On Integration of Congestion Control with Internet Streaming Applications

Reza Rejaie
Computer Science Department
University of Oregon
reza@cs.uoregon.edu

Abstract— **During recent years, the importance of integrating congestion control (CC) into streaming application has been frequently justified and several CC mechanisms for streaming applications have been proposed. However, many of existing streaming applications either do not incorporate any CC mechanism at all or do not integrate an existing CC mechanism properly, or implement their own untested CC mechanism. We believe that lack of a well-defined and flexible interface for CC mechanism has been a main obstacle in proper integration of CC into streaming application.**

In this paper, first we discuss various design choices, challenges and tradeoffs in devising a flexible interface for a CC module. This discussion leads us to a set of requirements for such an interface. Then we present *cclib*, an application-level library for congestion control that provides a flexible interface. Design and implementation of *cclib* is presented. We use *cclib* to illustrate proper integration of CC into streaming applications and address various practical issues.

Keywords— **Congestion Control, Internet Streaming Applications, Integration.**

I. Introduction

Shared nature of Internet resources requires all applications to behave in a network-friendly fashion and perform congestion control (CC). Performing CC ensures network stability and improve fairness among coexisting flows [1] but could result in unpredictable and potentially wide variations in transmission rate.

The need for integration of CC into Internet streaming applications have inspired design and evaluation of several new CC mechanisms (*e.g.*, [2], [3], [4], [5]). These new rate-based CC mechanisms often regulate transmission rate by adjusting inter-packet-gap (IPG) based on observed round-trip time (RTT) and loss rate for a given connection. Therefore, in design and evaluation of these new CC mechanism, the ideal scenario is implicitly assumed where packet payload is always ready and evenly spaced packets are sent once every IPG, *i.e.*, the CC mechanism fully controls packet transmission schedule. In practice, however, to integrate a CC mechanism into streaming applications, the CC mechanism is usually implemented as a module that periodically determines available bandwidth and reports it to the application. Then the application is expected to match its transmission rate with the reported available bandwidth. While this framework seems straightforward, to properly integrate CC module into streaming applications, the following key issues should be addressed:

1) How should the available bandwidth be defined by the CC module and how often it should be reported to the application? Available bandwidth is not a clearly-defined term. There should be a close relationship between definition of available bandwidth and frequency of reporting available bandwidth to the application. When the available bandwidth is reported to the application once every interval τ (*e.g.*, 5*RTT), it usually presents the average bandwidth over the next interval. Therefore, the CC module should associate available bandwidth with a timescale (τ) over which it is averaged, and also specify frequency of reporting bandwidth. Obviously, if the available bandwidth is not defined properly or it is not reported to the application frequently, the resulting flow will not be well-behaved.

2) How closely should the application's transmission rate over short timescale match with available bandwidth? i.e., *How bursty can the application's transmission rate become?*. The application can control micro-level packet departure time within the interval τ as long as its average transmission rate matches average available bandwidth. However, changing packet transmission schedule could affect performance of underlying CC mechanism. For example, if the application is allowed to send 10 packets during the next interval τ, it can send either 10 evenly spaced packets or a burst of 10 back-to-back packets. In both of these transmission schedules, the average transmission rate matches the average available bandwidth over interval τ. However, the bursty schedule could result in a significantly higher transmission rate over shorter timescales. The more the application changes packet transmission schedule from an evenly-spaced to a bursty packet transmission schedule, the more the observed loss rate and RTT of the connection could be affected. This in turn

changes behavior of the underlying CC module from the ideal scenario where the CC module is in full control of packet departure schedule.

Streaming applications often tend to send bursty data stream Therefore, it is important to accommodate small bursts over shorter timescale as long as this bursty transmission schedule does not significantly change behavior of the underlying CC mechanism.

3) Should the CC module operate in a passive or active mode? The CC module could passively monitor connection behavior and report available bandwidth to the application. This passive rate control scheme solely rely on the application to properly adapt its transmission rate. Therefore, a misbehaved application (that does not integrate the CC module properly, or does not properly match its transmission rate with available bandwidth, or is simply buggy) could easily result in a misbehaved flow. To prevent this, the CC module can not only report available bandwidth, but also can actively control packet departure schedule. For example, when application's transmission rate exceeds available bandwidth, the CC module in active mode can either buffer excess data or signal the application to slow down. This active CC mechanism could protect the network from misbehaved flows. However if the CC module does not provide a flexible interface to exchange bandwidth information with the application effectively, application developers will not have any incentive to integrate a CC module in active mode within their adaptive streaming applications.

4) Should the CC machinery be implemented within the kernel or at the application level? Clearly, each case has certain benefits and drawbacks. The location of the CC module also affects its interactions with the application. For example, an in-kernel CC module can signal the application to send a packet at proper time. However, an application-level CC module may need to be invoked by the application to examine whether next packet departure time has arrived.

Finally, there are several functions (*e.g.*, loss detection or RTT measurement) that are required by many adaptive streaming applications as well as CC mechanisms. To avoid duplicating these functionalities, a CC module can implement these functions and provide a proper interface for the applications to access updated information.

The above discussion illustrates that effective integration of CC mechanism into streaming application requires: 1) a well-defined and flexible interface for the CC module, and 2) a clear methodology for integrating CC module into the application. Defining such an interface for the CC module can decouple design of the CC mechanism from design of streaming applications to a large extent.

Despite extensive work on design of new CC mechanisms for streaming applications, various issues related to integration of CC into streaming applications have not received sufficient attention. The congestion manager (CM) architecture [6] appears to be the only work that presents an interface between congestion manager (an entity that performs congestion) and streaming applications. But as we discuss in section IV their proposed interface is not appropriate for adaptive streaming applications. Due to the lack of a well-defined and flexible interface for the CC module, application designers were not able to properly integrate new CC mechanisms (*e.g.*, [3], [2], [5]) that have been carefully designed and extensively evaluated, into their streaming applications. Instead, many application designers either do not incorporate any CC mechanism into their streaming applications, or even implement their own CC mechanism without sufficient evaluations. Design of a network-friendly CC mechanism is not a trivial task and requires a good understanding of network dynamics and extensive evaluation under various network conditions. Therefore, to leverage previously designed CC mechanisms, it is important to define a good interface for CC module.

