Congestion Control in Distributed Media Streaming

Lin Ma
School of Computing
National University of Singapore
malin@comp.nus.edu.sg

Wei Tsang Ooi
School of Computing
National University of Singapore
ooiwt@comp.nus.edu.sg

***Abstract*— Distributed media streaming, which uses multiple senders to collaboratively and simultaneously stream media content to a receiver, poses new challenges in congestion control. Such approach establishes multiple flows within a session. Since conventional congestion control only aims to make each of these flows TCP-friendly, selfish users can increase the number of flows to grab a larger share of the bandwidth, introducing more congestion and degrading the overall network performance. To address this issue, we propose the idea of task-level TCP-friendliness, which enforces TCP-friendliness upon a set of flows belonging to a task instead of upon individual flow. We design DMSCC, a congestion control scheme, to achieve task-level TCP-friendliness in distributed media streaming. By observing shared congestion, DMSCC identifies the set of flows experiencing congestion and dynamically adjusts those flows such that their combined throughput is TCP-friendly. To achieve this goal, DMSCC addresses two issues: (i) given a β ($\beta < 1$), how to control a flow using AIMD such that it consumes β-times the throughput of a TCP flow, and (ii) how to identify the set of flows that share a bottleneck. In our simulations, DMSCC can effectively regulate the throughput of flows on every bottleneck, resulting in a TCP-friendly combined throughput.**

I. Introduction

The term *distributed media streaming*, coined by Nguyen and Zakhor [1], refers to a new model in media streaming in which a client receives a media stream from multiple servers simultaneously. This new framework can be applied to several settings. In media-on-demand services, media content can be mirrored by different CDN servers, and a client can stream concurrently from these servers. In peer-to-peer file sharing applications, a receiver can stream from multiple peers that seed the same media file.

There are two main advantages in streaming from multiple senders concurrently. First, it improves robustness. By exploiting path-diversity among the senders, the receiver experiences average loss and congestion behavior of the paths. Careful selection of senders and allocation of packets to different senders lead to significant reduction in bursty loss [2]. In peer-to-peer settings with transient peers, the receiver can still receive and play parts of the media stream when one of the sending peer fails [3].

Second, distributed media streaming allows aggregation of bandwidth among peers. Most home computers have asymmetric Internet connections, where the down-link data rate is higher than the up-link data rate. Such asymmetry limits the ability of a peer to stream to another peer at full quality. For instance, a peer might request for a 128Kbps audio, but few peers would be able to, or willing to, dedicate this amount of bandwidth to serve another peer. A more likely situation is for, say, four peers, each contributing 32Kbps, to serve the requesting peer.

Streaming from multiple senders simultaneously poses many new research challenges. Existing work in the literature has studied issues such as rate allocation [1], allocation of packets [4], and selection of senders [3]. The existence of multiple senders has brought a new dimension to classic problems in media streaming as well. For instance, we recently studied error recovery via retransmission of loss packets in the context of distributed media streaming [5]. In this paper, we investigate another fundamental issue in media streaming – TCP-friendly congestion control.

Congestion control in media streaming with one sender is an important and well studied problem (e.g., see the survey by Widmer et al. [6] and references therein). In distributed media streaming, however, the problem of congestion control is more complicated than the single sender scenario.

A distributed media streaming session contains multiple media flows (called *DMS flows*) from different senders. These flows may or may not pass through the same bottleneck. Ensuring TCP-friendliness of each DMS flow is not sufficient: their combined throughput is larger than the other TCP flows on the same bottleneck. This unfairness encourages abuse by selfish users — by increasing the number of concurrent flows, a user can grab larger bandwidth share at the bottlenecks. We need a different type of congestion control – one that controls the *aggregate* throughput of the DMS flows such that their combined throughput is TCP-friendly. We call such aggregate congestion control as *task-level congestion control*.

Aggregate congestion control methods exist in the literature [7]–[10], but do not apply to distributed media streaming. In distributed media streaming, the flows from multiple senders converge on their way to the receiver, forming a reverse tree (see Fig. 1 for an example). The DMS flows only share parts of their links, so they may experience different delay and congestion. The existing methods of aggregate congestion control, however, assume that the flows traverse through the same path and share the same bottleneck.

We now illustrate the problem of congestion control in distributed media streaming through an example (see Fig. 1). A host R requests for some media content from senders S_i. DMS flows (f_i) from the senders travel through different IP-level paths and join each other at routers A and B. We term

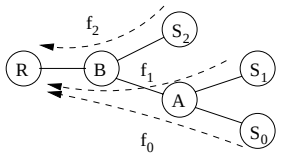


Fig. 1. Reverse Tree Topology in Distributed Media Streaming.

routers like A and B as *aggregation points*. Throughout this paper, we use the term *link* to refer to the set of physical links between a sender and an aggregation point (e.g., S_0-A), two aggregation points (e.g., A-B), or an aggregation point and the receiver (e.g., B-R). The set of DMS flows on a link is unique. Determining the set of DMS flows on a link is important, as it is the element upon which TCP-friendliness is enforced. Section II elaborates on this point.

In the reverse tree, congestion can occur on any link. If it occurs on R-B, the aggregate of f_0, f_1 and f_2 should be friendly to TCP flows on link R-B. But if the congestion occurs on A-B, only the aggregate of f_1 and f_2 needs to be friendly to TCP flows on link A-B. Flow f_2, on the other hand, can consume as much bandwidth as it wants. Similarly, if the congestion occurs on link S_0-A, only f_0 needs to be TCP-friendly.

The above example shows the difficulty in congestion control of distributed media streaming – the set of DMS flows to be controlled depends on where congestion appears. So the solution needs to first identify the flows sharing the same congestion, and then regulate them accordingly.

