Multimedia Streaming Using Multiple TCP Connections

Thinh Nguyen
thinhq@eecs.oregonstate.edu
School of EECS
Oregon State University
Corvallis, OR 97330

Sen-ching S. Cheung
cheung@engr.uky.edu
ECE Department
University of Kentucky
Lexington, KY 40506

Abstract

As broadband Internet becomes widely available, multimedia applications over the Internet become increasingly popular. However, packet loss, delay, and time-varying bandwidth of the Internet have remained the major problems for multimedia streaming applications. As such, a number of approaches, including network infrastructure and protocol, source and channel coding have been proposed to either overcome or alleviate these drawbacks of the Internet. In this paper, we propose the MultiTCP system, a receiver-driven, TCP-based system for multimedia streaming over the Internet. Our proposed algorithm aims at providing resilience against SHORT TERM insufficient bandwidth by using MULTIPLE TCP connections for the same application. Furthermore, our proposed system enables the application to achieve and control the desired sending rate during congested periods, which cannot be achieved using traditional TCP. Finally, our proposed system is implemented at the application layer, and hence, no kernel modification to TCP is necessary. We analyze the proposed system, and present simulation results to demonstrate its advantages over the traditional single TCP based approach.

1 Introduction

In recent years, there have been an explosive growth of multimedia applications over the Internet. All major news networks such as ABC and NBC now provide news with accompanying video clips. Several companies, such as MovieFlix [1], also offer video on demand to broadband subscribers. However the quality of videos being streamed over the Internet is often of low quality due to insufficient bandwidth, packet loss, and delay. To view a DVD quality video from an on demand video service, a customer must download either the entire video or a large portion of the video before playback time in order to avoid pauses caused by insufficient bandwidth during a streaming session. Thus, many techniques have been proposed to enable efficient multimedia streaming over the Internet. The source coding community has proposed scalable video [2][3], error-resilient coding, and multiple description [4] for efficient video streaming over the best-effort networks such as the Internet. A scalable video bit stream is coded in such a way to enable the server to easily and efficiently adapt the video bit rate to the current available bandwidth. Error-resilient coding and multiple description are aimed at improving the quality of the video in the presence of packet loss and long delay caused by retransmission. Channel coding techniques are also used to mitigate long delay for real-time applications such as video conferencing or IP-telephony [5]. The main disadvantages of these approaches are first, specialized codecs are required and second, their performances are highly affected by the network traffic conditions.

From a network infrastructure perspective, Differentiated Services [6][7] and Integrated Services [8][7] have been proposed to improve the quality of multimedia applications by providing preferential treatments to various applications based on their bandwidth, loss, and delay requirements. More recently, path diversity architectures that combine multiple paths and either source or channel coding have been proposed to provide larger bandwidth, and to combat efficiently against packet loss [9][10][11]. Nonetheless, these approaches cannot be easily deployed as they require significant changes in the network infrastructure.

The most straightforward approach is to transmit standard-based multimedia via existing IP protocols. The two most popular choices are TCP and UDP. A single TCP connection is not suitable for multimedia transmission because its congestion control may cause a large fluctuation in the sending rate. Unlike TCP, an UDP-based application is able to set the desired sending rate. If the network is not too much congested, the UDP throughput at the receiver would approximately equal to the sending rate. Since the ability to control the sending rate is essential to interactive and live streaming applications, majority of multimedia streaming systems use UDP as the basic building block for sending packets over the Internet. However, UDP is not a congestion aware protocol since it does not reduce its sending rate in presence of network congestion, and therefore potentially results in a congestion collapse. Congestion collapse occurs when a router drops a large number of packets due to its inability to handle a large amount of traffic from many senders at the same time. TCP-Friendly Rate Control Protocol (TFRC) has been proposed for multimedia streaming with UDP in order to incorporate TCP-like congestion control mechanism [12]. Another drawback of using UDP is its lack of reliable transmission and hence the application must deal with the packet loss.

Based on these drawbacks of UDP, we propose a new receiver-driven, TCP-based system for multimedia streaming over the Internet. In particular, our proposed system, called *MultiTCP*, is aimed at providing resilience against *short-term* insufficient bandwidth by using *multiple* TCP connections for the same application. Furthermore, our system enables the application to achieve and control the sending rate during congested period, which in many cases, cannot be achieved using a single TCP connection. Finally, our proposed system is implemented at the application layer, and hence, no kernel modification to TCP is necessary.