In this paper, we study the above key design issues for integration of CC into streaming applications and discuss implications of various design choices. Our discussion leads us to a set of required features for an interface between he CC module and streaming applications. We then present a prototyped CC module called *cclib*. *cclib* is an application-level library with a flexible interface that supports all the required features. We describe *cclib*'s interface along with a simple methodology for its integration into streaming applications. Although the library implements a specific CC algorithm, its interface and integration methodology is generic and should be applicable to other CC mechanisms with minor or no modification. Finally, we address several practical considerations in design of such a CC module and describe our solutions.

Note that our goal in this paper is to explore various design issues to design an interface for a CC module, and present a CC module with such a flexible interface. However, we do not intend to find the optimal setting for such an interface. For example, we show how our proposed interface allows the application to accommodate a given burst size, but we do not examine what the optimal burst size should be.

A. Target Applications

Our target applications in this paper are adaptive streaming applications, ranging from playback streaming to live but non-interactive multimedia (*i.e.*, lecture mode) ses-

sions. This class of streaming applications perform quality adaptation by adjusting quality of delivered stream to match with available bandwidth. To effectively perform quality adaptation, the application requires information about available bandwidth over short timescales. We assume that the application fully utilizes the available bandwidth that is reported by the CC module (instead of sending only at certain rate as was assumed by the CM interface [7].

The rest of this paper is organized as follows: in section II, we explore various issues for designing a flexible interface for a CC module and derive a set of required feature for such an interface. Section III presents $cclib$, a prototyped application-level CC module that supports all the required features. We describe how $cclib$ implements these features. Section IV reviews related work on this topic. We conclude the paper in section V and presents our future directions.

II. Integrating CC into Streaming Applications

The primary function of CC mechanism is to estimate available bandwidth as the network condition changes. The CC module maintains state of a connection by monitoring packet departure and ACK arrival times to determine individual packet losses (or loss rate) as well as RTT samples. Each rate-based CC mechanism deploys an increase/decrease algorithm (*e.g.*, AIMD, TCP-equation) to adjust inter-packet-gap (IPG) to regulate transmission rate based on observed connection loss rate and RTT.

When a CC module is integrated into an application, it determines and reports available bandwidth (BW_{ave}) to the application. Then the application should match its transmission rate (BW_{app}) with available bandwidth as shown in Figure 1. In this section, we explore key design issues in such a framework, present a set of preferred design choices and justify our choices.

A. Reporting Bandwidth Information

If the application is allowed to send at the rate of BW_{ave} for T seconds, it can send n packet of size b bytes where $n = \frac{T*BW_{ave}}{b}$. Therefore, for a given interval T and

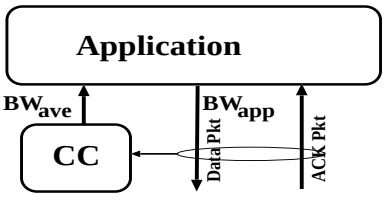

Fig. 1. Interactions between the CC module and application

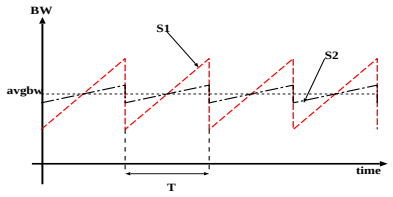

Fig. 2. Average bandwidth over different timescale

packet size b, available bandwidth can be presented as either actual bandwidth (BW_{ave} in terms of byte per second) or number of packets to be sent (*i.e.*, n). Throughout this paper, we present available bandwidth in one of these two forms.

When the CC module reports BW_{ave} to the application once every period τ, the most useful definition of available bandwidth to report is average bandwidth over each period. If the available bandwidth is reported less often (*i.e.*, long τ), the application needs to adapt to potentially variable bandwidth less frequently. Furthermore, the application might be able to plan packet transmission schedule for each period more efficiently. For example, the application requires to be less adaptive if it determines next 100 packets (rather than next 5 packets) to be sent during the next 10 RTTs at once. However, reporting available bandwidth less frequently has the following two major drawbacks. First, by reporting average available bandwidth during each period, the CC module has to ignore details of bandwidth variations that are important for quality adaptation. To illustrate this point, Figure 2 shows two different patterns of variation in available bandwidth that have the same average bandwidth. If BW_{ave} is reported once every T, both patterns have the same average bandwidth whereas reporting the bandwidth more frequently reveals differences among these patterns. Quality adaptive streaming applications require details information about variations of available bandwidth to adapt effectively. For example, pattern S1 exhibits a wider variations in bandwidth, therefore a layered quality adaptation mechanism (*e.g.*, [8]) requires more total buffered data at the client, and different buffer distributions among active layers. Second, when reporting only an average bandwidth from the CC module, the application can modify packet departure schedule during each period τ in any arbitrary way (*e.g.*, sending evenly-spaced packets or burst of packets). Obviously, the longer the period tau, the more freedom the application has to arbitrarily change packet departure schedule and become bursty. As we discuss in subsection II-B, this is not a desirable property and could easily lead to misbehaved flows.