This paper proposes a complete framework called DMSCC to achieve the above tasks. DMSCC tracks packet losses at the receiver as an indication of congestion and identifies the location of congestion by correlating the one-way delays between sender/receiver pairs. Additive increase, multiplicative decrease (AIMD) algorithm, with carefully adjusted increasing factor, regulates the throughput of the DMS flows on a bottleneck and produces a TCP-friendly flow aggregate. If there are k DMS flows on a bottleneck, they are regulated such that, in ideal situation, each flow consumes $1/k$ of the bandwidth of a TCP flow in a comparable network condition. As a result, the flow aggregate consumes as much as one TCP flow and is friendly to TCP. We use only TCP Reno in this paper, but the scheme is applicable to other versions of TCP.

When the throughput of the each flow is regulated, the receiver needs to decide which packets each sender should send to conform to the new throughput constraint. This and other issues (e.g., what to retransmit, media coding methods used) are orthogonal to congestion control and are beyond the scope of this paper.

The rest of the paper is organized as follows. In Section II, we make a case for task-level TCP-friendliness and formulate the congestion control problem to achieve task-level TCP-friendliness in distributed media streaming. Section III describes the framework of DMSCC and presents our assumptions. Section IV presents the methods to control throughput of DMS flows. Section V describes how DMSCC locates congestion in a reverse tree. Section VI shows how DMSCC combines congestion location and throughput control to achieve TCP-friendliness at the task level. Section VII presents simulation results, which validate our design. Some related work is presented in Section VIII. Section IX concludes the paper.

II. Problem Formulation

A. Task-level TCP-Friendliness

The term TCP-friendly is commonly used to describe a flow whose arrival rate at steady state is no more than the arrival rate of a TCP flow under the same network condition (such as packet loss rate and round trip time). We refer to congestion control schemes that aim to produce TCP-friendly flows as *flow-level congestion control*. Several work in the literature extends the notion of TCP-friendliness to coarser granularity. Hacker et al. [10] consider parallel TCP flows and propose an approach where multiple parallel TCP flows in one download session are friendly to a single (unmodified) TCP flow. We call this approach as *task-level congestion control*. Finally, congestion manager [7] seeks fairness of flow aggregate between a pair of hosts. We refer to this approach as *host-level congestion control*.

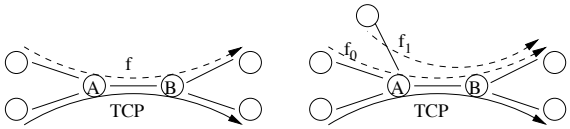
We believe that task-level congestion control is appropriate for Internet applications, including distributed media streaming. Congestion control pursues fair sharing of bandwidth at a bottleneck, and fairness is meaningful only when the entity of bandwidth consumption is identified. Such entity should have two properties: (i) An entity consumes bandwidth to complete a well-defined task for an end user; (ii) Creating more entities does not make completing the task better or faster. The second property is crucial in removing the motivation to abuse the network using multiple entities.

For example, an FTP file downloading session is an entity – the task is well defined, and downloading another file does not accelerate the completion of the current task. In this single-flow task, task-level congestion control is equivalent to flow-level congestion control. On the other hand, some applications (e.g., FlashGet[1]) allow users to download the same file with multiple flows concurrently. In this case, the multi-flow downloading session is one entity – (i) the task is still downloading of a file, and (ii) creating another multi-flow session for the same file does not speed up the current downloading. Task-level congestion control takes the whole downloading task as the entity of bandwidth consumption and keeps the total throughput friendly to TCP. Contrarily, flow-level congestion control only requires TCP-friendliness of individual flow. Therefore, the task consumes more bandwidth than a TCP flow, gaining advantage over other single-flow tasks. Without task-level TCP-friendliness, selfish users can use more flows to grab more bandwidth on bottlenecks.

B. The Criterion for Task-Level TCP-Friendliness

We now formally describe the goal of task-level congestion control.

[1] www.flashget.com



(a) a single flow task (b) a two–flow task

Fig. 2. A Single-Flow Task and a Two-Flow Task.

1) A Single-Flow Task: First, let's consider a task with only one flow, as shown in Fig. 2(a). The flow f and a TCP flow share bottleneck A-B. As the task has only one flow, task-level TCP-friendliness is equivalent to flow-level TCP-friendliness. Assuming that the RTT of both flows are the same, TCP-friendliness is achieved if the following equation holds:

$$B = B_{TCP}$$

where B and B_{TCP} are the throughput of f and the TCP flow, respectively.

Consider a more general case where the two flows experience different RTT. TCP's congestion control algorithm is biased against flows with larger RTT [11]. Despite efforts to correct such unfairness (e.g., TCP Libra [12]), this unfairness persists in current TCP implementations. On the other hand, B×RTT of the two TCP flows remain the same if they experience the same loss rate. For flow f and the TCP flow in Fig. 2(a), it is reasonable to assume a similar loss rate: A-B is the only bottleneck on their paths, and active queue management, such as RED, tries to drop packets from both flows in a fair manner. Therefore, under different RTT, TCP-friendliness is ensured by:

$$B \times RTT = B_{TCP} \times RTT_{TCP} \tag{1}$$

where RTT and RTT_{TCP} are the RTT of flow f and the TCP flow, respectively.

2) A Multi-Flow Task: We now extend Equation 1 to handle a multi-flow task sharing the same bottleneck with other TCP flows.

Consider a multi-flow task (e.g., Fig. 2(b)). The two flows f_0 and f_1 share bottleneck A-B with a TCP flow. Task-level TCP-friendliness requires the flow aggregate to be friendly to a TCP flow. If we treat the flow aggregate as a single flow, task-level TCP-friendliness is the same as flow-level TCP-friendliness. Therefore, Equation 1 holds; except that, B is now the combined throughput of f_i, and RTT is the average round trip time of f_i:

$$RTT = \frac{1}{B} \sum_{f_i \in O} (b_i \times rtt_i)$$

where O is the set of flows in the flow aggregate, b_i and rtt_i are the throughput and round trip time of flow f_i. By replacing B and RTT, we extend Equation 1 to consider multi-flow tasks:

$$\sum_{f_i \in O} (b_i \times rtt_i) = B_{TCP} \times RTT_{TCP} \tag{2}$$

Equation 2 provides the criterion for task-level TCP-friendliness on a given bottleneck. *Formally, a task is TCP-friendly if the combined $B \times RTT$ of its flows is equal to that of a TCP flow on the same bottleneck.*

3) The Goal of DMSCC: We now apply Equation 2 to the problem of congestion control in distributed media streaming. Consider a distributed media streaming session as shown in Fig. 1. As bottlenecks form on different links, the flow aggregates on them contain different sets of DMS flows. The criterion of task-level TCP-friendliness for distributed media streaming should consider multiple bottleneck locations with different sets of flows.