The rest of the paper is organized as follows. In Section 2, we describe the two major drawbacks of using TCP for multimedia streaming: short-term insufficient bandwidth and lack of precise rate control. These drawbacks motivate the use of multiple TCP connections in our proposed system, which is described in Section 3. In Section 4, we demonstrate the performance of our system based on simulations results using NS[13]. We then describe other related works that utilize multiple network connections in Section 5. Finally, we summarize our contributions in Section 6.

2 Drawbacks of TCP for multimedia streaming

In subsections

As discussed briefly in Section 1, TCP is unsuitable for multimedia streaming due partly to its fluctuating throughput and its lack of precise rate control. TCP is designed for end-to-end reliability and fast congestion avoidance. To provide end-to-end reliability, a TCP sender retransmits the lost packets based on the packet acknowledgment from a TCP receiver. In order to have fast response to network congestion, TCP controls the sending rate based on a window-based congestion control which works as follows. The sender keeps track of a window of maximum number of unacknowledged packets, i.e., packets that have not been acknowledged by the receiver. In the steady state, the sender increases the window size W by $1/W$ upon successfully receiving an acknowledged packet, or equivalently, it increases the sending rate by one packet per round trip time. Upon encountering a loss, the window size is reduced by half, or equivalently, the sending rate is cut in half. In TCP, the receiver has the ability to set a maximum window size for the unacknowledged packets, hence imposing a maximum sending rate. Thus, in a non-congestion scenario, the application at the receiver can control the sending rate by setting the window size appropriately. On the other hand, during congestion, the actual throughput can be substantially low as the maximum window size may never be reached.

Based on the above discussion, we observe that a single packet loss can drop the TCP throughput abruptly and the low throughput lingers due to the slow increase of the window size. If there is a way to reduce this throughput reduction effect without modifying TCP, we can effectively provide higher throughput with proper congestion control and reliable transmission. In addition, if there is a way to control the TCP sending rate during congestion, then TCP can be made suitable for multimedia streaming. Unlike non real-time applications such as file transfer and email, precise control of sending rate is essential for interactive and live streaming applications due to several reasons. First, sending at too high a rate can cause buffer overflow in certain receivers with limited buffer such as mobile phones and PDAs. Second, sending at a rate lower than the coded bit rate results in pauses during a streaming session, unless a large buffer is accumulated before playback.

In the following section, we propose a system that can dynamically distribute streaming data over multiple TCP connections per application to achieve higher throughput and precise rate control. The control is performed entirely at the receiver side and thus, suitable for streaming applications where a single server may serve up to thousands of receivers simultaneously.

3 MultiTCP overview and Analysis

As mentioned in Section 2, the throughput reduction of TCP is attributed to the combination of (a) reduction of the sending rate by half upon detection of a loss event and (b) the slow increase of sending

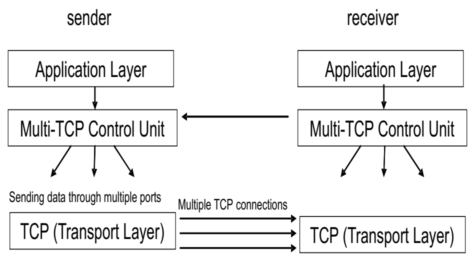


Figure 1: *MultiTCP* system diagram.

rate afterward or *congestion avoidance*. To alleviate this throughput reduction, one can modify TCP to (a) reduce the sending rate by a small factor other than half upon detection of a loss, or (b) speed up the congestion avoidance process, or (c) combine both (a) and (b). There are certain disadvantages associated with these approaches. First, these changes affect all TCP connections and must be performed by recompiling the OS kernel of the sender machine. Second, changing the decreasing multiplicative factor and the additive term in isolated machines may potentially lead to instability of TCP in a larger scale of the network. Third, it is not clear how these factors can be changed to dynamically control the sending rate.

As such, we propose a different approach: instead of using a traditional, single TCP connection, we use multiple TCP connections for a multimedia streaming application. Our approach does not require any modification to the existing TCP stack or kernel. Figure 1 shows a diagram of our proposed *MultiTCP* system. The MultiTCP control unit is implemented immediately below the application layer and above the transport layer at both the sender and the receiver. The MultiTCP control unit at the receiver receives the input specifications from streaming application which include the streaming rate and the throughput resilience. The throughput resilience can be thought of as the amount of throughput reduction an application can tolerate in presence of sudden burst traffic. A higher throughout resilience leads to a lower short-term throughput reduction. The MultiTCP control unit at the receiver measures the actual throughput and uses this information to control the rate and the throughput reduction by using multiple TCP connections and dynamically changing receiver's window size for each connection. In the next two sections, we show how multiple TCP connections can mitigate the throughput reduction problem in a lightly loaded network and describe our mechanism to maintain the desired throughput in a congested network.

