Accepted for Publication in the Proc. of the 21st International Performance, Computing, and Communications Conference (IPCCC 2002), Phoenix, AZ.

TCP-CM: A Transport Protocol for TCP-friendly Transmission of Continuous Media

Yongxiang Liu, K. N. Srijith, Lillykutty Jacob, A. L. Ananda

Centre for Internet Research
School of Computing
National University of Singapore
3, Science Drive 2, Singapore 117543
{liuyx, srijith, jacobl, ananda}@comp.nus.edu.sg

Abstract

We propose a new TCP-friendly transport protocol, called TCP-CM, for the continuous media applications over the Internet. TCP-CM is a direct modification of the TCP to support continuous media applications without compromising the congestion control feature of the TCP, which is critical to the stable functioning of the Internet. We design TCP-CM API to be compatible to the BSD socket interface, which requires minimum changes for the applications to adopt TCP-CM. Continuous media applications that adopt TCP-CM as the transport protocol can be relieved from the burdens such as rate control and scheduling for timely delivery, and hence can focus solely on advanced coding or compression techniques for adapting the content according to the available network bandwidth. We implement the TCP-CM in Linux 2.2.15 TCP/IP protocol stack, and run extensive experiments on TCP-CM using emulated video flows. Our experiments show that TCP-CM can be used for the timely delivery of continuous media data within the constraints of the available network bandwidth, and can compete with TCP connections fairly.

Key Words: Congestion Control, TCP-friendly, Pushed LIFO, Pushed FIFO

I. INTRODUCTION

Recent measurements across the transatlantic links have shown TCP flows being in majority with almost 95% of the byte share [1]. However, due to the growing popularity of continuous media applications such as audio/video streaming and audio/video conferencing and because TCP is not suitable for the delivery of time-sensitive data, a growing number of applications are being implemented using UDP. In [2], the authors discuss the problems of unresponsive flows and the danger of congestion collapse of the Internet. Unlike TCP, the rates of UDP flows are not limited by congestion control and such aggressive flows only starve TCP flows of their rightful share of bandwidth. In addition, UDP flows typically exacerbate congestion problems and waste precious bandwidth when their packets are dropped during the onset of congestion.

To date, several TCP-friendly end-to-end congestion control mechanisms have been put forward to address the above issue. Of the known proposals, which are discussed in Section II, some are not able to share bandwidth fairly with the common implementations of TCP; some require substantial amount of modification to existing applications; some cannot provide timely data delivery, which is a critical requirement for continuous media applications; and some are not stable in dynamically changing network conditions.

Design of a new TCP-friendly transport protocol involves significant amount of work, including redesign of roundtrip time calculation, packet sequencing, congestion discovery and rate adjustment strategies etc. Due to the complexities involved in these functionalities, an inappropriate design or implementation is highly possible.

In this paper, we design and implement a TCP-friendly protocol, called TCP-CM, using direct TCP modifications, which possesses inherited TCP-friendliness, efficiency, and robustness. TCP-CM does not attempt to retransmit a packet when it is lost, rather give up retransmission in favor of new data. TCP-CM has in-kernel scheduling mechanisms to allow timely delivery of data. It allows applications to use the same programming interfaces (BSD sockets) as that of TCP/UDP, which is widely known, thus enabling applications to adopt proper congestion control with ease. An application needs only to create a TCP-CM socket in replacement of UDP, and send data to the socket in the same way as it sends to UDP. TCP-CM will handle the congestion control and timely scheduling of packets automatically.

Continuous media applications that adopt TCP-CM as the transport protocol can be relieved from the burdens such as rate control and scheduling for timely delivery, and hence can focus solely on advanced coding or compression techniques for adapting the content according to the available network bandwidth.

The remainder of this paper is organized as follows. In Section II we describe related work and in Section III we explain the various aspects of TCP which are subjected

to modifications in the design of TCP-CM. How TCP-CM is implemented is described in Section IV. The results from our experiments on a testbed are presented in Section V which is then followed by the conclusion.

II. RELATED WORK

Previous research on providing TCP-friendliness for continuous media applications falls into two main categories, window-based and rate-based approaches.

Most of the recent works have focused on TCP-friendly congestion control based on rate-adaptation. Typically the sender regulates the rate at which it injects packets into the network, the rate being computed by the application based on models derived from the TCP congestion control mechanisms and also on model parameters estimated by the application. Sisalem and Schulzrinne's Loss-Delay Adjustment (LDA) algorithm [3] uses an additive increase multiplicative decrease (AIMD) rate control based on RTCP feedbacks. Rate Adaptation Protocol (RAP) by Rejaie et al [4] adjusts its rate in AIMD fashion based on frequent receiver acknowledgements. In [5,6,7], algorithms based on derived throughput equations from more detailed TCP models are proposed.