The unpredictable and potentially wide variations in congestion controlled bandwidth motivate the need for fre-

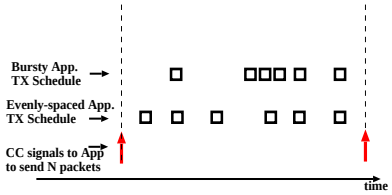

Fig. 3. per-interval bandwidth reporting

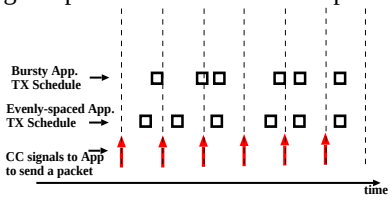

Fig. 4. Per-packet bandwidth reporting

quently reporting $BW_{ave}$to the application. Reporting BW_{ave}once every RTT seems to be the lowest acceptable frequency for adaptive applications since many CC mechanisms do not increase their bandwidth faster than once per RTT. Besides periodic reporting, the CC module should still inform the application when BW_{ave}suddenly reduces due to a congestion. To provide detailed variations of available bandwidth to the application, the CC module can signal the application on each departure time *i.e.*, reporting per-packet or instantaneous available bandwidth as $\frac{PacketSize}{IPG(t)}$. In this case, the CC module essentially presents the ideal evenly-spaced transmission schedule to the application by signaling the application on each packet departure time. However, the application may (or may not) strictly follow the reported transmission schedule within shorter timescales Note that in both bandwidth reporting schemes, the application has the freedom to arbitrarily control packet departure schedule. However, in the later case, the application 1) has more info about bandwidth variations over short timescales, and 2) is able to follow the ideal schedule if it is desired. Figure 3 and 4 illustrate application transmission schedules with a per-interval and per-packet bandwidth reporting schemes respectively. It clearly shows that per-packet bandwidth reporting provides more details, and thus better hints, for the application to adjust its transmission rate over short timescales.

We believe that per-packet bandwidth reporting is the most appropriate scheme for adaptive streaming applications because it provides fine-grained details about available bandwidth. The key question in per-packet bandwidth reporting is how "closely" the application follows the reported transmission schedule. We address this issues in the next subsection.

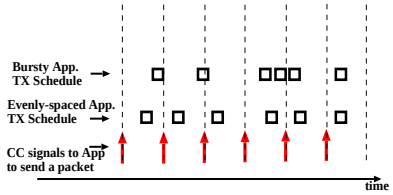

Fig. 5. Allowing bursty streams

B. Supporting Bursty Transmission Schedule

Streaming applications often tend to send bursty data stream. Various factors could result in a bursty data stream. For example, efficient encodings often generate bursty data stream, or a non-realtime operating system could easily fall behind a transmission schedule and sends a burst of packets at once. Even when the application limits its average transmission rate to the reported bandwidth over longer timescale, variation of application's transmission rate over shorter timescales could affect behavior of underlying CC module. The burstier the transmission schedule becomes, the more likely the resulting flow observes different loss rate and RTT samples which in turns affects the behavior of the underlying CC module. Therefore, it is important to allow streaming applications to send bursts of packets over shorter timescales as long as this burstiness does not significantly affect observed behavior of the underlying CC module.

When the CC module reports available bandwidth at a per-packet granularity, it allows the application to properly match its transmission rate even at shorter timescales while limiting its level of burstiness. Figure 5 illustrates transmission schedule of a well-behaved applications that uses per-packet available bandwidth reports but control its level of burstiness to 3 packets. This figure clearly shows how the application can micro-manage (mainly postpone) departure time of individual packets to accommodate a potential bursty transmission. In practice the CC module should keep track of maximum burst size that the application is allowed to send at any point of time. This would simplify application design since it does not require to maintain timing information for individual packets.

C. Up-call vs Probe Approach for Bandwidth Reporting

There are two ways to exchange bandwidth information between application and CC module. Application can probe the CC module to obtain available bandwidth, we call this probe approach. This approach is similar to the request/callback interface in the CM architecture [6]. The application request the CC module for permission to send a packet, then the CC module grant the permission, before the application sends a packet.

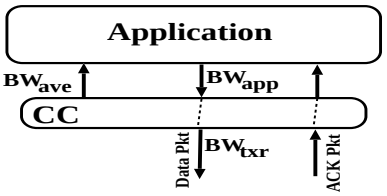

Fig. 6. The Confi guration for the CC module in active mode

Alternatively, the CC module can pro-actively report available bandwidth to the application. This is called up-call approach. As we mentioned earlier, available bandwidth can be exchanged at different granularities (*e.g.*, per-interval or per-packet) in both approaches.

We believe that the up-call approach for bandwidth is more appropriate for quality adaptive streaming applications than probe approach due to the following reason. Since the application can probe the available bandwidth at an arbitrary pattern, the reporting interval in probing approach is not known a priori. Thus it is hard for the CC module to properly define average available bandwidth.

D. Passive vs Active CC

The CC module can passively monitor packet departure and ACK arrivals to determine loss rate and RTT and report available bandwidth as shown in Figure **??**. Then the application is expected to implement various functions 1) to match its transmission with available bandwidth (over different timescale), 2) to limit degree of burstiness, and 3) to frequently receive bandwidth information from the CC module. This framework solely depends on the application to correctly implement these functions. Therefore, any application-level error in these functions could easily lead to a misbehaved flow. To avoid this problem, the CC module can *actively* regulate transmission rate by decoupling application's transmission schedule (*i.e.*, BW_{ave}) from actual transmission departure schedule (*i.e.*, BW_{txr}) as shown in Figure6. When the application does not implement the above functions properly, its transmission rate (over a certain timescale) exceeds the available bandwidth ($BW_{ave} < BW_{app}$). This in turn triggers the CC module to actively modify the transmission schedule to fit it into the acceptable range of transmission schedules. However, the key point is that an adaptive application should be aware of such a modification to behave properly, otherwise any modification of application's transmission schedule will confuse the application and adversely affect adaptation mechanism.

The CC module has the following two alternative ways to actively control application's transmission rate:

1) *Reshaping application's transmission rate*: The CC module can buffer extra packets, inform the application about the volume of buffered data and send the buffered data with the rate equal to available bandwidth. This approach essentially reshapes application's transmission schedule [1]. Such server-side buffering scheme adds a random component to the end-to-end delay that could not be easily tolerated by quality adaptive streaming applications. Note that the amount of buffering delay depends on the difference between application's transmission rate and available bandwidth which could be high when the application does not slowdown. Such a random delay could easily cause an important packet to arrive after its playout time. Obviously, the application can use the amount of buffered data as a signal to adjust its transmission rate as suggested by Jacobs et al. [9] (*e.g.*, if the volume of buffered data in the CC module is more than a threshold, then reduce its transmission rate in a certain way). However, such adaptation scheme only reacts to "smoothed" variations of bandwidth, thus it is unable to perform quality adaptation effectively [2].

