PALS: Peer-to-Peer Adaptive Layered Streaming

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

Antonio Ortega
Integrated Media Systems Center
University of Southern California
ortega@sipi.usc.edu

ABSTRACT

This paper presents a new framework for Peer-to-Peer Adaptive Layered Streaming, called PALS. PALS is a receiver-driven approach for quality adaptive playback of layer encoded streaming media from a group of congestion controlled sender peers to a single receiver peer. Since the effective throughput from each sender is variable and not known a priori, it is challenging to coordinate delivery among active senders. In PALS, the receiver orchestrates coordinated delivery among active senders by adaptively determining: 1) a subset of senders that maximize overall throughput, 2) overall quality (i.e. number of layers) that can be delivered from these senders as well as distribution of overall throughput among active layers, and most importantly 3) required packets to be delivered by each active sender in order to effectively cope with any sudden change in throughput from individual senders. We describe PALS framework, identify key components of the framework and their interesting design challenges, present sample solution for the key components, and present our preliminary results.

Categories and Subject Descriptors

C.2 [Computer-Communications Networks]: Distributed Systems-*Distributed Applications*.

General Terms

Design, Performance, Measurement.

Keywords

Peer-to-peer networks, Congestion control, Quality Adaptive streaming, Layered encoding.

1. INTRODUCTION

Peer-to-peer (P2P) networks are becoming increasingly popular as an alternative communication paradigm to traditional client-server architecture. Most of the research on P2P networks has focused on two questions: "how to form a network of cooperative peers?" and "how to locate a piece of content among in such a network?" (*e.g.*, [1],[2]). Since most applications in P2P networks exchange files among peers (*e.g.*, Napster), the actual delivery of content in a P2P network has received little attention, because it simply requires a file transfer between two (or more) peers. However $streaming$ realtime multimedia (audio, video) content in P2P networks is a challenging problem because a single peer may not be willing or able to commit sufficient resources (*i.e.*, bandwidth or CPU) to stream a media file to another peer. Therefore, it is more likely that several peers collectively stream requested content to another peer. This approach should result in a better load balancing among sender peers and less congestion across the network. Furthermore, assuming receiver's access link is not the bottleneck, a group of sender peers is more likely to provide a higher overall throughput to the receiver peer and thus deliver a higher quality stream.

To enable streaming of content from multiple peers several challenging issues should be addressed: First, there should be a mechanism to identify a subset of available peers with the desired content, that can provide maximum overall throughput to the receiver peer. Note that the overall throughput of a group of sender peers may not increase when additional senders are added, because multiple senders may reside behind the same bottleneck. Second, each sender peer should perform TCP-friendly congestion control (CC) such as RAP [3] or TFRC [4]. This implies that the throughput available from each sender is not known a priori and could significantly change during a session. Furthermore, because of the Internet heterogeneity, characteristics (*e.g.*, bandwidth and RTT) of connections from different senders could be significantly different. Therefore, any delivery mechanism designed to operate from multiple senders should be quality adaptive, i.e., it should be able to adjust smoothly the quality (and thus the bandwidth) of the delivered stream, as the overall throughput changes. Third, given a subset of sender peers, the main problem is *how to coordinate a group of active senders so that they can cooperate in streaming the maximum deliverable quality that can be supported with the given overall throughput?*. For example, when three peers are cooperatively streaming a video they need to coordinate which peer would be responsible for in time delivery of each segment of the video. Fourth, since each sender peer can potentially leave during a session, any delivery mechanism should be able to cope with the dynamics of peer participation and minimize any negative impact of such dynamics on delivered quality.