In common, these rate-based approaches measure the model parameters such as packet loss rate and round trip time periodically and adjust the rate based on throughput equations or other strategies. A fundamental problem associated with these approaches lies on the difficulties to determine their adjustment periods. If the period is too short, spontaneous changes in loss rate or round trip time can cause significant oscillations. On the other hand, if the adjustment period is too long, connections will become more reluctant to congestion signals, thus the TCP-friendliness is sacrificed. Another problem in trying to model TCP congestion control behavior is that it differs significantly under different conditions, thus an equation that accurately characterizes TCP under one set of conditions may fail under others. In addition, most of these approaches leave the congestion control responsibility to an application, which is dangerous. Due to the complexities involved, applications might implement congestion control in wrong ways, thereby causing network instabilities; e.g., inaccurate estimation of the model parameters can lead to under- or over-allocation of bandwidth.

Proposals in [8,9] are based on window-based congestion control. Andersen *et al* developed an end system support called Congestion Manager (CM) [8] to provide exclusive congestion control service independent of TCP and any applications. The CM maintains a congestion window, which increases gradually and decreases upon a packet loss. However, protocols or applications that make use of the CM need to update the CM frequently with the congestion and packet delivery status. Thus existing applications need to do considerable amount of extra work, including packet loss discovery, ACK for every pieces of received data, and frequent updating of these information to the CM. These applications also need to incorporate additional scheduling mechanisms to ensure timely delivery of data. In addition, CM introduces a new set of application programming interfaces (API), which requires considerable amount of modification to these applications.

In another window-based approach, Mukherjee and Brecht proposed a TCP modification for continuous media, called Time-lined TCP [9], in which a deadline is specified for each packet, to ensure timely data delivery. As the deadline timer expires, the obsolete packets are discarded. Because packet discarding will create discontinuities in TCP's sequence number, the sender has to inform the receiver these discontinuities. This approach deviates from the current TCP's implementation significantly. Complexities can occur when multiple discontinuities occur, or packets carrying the discontinuity information are lost. As their work is simulation based, no details are given on these practical issues. In addition, TLTCP requires sender to specify a deadline for each packet, which requires considerable amount of modifications to existing application codes. Also it is not the sender but the receiver that is in a position to tell the playback deadlines. In their simulation, the timely delivery property of TLTCP is not clearly shown.

III. TCP-CM DESIGN OVERVIEW

One of our design principles is to retain as much as possible the current TCP congestion control mechanisms to ensure the robustness and TCP-friendliness of TCP-CM. Another important principle of our design is to provide user-friendliness to applications that want to adopt congestion control. We believe that any additional burden or complexity caused to the applications for performing some kind of congestion control will greatly hinder them from adopting congestion control. Our design addresses each of the deficiencies when TCP is used for continuous media applications. As a byte-stream reliable transport protocol, TCP in many aspects is not suitable for continuous media applications. We now address each of these issues and the solutions incorporated in the design of TCP-CM.

A. TIMELY DELIVERY OF DATA

An application using TCP is constrained from writing into the TCP send buffer when it is full. That is, when TCP *send buffer* is full due to congestion, new data cannot be admitted. When the congestion is so severe that *congestion window* is small and too few ACKs can

be received to clock the transmission, the delivery time of those already admitted data as well those waiting at the application could be extremely long. However, timely delivery of data is crucial for continuous media applications because any data that arrive at the receiver later than its scheduled arrival time will render itself useless. Thus, TCP's way of scheduling data is one of the inappropriate features that need to be changed.

Another concern is that applications need complex mechanism to schedule its data on time. Application cannot fill the *send buffer* in a continuous unblocked manner; rather it needs to check constantly for available buffer space and determine which data to send. Consequently, applications are burdened with complex scheduling of data, which will further hinder applications from adopting congestion control.

We address these problems as follows: The application data are admitted to TCP-CM's buffering with no blocking. Application will send its data stream continuously to TCP-CM, as if it is using UDP. The TCP-CM schedule data in a timely fashion and this is transparent to the applications. This provides a very user-friendly transport protocol for continuous media applications to adopt TCP-friendly congestion control.

We propose two queuing schemes for TCP-CM, namely Pushed FIFO (P-FIFO) and Pushed LIFO (P-LIFO), to ensure timely delivery of data.

