_ack with wnd_base, and inflates the transmission window by one as $hi\_ack \leq wnd\_base$. The sender transmits packet 16 and 17 as it can transmit up to $max\_seq - wnd\_base$ + the retransmission replacement inflation (rrp_inf). Additionally, packets 16 and 17 are marked as able to replace retransmissions as the MTP sender is still in timeout recovery mode. Upon receiving the acknowledgment for packet 16 ($> wnd\_base$), the sender comes out of the timeout recovery by setting $wnd\_base = hi\_ack = 16$ and returns to congestion avoidance mode.

In general, TCP and MTP transmission timings are identical for timeouts due to a single packet loss for a small $cwnd$ or due to two packet losses in a window where the most recently sent packet is lost as in Figure 3. In other cases, MTP behaves slightly more efficiently than TCP.

Figure 4 illustrates the behavior of TCP and MTP under the transmission scenario that is identical to the scenario of Figure 3 except that packet 13 is lost instead of packet 14. In TCP, the retransmitted packet 14 is wasted, since the receiver already has packet 14 in the receiver buffer. This also generates a duplicate acknowledgment that delays advancement of $cwnd$ at the sender. On the other hand, by transmitting new packets each time, MTP avoids duplicate acknowledgments during timeout recovery and may achieve a higher goodput than TCP. Although TCP and MTP transmission timings may be slightly different after a timeout in general, this makes little difference on their throughput as long as the TCP sender does not face three duplicate acknowledgments by unwisely retransmitting packets that the TCP receiver already has in the buffer. In such a case, the TCP sender unnecessarily reduces its congestion window.

[2]In case there is no new application data to send, the MTP sender waits until new data is available.

[3]In an OS implementation stack, the packet sequence number and window advancement should use bytes instead of packets.

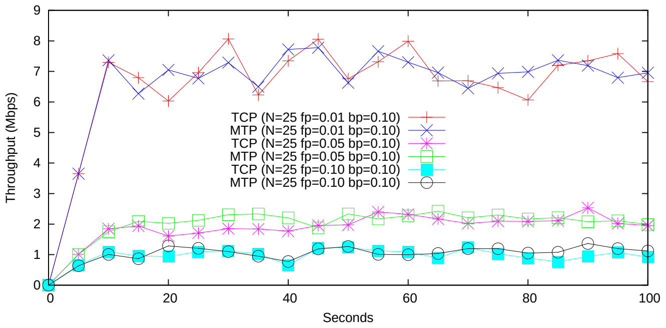

Fig. 5. Aggregate Throughput of 25 TCP and 25 MTP Flows

C. *TCP-Friendliness*

MTP may improve goodput over TCP by avoiding retransmission during a timeout recovery. However, the fact that TCP may falsely back off due to unnecessary retransmissions while MTP does not may bring up a concern about the TCP-Friendliness of MTP. In practice, TCP may rarely get triple duplicate acknowledgments during a timeout recovery, and even so, it does not significantly reduces the TCP throughput. Moreover, newer versions of TCP have option to use selective acknowledgment (SACK) [22], which remedies this drawback of Reno TCP.

Figure 5 compares the aggregate throughput of 25 MTP flows with that of 25 TCP flows for a ranges different packet loss rates in the simulated network. This simulation uses a dumbbell topology with a 100 Mbps links, where the round-trip delay of 25 source-edge node pairs are randomly distributed over the range [140,480] ms. Random packet drop modules are installed at the forward and backward backbone links and backward packet drop rate is set to $p_b = 0.10$ in order to create a fair amount of acknowledgment compression. Then, the forward packet loss rate p_f is varied over 0.01, 0.05 and 0.10. The same simulation is run with same random seed twice for each p_f, once with 25 TCP flows and once with 25 MTP flows. The nearly identical aggregate throughput of 25 TCP flows and 25 MTP flows for all forward packet drop rates in Figure 5 verifies the TCP-Friendliness of MTP flows.

D. *Application Programming Interface*

MTP offers two modes of transmission at the application programming interface (API): block-on-full-queue mode offering TCP transmission API semantics, and non-blocking transmission mode offering UDP transmission API semantics.

When the block-on-full-queue transmission mode is used, a user process sending a packet is blocked while the MTP sender buffer is full. The block-on-full-queue mode is for prospective streaming applications or existing TCP streaming applications that use non-blocking socket write system calls in a polling manner to detect whether the transport sender queue is full and use the information to make media scaling decisions. For the block-on-full-queue transmission mode, an MTP sender may also use the dynamic queue length adaptation scheme introduced in [16] to reduce MTP queuing delays. The block-on-full-queue transmission mode is not discussed or evaluated further in this paper.

The non-blocking transmission mode, which can also be used with dynamic queue length adaptation for delay critical interactive streaming, is designed for existing UDP streaming applications, so that they can easily adapt to MTP with little modification. In order to implement the non-blocking transmission API, MTP uses drop-front queue management on the MTP sender input queue. Drop-front queue management works better with delay sensitive streaming media transmissions than does drop-tail queue management, because a prompt notification of packet loss helps streaming applications to timely perform media scaling or selective retransmission. Especially when scaling down, dropping packets from high quality media frames in the queue helps timely transmission of the lower quality media frames. Thus, drop-front queue management helps prevent the media buffer at the application receiver from running out of frames avoiding an interrupt in the media playout while re-buffering. As in the block-on-full-queue mode, the non-blocking transmission mode can also be used with a dynamic queue length adaptation mechanism to enhance the support for delay-critical interactive streaming. Evaluation of such a combination is left as future work.

Although MTP offers an API that transparently provides underlying network information such as packet losses or round-trip time to applications above, MTP can also explicitly provide network information such as effective MTP transmission rate to help streaming applications. However, the exact network information, computation and format that should be provided to be useful for media scaling requires further study.

III. GODDARD STREAMING CLIENT AND SERVER

We design and implement in NS a streaming system (client and server) called *Goddard*.[4] Goddard is designed based on the behaviors of Real Networks streaming media and Windows Stream media, as observed in measurements studies [2] and [3], respectively. The Goddard streaming client and server use packet-pairs [23], [24], [25] to estimate the bottleneck capacity and select an appropriate media encoding level before streaming. During streaming, Goddard client and server reselect the media to stream (i.e., perform media scaling) in response to network packet losses or re-buffering events that occur when the client playout buffer empties. Goddard also simulates frame playout of the received media at the client, allowing frame rate and jitter to be measured for performance evaluation. To the best of our knowledge, Goddard is the first and only realistic streaming system available in NS. Our contribution of Goddard to the research community[5] represents an additional contribution of this work.