2) *Clipping Application's transmission rate*: Alternatively, the CC module can simply return an error (in respond to application's send request) whenever application's transmission rate exceeds available bandwidth without buffering any packet. Acceptable bursts are sent through the CC module without any reshaping. In essence, the application can fully control transmission departure schedule, and the CC module only performs a policing function to ensure that application's behavior (in terms of rate, burstiness, etc) is within an acceptable range. This approach works really well with the per-packet bandwidth reporting scheme since the CC module passes the bandwidth budget (*i.e.*, number of packets that can be currently sent) as a hint to the application.

E. Kernel-level vs Application-level CC

CC functionality can be implemented within the kernel or at the application level. Kernel-level CC (*e.g.*, congestion manager [6]) has two benefits: 1) an in-kernel CC module would be able to better meet a specific packet departure time since it does not depend on proper invocation by the application, and 2) an in-kernel CC module can easily interrupt the application to report available bandwidth. Furthermore, an in-kernel CC module can easily operate in active mode.

Placing the CC module in the kernel has its own drawbacks: 1) the first issue is incremental deployment. A

[1] This is somewhat similar to the way TCP socket works inside the kernel.

[2] This is similar to reporting average bandwidth over long intervals that ignores details of bandwidth variations as we discussed in sec **??**.

kernel-level CC module should be available within the kernel on both end-systems. 2) this would discourage research and development on design of new CC modules because adding new modules to a kernel requires special privilege that an average user does not usually have. 3) kernel-level CC modules may not be easily ported to other operating systems.

In contrast, an application-level CC module could fall behind a given packet departure schedule due to 1) lack of sufficiently fine-grain timers by most operating systems, or 2) improper integration of the CC module into the application. Furthermore, an application-level CC module can work properly if the application properly and frequently passes the control of execution to the CC module. This requires that the CC functions would be included in the main application event loop as we discuss in section III-A. Proper integration of the CC module into the application is crucial not only to closely control packet departure schedule but also to quickly read ACK packets and measure RTT samples accurately. The CC module should quickly perform time-sensitive functions and return the control to the application. An application-level CC module can be easily ported to any operating systems and allows incremental deployment. Both kernel and application-level CC modules could be used in active or passive mode.

F. Supporting Aggregate CC

A multimedia application may require to maintain several sessions between two end points. For example, many streaming applications require an audio sessions and a separate video session. Another example is a streaming application that delivers layered encoded streams where each layer is sent to a separate session[3].

The CC module should be able to map all these related sessions between two endpoints into a single CC flow and perform CC over all these sessions collectively. In other words, overall transmission rate of all these sessions are regulated by the CC module[6]. The application can determine what portion of the available bandwidth should be allocated to each flow (*i.e.*, perform quality adaptation). The CC module should also allow the application to group the sessions into separate CC flows based on their semantics. For example, if an application supports two video sessions, each video session could be delivered through a separate CC flow.

G. Duplicating Functionalities

A CC module requires to derive path properties (mainly RTT and loss rate, or individual packet losses) to determine available bandwidth. Adaptive streaming applications also need this information and should implement required functions to work properly. For example, some encodings can gracefully tolerate up to certain loss rate but higher loss rate significantly degrades observed quality. It would be crucial for such streaming application to continuously monitor observed loss rate. In many encodings, some frames have a bigger impact on quality (*e.g.*, I frames in MPEG, or frames of base layer in layered encoding) and thus it is crucial to deliver these frames. Streaming applications should keep track of individual losses to ensure that the important frame have been delivered.

To avoid duplicating these functionalities (*i.e.*, RTT and loss-rate measurement) in both CC module and the application, the CC module should implement these functions and provide a flexible interface to the application.

H. Role of Client Buffering

It is important to clarify that client buffering does not compensate for inaccurate implementation of a CC mechanism or improper integration of a CC module into the application. Our goal is to properly integrate CC into streaming application such that the application has some freedom to change the packet departure time and possibly send bursts, but these changes should not significantly modify overall behavior of the CC mechanism so that the resulting flow no longer behave in a network-friendly fashion (and may receive less bandwidth share). Client buffering only allows the application to cope with variations of available bandwidth due to congestion control which is an orthogonal issue.

I. Summary of Desired Features for CC module

Here we summarize our desired features for a CC module from the above discussion:

- Available bandwidth should be reported at per-packet granularity.
- Bursty transmission schedule should be accommodated.
- Up-call bandwidth reporting should be supported.
- The module should be able to operate in both active and passive modes. However, active mode is preferred.
- Application-level implementation is preferred.
- A flexible interface for reporting link characteristics should be provided.
- Aggregate CC should be supported with flexibility for the application to group active sessions in any arbitrary manner.
- The CC module should provide a flexible interface to be easily integrated into an application.

In the following section, we present an application level library that implements such an interface.

[3]Each session is sent to a separate port

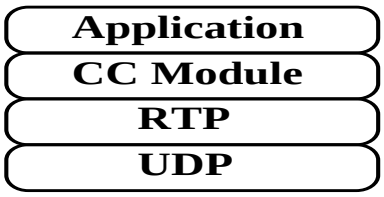

Fig. 7. Location of CC module in active mode

III. *cclib*: An Application-Level CC Library

In this section, we describe *cclib*. $cclib$ is an application-level CC library that supports all the desired features listed in section II-I. $cclib$ can be used in both active and passive mode. However, throughout this paper we present its operation in active mode since it is more comprehensive. Figure7 illustrates how $cclib$ is layered on top of RTP[10] and below the application. [4]. We have prototyped $cclib$ as a C library and evaluated it on top of UCL RTP library [11]. Current version of $cclib$ implements an AIMD-based CC algorithm, but its interface as well as integration methodology are rather generic and can be used with other CC algorithms with minimal or no modification.

cclib allows congestion control on a *per-flow* basis, *i.e.*, the library maintains state information for each active flow. Table I shows some of the state information that are kept for each flow. A flow consists of one or more RTP *sessions* where each RTP session could be a separate layer of a layered encoded stream, or a video (or audio) stream in a multi-stream application between two hosts. To open a new flow, the application calls $cc\_open(dst, n_s, callback)$ and specifies destination address (dst), number of sessions (n_s) and a callback function ($callback()$) among other things. The $cclib$ then opens n_s RTP sessions and group them into a single flow and returns a fid. These RTP sessions are multiplexed into a single congestion controlled flow (fid) and are directly managed by $cclib$.