1) Pushed FIFO (P-FIFO)

The terms, namely "*over-loaded*", "*under-loaded*", and "*well-loaded*" are used in our description of queuing behaviours: The *over-loaded* condition exists when the rate of incoming data from the application exceeds the outgoing rate (i.e. the available bandwidth) as determined by the congestion control algorithms; the *under-loaded* condition exists when the reverse happens; and the *well-loaded* condition exists if the rate of incoming data from the application matches the outgoing rate.

A pushed FIFO (P-FIFO) scheme admits new segments regardless of whether the queue is full or not. When it is full, the new segment will be admitted by discarding the oldest one. The newly admitted segment is inserted at the queue tail. When the congestion control window allows a segment to be sent out, the oldest one in the queue (the segment at the queue head at the instant) gets the turn. Thus, at any instant under the overloaded condition, only the most recent up-to-the-queue-length number of segments are kept in the queue.

Traffic rates from the applications can have spontaneous variations. Also the speed at which packets are dispensed is variable depending on the available bandwidth. It is therefore important to have sufficient buffer size to cater these dynamics. However, continuous media applications generally have stringent delay bound requirements. Assuming that an application needs a delay bound less than D seconds, and generates data at an average rate R bytes/sec, its TCP-CM buffer size should be less than or equal DR *bytes.*

2) Pushed LIFO (P-LIFO)

Similar to P-FIFO, a pushed LIFO (P-LIFO) scheme also admits new segments regardless of whether the queue is full or not; and when it is full, the new segment will be admitted by discarding the oldest one. However, the newly admitted segment is inserted at the queue head instead of queue tail. When the congestion control window allows a segment to be sent out, it is the newest one in the queue (the segment at the queue head at the instant) that gets turn.

The rationale for choosing LIFO queuing is that at any instant when a congestion control engine allows some segments to be transmitted out, the latest segments should be sent out. This is because continuous media applications can afford to drop some older segments, but not to queue a segment too long to wait for congestion control engine to schedule its transmission.

The advantage of a P-LIFO scheme is its real-time properties, i.e. it causes the least queuing delay to the source host. Obviously, P-LIFO will cause some packets reordering, but generally the reordering will involve only very few packets. In Section V, we will show some experimental results to verify this statement. Fortunately, existing continuous media applications typically provide receiver play-out buffering in order to take care of the end-to-end transmission delay jitter. Therefore, P-LIFO's packet reordering, which involves only a few packets, will not cause deployment issues for most of the existing applications.

B. RETRANSMISSION MECHANISMS

In TCP, Packet retransmission can take place, at the earliest, only after three duplicate ACKs are received, which takes more than one round trip time. As a packet loss signifies congestion, it means that available bandwidth is not enough to deliver application data. In other words, some data from the application have to be sacrificed due to insufficient network bandwidth. Instead of retransmitting an older packet, it is justifiable to discard it and allocate its transmission opportunity to new data.

However, because TCP is a reliable transport protocol, any data missing in a window will freeze the transmission process, causing repeated ACKs being sent for the missing data. Therefore, to keep the normal data flowing for a TCP connection, retransmission cannot be avoided completely.

Our design avoids the retransmission of old data, at the same time keeps the normal data flowing. The lost segment is discovered either upon observation of three duplicate ACKs, or retransmission timeout when the connection's path is severely congested. Our scheme is to transmit a dummy segment whenever a lost segment is discovered. This dummy segment contains the information of the starting and ending sequence numbers of the lost data in the TCP header option field (as described in the next section) and it is piggybacked on a new data segment. Upon receiving the dummy segment, the receiver will use the starting and ending sequence numbers to fill in the hole, thereafter the normal data flow continues.

This approach has several advantages. Firstly, it causes the least changes to the existing TCP implementation. Similar to TCP, duplicate ACKs are fedback to the sender to convey implicitly the congestion signal, until the dummy segment is received. Secondly, each retransmission is effectively utilized to transmit new data.

C. PACKETIZATION

Continuous media applications typically use some compression techniques in their source coding schemes. Generally, these encoding techniques reduce dependencies among application protocol data units (APDUs) and maximize the amount of decodable data at the receiver in case of packet losses [10]. Because only the application has the knowledge of the internal structure of their data, packetization at the application level is suitable and is the only way to ensure the error resilience in case of packet loss due to congestion or corruption.

However, TCP is a byte-stream protocol which assumes that the incoming data is continuous bytes stream. If no changes are made to TCP, it will attempt to repacketize any data received from the application, thus breaking the internal structure of APDUs and largely impacting the application data's error resilience.

For TCP-CM we modify TCP to support application level framing. Our API for TCP-CM provides a read-only socket option, *SO_MSS*, which allows an application to query the TCP-CM's current MSS value. The usage and descriptions on the new socket options provided by the TCP-CM are outlined in the Appendix. If the application sends data that is smaller than MSS, TCP-CM will not merge it with another segment. Instead it creates an independent segment for such an APDU. If the APDU has larger size than current MSS, TCP-CM will break it into segments of MSS size. The remaining bytes become a segment by itself, without being merged with other segments.