As in commercial systems, the Goddard server supports multiple levels of encoded media that are configured by giving the frame size and the frame rate for each scale level. A sample media scale configuration that simulates multiple level encoding of a high quality Internet video is shown in Figure 6. In addition, the Goddard server has an option for setting the maximum fragment size for fragmenting large media frames

[4]Our streaming system is named after Robert Goddard, the "Father of Modern Rocketry" and a WPI alumnus.

[5]http://perform.wpi.edu/downloads#goddard

Level	Frame Size	Frame Rate	Bitrate
0	1 KB	10 FPS	80 Kbps
1	1 KB	15 FPS	120 Kbps
2	2 KB	15 FPS	240 Kbps
3	2 KB	20 FPS	320 Kbps
4	4 KB	20 FPS	640 Kbps
5	4 KB	30 FPS	960 Kbps
6	8 KB	30 FPS	1,920 Kbps

Fig. 6. Sample Media Scale Levels

Parameter	Default Value	Description
$pkp\_timeout\_interval$	2 seconds	Packet-pair timeout interval
buf_factor	1.5	Buffering rate factor
$play\_buf\_thresh$	5 Seconds	Threshold to start playout
$loss\_monitor\_interval$	5 seconds	Loss monitoring interval
$downscale\_frame\_loss\_rate$	0.05	Down-scale frame loss rate
$upscale\_interval$	60 seconds	Up-scale decision interval
$upscale\_frame\_loss\_rate$	0.01	Up-scale frame loss rate
$upscale\_limit\_time\_factor$	3	Up-scale limit time factor

Fig. 7. Goddard Client (Gplayer) Parameters with Default Values

before transmission. Typically, the maximum fragment size would be set to the maximum transmission unit (MTU) of the underlying network. The Goddard client, also called *Gplayer* for Goddard Player, has the configuration parameters shown in Figure 7. The default parameter values are set based on the observations in [2], [3].

Similar to commercial streaming systems, the Goddard client and server use three communication channels for a streaming session: a control cannel using a TCP connection, a UDP packet-pair channel, and a media streaming channel that can be TCP, UDP or MTP. When setting up a streaming session, the Goddard server sends the list of supported media scale levels to the Gplayer using the control channel. Then, Gplayer sets a timer with $pkp\_timeout\_interval$ and requests the server to send a pair of UDP packets to estimate the capacity of the network path. If any one of the packet-pairs is lost, the packet-pair timer expires and Gplayer will send a request for another packet-pair to the Goddard server. On successful reception of a packet-pair, the capacity of the network path is computed by dividing the packet size by the dispersion [26]. Then, Gplayer selects the largest media scale level with a bitrate less than the computed capacity and notifies the server.

Gplayer also notifies the server of the buf_factor before starting streaming to determine how much the server should increase the transmission rate during media buffering periods. The Goddard client and server operate in two modes: *buffering* or *streaming*. During buffering, the Goddard server transmits the chosen media frames at the rate of buf_factor times the streaming bitrate, where buf_factor used for commercial streams typically ranges from 1.5 to 4 [2], [3]. Gplayer maintains a media playout buffer and a playout threshold ($play\_buf\_thresh$). When the Goddard server starts media transmission in buffering mode, the Gplayer buffers the frames received in the media buffer. When the media buffer size (given in playout time) reaches the $play\_buf\_thresh$, Gplayer tells the server to switch to streaming mode and starts playing the media according to the timing described for the current media scale level. If the media buffer runs out of frames, Gplayer stops media playout and switches back to buffering mode. At this time, Gplayer re-selects the largest media scale level with a bitrate less than the average received throughput for the previous control interval. Then, Gplayer tells the server to transmit frames of the new scale level at the buffering rate, that is the streaming bitrate times buf_factor.

When a Goddard streaming session uses UDP or MTP for the media channel, Gplayer can also use frame loss information to make media scaling decisions. In this case, Gplayer monitors the frame loss rate each time it receives a media frame. When it is at least $loss\_monitor\_interval$ since the last scale adjustment decision was made and the frame loss rate is greater than $downscale\_frame\_loss\_rate$, Gplayer scales the media down one level if the current scale level is not already at the minimum. If the current scale is at the minimum, Gplayer maintains the current scale level. The default value for $downscale\_frame\_loss\_rate$ is set to 0.05 according to [3].

Gplayer also makes decisions to scale the media up to a higher level, but does so slowly and gently. Gplayer increases the scale level by one if the frame loss rate of the stream is less than $upscale\_frame\_loss\_rate$ for $upscale\_interval$ since the last time scaling decision was made and the bitrate of the stream after the increase is less than or equal to the estimated network capacity. Also, in order to reduce the chance of playout interruption, Gplayer limits scaling up to one below the last scale level that caused media re-buffering. This limit on scaling up is heuristically relaxed by one scale level if the stream maintains good quality (i.e., no scale down events) for $upscale\_limit\_time\_factor$ times the $upscale\_interval$. The default for $upscale\_interval$ is set to 60 seconds, a value from the observed range (30 to 90 seconds) during the streaming measurement studies in [2], [3].

Thus, Goddard simulates a realistic streaming video application that performs media scaling, buffering and playout. Implementation of support for video frame dependencies, selective retransmission or other media repair mechanisms are left as future work.

IV. Evaluation of MTP

This section evaluates MTP by comparing the performance of Goddard streaming video over TCP, UDP and MTP through detailed simulations. MTP is evaluated under two different network configurations: a network with limited capacity on the local link, and a network with a congested backbone link. The limited capacity local link simulations are designed to evaluate the media scale adaptation of Goddard over TCP, UDP and MTP on the network where the end-host connection link capacity is less than the maximum stream capability, as will often occur for the home user. The congested backbone link simulations are designed to evaluate the performance of the Goddard streams over TCP, UDP and MTP on a network path congested with many flows.