The callback function is used by the $cclib$ to notify the application on a packet departure or an ACK arrival event. Since $cclib$ in an application level library, it is crucial to properly integrate it into the application's main event loop so that the control of execution is frequently exchanged between the application and the library. This allows the $cclib$ to function properly. More specifically, the library can 1) examine transmission schedule of all active flows to signal each flow on its next departure time and 2) read any arrived ACK as soon as possible to accurately measure RTT samples.

Figure 8 depicts interactions between the application and the CC module for sending data packet and receiving ACK packets. When the next packet departure time for flow fid arrives, $cclib$ invokes its callback function to notify the application to send a packet to this flow. The callback function should determine 1) content of the next packet (pkt), and 2) the session that the packet is sent to (sid), [5] and then send the packet by calling $cc\_send(fid, sid, pkt, ..)$ interface of the CC module in the active CC mode. This allows the application to multiplex contributing sessions of an active flow at a per-packet granularity. When an ACK packet arrives, $cclib$ reads the ACK as soon as possible, measures RTT sample, performs ACK-based loss detection, and then passes the ACK packet to the application via the callback function because the application may still use the ACK packet as its own end-to-end feedback. For example, the receiver can report its most recent playout time to allow the sender to determine amount of buffered data at the receiver. Furthermore, the CC module passes the callback function to the RTP library to notify the application when it parses an RTCP message.

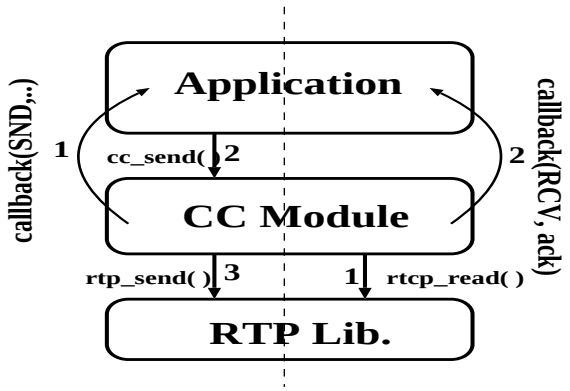

Fig. 8. Packet transmission and ACK arrival process

$cclib$ provides access to main characteristics of each active connection including: exponentially weighted moving average (EWMA) RTT, last RTT sample, loss rate, and individual packet losses for different RTP sessions. The application can obtain updated value for all of these variable (except individual losses) by calling $cc\_getopt(optname, \&optval)$ where $optname$ specifies the information of interest and $optval$ is the return value of the requested variable. Individual losses are maintained and presented through a separate interface that is described in section III-C

A. Integration into Application

As we mentioned earlier, an application-level CC module should be properly integrated into the application's

[4]Note that for non realtime application, $cclib$ can similarly operate on top of UDP.

[5]The callback function should in fact implement both quality adaptation and adaptive loss repair functions

State Parameter	Definition
cc_seqno	a unique seqno per flow
$last\_depart\_t$	departure time of last pkt
ipg	current inter packet gap
lossrate	loss rate
$ewma\_rtt$	EWMA RTT
$sample\_rtt$	last sample RTT
$app\_callback$	ptr to app. callback func.
NoLayer	No of rtp_session
lossq[MAXLAYER]	lost packet per session
$rtp\_session$[MAXLAYER]	pointer to RTP sessions

TABLE I
PER-FLOW STATE INFORMATION BY $cclib$

main event loop. The event loop for most networking applications is formed around a $select()$ call. cc_lib provides two simple and intuitive functions to be called before and after the $select$ call in the main event loop as follows:

```
/* event loop */
WHILE(EndofSession == FALSE){
  FOR(all active fids)
    cc_select_info(fid, &maxfd, &rfds,
                   &timeout,..);

  SELECT(maxfd, rfds, wfds, timeout);

  FOR(all active fids)
    cc_select_handle(now, fid,
                     rfds,..);
}
```

The application should call $cc\_select\_info()$ for each flow prior to $select$. This function updates all the select input parameters to include all the rtp session (*i.e.*, sockets) that are associated to a flow and managed by $cclib$. $fdset$ bitmap is updated for all file descriptors of active sessions and $maxfds$ is updated only if one of the new file descriptors has a higher value than current $fdset$ value. If the time interval between the next departure time ($next\_deprt\_t$) and current time (now) is less than the current $timeout$ value, $timeout$ is reduced accordingly, more specifically

```
IF(next_depart_t - now < timeout)
   timeout = next_depart_t - now
```

$cc\_select\_handle()$ examines state of an active flow after select. If $next_departure_t$ for a flow has arrived, that flow is notified through its registered callback function to send a packet. If an ACK has arrived for a flow, $cclib$ uses the ACK to measure a sample RTT and perform loss detection, then the ACK is passed to the application through the callback function as shown in Figure 8.

B. Supporting Bursty Streams

As we argued in the previous section, it is important that a CC module allows bursty flows while limits their burstiness below the acceptable degree. In other words, an application should be able to micro-manage its transmission schedule as long as its transmission schedule does not significantly diverge from per-packet available bandwidth (*i.e.*, CC's transmission schedule). Thus, the CC module should implement a policing mechanism that detects any unacceptable change in per-packet available bandwidth, and prevents it.