Let l_j be a link, and a TCP flow passing through l_j be TCP_j. As the set of DMS flows flowing through each link is distinct, we can represent a link using its set of DMS flows. We use set notations to represent relationships among the flows and the links. The notation $f_i \in l_j$ means that flow f_i passes through link l_j; and $l_i \supset l_j$ means that the flows on l_i are a proper superset of flows on l_j, or l_i *dominates* l_j for short.

Distributed media streaming is task-level TCP-friendly when, *on any bottleneck* l_j, the following inequality holds:

$$\sum_{f_i \in l_j} (b_i \times rtt_i) \leq B_{TCP_j} \times RTT_{TCP_j} \tag{3}$$

The above criterion is an inequality, as a DMS flow may experience multiple bottlenecks.

III. Model and Assumptions

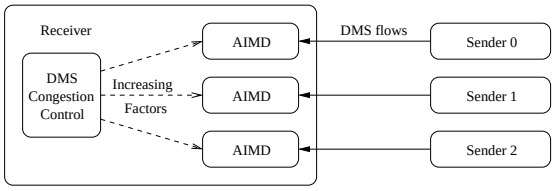


Fig. 3. A Three-Sender Session.

Our congestion control scheme, DMSCC, is designed to ensure that Inequality 3 is satisfied on any congested link in a distributed media session. DMSCC is a receiver-driven protocol – the receiver pulls the data from the senders by sending requests with sequence numbers, and the senders reply with data. The receiver therefore controls the sending rate of each senders and is the natural place to implement the congestion control protocol.

Fig. 3 shows the relationship between DMSCC and the DMS flows in a distributed media streaming session with three senders. There are three connections between the receiver and the senders. At the receiver, each connection is controlled by an AIMD loop similar to TCP. The increasing factors of these AIMD loops are controlled by the DMSCC module in the receiver. We will show in Section IV how the increasing factors of individual DMS flows are determined. But first, in this section, we introduce the framework of DMSCC and present our assumptions in the design of our protocol.

A. AIMD versus TFRC

AIMD and equation-based method [13] are two common methods for regulating the throughput of a non-TCP flow. We use AIMD method to regulate DMS flows in DMSCC for the following reason.

Equation-based methods rely on long term observation of network parameters such as loss rate and smoothed RTT. These parameters are used in an equation to estimate the long term throughput that is fair to TCP. This long term observation is meaningful only in cases where flows share the same path, and bottlenecks affect the same set of flows. In distributed media streaming, the congestion may affect different set of DMS flows at different bottlenecks. Thus, a long term observation might become outdated and fail to capture the congestion on a particular bottleneck. On the other hand, AIMD methods respond quickly to a packet loss and adapt swiftly to congestion on new bottleneck. Although it is argued that AIMD produces saw-tooth like throughput, in non-interactive streaming, as in the case of distributed media streaming, buffering can be used to smooth the playback at the receiver.

B. DMSCC

The framework of DMSCC is shown in Fig. 4. DMSCC has two relatively independent functionalities: throughput control (Section IV) and congestion location (Section V). These two functionalities cooperate to perform task-level congestion control on DMS flows. When congestion occurs, the congestion location module identifies the bottleneck. The throughput control module then updates the increasing factor of AIMD loops of each DMS flow on that bottleneck.

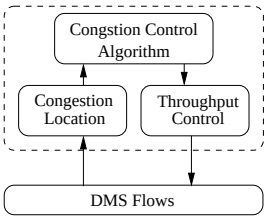


Fig. 4. Framework of DMSCC

C. Assumptions

Before proceeding to descriptions of DMSCC, we first clarify our assumptions. First, we assume that the paths among the receiver and senders form a reverse tree rooted at the receiver, and this topology is known by the receiver. Second, we assume that DMS flows on the same bottleneck link experience similar loss rate. This assumption is reasonable when active queue management schemes such as RED is used. Third, we focus on links with high multiplexing factors, where loss rate is decided by the background traffic rather than the DMS flows. Lastly, we can reasonably assume that the number of senders in a DMS session is typically small (less than 10). Thus, scaling DMSCC to large number of senders is not an issue.

IV. Throughput Control

In this section, we describe how to control the throughput of a DMS flow using AIMD algorithm such that it achieves a fixed fraction of the throughput of a TCP flow. In order for an aggregate of k DMS flows to be fair to a single TCP flow, DMSCC tries to control the throughput of each of the DMS flow to be $1/k$ of the throughput of a conformant TCP flow.

We derived our method from the well-known Mathis Equation [14]. Mathis et al. assume that packet losses are distributed in such a way that, if the loss rate is p, then for every $1/p$ packets, one packet is lost. Fig. 5 shows the variation of congestion window in such an ideal lossy channel. W denotes the size of the congestion window (in number of packets) before packet loss. Every packet loss reduces the congestion window to $W/2$. The congestion window then increases by α packets for every RTT, until the next packet loss occurs.

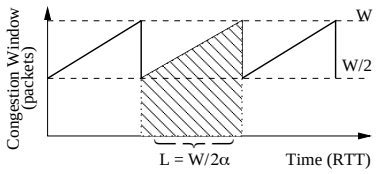


Fig. 5. Evolution of Congestion Window Under Periodic Loss.