The simulations use an extended dumbbell network topology that adds an intermediate node for each of the end-user nodes (e_k) as shown in Figure 8. The intermediate nodes (i_k) simulate a 750 Kbps DSL modem (symmetric up- and down-link capacity) for the limited local bandwidth study

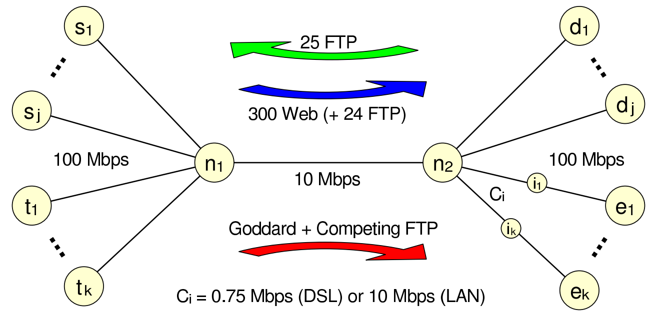

Fig. 8. Network Topology

in Section IV-A, and a 10 Mbps router for the congested backbone link simulations in Section IV-B. The backbone link ($n_1 \leftrightarrow n_2$) capacity is set to 10 Mbps, whereas all the other network link capacities are set to 100 Mbps. The link delays of the streaming and the competing FTP paths ($t_k \rightarrow e_k$) are set to 70 ms giving 140 ms of round-trip link delay. The round-trip link delays of the other normal dumbbell paths ($s_j \rightarrow d_j$) are randomly uniformly selected over the range [60:1000] ms, based on measurements in [27]. All the links use drop-tail queues. The physical queue limit is set to 20 Kbytes for the 750 Kbps links, and 500 Kbytes for the 10 Mbps and 100 Mbps links. The queue limits have little effect on the non-saturated 100 Mbps link traffic, and are approximately equal to the bandwidth-delay product for the 750 Kbps and 10 Mbps links given the mean round-trip time. The maximum network packet size is set to 1000 bytes.

On the normal dumbbell paths ($s_k \leftrightarrow d_k$), each simulation has 25 backward direction bulk transfer FTP flows and 300 forward direction background Web sessions (using the Webtraf code built into NS) that start evenly distributed during the first 30 seconds. Based on settings from [28], [29], each Web session requests pages with 2 objects drawn from a Pareto distribution with a shape parameter of 1.2 and an average size of 5 Kbytes. The Web sessions have an exponentially distributed think time with a mean of 7 seconds, which results in an average utilization of about 2.5 to 3 Mbps of the 10 Mbps capacity, a fraction typical of some Internet links, such as in [30]. For simulations in Section IV-B, additional 24 forward direction bulk transfer FTP flows are used on the normal dumbbell paths to load the backbone link.

The Goddard server uses configuration with the seven media scale levels shown in Figure 6 and a fragment threshold of 1 Kbyte. The Goddard client uses the default configuration parameters shown in Figure 7.

MTP is evaluated by analyzing the network and application layer performances of Goddard video streams over TCP, UDP and MTP. The streaming performance is measured in terms of average sustained scale level, frame rate, durations of initial buffering and media playout, re-buffering event count, TCP-Friendliness, TCP-Friendly rate adaptation time, and frame reception jitter. When there are lost frames, frame reception jitter is computed by taking the inter-frame arrival time of two consecutively received frames divided by the number of lost frames between the two received frames plus one.

A. Limited Local Bandwidth

This experiment shows the media scale adaptation of the Goddard streams over TCP, UDP and MTP on the network where the end-host connection link capacity is less than the maximum stream capability. For this study, the intermediate link capacity (C_i) is set to 750 Kbps, a typical capacity of a DSL modem. This creates a capacity limited streaming condition since the Goddard streams are configured to support the maximum streaming bitrate of 1,920 Kbps. On the capacity limited paths ($t_k \rightarrow e_k$), 3 Goddard streams (TCP, UDP and MTP) and 4 bulk FTP flows are simultaneously started at 100 seconds and stopped at 400 seconds, where the background Web traffic and reverse FTP traffic are running from the beginning of the simulation. This traffic aggregate does not cause a congestion at the backbone link, since the total maximum expected traffic rate is well under the backbone link capacity (10 Mbps). The maximum expected traffic rate is 8.75 Mbps (7×0.75 Mbps + 3 Mbps of Web and acknowledgments traffic). However, the traffic is to simulate a utilized backbone link for realistic transmission timing variations to the Goddard streams. We run the simulation 7 times with different random seeds to avoid biases in any one measurement.

Figure 9 shows the media scaling dynamics of the TCP, UDP and MTP streams from one of the simulations, and Figure 10 shows frame reception jitter for the corresponding streams. Figure 9 show that for all three streams the packet-pairs estimation of capacity choose the initial scale level of 4 providing streaming bitrate of 640 Kbps. The streams start transmission in the buffering mode at the rate of 960 Kbps, 1.5 times faster than the streaming bitrate and larger than the local connection capacity of 750 Kbps.

The high buffering bitrate causes TCP to probe for available bandwidth, incurring packet losses at the local link. The TCP stream experiences an unfortunate sequence of packet drops at about 140 seconds, which causes the large frame jitter and results in a media re-buffering event. TCP scales down a media level and stays at that level for the life of the stream. The smooth TCP stream jitter after 240 seconds in Figure 10 indicates that the TCP sender does not have enough packets for bandwidth probing as the stream scales down and switches to the streaming mode.

The high buffering bitrate also causes frame losses for the UDP stream in the buffering period. However, the UDP stream tolerates the initial frame losses and maintains its initial scale level, since it can tolerate the packet loss rate for the initial measurement interval is less than 5%. In addition, the UDP stream mains low frame reception jitter at the mean (μ) of 50 ms that is the playout interval of the 20 frames-per-second (FPS) video, and a standard deviation (θ) of 6 ms.

The MTP stream scales down the media bitrate like the TCP stream but for to a different reason. The small number of packet losses (or frame losses) during MTP bandwidth probing scarcely affect the performance of the stream, since MTP immediately delivers received packets to the application and does not increase the frame reception jitter at the client. However, scale-down decisions are made due to the frame losses that occur during the initial buffering period at the

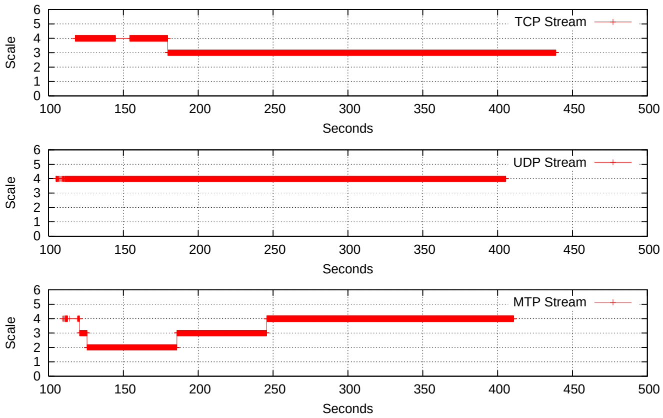

Fig. 9. Example Media Scale Dynamics (Run 0)