To devise such a policing mechanism, $cclib$ should limit the following two basic change in packet transmission schedule: 1) maximum burst size ($MAXBURST$), 2) maximum delay in sending a packet *i.e.*, maximum shift of schedule in time (τ). $MAXBURST$ simply limits number of back-to-back packets that can be sent by the applications whereas tau determines how long the application can postpone transmission of a packet. To implement such a mechanism, $cclib$ uses a sliding window of length tau to implement such a mechanism. Furthermore, $cclib$ maintains a variable mbs which keeps track of maximum burst size that the application can send at any point of time. mbs is increased by one before the callback function notifies the application to send a packet, and decreased by one after the application sends a packet via $cc\_send()$. To reduce the book-keeping overhead for the application, the recent value of mbs (*i.e.*, current bandwidth budget) is reported to the application in the callback function for transmission. Reporting the bandwidth budget allows the application to effectively match its transmission schedule with available bandwidth over short timescale. Furthermore, the application can probe $cclib$ to obtain the updated value of msb.

Clearly, if the application does not send a packet when it is notified, msb could increase up to the $MAXBURST$ limit. However, to ensure that the application does not shift the per-packet available bandwidth (*i.e.*, ideal transmission schedule), a credit for sending a packet is expired after period τ, *i.e.*, msb is decreased by one if the application does not send any packet within period τ. This is somewhat similar to the idea of aging TCP congestion window when no feedback is received from the channel for a certain period.

Figure 9 illustrates how mbs is updated by these two mechanisms in parallel. Besides simple decrease of mbs before each packet transmission in $cc\_send()$, the following pseudo code illustrates main modifications in

$cc\_select\_handle()$ to implement the above sliding window mechanism:

```
CC_SELECT_HANDLE(now, ...){
  <...>
  /* check for expired pkt tx */
  IF(fst_depart_t < now - T)
     mbs--;
     fst_depart_t += ipg;

 /* update mbs */
 IF(++mbs > MAXBURST)
     mbs = MAXBURST
     callback(SEND, mbs, ...)
```

Both τ and $MAXBURST$ are configuration parameters that both limit max burst size in different ways. $MAXBURST$ limits the absolute value of the burst size whereas τ limits the interval over which these mbs can be accumulated. Depending on source transmission rate one of these two parameters would be a limiting factor. For example, in high transmission rate, $MAXBURST$ is more likely to be a limiting factor while in low transmission rates τ could limit the level of freedom for the application to control packet departure. Therefore, for proper operation of the sliding window scheme, it is essential to set these parameters appropriately. In the current implementation of *cclib*, τ is set to smoothed RTT (*i.e.*, EWMA RTT) and $MAXBURST$ is set to 3.

We are currently conducting more investigations to understand how these parameters adversely affect the behavior of different congestion control algorithms to the point where a CC algorithm is no longer behave in a network-friendly fashion. We should add that the sensitivity of various congestion control algorithms to these parameters could be different. Our goal in this paper is not to find proper values for these configuration parameters. Instead, we present a mechanism that allows the CC module to regulate application's transmission schedule for given configuration parameters.

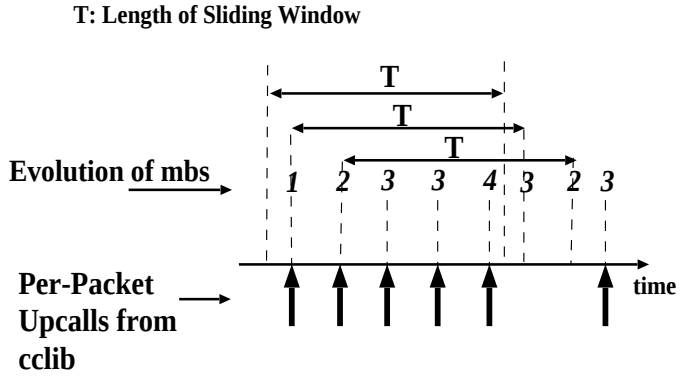

Fig. 9. Controlling burstiness and shift of transmission schedule by application

Note that a misconfigured $cclib$ could lead to a misbehave flow which is the original problem that we are trying to avoid. There are two points to notice. First, the damage that is caused by a bad CC implementation or buggy application that does not integrate a passive CC module properly, is worse than a carefully designed but misconfigured $cclib$. Second, as we learn more about the impact of these configuration parameters, we reduce the side effects of misconfiguration.

C. Support for Loss Repair

To avoid duplicating the same functionalities by both applications and the CC module, $cclib$ keeps track of various path variables including EWMA RTT, loss rate, individual packet losses. The application an use $cc\_getopt(optname, \&optval)$ to query these value except for individual loss packets. Keeping track of individual losses are specially important for those streaming application that perform retransmission-based loss repair. $cclib$ deploys both a timer-based and an ack-based loss detection mechanisms in parallel to detect individual losses. $cclib$ also adds some redundancy to the ACK stream to cope with some degree of packet reordering[5]. Detected losses for each RTP session of an active flow are kept in a separated link-list (called $lossq(i)$) where they are sorted by their RTP timestamp.

The application can call $cc\_get\_lossq(fid, sid, \&pkt)$ to get a pointer to the first lost packet for RTP session number sid. The application can later remove this lost packet from the $cclib$ list by calling $cc\_rm\_lossq(fid, sid, pkt)$ when the packet is no longer considered for repair. (*i.e.*, the packet is not sufficiently important to retransmit or it is too late to retransmit the packet). $cclib$ also controls the length of $lossq(i)$ in cases where a buggy application does not effectively free up old loss packets. Clearly, this is not the only way that a loss queue can be implemented.

D. Practical Considerations

An operational CC library should be able to cope with a couple of practical limitations. One key issue on most existing operating systems is a lack of fine-grained timers. The best timer granularity in many existing operating systems is around 10ms. Thus, a naive implementation of $cclib$ can delay a packet departure by as much as 10ms just due to bad scheduling. A similar problem might occur

when a system is heavily loaded or the application spends a relatively long time for its book-keeping activities.