The variable α is the *increasing factor*. If we let the period (in RTT) between every two packet losses be L, then

$$L = \frac{W/2}{\alpha}.$$

The total number of packets received during that period can be calculated as the size of the shaded area S:

$$S = \frac{3}{2} \times \frac{W}{2} \times L = \frac{3W^2}{8\alpha}. \quad (4)$$

From the assumption of ideal packet loss pattern, we know that the number of packets between two packet losses is $1/p$, that is,

$$S = \frac{1}{p}. \quad (5)$$

From Equation 4 and 5, we obtain:

$$W = \sqrt{\alpha} \times \sqrt{\frac{8}{3p}}.$$

The throughput of a flow is proportional to the average size of congestion window, which is:

$$\overline{W} = \frac{3}{4}W = \sqrt{\alpha} \times \frac{3}{4}\sqrt{\frac{8}{3p}}. \quad (6)$$

Equation 6 provides us a way to change the throughput by adjusting its increasing factor α. If we want a DMS flow to have β times the throughput of a TCP flow, whose increasing factor is 1, then

$$\begin{aligned} &\overline{W} = \beta \times \overline{W_{TCP}} \\ \Rightarrow &\sqrt{\alpha} \times \frac{3}{4}\sqrt{\frac{8}{3p}} = \beta \times \frac{3}{4}\sqrt{\frac{8}{3p}} \\ \Rightarrow &\alpha = \beta^2. \end{aligned} \quad (7)$$

Equation 7 tells us that, for the throughput of a DMS flow to be β times of a conformant TCP flow, we need to set its increasing factor α to β^2. We tested this observation in the following simulation (Simulation 1) using ns-2.28.

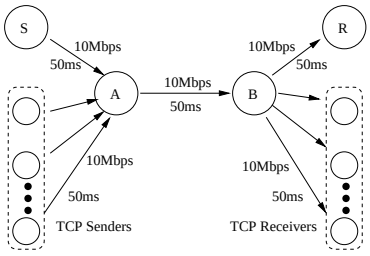


Fig. 6. Topology of Simulation 1.

The topology of the simulation is shown in Fig. 6. The bottleneck between nodes A and B has a bandwidth of 10Mbps and a delay of 50ms. Node A is a RED gateway using ns-2.28 default setting[2]. Fifty TCP Reno flows pass through the bottleneck and produce congestion. A DMS flow is sent from S to R. Its increasing factor α changes based on the value of β according to Equation 7. We increased β from 0.1 to 1.4 (note that in DMSCC, we are interested only in $\beta \leq 1$) and observed the ratio of the throughput of the DMS flow to the average throughput of TCP flows. For each value of β, we repeated the simulation 20 times and computed the average ratio.

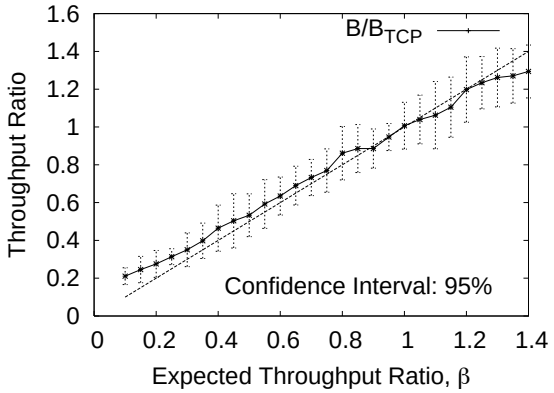


Fig. 7. Simulation 1: Throughput Ratio as β Changes.

The result is plotted in Fig. 7. The x-axis, β, is the expected throughput ratio (β). The y-axis is the ratio observed when setting α to β^2. Fig. 7 shows that as β changes, the actual throughput ratio is close to β when β ranges from 0.2 to 1.0. The result shows the effectiveness of Equation 7.

Mismatch between the actual throughput ratio and β is observed in Fig. 7 for small β and large β. This mismatch is due to bursty packet losses in the simulation, which violates the assumption that packet losses are evenly distributed. During the bursty loss period, the congestion window becomes small.

[2]queue length = 50, min_thresh = 5, max_thresh = 15, gentle-enabled, and mark_p = 0.1

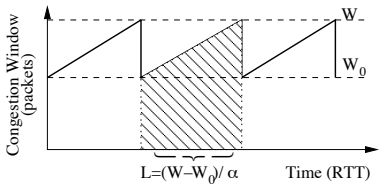


Fig. 8. Effects of Minimum Congestion Windows.

When the halved congestion window is less than the minimum window, the latter dominates the throughput and skews the throughput ratio from β. We elaborate on this below.

To study the effects of minimum congestion window over throughput, we make a similar deduction as in Fig. 5. Let W_0 be the minimum window. When the loss rate is high, congestion window is rarely greater than $2 \times W_0$, since it encounters packet losses frequently. On every packet loss, as $W_0 > W/2$, the congestion window is reduced to W_0. Fig. 8 shows the evolution of window size in this situation. We can derive W as:

$$\begin{cases} L = \frac{W - W_0}{\alpha} \\ S = WL - \frac{(W-W_0)L}{2} \\ S = \frac{1}{p} \end{cases}$$
$$\Rightarrow W = \sqrt{\frac{2\alpha}{p} + W_0^2}.$$

In a TCP flow, α equals to 1, hence the throughput ratio can be represented as:

$$\frac{B}{B_{TCP}} = \frac{W}{W_{TCP}} = \sqrt{\frac{2\alpha + pW_0^2}{2 + pW_0^2}}$$

We further divide this value by β and denote the resulting value as R. Ideally, R equals to 1 (i.e., throughput ratio equals β).

$$R = \frac{B/B_{TCP}}{\beta} = \sqrt{\frac{2 + pW_0^2/\alpha}{2 + pW_0^2}} \quad (8)$$
$$= \sqrt{1 + \frac{(1-\alpha)W_0^2}{2\alpha/p + \alpha W_0^2}} \quad (9)$$

Equation 8 tells us that when the size of congestion window is dominated by the minimum window size, smaller α (therefore smaller β) increases R, i.e., the throughput of the DMS flow becomes larger than expected. Similarly, larger α (and β) decreases R, and the throughput of DMS flow is less than expected. This equation explains the discrepancy between B/B_{TCP} curve and the expected line in Figure 7.