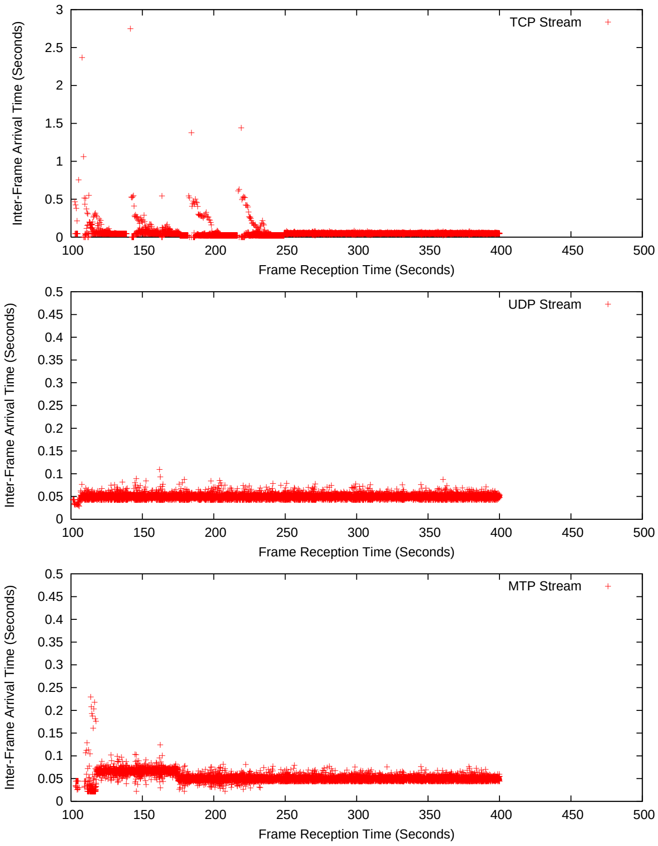

Fig. 10. Example Media Frame Reception Jitter (Run 0)

input buffer of the slow-starting MTP sender. Figure 9 shows that large frame loss bursts at the start of the MTP stream cause it to scale down two levels consecutively in 10 seconds. The MTP stream does scale back up to the initially selected scale level, but slowly and conservatively due the large up-scale decision interval (60 seconds). These inefficient scaling decisions made by the MTP stream at the initial buffering period can be avoided by using selective retransmission, which is supported by most commercial streaming products [31]. In addition, Figure 10 shows that the MTP frame jitter (μ = 52 ms, θ = 11 ms) is as compatibly low as the UDP jitter throughout the stream lifetime for the local connection

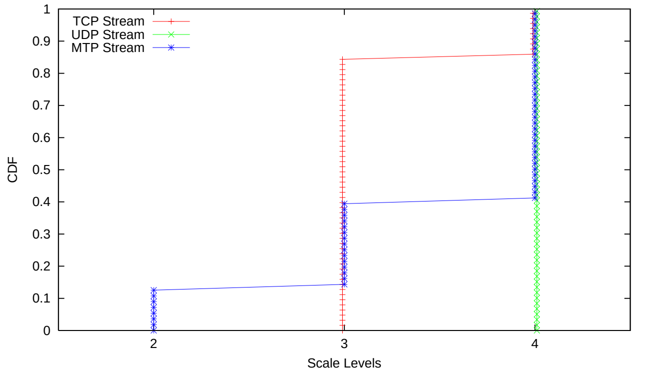

Fig. 11. CDF of Streamed Media Scale Levels

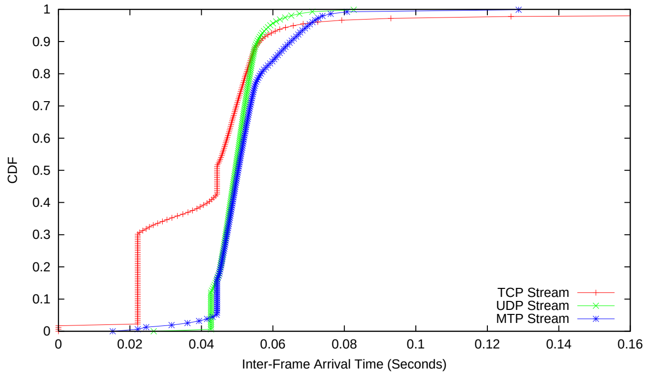

Fig. 12. CDF of Media Frame Reception Jitter

bandwidth constrained condition.

Figure 11 and Figure 12 provide cumulated density function of the streamed media scale level dynamics and the inter-frame reception time (jitter) of all seven simulations. Figure 11 shows that the UDP streams stay at level 4, the highest level allowed by the local link, all the times, and the TCP streams stay at level 3 about 85% of the times. The MTP streams achieve the scale level of the UDP streams (level 4) about 60% of times, although about 12% of the times they stay at the level 2 due to the inefficient scaling decisions made during the initial buffering periods.

Figure 12 confirms that that the frame reception jitter of the MTP streams is as smooth as that of the UDP streams under a normal condition where the local connection link is the only bottleneck of the network path. In addition, Figure 12 reveals the ineffectiveness of media scaling over TCP as well as the effect of TCP's reliable in-order packet delivery on frame reception jitter. The TCP streams' inter-frame arrival times less than 50 ms, the playout interval of the 20 FPS video (level 3 and 4), indicate that about 40% the frames are sent in the buffering mode and/or delayed in the TCP receiver buffer due to packet losses. Especially, the inter-frame arrival time of zero in Figure 12 indicate that two or more consecutive frames are delayed in the TCP receiver buffer and delivered to the Gplayer at once.