IV. IMPLEMENTATION OVERVIEW

We implement a new socket type, *SOCK_TCPCM*. Applications need to create sockets with type of *SOCK_TCPCM* to make TCP-CM connections. To support TCP-CM connection, a new TCP header option named *TCP-CM Permit Option* (shown in Figure 1), is set in the SYN segments during handshaking. Both sides need to set this header option in their SYN segments for a successful TCP-CM connection setup.

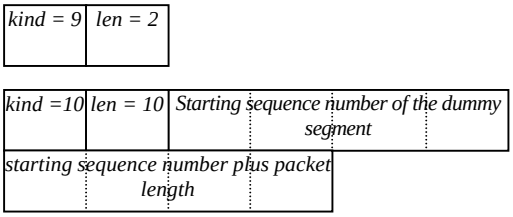

Figure 1. Two New TCP Header Options: *TCP-CM Permit (kind = 9), Dummy Segment Option (kind = 10)*

The changes to the original TCP's retransmission mechanisms, from a sender's and a receiver's point of views, are shown in Figure 2 and Figure 3 respectively. We introduce a *Dummy Segment Option* (shown in Figure 1) to convey the starting and ending sequence numbers of a segment being retransmitted via the TCP header option. As shown in Figure 2, when the TCP-CM realizes that a segment is to be retransmitted, it will trim the payload to 0 bytes, and put the starting and ending sequence numbers into a *Dummy Segment Option*. This dummy segment information is piggybacked on the next new segment to be transmitted. As shown in Figure 3, when the TCP-CM at the receiver's end receives a segment, the TCP header is parsed to check for the presence of the *Dummy Segment Option*. If it is found, a dummy segment will be regenerated and filled into the receiving queue.

We implement a Queue Manager (QM) to store packets from the application and to schedule packets to TCP's congestion control module. Application's write routines (such as *write ()*, *sendmsg ()*) will invoke the TCP-CM's write routine in the kernel, which instead of storing data to its socket buffer as in the case of TCP, stores the data into QM by calling its enqueue function. In turn, the QM packetizes data into segments in accordance with the current MSS and enqueues them. Events such as writing of application data to TCP-CM socket, or receiving acknowledgement, will trigger TCP-CM to decide whether to send out a new segment by testing its congestion window against the number of outstanding segments. If a new segment can be sent out, TCP-CM will call QM's dequeue function to dequeue a segment and dispense it to the network. Currently two scheduling schemes are implemented in the QM, P-LIFO and P-

FIFO. The QM has interfaces to include more scheduling schemes such as priority queuing or class-based queuing, if required.

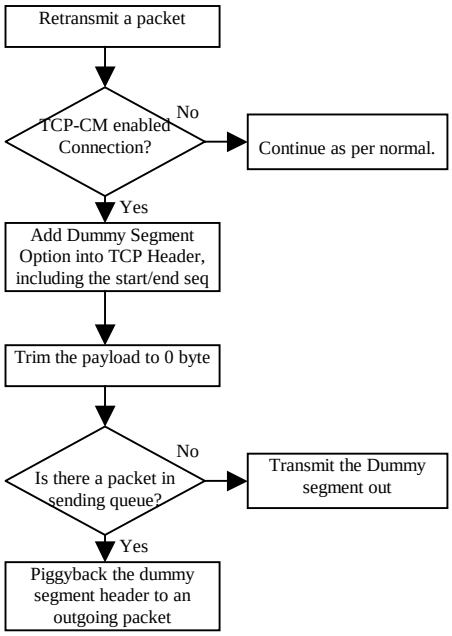

Figure 2. Modifications to TCP Retransmission Engine

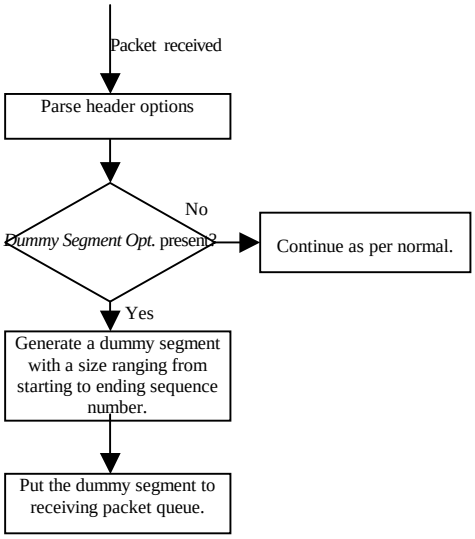

Figure 3. Modifications to Packet Receiving Process

API for TCP-CM

TCP-CM provides user-friendly interface to the applications. The programming interface is exclusively based on BSD socket interface, the most widely accepted network programming interface. In the Appendix, we highlight the usage of the new socket operations associated with TCP-CM.