Equation 9 tells us that, for a DMS flow with a given α ($\alpha < 1$), if loss rate p increases, R ($R > 1$) will increase, i.e., the actual throughput will be larger than the expected value, and the difference will be enlarged.

Although mismatch of the throughput ratio exists and is found to be inevitable in lossy environment, the method still manage to control the throughput of a flow to reasonable level

of accuracy. Note that when the channel is highly lossy, media streaming is generally not usable anyway. Thus the larger mismatch in throughput ratio in this case is less of a concern in our context.

We have described our method to control the throughput of a DMS flow on a bottleneck. To apply it in DMSCC, we need to find out where the bottlenecks are, so that we can regulate the throughput of DMS flows on these bottlenecks. We describe our approach to locate the congested bottlenecks in the next section.

V. Congestion Location

An ideal solution to locate a congestion should work as follow: (i) when a congestion causes a packet loss on a DMS flow, the solution should be able to tell which link is congested, so that DMS flows on the affected link can be regulated, (ii) when the congestion subsides, the solution should sense it, so that the regulation on the DMS flows previously imposed can be lifted. Such ideal solution is difficult to achieve in a tree topology: (i) there may be multiple, simultaneous congestion on different links in the tree, and (ii) the same flow might experience congestion on different links.

Rubenstein et al. [15] partially solved this problem for the case with one shared bottleneck. Based on the observation that a shared congestion produces highly correlated one-way delay on flows, they compare the cross-correlation of two flows and the auto-correlation of one of them. The shared bottleneck link is identified as one where the cross-correlation is larger. For details of Rubenstein's technique, please refer to the original paper [15].

Rubenstein's method works well when each flow experiences one congestion. To use the same correlation test when a flow passes through multiple congested links is difficult. On a shared bottleneck, the delay of the flows might contain too much noise induced by other congested links. Solving the congestion location problem completely in the distributed media streaming scenario remains a difficult and open problem. In this paper, we extend Rubenstein's method to identify multiple bottlenecks in the case where the delay values on the shared bottleneck has limited interference from other congested links.

We use *CorrTest(i,j)* to denote the correlation test of Rubenstein applied on flow i and flow j. When *CorrTest(i,j)* returns 1, the two flows share a bottleneck; when it returns 0, no shared bottleneck is detected. We apply the test over a window of one-way delays recorded using probe packets sent together with flows i and j. We use probes to maintain certain minimum sampling frequency. Without probes, flows i and j may not send any packet for a long period due to congestion windows. Probes are tiny packets that consume negligible bandwidth (0.8KBps in our simulation). In the rest of this section, we explore congestion location step by step, and then propose our method.

First, consider a simple case where only one link is congested. In this case, we can directly apply the correlation test. The method is listed as Algorithm 1, The method is called whenever a packet loss is detected on flow f_i. It applies correlation test on (the probes of) f_i and other DMS flows and adds DMS flows that are correlated with f_i into a set C_f. The least dominant link that contains the set of flows in C_f is returned as the shared bottleneck.

1 OneBottleneck(f_i)

INPUT: f_i {the flow whose packet is lost}
Let F be the set of all flows and L be the set of all links;
$C_f \leftarrow \{f_j | CorrTest(i,j) = 1, \forall f_j \in F\}$;
$C_l \leftarrow \{l | l \supseteq C_f, l \in L\}$;
OUTPUT: Link $l \in C_l$ such that $l_k \supseteq l, \forall l_k \in C_l$;

The situation is more complex when two links are congested simultaneously. For instance, in Fig 1, when two bottlenecks S_0-A and A-B coexist, one-way delay of f_0 is worsen by both congestion, but one-way delay of f_1 is only affected by congestion at A-B. When a packet is lost, $CorrTest(0,1)$ can return either 1 or 0, depending on which bottleneck dominates value of delay during that sampling period. When the queuing delay on one bottleneck is temporally reduced by congestion control of background traffic or packet dropping, the queuing delay on the other bottleneck can remain high and continue to dominate the end-to-end delay. So, $CorrTest(0,1)$ may return 0 even when A-B is congested due to domination of bottleneck S_0-A on the one-way delay of f_0, making it less correlated with f_1.

Whereas a $CorrTest(0,1)$ value of 0 does not necessary imply no shared bottleneck, a value of 1, however, does confirm the existence of shared congestion on f_0 and f_1. Our observation is that, if the congestion is shared, $CorrTest(0,1)$ may return 1 from time to time after every packet loss. Based on this observation, we use a history-based method to update the set of current bottlenecks. We denote C as a set of current congested links and H as a FIFO queue of previously detected congested links due to the most recent h packet loss. When a packet loss is detected on f_i, H is updated as in Algorithm 2.

2 OnPacketLoss (f_i)

INPUT: f_i, H, h
$l \leftarrow OneBottleneck(f_i)$;
if $|H| = h$ **then**
  $dequeue(H)$; {phase out old bottleneck}
end if
$enqueue(H, l)$; {phase in new bottleneck}
$C \leftarrow \{l | l \text{ in } H\}$;
OUTPUT: C, H

We can view H as a history of bottleneck detection record. On every packet loss, the oldest record in H is phased out. If no other record in H refers to the same bottleneck, the bottleneck is removed from the output. In other words, if a link is not identified as a bottleneck during the most recent h packet loss event, the congestion on the link is likely to have subsided. The length of the queue, h, should be long enough

so that H is able to buffer all current congested links. If h is too small, H may phase out existing bottlenecks and update C incorrectly. On the other hand, h needs not be too large, as the probability of having many simultaneous bottlenecks is small. Our experiments on a four-sender session show that value of h beyond 8 produces little improvement in accuracy of C, so we use $h = 8$ in our protocol.