Figure 13 shows the initial buffering time (top), the streamed video play duration including re-buffering time (bottom) for the TCP, UDP and MTP streams. Figure 13 (top) shows that

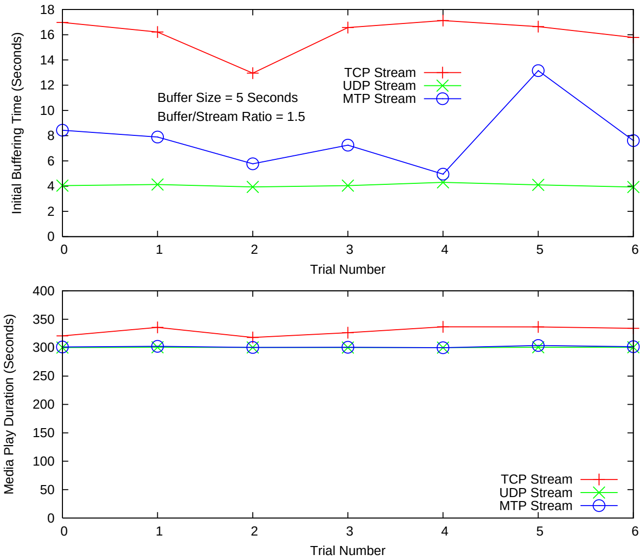

Fig. 13. Initial Buffering Time, Video Play Duration (including intermediate buffering times)

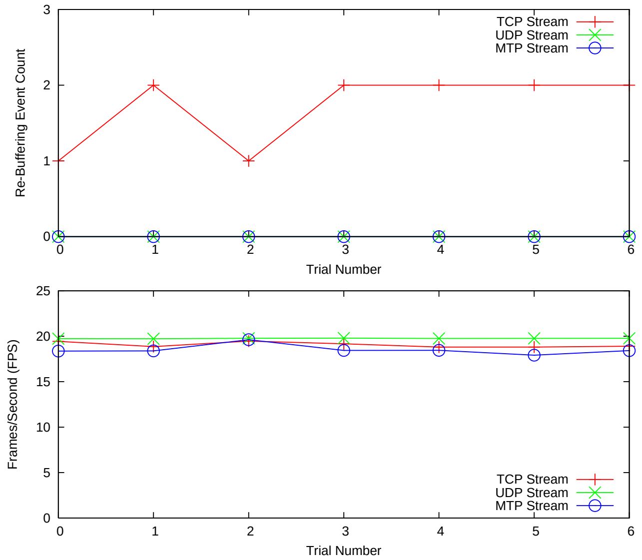

Fig. 14. Re-buffering Event Counts and Average Frame Playout Rate

the TCP streams need significantly more initial buffering time than the UDP streams. The MTP streams also require more time for initial buffering than the UDP streams, however a half as much as the time taken by the TCP streams in average. Figure 13 (bottom) shows that the media play durations of the UDP and MTP streams are bounded almost exactly by the playout length of the video. TCP streams do take a little more time to finish playing the video due to re-buffering events.

Lastly, Figure 14 shows the re-buffering event count (top) and the average frame playout rate (bottom) computed as the number of played frames divided by the play durations including the play-halted re-buffering periods. Figure 14 (top) shows that all the TCP streams incur one or two re-buffering events, while the UDP and MTP streams have none. Figure 14 (bottom) shows that the UDP streams achieved frame rate close to the maximum, 20 FPS. The TCP streams achieve about 19 FPS on average although they experience couple of long pauses in media playout due to media re-buffering. The MTP streams achieved between about 18 to 19 FPS, since the MTP streams often scale down to the 15 FPS level-2 encoded media in the beginning. Thus, all three types of stream perform similarly to one another in terms of frame playout rate, although the TCP streams have a couple of pauses during the playout.

This section showed that streaming over TCP, UDP and MTP can achieve compatible performances under a lightly-loaded network condition where the network usage is mostly limited by the local connection capacity. In addition, it is shown that MTP eliminates media play interruptions caused by TCP, and works well with a streaming application designed for UDP with little modification.

B. Backbone Link Congestion

This experiment shows the performance of the Goddard streams over TCP, UDP and MTP on a network path congested with many flows. For this study, the intermediate link capacity (C_i) is set to 10 Mbps, not to be a bottleneck in the network path. As each simulation starts, the backbone link is loaded with 300 Web sessions (using the Webtraf code built into NS) and 24 FTP flows in the forward direction dumbbell path ($s_j \rightarrow d_j$), and acknowledgments of 25 FTP flows in the backward direction ($s_j \leftarrow d_j$). A Goddard stream and a bulk FTP flow are started at 100 seconds on $t_1 \rightarrow e_1$ and $t_2 \rightarrow e_2$, respectively, and stopped at 400 seconds. The simulation set is composed of three simulations with the same random seed but different streaming transport protocols, TCP, UDP and MTP. Each simulation set is repeated 7 times with different random seeds to account for variance in the measurements.

Figure 15 shows the media scaling dynamics of the TCP, UDP and MTP streams from one of the simulation sets, and Figure 16 shows the frame reception jitter of the corresponding streams. During streaming (100 to 400^+ seconds), a fair bandwidth share with this traffic load is about 270 Kbps ((10 Mbps − background_traffic) / 26 flows), where the background traffic takes about 2.5 to 3 Mbps. Thus, a TCP-Friendly streams' will chose a media scale level of 2 (a 240 Kbps stream) given the traffic conditions.

Figure 15 shows that the TCP stream suffers from overestimating the available network bitrate in the beginning. The TCP stream makes a scale-down decision about 42 seconds after it starts streaming. During this period, the Goddard server is in buffering mode and pushes about 15 Mbytes of frames into the underlying TCP buffer, taking about 450 seconds to transmit all the high quality frames based on the 270 Kbps fair share assumption. In fact, the TCP stream takes about 500 seconds to drain the frames from the TCP sender queue. This huge TCP sender queue size is due to the unrealistic NS TCP implementation that does not simulate a limited input buffer size. For most current Unix kernels, TCP uses a send buffer of at least 64 Kbytes [16]. Although exaggerated, the TCP streaming simulation results show the difficulty in media scaling over TCP.

Unlike the TCP stream, the Goddard server and client over

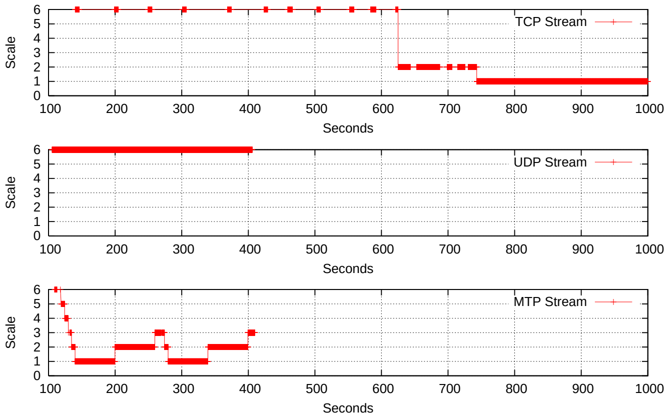

Fig. 15. Example Media Scale Dynamics (Set 0)