$cclib$ incorporates two simple mechanisms to minimize the negative impact of coarse timers or ill-behaved (or busy) application on packet scheduling. First, $cclib$ performs loose scheduling by assigning a departure window instead of a departure time to each packet. Packet n is assigned an ideal departure time ($t_{snd}(n)$, but it could be sent any time during the window $t \leq t_{snd}(n)$ - δ*$ipg(t)$ where 0 < $delta\ leq$ 1. This means that a packet can be sent up to one ipg earlier than its ideal departure time. Second, arrival of an ACK packet provides an opportunity for the application to pass the control to $cclib$ (by calling $cc\_select\_handle()$) which in turns allows the library to follow packet departure schedule more closely. Thus a server with a few active flows (to different end points), is "clocked" sufficiently by the incoming ACKs so that any packet departure deadline can be closely met. A combination of loose scheduling and ack-clocking [6] significantly limit the negative impact of coarse timers on accuracy of packet scheduling by $cclib$.

IV. Related Work

The benefits of implementing CC for streaming applications have been discussed in [12], [13] where the main motivation for performing CC was to improve quality of delivered stream rather than achieving inter-protocol fairness and network stability. The importance of integrating CC into streaming applications was illustrated by Floyd et al. in [1]. During the past few years, design and evaluation of new CC mechanisms for streaming applications has been an active area of research (*e.g.*, [5], [2], [3], [4]). These new CC mechanisms were carefully designed and extensively evaluated to achieve inter-protocol fairness (*i.e.*, tcp-friendliness) under various network conditions. However, in design and evaluations (often with simulation) of these new CC mechanisms, it is implicitly assumed that these CC mechanism can fully control and achieve ideal packet departure schedule,*i.e.*, packets are evenly spaced and sent once every inter-packet-gap. Therefore, the following practical are not often addressed: "how available bandwidth should be reported to applications?, and "how is the behavior of a congestion controlled flow affected when the application changes packet departure time?"

Jacobs et al. [9] uses a window-based CC scheme similar to TCP to regulate transmission rate. Rejaie et al. [8] presents a quality adaptation mechanism for layered encoded stream over Rate Adaptation Protocol [5]. This work is extended to Binomial congestion control by Feamster et al. [14]. In these schemes the CC machinery is implemented within the application. Therefore, there is not separate modular CC mechanism that can be reused in other applications.

There have been several application-specific solutions that added CC into streaming applications (*e.g.*, [9], [8], [14], [15], [16], [17]. Tan et al. [16] uses TCP equation [18] to estimate TCP-equivalent available bandwidth. Their work is probably the first attempt to use TCP-equation for rate control in streaming applications. [17] uses TCP-equation to *sample* available bandwidth and further *smooth out* sample bandwidth estimates from the equation. Clearly, applying such a sampling and smoothing functions on estimated bandwidth could significantly affect behavior of an equation-based CC mechanism and could result in a non-TCP-friendly CC mechanism. The above two solutions only examine overall performance (*i.e.*, quality) of their congestion controlled streaming applications and does not evaluate fairness of their proposed CC mechanism at all. Merely using TCP-equation to estimate bandwidth does not necessarily imply that application's behavior would be network friendly. As discussed by Floyd et al. in [3], to engineer an equation-based CC mechanism, several important design issues (*e.g.*, proper calculation of loss event rate) must be properly addressed. But even a carefully-engineered CC mechanism like TFRC could exhibit some undesirable behavior in certain circumstances[19]. Work in [20] and [21] proposed new CC mechanisms for streaming applications without sufficiently evaluating performance of their proposed CC schemes.

Several solutions have shown how carefully integrated CC mechanism into streaming applications (*e.g.*, [9], [8], [14]); Jacobs et al. [9] uses a window-based CC scheme similar to TCP to regulate transmission rate. Rejaie et al. [8] presents a quality adaptation mechanism for layered encoded stream over Rate Adaptation Protocol [5]. This work is extended to Binomial congestion control by Feamster et al. [14]. In these solutions, the CC machinery is implemented within the application. Therefore, there is not a separate modular CC mechanism that can be reused in other applications.

Researchers who study encoding scheme for streaming over the Internet (*e.g.*, [22] and [23]) often model an Internet connection as fix-bandwidth channel with a known loss distribution. Then, they formulate the problem as an optimization problem to minimize distortion. These studies nicely relate channel characteristics with delivered quality, however, the dynamics of congestion control which is admittedly hard to model, is not incorporated in these studies.

[6] Here by ack-clocking, we refer to the fact that the application is clocked by any ACK arrival. This does not necessarily mean that a packet is sent upon each ack arrival similar to TCP ack-clocking

There has also been efforts at IETF to present a new end-to-end protocols to encourage streaming applications to deploy congestion control. Streaming Control Transmission Protocol (SCTP) [24] is a transport protocol suited for PSTN signaling across the IP network. SCTP deploys a window-based congestion control scheme and implements error control mechanism among other things. Datagram Congestion Control Protocol (DCCP) [25] is a new protocol to the family of TCP, UDP, SCTP. DCCP implements a congestion-controlled, unreliable flow of datagrams for streaming applications but it allows applications choose among several forms of CC. As of the current version, DCCP still neither presents any API for application nor discusses its relationship with RTP.

The most relevant previous work to our work is the Congestion Manager (CM) by Balakrishnan et al. [6]. CM is an operating system module with a convenient programming interface that implements CC functionality[7]. CM allows multiple concurrent streams between two endpoints to share a congestion controlled connection. The current version of CM can be used with those applications that implement their own end-to-end feedback to determine packet loss. However, many of today's streaming applications either do not implement any CC mechanism at all or at least do not implement it properly. According to our simple taxonomy, CM is a passive Kernel-level CC module that

CM presents two interfaces for streaming applications. The rate-control interface is intended for layer encoded stream where the application specifies a minimum bandwidth changes for which it should be notified by the CM. The assumption is that the source can only send at discrete rate depending on number of layers. However, quality adaptive application can send at any rate and use extra bandwidth to send more important layers faster (*e.g.*, [8]). For example, when the available is equal to 2.7 layer bandwidth, although the available bandwidth is never sufficient for delivery of three layers, but the server can still deliver he third layer 70% of the time.