3.1 Alleviating Throughput Reduction In Lightly Loaded Network

In this section, we analyze the throughput reduction problem in a lightly loaded network and show how it can be alleviated by using multiple TCP connections.

When there is no congestion, the receiver can control the streaming rate in a single TCP connection quite accurately by setting the maximum the receiver's window size W_{max}. The effective throughput during this period is approximately equal to

$$T = \frac{W_{max} MTU}{RTT} \qquad (1)$$

where RTT denotes the round trip time, including both propagation and queuing delay, between the sender and the receiver. MTU denotes the TCP maximum transfer unit, typically set at 1000 bytes. If a loss event occurs, the TCP sender instantly reduces its rate by half as shown in Figure 2(a). As a result, the area of the inverted triangular region in Figure 2(a) indicates the amount of data that would have been transmitted if there were no loss event. Thus, the amount of data reduction D equals to

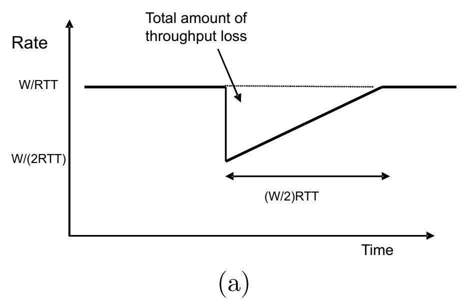
$$D = (\frac{1}{2})(\frac{W_{max} MTU RTT}{2})(\frac{W_{max}}{2RTT}) = \frac{W_{max}^2 MTU}{8} \qquad (2)$$

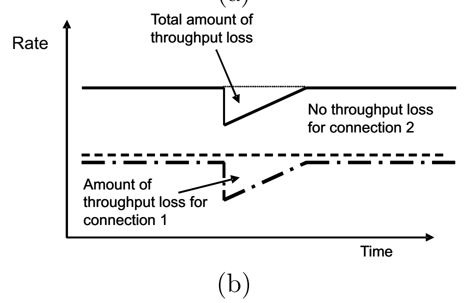
Note that the time it takes for the TCP window to increase from $W_{max}/2$ to W_{max} equals to $W_{max} RTT/2$ since the TCP window increases by one every round trip time. Clearly, if there are a burst of loss events during a streaming session, the total throughput reduction can potentially be large enough to deplete the start up buffer, causing pauses in the playback.

Now let us consider the case where two TCP connections are used for the same application. Since we want to keep the same total streaming rate W_{max}/RTT as in the case of one TCP connection, we set $W_{max}^{'} = W_{max}/2$ for each of the two connections as illustrated in Figure 2(b). Assuming that only a single loss event happens in one of the connection, the total throughput reduction would be equal to

$$D^{'} = \frac{(W_{max}^{'} MTU)}{8} = \frac{(W_{max}^2 MTU)}{32} = \frac{D}{4} \qquad (3)$$

Equation (3) shows that, for a single loss event, the throughput reduction of using two TCP connections is four times less than that of using a single TCP connection. Even in the case when there are simultaneously losses on both connections as indicated in Figure 2(c), the throughput reduction is half of that of the single TCP. In general, let N denote the number





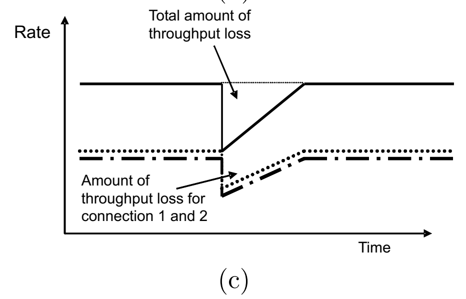


Figure 2: Illustrations of throughput reduction for (a) one TCP connections with single loss; (b) two TCP connections with single loss; (c) two TCP connections with double losses.

of TCP connections for the same application and n be the number of TCP connections that suffer simultaneous losses during short congestion period, the amount of throughput reduction equals to

$$D_N = \frac{nW_{max}^2 MTU}{N^2} \tag{4}$$

As seen in Equation (4), the amount of throughput reduction is inversely proportional to the square of the number of TCP connections used. Hence, using a only small number of TCP connections can greatly improve the resilience against TCP throughput reduction.