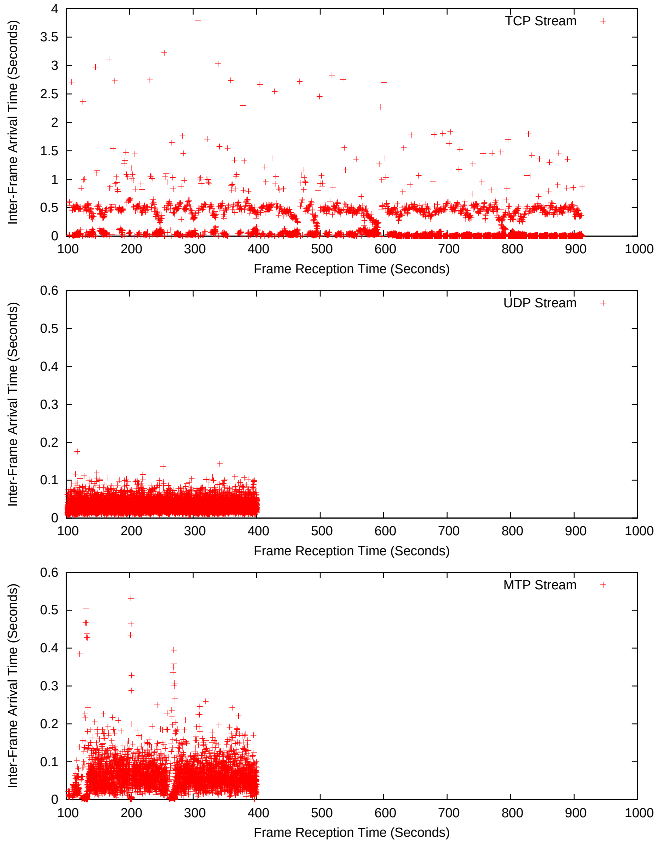

Fig. 16. Example Media Frame Reception Jitter (Set 0)

UDP finishes streaming and playout of the media with no delay. However, it streams the highest quality media that has a streaming bitrate much higher than the fair share, since the streamed media quality is not significantly degraded by small network packet losses. The measured packet loss rate at the backbone link during streaming is about 0.005. This illustrates how a UDP stream can be TCP-unfriendly during congestion, although the stream can adapt to the limited local connection capacity.

The MTP stream shown in Figure 15 inherits the good characteristics of both the TCP and UDP streams. That is, the MTP stream quickly finds an appropriate scale level in

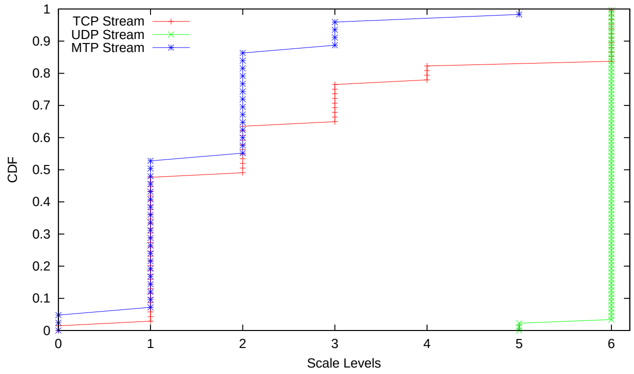

Fig. 17. CDF of Streamed Media Scale Levels

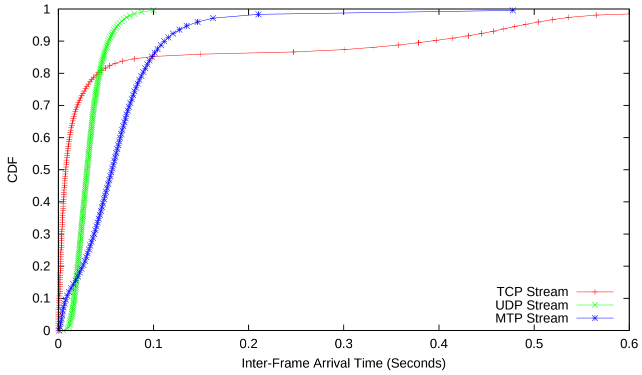

Fig. 18. CDF of Media Frame Reception Jitter

response to network congestion and achieves both uninterrupted stream playout and a TCP-Friendly streaming bitrate. In the beginning, the MTP stream quickly scales down to the level-1 media, and then at 200 seconds, scales up to level-2 media. When the MTP stream tries level-3 media at around 260 seconds, the buffer in the MTP sender overflows and the Goddard client and server scale back.

Figure 16 shows that the TCP stream has the highest frame reception jitter (μ = 82 ms, θ = 213 ms), followed by the MTP stream (μ = 64 ms, θ = 60 ms) and the UDP stream (μ = 33 ms, θ = 16 ms). Comparing the MTP and UDP streams, the UDP jitter is lower than that of MTP. The mean UDP inter-frame arrival time of about a half that of the MTP stream is due to the differences in the scale level. The UDP stream uses the highest quality media with playout interval of 33 ms, while the MTP stream's media playout interval is about 67 ms. Comparing the θ/μ ratio shows UDP is a little smoother than, but comparable to MTP. However, the jitter of the TCP stream is an order of magnitude higher than UDP or MTP , even after the TCP sender clears the highest quality frames from its buffer at 600 seconds. This confirms that the main source of the TCP's streaming unfriendly delay and jitter is the retransmission mechanism that provides reliable in-order packet delivery.

Figure 17 and Figure 18 summarize the streamed media scale level dynamics and the inter-frame reception time (jitter) of all seven sets of simulations in cumulative density functions. Figure 17 confirms that Goddard over MTP streams chooses

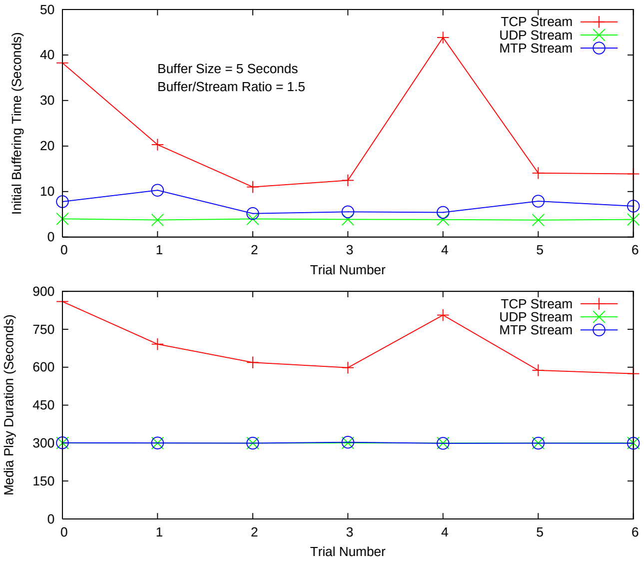

Fig. 19. Initial Buffering Time and Video Playout Duration (including buffering times)