The second interface is request/callback interface for adaptive applications. In this interface the application first requests from CM for permission to send a packet, then CM calls back the application to send a packet. Finally, the application informs CM that a packet was sent. There seems to be two problems with this probe interface. First, the CC module requires to send sufficient number of packets in order to probe the channel for excess capacity. Therefore, if the application does not aggressively request for packet transmission, it may not receive its fair share of bandwidth. Second, such an aggressive adaptive application should continuously request for packet transmission in a busy-waiting-like manner to probe the connection for excess capacity. As we argued in section II-C, up-call interface or CC module would eliminate this problem. To the best of our knowledge, no other work has addressed issues and challenges for integration of CC mechanism into streaming applications.

V. Conclusions and Future Work

In this paper, we investigated how a CC mechanism should be effectively integrated into adaptive streaming applications. We addressed challenges for such integration and argued that CC module should be designed and evaluated independent of applications, and therefore it should provide a well-defined and flexible interface to facilitate the integration process. Then, we explored design space for defining such an interface and presented a set of preferred features. Finally, we described a prototyped application-level CC library that implements a well-defined and flexible interface, and supports all the preferred features.

We plan to continue this work in a couple of directions. First, we plan to examine the effect of bursty transmission schedule and shift in transmission schedule on observed channel behavior. Our findings allows us to select better configuration parameters for our propose CC interface. Second, we need to evaluate performance of $cclib$ to ensure that various proposed mechanisms can perform effectively under various adaptive streaming applications.

References

[1] S. Floyd and K. Fall, "Promoting the use of end-to-end congestion control in the internet," *ACM/IEEE Transactions on Networking*, 1999, http://www-nrg.ee.lbl.gov/fbyd/papers.html/end2end-paper.html.

[2] D. Bansal and H. Balakrishnan, "Binomial congestion control algorithms," in *Proceedings of the IEEE INFOCOM*, 2001.

[3] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicaqt applications," in *Proceedings of the ACM SIGCOMM*, 2000.

[4] I. Rhee, V. Ozdemir, and Y. Yim, "Tear: Tcp-emulation at receivers-fbw control for multimedia streaming," in *Technical Report NCSU*, 2000, http://www.csc.ncsu.edu/faculty/rhee/export/tear-page.

[5] R. Rejaie, M. Handley, and D. Estrin, "RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet," in *Proc. IEEE Infocom*, New York, NY., Mar. 1999.

[6] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan, "An integrated congestion management architecture for internet hosts," in *Proceedings of the ACM SIGCOMM*, Cambridge, MA., Sept. 1999.

[7] Deepak Bansal David Anderson, Srinivasan Seshan Dorothy Curtis, and Hari Balakrishnan, "Systems support for bandwidth management and content adaptation in the internet," in *OSDI*, San Diego, CA, Oct. 2000.

[8] R. Rejaie, M. Handley, and D. Estrin, "Quality adaptation for

congestion controlled playback video over the internet," in *Proceedings of the ACM SIGCOMM*, Cambridge, MA., Sept. 1999.

[9] S. Jacobs and A. Eleftheriadis, "Real-time dynamic rate shaping and control for internet video applications," *Workshop on Multimedia Signal Processing, Invited Paper*, pp. 23–25, June 1997.

[10] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for realtime applications," in *Internet Engineering Task Force, Audio-Video Transport Working group*, Jan. 1996, RFC 1889.

[11] "Ucl common library,".

[12] J. Bolot and T. Turletti, "A rate control mechanism for packet video in the internet," in *IEEE INFOCOM*, June 1994, pp. 1216–1223.

[13] T. Turletti and C. Huitema, "Video conferencing in the internet," in *ACM/IEEE Transactions on Networking*, June 1996, pp. 340–351.

[14] N. Feamster, D. Bansel, and Hari Balakrishnan, ," in *Packet Video Workshop*, 2001.

[15] K. Ross P. de Cuetos, "Adaptive rate control for streaming stored fi ne-grained scalable video," in *Workshop on Network and Operating System Support for Digital Audio and Video*, Miami, Florida, May 2002.

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

[17] F. Licandro and G. Schembra, "Rate/quality controlled mpeg video transmission system in a tcp-friendly internet scenario," in *Packet Video Workshop*, Pittsburgh, PA, Apr. 2002.

[18] J. Mahdavi and S. Floyd, "TCP-friendly unicast rate-based fbw control," *Technical note sent to the end2end-interest mailing list*, Jan. 1997, http://www.psc.edu/networking/papers/tcp-friendly.html.

[19] D. Bansal, H. Balakrishnan, S. Floyd, and Scott Shenker, "Dynamic behavior of slowly-responsive congestion control algorithms," in *Proceedings of the ACM SIGCOMM*, 2001.

[20] K-W Lee, R. Puri, T. Kim, K. Ramchandran, and V. Bharghavan, "An integrated source coding and congestion control framework for video streaming in the internet," in *IEEE INFOCOM*, Tel-Aviv, Isreal, Mar. 2000.

[21] E. Jammeh, M. Paredes-Farrera, and M. Ghanbari, "Transporting real time video over the internet using end-to-end feedback control," in *Packet Video Workshop*, Pittsburgh, PA, Apr. 2002.

[22] P. A. Chou and Z. Miao, "Rate-distortion optimized streaming over best-effort networks," in *Submitted toIEEE Transactions on Multimedia*, 2001.

[23] Z. Miao and A. Ortega, "Optimal scheduling for streaming of scalable media," in *Asilomar Conference*, Pacifi c Grove, CA, Nov. 2000.

[24] R. Stewart et al., "Stream control transmission protocol," in *RFC2960*, 2000, http://www.ietf.org/rfc/rfc2960.txt.

[25] E. Kohler, M. Handley, S. Floyd, and J. Padhye, "Datagram congestion control protocol (dccp)," in *Internet Draft (draft-ietf-dccp-specs-00*, 2002, http://www.icir.org/kohler/dcp.