After C is updated by Algorithm 2, C contains the set of current bottlenecks. For instance, in the previous example with simultaneous congestion on link S_0-A and A-B, Algorithm 2 may return C =$\{S_0$-A, A-$B\}$ or $\{S_0$-A, A-B, S_1-$A\}$. In the second set, S_1-A is a false detection. To understand this, imagine that the bottleneck A-B causes a packet loss on f_1. When performing $CorrTest(1, 0)$, the result can be 0 as we have analyzed. Therefore, Algorithm 1 returns S_1-A as a bottleneck. But, fortunately, the false detection does not affect the correctness of DMSCC, as we shall see in the next section.

VI. Congestion Control

A. Updating the Increasing Factors

After identifying the set of bottlenecks, the next step is to adjust the increasing factors of the DMS flows on the bottlenecks so that their combined throughput is TCP-friendly. Given C, the set of current bottlenecks, Algorithm 3 constructs another set C' containing the set of bottlenecks that are not dominated by any other bottlenecks in C. For each of the bottlenecks in C', the algorithm sets the increasing factor of the DMS flows that pass through it to $1/n^2$ according to Equation 7, where n is the number of DMS flows going through a bottleneck.

3 UpdateAlpha (C)

INPUT: C
$C' \leftarrow \{l|\ \nexists l_i \in C : l_i \supset\ l, l \in C\}$;
for all $l \in C'$ **do**
  $n \leftarrow |\{f_i\}|, f_i \in l$; {number of DMS flows}
  $\alpha_i \leftarrow 1/n^2, \forall f_i : f_i \in l$; {increasing factor}
end for

To understand the reason why DMS flows are adjusted according to the dominant bottlenecks, let us consider the previous example of simultaneous congestion on S_0-A and A-B in Fig. 1. Suppose that, after a packet loss, Algorithm 2 returns C =$\{S_0$-A, A-B, S_1-$A\}$. Link A-B dominates the other two links. Congestion on A-B requires the aggregate of f_0 and f_1 to be TCP-friendly. According to Equation 7, α_0 and α_1 should be set to $1/4$. Congestion on the other two links requires each of f_0 and f_1 to be TCP-friendly and thus both α_0 and α_1 should be set to 1. Setting the increasing factor to 1, however, makes the flow aggregate on A-B unfriendly. Considering the goal of DMSCC (Equation 3), α_i should be set conservatively to $1/4$. In short, the dominant bottleneck restricts the aggresiveness of the DMS flows, and therefore the increasing factor should be set according to the dominant links. This property also allows Algorithm 2 to return false bottlenecks (e.g. S_1-A) that are dominated by the shared bottleneck (e.g., A-B). Such false bottlenecks do not affect the correctness of DMSCC.

B. Bottleneck Recovery

The above mentioned algorithms run whenever a packet loss is detected. When congestion subsides and there is no more packet loss, we need to reset α_i to 1 so that the network bandwidth can be fully utilized. Having no packet loss to trigger the reset of α_i, we adopt a timer-based method. A timer is refreshed when packet loss is detected. If no packet loss is detected within t seconds, the increasing factors of all DMS flows are reset to 1. This method ensures that after congestion disappears, in at most t seconds, αs are reset to allow DMS flow to fully utilize available bandwidth. But if the bottleneck is still there when timer expires, resetting all α will make the flow aggregate unfriendly to TCP. To prevent such over aggresiveness of DMS flows, we (i) set t conservatively long (15 seconds in our simulation), and (ii) retain the value of C and H while resetting α_i. The latter helps Algorithm 2 to set α_i back to the right value immediately if packet loss reappears.

VII. Simulation and Discussion

We constructed Simulation 2 in ns-2.28 to validate our design. Fig. 9 shows a topology with four senders S_0, S_1, S_2, and S_3, and one receiver R. DMS flows converge on the way to R in the order of S_0, S_1, S_2, and S_3. Besides f_i, the senders also send CBR probes to the receiver using UDP, at 40 bytes per packet, 20 packets per seconds. The sample length for one-way delay records is 20 (one second in length) for correlation computation; according to Rubenstein et al. [15] this length gives nearly 90% of accuracy in correlation test. All links are configured with bandwidth of 5Mbps, delay of 20ms, and default RED setting in ns-2.28. Background traffic may congest link l_0, l_1, l_2 or l_3 to produce bottleneck. The background traffic consists of 20 TCP Reno flows on every bottleneck. The RTTs of background TCP flows are set to 120ms.

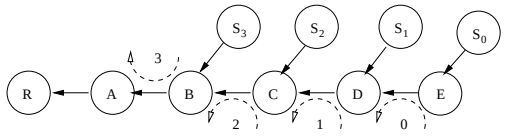


Fig. 9. Topology of Simulation 2.

The simulation aims to show that DMSCC leads to task-level TCP friendliness, achieving our goal stated at the end of Section II. When background traffic produces congestion on a link, the throughput of f_i and the RTT_i are measured to calculate B×RTT of the flow aggregate on the link. The average B×RTT of the TCP background flows is also calculated. If B×RTT of the flow aggregate is less than or equal to the average of a TCP flow, then task-level TCP-friendliness (Equation 3) is achieved. Fig. 10 shows B×RTT of the TCP flows (average) and the flow aggregate; each subgraph corresponds to one link.