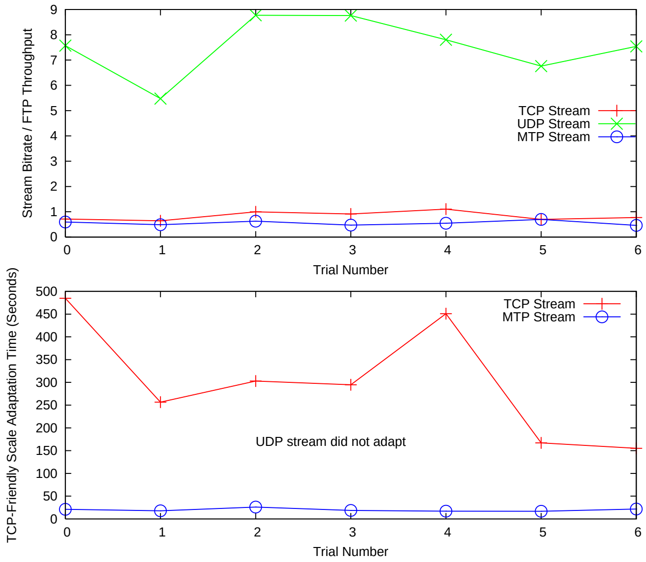

Fig. 21. TCP-Friendliness of Media Streams and TCP-Friendly Rate Adaptation Time

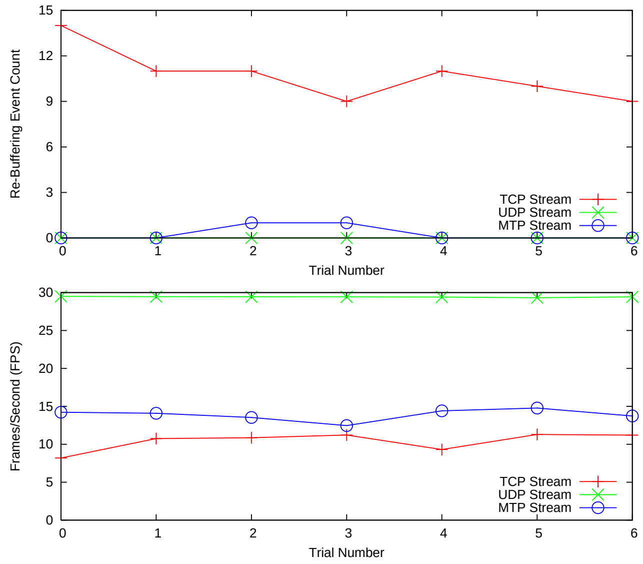

Fig. 20. Re-buffering Event Counts and Average Frame Playout Rate

the correct media scale level (it has a media TCP-Friendly bitrate so it must choose a scale level 2 or lower) most of the time. In addition, Figure 18 shows that the MTP frame reception jitter is consistently low. About 95% of MTP's inter-frame arrival times are under 134 ms, two times the playout interval (67 ms) of the streamed media. This means that Gplayer can play 95% of the received video frames after a buffering delay of only 67 ms when the one-way delay of the stream path is about 100 ms and the average packet loss rate is 0.44%. Given that streamed video bitrate is about 120 to 240 Kbps, typical of Internet videoconferencing, these results suggest the potential for MTP to be used as a streaming transport protocol for interactive applications as well as non-interactive applications.

Figure 19 shows the initial buffering time of the streamed video (top) and the playout duration including re-buffering time (bottom), and Figure 20 shows the re-buffering event count (top) and the average frame playout rate (bottom) for the TCP, UDP and MTP streams. Figure 19 (top) shows that the TCP streams need significantly more initial buffering time than the UDP and MTP streams. Figure 19 (bottom) shows that the media play durations of the UDP and MTP streams are bounded almost exactly by the playout length of the video, while the TCP streams take two to three times longer to playout. Figure 20 (top) shows that the TCP streams incur frequent re-buffering events, while the UDP and MTP streams have none or at most one. Figure 20 (bottom) shows that the TCP streams achieve a low average frame playout rate due to frequent re-buffering pauses, while the UDP and MTP streams achieve frame playout rates close to the encoded frame rates.

Lastly, Figure 21 (top) shows the average throughput ratio of the media streams in comparison to the competing bulk FTP flow, and Figure 21 (bottom) shows the time each stream takes to adapt to the TCP-Friendly rate. The UDP streams are extremely TCP-unfriendly, taking about 6 to 9 times the throughput of the competing bulk FTP, and never adapt to a TCP-Friendly rate. Both the TCP and MTP streams have throughput less than that of the bulk FTP flow, since the Goddard servers do not always have frames available when MTP or TCP can send. The MTP streams quickly adapt their media scale bitrate to the TCP-Friendly rate, while the TCP streams take two to three minutes to adapt their media scale bitrate to the TCP-Friendly rate.

V. Summary

This paper presents the design and evaluation of Multimedia Transport Protocol (MTP). MTP is an alternate to UDP for streaming and other delay sensitive Internet applications that favor prompt and timely datagram delivery service over the reliable transmission service of TCP. Removing retransmissions from TCP removes delays due to retransmission at the TCP sender and packet ordered preservation at the TCP receiver, and allows MTP to reveal network information essential for media scaling. MTP supports non-blocking transmission using input queue management as well as block-on-full-queue transmission to help existing UDP-based streaming applications to switch to MTP with little modification. MTP has the exact congestion avoidance mechanism and proven stability of TCP, and can be implemented as a mode of TCP during incremental deployment to take advantage of firewall support for TCP traffic.

MTP is implemented in NS by modifying the built-in Reno TCP implementation. In order to evaluate MTP, the Goddard streaming client and server are designed and implemented. Goddard estimates the bottleneck capacity, selects the media level to stream, performs media scaling during streaming, and simulates playout of the received media at the client. To the best of our knowledge, Goddard is only realistic streaming application in NS.