3.2 Control Streaming Rate in a Congested Network

In the previous section, we discuss the throughput reduction problem in a lightly loaded network and show that using multiple TCP connections can alleviate the problem. In a lightly loaded network condition, one can set the desired throughput T_d by simply setting the receiver window $W_{max} = T_d RTT/MTU$. However, in a moderately or heavily congested network, the throughput of a TCP does not depend on W_{max}, instead, it is determined by the degree of congestion. This is due to the fact that in a non-congested network, i.e. without packet loss, TCP rate would increase additively until $W_{max}MTU/RTT$ is reached, after that the rate would remain approximately constant at $W_{max}MTU/RTT$. However, in a congested network, a loss event would most likely occur before the sending rate reaches its limit and cut the rate by half, resulting in a throughput lower than $W_{max}MTU/RTT$.

A straightforward method for achieving a higher throughput than the available TCP throughput would be to use multiple TCP connections for the same application. Using multiple TCP connections results in a larger share of the fair bandwidth. Hence, one may argue that this is unfair to other TCP connections. On the other hand, one can view this approach as a way of providing higher priority for streaming applications over other non time-sensitive applications under resource constraints. We also note that one can use UDP to achieve the desired throughput. However unlike UDP, using multiple TCP connections can provide (a) congestion control mechanism to avoid congestion collapse, and (b) automatic retransmission of lost packets. Assuming multiple TCP connections are used, there are still issues associated with providing the desired throughput in a congested network.

In order to maintain a constant throughput during a congested period, one possible approach is to increase the number of TCP connections until the measured throughput exceeds the desired one. This approach suffers from a few drawbacks. First, the total resulting throughput may still exceed the desired throughput by a large amount since the sending rate of each additional TCP connection may be too high. Second, if only a small number of TCP connections are required to exceed the desired throughput, this technique may not be resilient to the sudden increase in traffic as analyzed in Section 3.1. A better approach is to use a larger number of TCP connections but adjust the receiver window size of each connection to precisely control the sending rate. It is undesirable to use too many TCP connections as they use up system resources and may further aggravate an already congested network. In practice, our algorithm maintains a relatively stable number of TCP connections while varies the size of the receiver windows to achieve the desired throughput. In the next section, we describe the algorithm to adjust the receiver windows.

3.2.1 The Algorithm

Based on previous discussion, our current algorithm uses a fixed, default number of TCP connections and only varies receiver window size to obtain the desired throughput T_d. We envision that the application or the user can change the default number of TCP connections as deemed necessary. Hence, the inputs to the algorithm are the desired user's throughput T_d and the number of TCP connections. Below are the steps of our proposed algorithm.

Initializing steps:

1. Set N, the number of TCP connection to the user input.
2. Set the receiver window size $w_i = \frac{T_d RTT}{(MTU)N}$ for connection i.

Running steps:

The actual throughput T_m is measured at at every δ second and the algorithm dynamically changes the window size based on the measured T_m as follows.

3. If both of the following conditions
   (a) $T_m < T_d$, and
   (b) $W_s = \sum_i w_i \leq \frac{f T_d RTT}{MTU}$ where $f > 2$

   are true, run *AdjustWindow(T_d, T_m)*.
4. If $T_m > T_d + \lambda$, run *AdjustWindow(T_d,T_m)*.
5. Else, keep the receiver window size the same.

We now discuss each step of the algorithm in detail and show how to choose appropriate values for the parameters. In step 1, we found empirically, $N = 5$ works well in many scenarios. If user does not specify the number of TCP connections, the default value is set to $N = 5$. In step 2, we assume that the network is not congested initially, hence the total expected throughput and the total receiver window size W_s would equal to T_d and $T_d RTT/MTU$ respectively. Note that the average RTT can be obtained easily at the receiver.

In the running steps, δ should be chosen to be several times the round trip time since the sender cannot respond to the receiver changing window size for at least one propagation delay, or approximately half of RTT. As a result, the receiver may not observe the change in throughput until several RTTs later. In most scenarios, we found that setting the measuring interval $\delta = 8RTT$ works quite well in practice. In step 3, the algorithm tries to increase the throughput by increasing the window size of each connection via the routine *AdjustWindow*. The implementation details of *AdjustWindow* are discussed in Section 3.2.2. The first condition of step 3 indicates the measure throughput is still under the desired one. The second condition limits the maximum total receiver window size. Recall that in the congestion state, the average size of the receiver window is $W_{max} = \frac{2T_m RTT}{MTU}$. Hence, increasing w_i beyond this value would not increase the TCP throughput. However, if we let w_i increase without bound, there will be a spike in throughput once the network becomes less congested. To prevent unnecessary throughput fluctuation, our algorithm limits the sum of receiver window size W_s to $f\frac{T_d RTT}{MTU}$ where $f > 2$ is used to control the height of the throughput spike. Larger and small values of f result in higher and lower throughput spikes respectively, as discussed later in Section 4. We note that if the desired bandwidth is smaller than the physical bandwidth limit, an improved algorithm can open a new connection to potentially achieve the desired bandwidth. At present, we have not investigated this approach in details, as we have not found a good criterion for removing connections when the network becomes less congested.