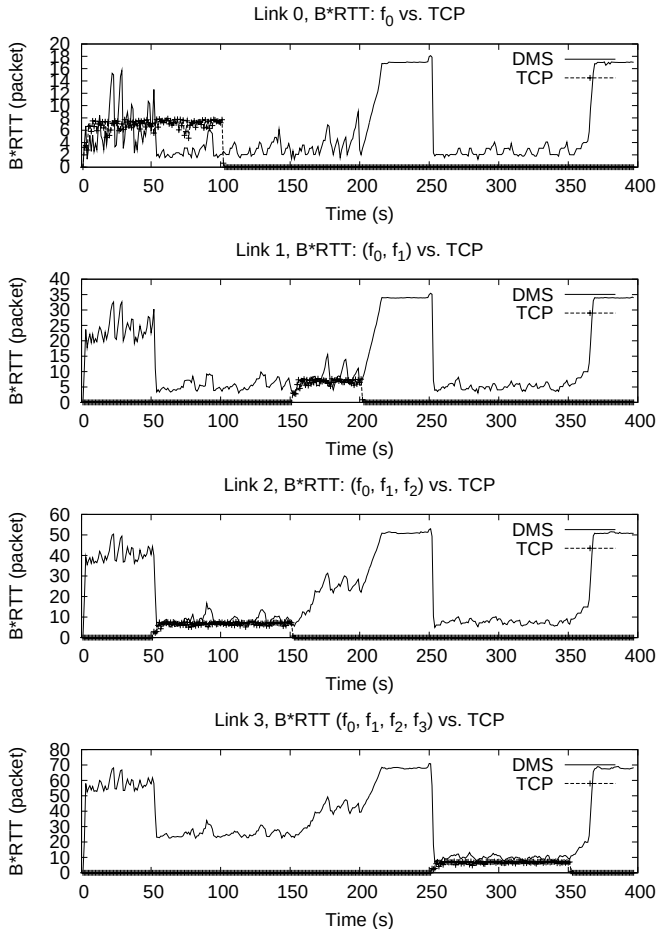


Fig. 10. B×RTT of Aggregate DMS flows and TCP Flow on Each Links.

To show the ability of DMSCC to identify the dominant bottleneck, we generate background traffic such that Link 0 is congested during time 0 - 100s and Link 2 is congested during time 50 - 150s. The first subgraph in Fig. 10 shows that during time 0 - 50s, B×RTT of the flow aggregate on Link 0 roughly equals to a TCP flow; and the third subgraph shows that during time 100 - 150s, B×RTT of the flow aggregate on Link 2 is also similar to a TCP flow. This confirms that task-level TCP-friendliness is achieved when only one congestion exists in the topology. During time 50 - 100s, when both Link 0 and Link 2 are congested, we find that B×RTT of the flow aggregate on Link 2 equals to a TCP flow, and that B×RTT of the flow aggregate on Link 0 is less than a TCP flow. This result demonstrates that DMSCC is able to identify that Link 2 dominates Link 0, and therefore sets the increasing factor accordingly to achieve task-level TCP-friendliness in the case of simultaneous congestion.

To show that DMSCC is able to utilize bandwidth fully when congestion disappears, Link 1 and Link 3 are congested during time 150 - 200s and 250 - 350s respectively. The streaming session completes at time 400s. In the second subgraph (Link 1), at 200s, TCP flows disappear. The value of B×RTT of the flow aggregate increases quickly to the maximum playback rate of the media, demonstrating that the available bandwidth is fully utilized when there is no congestion. We can also see in the last subgraph (Link 3), starting from time 350s, B×RTT of the flow aggregate increases slowly at first and increases faster later. The small slope is due to the small increasing factor ($\alpha_i = 1/16$), which is determined by the congestion on Link 3. But when packet loss is not seen for a period of 15 sec (value of t in this simulation), it is likely that the congestion has disappeared. DMSCC therefore sets α_i to 1, allowing throughput to increase quickly, achieving better bandwidth utilization.

A. The sensitivity of h

When presenting Algorithm 2, we mentioned that the length of congestion history h controls the update frequency of C, and, therefore, affects the accuracy of α. The value of h is empirically set to 8 in the simulation. We changed h from 1 to 20 and ran the above simulation 50 times each. On every packet loss, C (the detected bottlenecks) is compared with the actual bottlenecks, and the detected dominant bottlenecks (which affects the value of α) are compared with the real dominant bottlenecks. The accuracy is defined as the number of correctly detected bottlenecks over the number of bottlenecks. Fig. 11 shows the average accuracy of C and dominant bottlenecks, when h changes. The accuracy of dominant bottleneck is higher than that of C, indicating that even if false detection on bottlenecks exists, the dominant bottleneck could still be correctly detected. After the value of h exceeds 8, both curves increase slower: larger h contributes less to the accuracy of α.

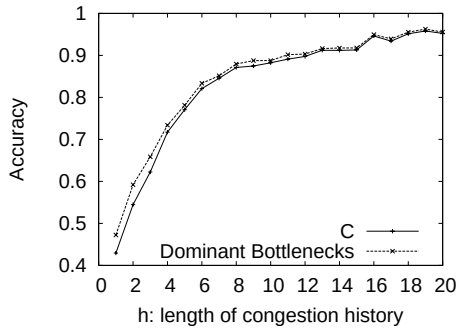


Fig. 11. Effects of h on Accuracy of C and Dominant Bottlenecks.

VIII. Related Work

Congestion control is a well studied problem in unicast [6]. We believe, however, that flow-level TCP-friendliness is not sufficient for multi-flow applications (Section II). Congestion control has also been studied in the context of multicast. Even though the topology of distributed media streaming resembles that of multicast, the research focus in congestion control is different: we focus on achieving task-level TCP-friendliness, while research on multicast congestion control [16] focuses on scalability as well as flow-level TCP-friendliness. Congestion control in multi-path streaming (e.g., [17]) also shares a similar

topology with distributed media streaming, but their goal is to achieve flow-level friendliness.

Our work is more related to the study on aggregate congestion control. Aggregate congestion control pursues the fairness of a group of flows. Congestion Manager (CM) [7] uses one AIMD congestion window adjustment loop for the flow aggregate to achieve a fair combined throughput. CP [9] adopts equation-based rate adaptation [13] with packets subsampling to achieve fair bandwidth share. MPAT [8] keeps multiple bandwidth estimation loops and allows the application to allocate bandwidth to different flows while ensuring that the total throughput is fair. Hacker et al. study parallel TCP flows [10] and mimic TCP flows with longer RTT, so that flows in the aggregate consume less bandwidth than a TCP flow, making the aggregate TCP-friendly.