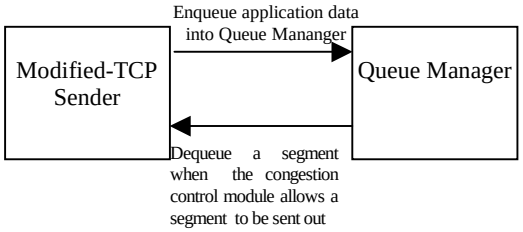

Figure 4. Queue Manager and its Interactions with the modified-TCP Sender

V. EXPERIMENTS ON TCP-CM

In this section, we first verify that TCP-CM performs congestion control properly by obtaining its fair amount of bandwidth when it is competing with TCP connections. We then examine if TCP-CM can utilize the bandwidth that it obtains to transmit useful data, by measuring the end-to-end delay and jitter of the data being received at the destination. We deploy a testbed with the simple topology shown in Figure 5.

A. Data Transmission Using TCP-CM

In this set of experiments, we show that TCP-CM connections compete with TCP connections fairly. In Figure 5, the link between router 1 and router 2 is rate limited at 4.5Mbps, which is representative of three T1 links. Router 1 deploys RED with buffer size of 60KB and RED parameters, respectively, min = 15KB, max = 45KB and probability = 0.4. A delay box is introduced to emulate the wide-area networks scenario where the end-to-end delay is much longer.

The network is initially lightly loaded with two TCP connections and two emulated video connections of data rates 1Mbps each, utilizing TCP-CM as their underlying transport protocol; the delay box is disabled. Each of these connections lasts for 30 minutes. The average bandwidth shares and loss rates are listed in Table 1. We notice that under such lightly loaded condition, each TCP-CM flow obtains a bandwidth close to 1Mbps, which matches to the rate of emulated video. The remaining bandwidth is shared between the two TCP flows. We then increase the network load by opening more TCP connections. When the number of TCP connections reaches four, each of the connections merely obtains near 0.75Mbps. We notice that the total sum of the bandwidth is slightly less than the bottleneck link bandwidth. This is due to the TCP-CM and TCP protocol overheads.

We repeat the same experiments in the context of wide area network emulated by introducing additional delay at the delay box. Based on Huffaker *et al.*'s measurement, cross-continental round-trip time typically ranges from 200ms to 500ms [11]. We set a round-trip time of 400ms. From the results shown in Table 2, we observe

that TCP-CM can compete with the TCP connections fairly, though the overall link utilization suffers slightly in long delay environment.

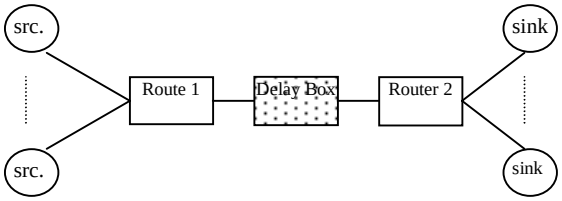

Figure 5. Experimental Setup

Loading Condition	Bandwidth of TCP-CM Flow	Bandwidth of TCP Flow	Loss Rate of the Emulated Video
Light (2xTCP-CM, 2xTCP)	0.971Mbps	1.206Mbps	2.6%
Moderate (2xTCP-CM,3xTCP)	0.875Mbps	0.873Mbps	11.2%
Heavy (2xTCP-CM, 4xTCP)	0.740Mbps	0.751Mbps	25.9%

Table 1. TCP-CM Data Transmissions in LAN environment

Loading Condition	Bandwidth of TCP-CM Flow	Bandwidth of TCP Flow	Loss Rate of the Emulated Video
Light (2xTCP-CM, 2xTCP)	0.960Mbps	1.150Mbps	4%
Moderate (2xTCP-CM,3xTCP)	0.865Mbps	0.860Mbps	13.3%
Heavy (2xTCP-CM, 4xTCP)	0.731Mbps	0.741Mbps	27%

Table 2. TCP-CM Data Transmissions in WAN Environment

We conclude from our experimental results that TCP-CM can compete fairly with other TCP connections. When the network is not heavily loaded, TCP-CM can grab enough bandwidth from the network for the emulated video. When the network becomes heavily loaded, TCP-CM grabs its fair share from the network.