Step 4 of the algorithm is similar to step 3 except the receiver window size now would be reduced, using the same *AdjustWindow* routine. λ in the inequality $T_m > T_d + \lambda$ is a small throughput threshold used to ensure T_m to be approximately equal to T_d, and at the same time, to prevent T_m from going below T_d. Finally, since the measured throughput can be noisy, we use the exponential average measured throughput computed recursively as $T_m = \alpha T_m + (1 - \alpha) T_n$ where T_n is the new throughput sample and $\alpha < 1$ is a smoothing parameter.

3.2.2 Adjusting the receiver window sizes

We now discuss *AdjustWindow* in detail. In this step, the algorithm increases (decreases) the window size w_i for a subset of connections if the measured throughput is smaller (larger) than the desired throughput. There exists an optimal way for choosing a subset of connections for changing the window size and the corresponding increments in order to achieve the desired throughput. If the number of chosen connections for changing the window size and the corresponding window increments are small, then the time for achieving the desired throughput maybe longer than necessary. On the other hand, choosing too large a number of connections and increments may result in higher throughput than necessary. For example, assuming we have five TCP connections, each with RTT of 100 milliseconds, MTU equals to 1000 bytes, and the network is in non-congestion state, then changing the receiver window size of all the connections by one can result in a total change in throughput of 5(1000)/.1 = 50 Kbytes per second. In a congested scenario, the change will not be that large, however, one may still want to control the throughput change to a certain granularity. To avoid these drawbacks, our algorithm chooses the number of connections for changing their window size and the corresponding increments based on the current difference between the desired and measured throughput. The pseudo codes of the algorithm is shown below.

```
AdjustWindow($T_d$, $T_m$)
1. $D_s = \lceil |T_d - T_m| RTT/MTU \rceil$
2. If $T_d > T_m$
    (a) Sort the connections in the increasing order of $w_i$
    (b) While $D_s > 0$
            $w_i := w_i + 1$
            $D_s := D_s - 1$
            $i := (i+1) \mod N$
3. If $T_d < T_m$
    (a) Sort the connections in the decreasing order of $w_i$
    (b) While $D_s > 0$
            $w_i := w_i - 1$
            $D_s := D_s - 1$
            $i := (i+1) \mod N$
```

The reasoning behind the algorithm is as follows. Consider the case when $T_m < T_d$. If there is no congestion, setting the sum of window size increments D_s from all the connections to $\lceil (T_d - T_m) RTT/MTU \rceil$ would result in a throughput increment of $T_d - T_m$, hence the desired throughput would be achieved. If there is a congestion, this total throughput increment would be smaller. However, subsequent rounds of window increment would allow the algorithm to reach the desired throughput. This method effectively produces a large or small total window increment at every sampled point based on a large or small difference between the measured and desired throughput, respectively. Steps 2a and 2b in the above algorithm ensure the total throughput increment is equally contributed by all the connections. On the other hand, if only one connection j is responsible for all the throughput, i.e. $w_i = 0$ for $j \neq i$, then we simply have a single connection whose throughput can be substantially reduced in a congested scenario. We note that using our algorithm, w_i's for different connections at any point in time differ from each other at most by one. The scenario where $T_m > T_d$ is similar.