Aggregate congestion control is relatively new, and researchers still have different views on the definition of TCP-friendliness of flow aggregate. Some believe that the flow aggregate should be fair to one TCP flow, so that software that uses concurrent downloading do not gain advantage by establishing multiple flows, and therefore does not encourage abuse using multiple flows [10] Others allow an aggregate of n flows to have equal bandwidth share to that of n TCP flows [8], [9]. Their argument is that, since traditional TCP-friendliness is between flows, granting n flows a throughput equivalent to n TCP flows does not breach TCP-friendliness.

Regardless of the differences, these studies apply congestion control on a fixed set of flows. In distributed media streaming, congestion control needs to be applied on different sets of flows on different links. A new congestion control method is therefore required.

IX. Conclusion

In this paper, we introduce the problem of congestion control in DMS system. It differs from previous congestion control problems as it involves multiple flows traversing through different paths. A better definition of TCP-friendliness is needed to further explore the problem. We therefore introduce the notion of task-level TCP-friendliness in this paper. We then formulate a criterion for task-level fairness in the context of distributed media streaming. We divide the problem of congestion control in distributed media streaming into two subproblems. The first is how to locate congestion in a reverse tree topology. The second is how to control the throughput of a DMS flow using AIMD loop such that the combined throughput on the bottleneck is TCP-friendly.

This paper is the first one to address the problem of congestion control in distributed media streaming. The concept of task-level TCP-friendliness gives a different perspective to the meaning of TCP-friendliness, and it is usable in other scenario such as peer-to-peer file sharing. Our method to control the aggregate throughput of DMS flows might be useful in other context as well, including controlling the throughput of parallel TCP connections.

DMSCC has several limitations. Our throughput control algorithm is based on Mathis equation, and therefore does not work accurately in all network conditions (e.g., when loss is frequent and bursty). Our congestion location algorithm relies on Rubenstein's method. Identifying location of congestion in multiple congestions scenario with high delay inteference remains a challenging problem. Our future work aims to address these limitations.

X. Acknowledgments

We would like to thank Xiuchao Wu, Wei Cheng, Eric Liu and the anonymous reviewers for their precious comments.

References

[1] T. Nguyen and A. Zahkor, "Distributed Video Streaming over the Internet," in *Proceedings of SPIE Conference on Multimedia Computing and Networking*, San Jose, California, USA, January 2002.

[2] J. Apostolopoulos, T. Wong, W. Tan, and S. Wee, "On Multiple Description Streaming with Content Delivery Networks," in *Proceedings of IEEE INFOCOM'02*, New York, NY, June 2002.

[3] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "PROMISE: Peer-to-Peer Media Stream-ing Using Collectcast," in *Proceedings of ACM International Conference on Multimedia (MM'03)*, Berkeley, California, USA, November 2003.

[4] R. Rajaie and A. Ortega, "PALS: Peer-to-Peer Adaptive Layered Streaming," in *Proceedings of the 13th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Monterey, California, USA, June 2003.

[5] L. Ma and W. T. Ooi, "Retransmission in Distributed Media Streaming," in *Proceedings of the 15th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, Stevenson, Washington, USA, June 2005.

[6] J. Widmer, R. Denda, and M. Mauve, "A Survey on TCP-Friendly Congestion Control," in *IEEE Network Magazine, Special Issue on Control of Best Effort Traffic*, vol. 15, no. 3, May 2001.

[7] H. Balakrishnan, H. Rahul, and S. Seshan, "An Integrated Congestion Management Architecture for Internet Hosts," in *ACM SIGCOMM*, Cambridge, MA, September 1999.

[8] M. Singh, P. Pradhan, and P. Francis, "MPAT: Aggregate TCP Congestion Management as a Building Block for Internet QoS," in *Proceedings of IEEE International Conference on Network Protocols (ICNP'04)*, Berlin, Germeny, October 2004.

[9] D. E. Ott, T. Sparks, and K. Mayer-Patel, "Aggregate Congestion Control for Distributed Multimedia Applications," in *Proceedings of IEEE INFOCOM'04*, Hong Kong, China, March 2004.

[10] T. J. Hacker, B. D. Noble, and B. D. Athey, "Improving Throughput and Maintaining Fairness Using Parallel TCP," in *Proceedings of IEEE INFOCOM'04*, Hong Kong, China, March 2004.

[11] S. Floyd and V. Jacobson, "Traffic Phase Effects in Packet-Switched Gateways," *ACM SIGCOMM Computer Communication Review*, vol. 21, no. 2, pp. 26–42, April 1991.

[12] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Sanadidi, and M. Roccetti, "TCP-Libra: Exploring RTT Fairness for TCP," *Accepted in IEEE Journal on Selected Areas in Communications, Special Issue on Non Linear Optimization of Communication Systems*, 2006.

[13] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-Based Congestion Control for Unicast Applications," in *ACM SIGCOMM'00*, Stockholm, Sweden, August 2000.

[14] M. Mathis, J. Semke, and J. Mahdavi, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," in *Computer Communication Review*, vol. 27, July 1997.

[15] D. Rubenstein, J. Kurose, and D. Towsley, "Detecting Shared Congestion of Flows Via End-to-end Measurement," in *IEEE/ACM Transactions on Networking*, vol. 10, June 2002.

[16] A. Matrawy and I. Lambadaris, "A Survey of Congestion Control Schemes for Multicast Video Applications," in *IEEE Communications Surveys & Tutorials*, vol. 5, no. 2, Fourth Quater, 2003.

[17] H. Han, S. Shakkottai, C. Hollot, R. Srikant, and D. Towsley, "Overlay TCP for Multi-Path Routing and Congestion Control," in *ENS-INRIA ARC-TCP Workshop*, Paris, France, November 2003.