Our simulation results show that MTP video streams inherit the good characteristics of both TCP and UDP streams. MTP streams are TCP-Friendly, but can still quickly and effectively perform media scaling and adapt to the available TCP-Friendly bitrate. Thus, most of the time, MTP streams avoid interruptions to the streamed media playout. Existing UDP streaming application can use MTP with little modification to their media scaling mechanisms, achieving better quality streams than TCP streaming applications. Additionally, our simulation results show that MTP dramatically reduces media frame reception jitter from TCP's reliable in-order packet delivery mechanism, and illustrates the potential of MTP as a streaming transport protocol for interactive as well as non-interactive applications.

Future work includes enhancing the MTP API to provide useful network information that can be utilized to improve media scaling and repair performance, and evaluate MTP with dynamic queue length adaptation for interactive streaming. Other future work is to implement and evaluate MTP under Linux with an open-source streaming application.

REFERENCES

[1] J. van der Merwe, S. Sen, and C. Kalmanek, "Streaming Video Traffic: Characterization and Network Impact," in *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, Boulder CO, USA, August 2002.

[2] J. Chung, M. Claypool, and Y. Zhu, "Measurement of the Congestion Responsiveness of RealPlayer Streaming Video Over UDP," in *Proceedings of the Packet Video Workshop (PV)*, Nantes, France, April 2003.

[3] J. Nichols, M. Claypool, R. Kinicki, and M. Li, "Measurements of the Congestion Responsiveness of Windows Streaming Media," in *Proceedings of the 14th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 2004.

[4] J.-C. Bolot, S. Fosse-Parisis, and D. Towsley, "Adaptive FEC-Based Error Control for Internet Telephony," in *Proceedings of IEEE INFOCOM*, March 1999.

[5] Y. Liu and M. Claypool, "Using Redundancy to Repair Video Damaged by Network Data Loss," in *Proceedings of IS&T/SPIE/ACM Multimedia Computing and Networking (MMCN)*, San Jose, California, USA, January 2000.

[6] C. Padhye, K. Christensen, and W. Moreno, "A New Adaptive FEC Loss Control Algorithm for Voice Over IP Applications," in *Proceedings of IEEE International Performance, Computing and Communication Conference*, Feburary 2000.

[7] K. Park and W. Wang, "QoS-Sensitive Transport of Real-Time MPEG Video Using Adaptive Forward Error Correction," in *Proceedings of IEEE Multimedia Systems*, June 1999, pp. 426–432.

[8] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-Based Congestion Control for Unicast Applications," in *Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August-September 2000, pp. 43–56.

[9] I. Rhee, V. Ozdemir, and Y. Yi, "TEAR: TCP Emulation at Receivers - Flow Control for Multimedia Streaming," Department of Computer Science, NCSU, Raleigh, NC, Tech. Rep., April 2000.

[10] R. Rejaie, M. Handley, and D. Estrin, "RAP: An End-to-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet," in *Proceedings of IEEE INFOCOM*, March 1999, pp. 1337–1345.

[11] Y. R. Yang and S. S. Lam, "General AIMD Congestion Control," in *Proceedings of the 8th International Conference on Network Protocols (ICNP)*, Osaka, Japan, November 2000.

[12] B. Wang, J. Kurose, P. Shenoy, and D. Towsley, "Streaming via TCP: An Analytic Performance Study," in *Proceedings of ACM Multimedia*, New York, NY, USA, October 2004.

[13] N. Seelam, P. Sethi, and W. chi Feng, "A Hysteresis Based Approach for Quality, Frame Rate, and Buffer Management for Video Streaming Using TCP," in *Proceedings of 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services*, Chicago, IL, USA, October 2001.

[14] C. Krasic and J. Walpole, "Priority-Progress Streaming for Quality-Adaptive Multimedia," in *Proceedings of the Ninth ACM International Conference on Multimedia*, Ottawa, Canada, October 2001.

[15] P. de Cuetos and K. W. Ross, "Adaptive Rate Control for Streaming Stored Fine-Grained Scalable Video," in *Proceedings of the 12th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Miami, Florida, USA, May 2002.

[16] A. Goel, C. Krasic, K. Li, and J. Walpole, "Supporting Low Latency TCP-Based Media Streams," in *Proceedings of International Workshop on Quality of Service (IWQoS)*, Miami Beach, FL, USA, May 2002.

[17] N. Spring, D. Wetherall, and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces," RFC-3540, June 2003.

[18] T. Miller, "ECN and it's impact on Intrusion Detection," SecurityFocus InFocus Article, http://www.securityfocus.com/infocus/1205, November 2000.

[19] T. Phelan, "Datagram Congestion Control Protocol (DCCP) User Guide," IETF Internet-Draft: draft-ietf-dccp-user-guide-02, July 2004.

[20] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," IETF Internet-Draft: draft-ietf-dccp-spec-11, March 2005.

[21] VINT, "Virtual InterNetwork Testbed, A Collaboration among USC/ISI, Xerox PARC, LBNL, and UCB," http://www.isi.edu/nsnam/vint/.

[22] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC-2018, October 1996.

[23] J. C. Bolot, "Characterizing End-to-End Packet Delay and Loss in the Internet," *Journal of High Speed Networks*, vol. 2, no. 3, pp. 289–298, September 1993.

[24] V. Jacobson, "Congestion Avoidance and Control," in *Proceedings of ACM SIGCOMM*, Stanford, CA, USA, August 1988.

[25] S. Keshav, "A Control-Theoretic Approach to Flow Control," in *Proceedings of the conference on Communications Architecture & Protocols*, 1991, pp. 3–15.

[26] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy, "Bandwidth Estimation: Metrics, Measurement Techniques, and Tools," *IEEE Network*, vol. 17, no. 6, November/December 2003.

[27] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Inferring TCP Connection Characteristics Through Passive Measurements," in *Proceedings of IEEE INFOCOM*, Hong Kong, China, March 2004.

[28] M. Arlitt and T. Jin, "Workload Characterization of the 1998 World Cup Web Site," Hewlett-Packard Laboratories, Tech. Rep. HPL-1999-35R1, October 1999.

[29] F. Hernandez-Campos, K. Jeffay, and F. Smith, "Tracing the Evolution of the Web Traffic: 1995-2003," in *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Orlando, FL, USA, October 2003.

[30] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy, "An Analysis of Internet Content Delivery Systems," in *Usenix Operating Systems Design and Implementation (OSDI)*, Boston, MA, USA, October 2002, pp. 315 – 327.

[31] J. Nichols, "Measurement of Windows Streaming Media," Master's thesis, Worcester Polytechnic Institute, February 2004, advisor: Mark Claypool and Robert Kinicki.