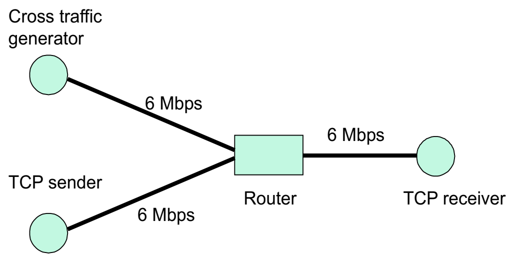


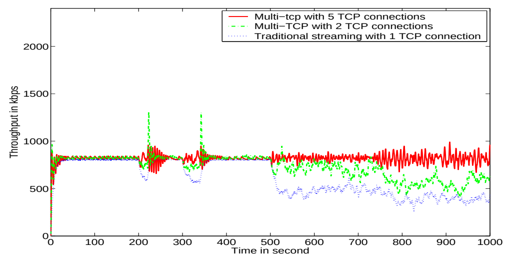
Figure 3: *Simulation topology.*

3.2.3 Remarks on Sender

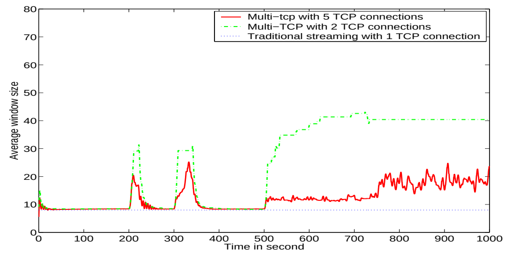
At the sender, data is divided into packets of equal size. These packets are always sent in order. The MultiTCP system chooses the TCP connection to send the next packet in a round robin fashion. If a particular TCP connection is chosen to send the next packet, but it is blocked due to TCP congestion mechanism, the MultiTCP system chooses the first available TCP connection in a round robin manner. For example, suppose there are 5 connections, denoted by TCP1 to TCP5. If none of TCP connection is blocked, packet 1 would be sent by TCP1, packet 2 by TCP2, and so on. If TCP1 is blocked, then TCP2 would send packet 1 and TCP3 would send packet 2, and so on. When it is TCP1's turn again and if TCP1 is not blocked, it would send packet 5. This is similar to socket striping technique in [14].

4 Results

In this section, we show simulation results using NS to demonstrate the effectiveness of our MultiTCP system in achieving the required throughput as compared to the traditional single TCP approach. Our simulation setup consists of a sender, a receiver, and a traffic generator connected together through a router to form a dumb bell topology as shown in Figure 3. The bandwidth and propagation delay of each link in the topology are identical, and are set to 6 Mbps and 20 milliseconds, respectively. The sender streams 800 kbps video to the receiver continuously for a duration of 1000s, while the traffic generator generates cross traffic at different times by sending packets to the receiver using either long term TCPs or short bursts of



(a)



(b)

Figure 4: (a) Resulted throughput and (b) average receiver window size when using 1, 2 and 5 TCP connections.

UDPs. In particular, from time $t = 0$ to $t = 200$s, there is no cross traffic. From $t = 200$s to $t = 220$s and $t = 300$s to $t = 340$s, bursts of UDPs with rate of 5.5 Mbps are generated from the traffic generator node to the receiver. At $t = 500$s the traffic generator opens 15 TCPs connections to the receiver, and 5 additional TCP connections at $t = 750$s. We now consider this setup under three different scenarios: (a) the sender uses only one TCP connection to stream the video, while the receiver sets the receiver window size to 8, targeting at 800 kbps throughput, (b) the sender and the receiver use our MultiTCP system to stream the video with the number TCP connections limited to two, and (c) the sender and the receiver also use our proposed MultiTCP system, except the number of TCP connections are now set to five. Table 1 shows the parameters used in our MultiTCP system.

Sampling interval δ	300 ms
Throughput smoothing factor α	0.9
Guarding threshold λ	7000 bytes
Throughput spike factor f	6

Table 1: *Parameters used in MultiTCP system*

Figure 4(a) shows the throughput of three described scenarios. As seen, initially without congestion, using the traditional single TCP connection can control the throughput very well since setting the size of the receiver window to 8 achieves the desired throughput. However, when traffic bursts occur during the intervals $t = 200s$ to $t = 220s$ and $t = 300s$ to $t = 340s$, the throughput of using a single TCP connection reduces substantially to only about 600 kbps. For the same congested period, using two TCP connections results in higher throughput, approximately 730 kbps. On the other hand, using five TCP connections produces approximately the desired throughput, demonstrating that a larger number of TCP connections results in higher throughput resilience in the presence of misbehaved traffic such as UDP flows. These results agree with the analysis in Section 3.1. It is interesting to note that when using two TCP connections, there are spikes in the throughput immediately after the network is no longer congested at $t = 221$s and $t = 341$s. This phenomenon relates to the maximum receiver window size set during the congestion period. Recall that the algorithm keeps increasing the w_i until either (a) the measured throughput exceeds the desired throughput or (b) the sum of receiver window size $W_s = \sum_i w_i$ reaches $f\frac{T_d RTT}{MTU}$. In the simulation, using two TCP connections never achieves the desired throughput during the congested periods, hence the algorithm keeps increasing the w_i. When network is no longer congested, the W_s already accumulates to a large value. This causes the sender to send a large amount of data until the receiver reduces the window size to the correct value a few RTTs later. On the other hand, when using 5 TCP connections, the algorithm achieves the desired throughput during the congestion periods, as such W_s does not increase to a large value, resulting in a smaller throughput spike after the congestion vanishes.