B. Delay Properties of TCP-CM Using P-lIFO

For the amount of bandwidth it grabs from the network, TCP-CM must transmit as much useful data (whose delay and jitter are within bounds) as possible. In this set of experiments we characterize the delay and jitter properties of TCP-CM when it uses P-LIFO, one of the in-kernel scheduling schemes that TCP-CM provides.

We first measure the delay properties in an under-loaded network. The under-loaded condition is created by running two TCP-CM connections together with two TCP connections. The bottleneck link rate remains at 4.5Mbps. The emulated video application timestamps each of its packets. The experiments are performed in a LAN environment, and thus delay caused by TCP-CM is the predominant component in the end-to-end delay. The average end-to-end delay and delay jitter are calculated and summarized in Table 3 (row 1). Next we add two more TCP connections to congest the network. The resulting average delay and jitter are still low (row 2).

Loading Conditions & Buffer Size	Ave. Delay (sec)	Jitter (sec)	Reordering Index
Non-congested (200KB)	0.005	0.003	0.37
Congested (200KB)	0.092	0.026	0.81
Congested (300KB)	0.155	0.033	1.60
Congested (450KB)	0.162	0.033	1.55

Table 3. TCP-CM Delay and Reordering Properties (P-LIFO)

We then study effects of P-LIFO buffer size on delay properties of TCP-CM. The delay measurements are summarized in Table 3. The cumulative distribution function (CDF) of delay for each of the test settings is shown in Figure 7.

We can conclude from these measurements that TCP-CM with P-LIFO introduces small delay and delay jitter, and can support timely delivery of data for reasonable network loading levels.

Figure 6 shows the packet sequence number vs the end-to-end delay characteristics. Each horizontal line is for one packet; the starting point of a line represent the sending time while the ending point represent received time. We can observe that for P-LIFO, some packets that arrive at the receiver are reordered. We quantify the degree of packet reordering using a newly defined variable called **reordering index**. We compare the order in which the receiver receives packets with the original order in which the sender sends the packets. For each packet received we compute the number of places it has shifted with respect to its original position. The averaged value is defined as the reordering index. As we can see from Table 3, each packet, on the average, is shifted from its original position by less than 2 places. The

number of shifting vs. the percentage of packets is shown in Figure 8. We can observe that packet reordering for majority of the packets involves very few places (mostly less than 5 places). This is acceptable at the receiver's side, because a receiving application normally needs to take care of packet reordering due to the IP datagram service.

Thus our experiments on the characterization of TCP-CM's P-LIFO in-kernel scheduling principle show that applications such as video conferencing and IP telephony that require stringent realtimeness can choose P-LIFO for their TCP-CM.

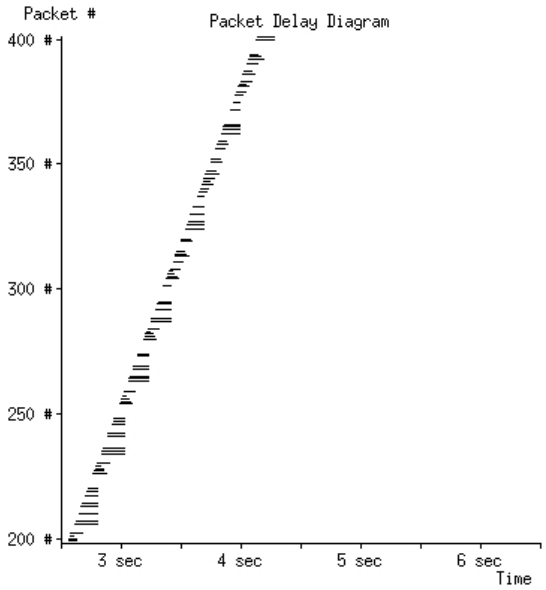

Figure 6. Zoom-in Version of Packet Delay Diagram for P-LIFO

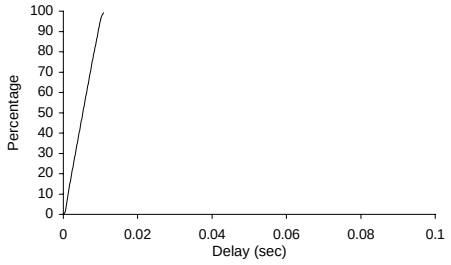

Figure 7.a. Cumulative Distribution Function of Delay, (Non-congested, P-LIFO, 200KB)

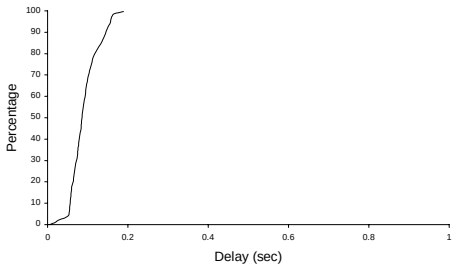

Figure 7.b. Cumulative Distribution Function of Delay (Congested, P-LIFO, 200KB)