Next, when 15 cross traffic TCP connections start at $t = 500$s, the resulting throughputs when using one and two TCP connections reduce to 700 kbps and 350 kbps, respectively. However, throughput when using 5 TCP connections stays approximately constant at 800 kbps. At $t = 750$s, 5 additional TCP connections start, throughput are further reduced for the one and two connection cases, but it remains constant for the five-connection case. These results demonstrate that our algorithm is able to achieve the desired throughput and maintain precise rate control under a variety congested scenarios with competing UDP and TCP traffic. We should emphasize again that, the applications based on our system indeed obtain a larger share of the fair bandwidth. However, we believe that under limited network resources, time-sensitive applications like multimedia streaming should be treated

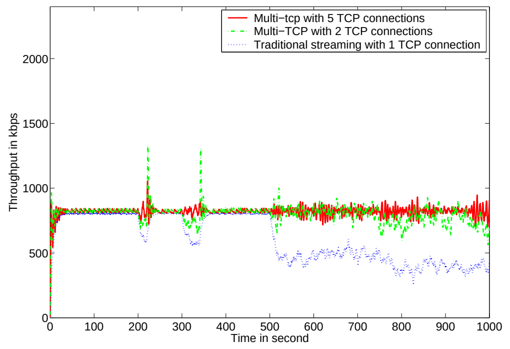


Figure 5: Resulted throughput when using 1, 2 and 5 TCP connections with cross traffic having larger RTT than that of video traffic.

preferentially as long as the performance of all other applications do not degrade significantly. Since our system uses TCP, congestion collapse is not likely to happen as in the case of using UDP when network is highly congested. In fact, DiffServ architecture uses the same principle by providing preferential treatment to high priority packets.

Figure 4(b) shows the average of the sum of window size W_s as a function of time. As seen, W_s increases and decreases appropriately to respond to network conditions. Note that using two connections, W_s increases to a larger value than when using 5 TCP connections during the intervals of active UDP traffic. This results in throughput spikes discussed earlier. Also, the average window size in the interval $t = 500$s to $t = 750$s is smaller than that of the interval $t = 750$s to $t = 1000$s, indicating that the algorithm responds appropriately by increasing the window size under a heavier load.

We now show the results when the cross traffic has different round trip time from that of the video traffic. In particular, the propagation delay between the router and traffic generator node is now set to 40 milliseconds. All cross traffic patterns stay the same as before. The new simulation shows the same basic results. As seen in Figure 4, the throughput of 5 connections is still higher than that of two connections which, in turn is higher than that of one connection during the congestion periods. The throughput of two connections in this new scenario is slightly higher that that of the previous scenario during the congested period from $t = 500$s onward. This is due to a well known phenomena that TCP connection with shorter round trip times gets a larger share of bandwidth for the same loss rate. Since the round trip time of the video traffic is now shorter than that of the TCP cross traffic, using only two connections, the desired throughput of 800 kbps can be approximately achieved during the period from $t = 500$s to $t = 750$s, which is not achievable in previous scenario. So clearly, the number of connections to achieve the desired throughput depends on the competing traffic. In practice, to avoid the potential complexity associated opening and closing connections, we recommend using a fix number of connections such as five.

5 Related Work

There have been previous work on using multiple network connections to transfer data. For example, *path diversity* multimedia streaming framework [10][11][9] provide multiple connections on different path for the same application. These work focus on either efficient source or channel coding techniques in conjunction with sending packets over multiple approximately independent paths. On the other hand, our work aims to increase and maintain the available throughput using multiple TCP connections on a single path. There is also a related work using multiple connections on a single path to improve throughput of a wired-to-wireless streaming video session [15]. This work focuses on obtaining maximum possible throughput and is based TFRC rather than TCP. On the other hand, our work focuses on eliminating *short term* throughput reduction of TCP due to burst traffic and providing precise rate control for the application. As such, the analysis and rate control mechanism in our paper are different from those of [15]. Another related work is Streaming Control Transmission Protocol (SCTP)[16], designed to transport PSTN signaling messages over IP networks. SCTP allows user's messages to be delivered within multiple streams, but it is not clear how it can achieve the desired throughput in a congestion scenario. In addition, SCTP is a completely new protocol, as such the kernel of the end systems need to be modified. There are also other work related to controlling TCP bandwidth. For example, the work in [17] focuses on allocating bandwidth among flows with different priorities. This work assumes that the bottleneck is at the *last-mile* and that the required throughput for the desired application is achievable using a single TCP connection. On the other hand, our work does not assume the *last-mile* bottleneck, and the proposed *MultiTCP* system can achieve the desired throughput in variety of scenarios. Also, the authors in [18], use weighted proportional fair sharing web flows to provide end-to-end differentiated services. The work in [19] uses the receiver advertised window to limit the TCP video bandwidth in

VPN link between video and proxy servers. Finally, the authors in [20] propose a technique for automatic tuning of receiver window size in order to increase the throughput of TCP.

6 Conclusions

We conclude our paper with a summary of contributions. First, we propose and implement a receiver-driven, TCP-based system MultiTCP for multimedia streaming over the Internet using multiple TCP connections for the same applications. Second, our proposed system is able to provide resilience against short-term insufficient bandwidth due to traffic bursts. Third, our proposed system enables the application to control the sending rate in a congested scenario, which cannot be achieved using traditional TCP. Finally, our proposed system is implemented at the application layer, and hence, no kernel modification to TCP is necessary. The simulation results demonstrate that using our proposed system, the application can achieve the desired throughput in many scenarios, which cannot be achieved by traditional single TCP approach.

References

[1] MovieFlix, http://www.movieflix.com/.

[2] W. Tan and A. Zakhor, "Real-time internet video using error resilient scalable compression and tcp-friendly transport protocol," *IEEE Transactions on Multimedia*, vol. 1, pp. 172–186, june 1999.

[3] G. De Los Reyes, A. Reibman, S. Chang, and J. Chuang, "Error-resilient transcoding for video over wireless channels," *IEEE Transactions on Multimedia*, vol. 18, pp. 1063–1074, june 2000.

[4] A. Reibman, "Optimizing multiple description video coders in a packet loss environment," in *Packet Video Workshop*, April 2002.

[5] H. Ma and M. El Zarki, "Broadcast/multicast mpeg-2 video over wireless channels using header redundancy fec strategies," in *Proceedings of The International Society for Optical Engineering (SPIE)*, November 1998, vol. 3528, pp. 69–80.

[6] S. Blake, D. Black, M. Carson, E. Davis, Z. Wang, and W. Weiss, "An architecture for differentiated services," in *RFC2475*, December 1998.

[7] Z. Wang, *Internet QoS, Architecture and Mechanism for Quality of Service*, Morgan Kaufmann Publishers, 2001.

[8] P. White, "Rsvp and integrated services in the internet: A tutorial," *IEEE Communication Magazine*, pp. 100–106, May 1997.

[9] T. Nguyen and A. Zakhor, "Multiple sender distributed video streaming," *IEEETransactions on Multimedia and Networking*, vol. 6, no. 2, pp. 315–326, April 2004.

[10] J. Apostolopoulos, "Reliable video communication over lossy packet networks using multiple state encoding and path diversity," in *Proceeding of The International Society for Optical Engineering (SPIE)*, January 2001, vol. 4310, pp. 392–409.

[11] J. Apostolopoulos, "On multiple description streaming with content delivery networks," in *InfoComm*, June 2002, vol. 4310.

[12] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast application," in *Architectures and Protocols for Computer Communication*, October 2000, pp. 43–56.

[13] Information Sciences Institute, http://www.isi.edu/nsnam/ns, *Network simulator*.

[14] J. Leigh, O. Yu andD. Schonfeld, and R. Ansari, "Adaptive networking for tele-immersion," in *Immersive Projection Techonology/Eurographics Virtual Environments Workshop(IPT/EGVE)*, May 2001.

[15] M. Chen and A. Zakhor, "Rate control for streaming over wireless," in *INFOCOM*, July 2004.

[16] Internet Engineering Task Force, RFC 1771, *Stream Control Transmission Protocol*, october 2000.

[17] P. Mehra and A. Zakhor, "Receiver-driven bandwidth sharing for tcp," in *INFOCOM*, San Francisco, April 2003.

[18] J. Crowcroft and P.Oeschlin, "Differentiated end-to-end internet services using weighted proportional fair sharing tcp," 1998.

[19] Y. Dong, R. Rohit, and Z. Zhang, "A practical technique for supporting controlled quality assurance in video streaming across the internet," in *Packet Video*, 2002.

[20] J. Semke, J. Mahdavi, and M. Mathis, "Automatic tcp buffer tuning," in *SIGCOMM*, 1998.