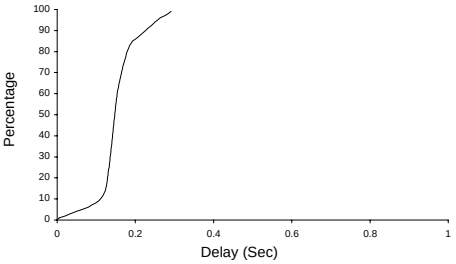

Figure 7.c. Cumulative Distribution Function of Delay (Congested, P-LIFO, 300KB)

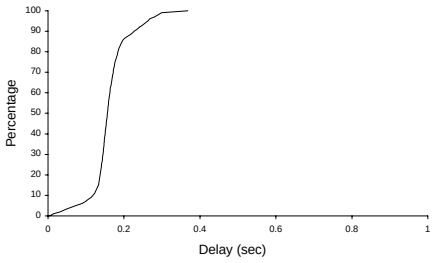

Figure 7.d. Cumulative Distribution Function of Delay (Congested, P-LIFO, 450KB)

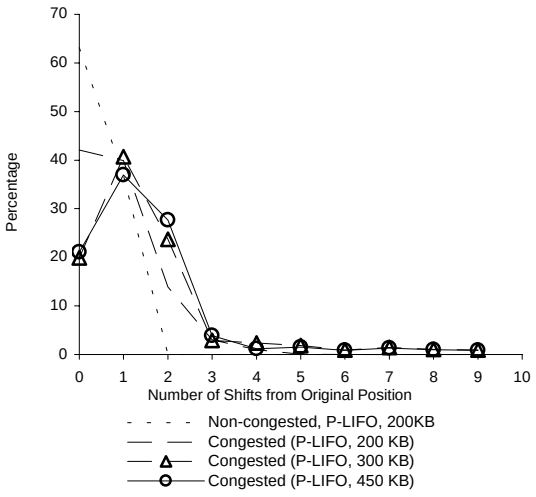

Figure 8. Percentage of Packets vs the Number of Places That are Shifted from the Original Positions

C. Delay Properties of TCP-CM Using P-FIFO

With similar experimental setup as in Part B, we run experiments to characterize the delay and jitter properties of P-FIFO, the other in-kernel scheduling principle that TCP-CM provides. Similarly we apply light traffic first to our testbed. As shown in the cumulative distribution diagram in Figure 9.a, majority of packets experience little delay and jitter. When network load is increased to 6 connections, congestion is experienced. Consequently, majority of packets suffer from a substantial amount of queuing delay. When the buffer size is increased, the queuing delay also increases. The jitter varies slightly when the buffer size is increased.

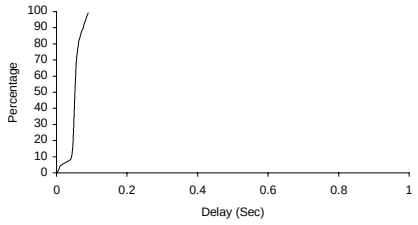

Figure 9.a. Cumulative Distribution Function of Delay (Non-congested, P-FIFO, 200KB)

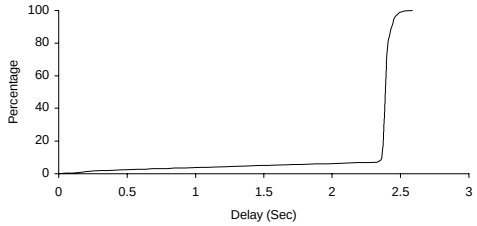

Figure 9.b. Cumulative Distribution Function of Delay (Congested, P-FIFO, 200KB)

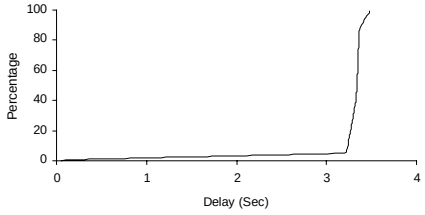

Figure 9.c. Cumulative Distribution Function of Delay (Congested, P-FIFO, 300KB)

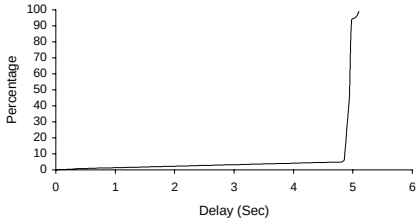

Figure 9.d. Cumulative Distribution Function of Delay (Congested, P-FIFO, 450KB)

Thus, an application that needs stringent in-order packet delivery can choose P-FIFO for its TCP-CM connection. However, the queue size must be carefully chosen to avoid excessive queuing delay.

Loading Conditions & Buffer Size	Ave. Delay (sec)	Jitter (sec)
Non-congested (200KB)	0.06	0.01
Congested (200KB)	2.30	0.183
Congested (300KB)	3.24	0.174
Congested (450KB)	4.81	0.250

Table 4. TCP-CM Delay and Reordering Properties (P-FIFO)

VI. CONCLUSIONS

We propose and implement a new TCP-friendly protocol based on direct modifications of the current TCP. TCP-CM modifies those features in TCP that are not suitable for continuous media applications, like retransmission, packetization and scheduling. At present, TCP-CM provides choice of the two queuing disciplines, P-FIFO, and P-LIFO for applications to make use of the available bandwidth in the best way. Extensions to BSD socket interface are implemented to provide a user-friendly interface to applications using TCP-CM.

Our experiments show that TCP-CM, in addition to its inheritance of TCP-friendliness from TCP, can deliver continuous media data in a timely fashion. We observe that TCP-CM with P-LIFO scheduling scheme provides the best realtime property with moderate reordering for continuous media.

VII. REFERENCES

[1] Simon L**.,** **"**UDP vs. TCP distribution", 01 Mar 2001,end2end-interest mailing list,

http://www.postel.org/pipermail/end2end-interest/2001-March/000218.html
[2] S. Floyd, K. Fall, “Router Mechanisms to Support End-to-End Congestion Control”, Technical Report, Network Research Group – Lawrence Berkeley National Laboratory, February 1997, ftp://ftp.ee.lbl.gov/papers/collapse.ps
[3] D. Sisalem and H. Schulzrinne. The loss-delay adjustment algorithm: A TCP-friendly adaptation scheme. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV*), July 1998.
[4] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate based congestion control mechanism for real-time streams in the Internet. In *IEEE Infocom*m, March1999.
[5] J. Mahdavi and S. Floyd. TCP-friendly unicast rate based flow control, January1997. http://www.psc.edu/network-ing/papers/tcp friendly.html.
[6] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its emperical valiations. In *ACM SIGCOM*M, 1998.
[7] J. Padhye, J. Kurose, D. Towsley, and R. Koodli. A model based TCP-friendly rate control protocol. In *IEEE NOSSDA*V, June 1999.
[8] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan, “System support for bandwidth management and content adaptation in Internet applications,” in *Proc. Symposium on Operating Systems Design and Im-plementatio*n, October 2000
[9] Biswaroop Mukherjee, and Tim Brecht, " Time-lined TCP for the TCP-friendly Delivery of Streaming Media over the Internet", In Proceedings of ICNP 2000, November 2000.
[10] M. Handley, C. Perkins, Guidelines for Writers of RTP Payload Format Specifications, INTERNET-DRAFT, 15th Oct 1999.

[11] Huffaker, B., M. Fomenkov, D. Moore, E. Nemeth and k. claffy. "Measurements of the Internet topology in the Asia-Pacific Region." in: Proceedings of Inet '00. Yokohama, Japan.

APPENDIX

- TCP-CM Connection Setup

A new socket type *SOCK_TCPCM* is introduced for setting up a TCP-CM connection.

Sample Usage:

```
mSock = socket( AF_INET, SOCK_TCPCM, 0 );
```

- Setting Queuing Parameters

Two new socket options are added to let an application customize their queuing settings:

SO_QTYPE: to set or get the type of queuing for a TCP-CM connection. Currently P-FIFO, P-LIFO are supported.

SO_QLEN: to set or get the buffer size of a TCP-CM connection. The default buffer size used is 100 ×MTU bytes. As a rule of thumb, if the maximum queuing delay that the application can tolerate is D_{max} seconds, then the buffer size shall statisfy:

$L \le R \times D_{\max}$, where R is the average rate at which an application generates its data.

Some coding examples are:

```
setsockopt(file_descriptor, SOL_SOCKET,
SO_QLEN, &option, optlen)
setsockopt(file_descriptor, SOL_SOCKET,
SO_QTYPE, &option, optlen)

getsockopt(file_descriptor,          SOL_SOCKET,
SO_QTYPE, &option, &optlen)
```

- Query MSS

This socket option (*SO_MSS*) is read-only, thus only associated with function *getsockopt*. This is an example on how to query the current MSS value of TCP-CM:

```
getsockopt(fd, SOL_SOCKET, SO_MSS, &option,
&optlen)
```