In this paper, we present a new framework for streaming in P2P networks that is called *P2P Adaptive Layered Streaming* or *PALS*. PALS is a receiver-driven approach that allows a receiver to orchestrate the adaptive delivery of stored layer encoded streams from multiple sender peers to a single receiver. Given a set of sender peers, PALS progressively evaluates various combinations of senders to determine a subset of the senders that can collectively provide maximum throughput. Once such a subset of senders is selected, the receiver monitors the overall throughput and periodically determines what is the target overall quality (i.e., the total number of coding layers) that can be delivered from all senders. Then, the receiver determines a proper distribution of the overall throughput among active layers (*i.e.*, what portion of the overall throughput should be allocated for delivery of each layer), and finally divides allocated bandwidth to each layer among active senders (*i.e.*, which segments of each layer should be delivered by each sender) in order to effectively cope with any sudden change in throughput of individual senders.

PALS is built on our previous work on the design of a quality adaptation (QA) mechanism for congestion controlled playback of layer encoded video over the Internet [5]. While the goal of PALS is in essence similar to our previous work, there are a few key differences between PALS and the unicast QA mechanism: 1) PALS performs QA across multiple independent connections, with potentially different characteristics, rather than a single connection, 2) PALS is receiver-driven rather than sender driven. A receiver-driven approach to QA should address two new problems: First, the receiver may not have any information about patterns of change in overall throughput. Second, the receiver should effectively monitor and manage the delivery of segments from multiple senders, which requires several other mechanisms besides the receiver-driven QA. These differences introduce a new set of challenges that do not exist in a unicast QA mechanism. *The main contribution of PALS is a receiver-driven coordination and adaptation framework for streaming from multiple, congestion-controlled, sender peers that is able to cope with unpredictable variations in throughput from each sender peer, as well as with dynamics of peer participation*. Although we motivated the PALS framework for streaming in P2P networks, one can use PALS framework for streaming multimedia content from a potentially distributed group of multimedia servers across a best-effort network to a single client.

Our main goals in this paper are: 1) to describe the framework and its key components, i.e., the required mechanisms for coordination among senders and adaptation of delivered quality to dynamics of both network connection and peer participation, 2) to provide sample mechanisms for the key components, 3) to discuss interesting challenges that arise in the design of such mechanisms and present our preliminary results. In this paper, however, we do not discuss how an initial set of sender peers is identified. This issue appears to be similar for both realtime and non-realtime content and can be achieved in different ways, such as by contacting a server (*e.g.*, [6]) or using a hash function (*e.g.*, [1]). A detailed analysis of the various design issues, including the analysis of more general mechanisms to address the key challenges, and a more extensive evaluation of PALS remain as future work.

The rest of this paper is organized as follows: In Section 2, we present related work. We justify our two key design choices in Section 3. Section 4 describes the PALS framework, its key components and sample solutions for each component. We present our preliminary result in Section 5. Finally, Section 6 concludes the paper and presents our future plans.

2. RELATED WORK

The basic idea of streaming from multiple servers to a single client is not new. Apostolopoulos et al. [7] proposed a mechanism for streaming from multiple servers using multiple description (MD) encoding. They presented a distortion model for MD encoding and used this model to study the effect of server and content placement on delivered quality of both MD and single description (SD) encodings. They did not consider variations of throughput from each sender to the receiver and focused on content placement and server selection mechanisms. Nguyen et al. [8] presented a mechanism for delivery of streaming media from multiple mirror servers to a single client. They assumed that different flows do not share any bottleneck and overall bandwidth from all servers is always higher than stream bandwidth. In their proposed solution, the receiver periodically reports throughput and delay of all senders back to them using control packets. Then, senders run a distributed algorithm to determine which one should send each packet. The main difference between their approach and PALS is that PALS is receiver-driven and adapts the quality of delivered stream with variations of overall throughput.

There has been some related work on streaming in P2P networks. CoopNet[6] is a mechanism for distributing MD encoded streams among peers in both live and on-demand sessions that is able to cope with flash-crowd. CoopNet leverages MD encoding to send different description to separate tree of interested peers. For on-demand sessions, a new client contacts a single sender peer from a set of candidate peers, and this peer is asked to deliver the content. If that sender peer is unavailable or unwilling to serve, the client contacts another peer until a sender peer is found. There are several systems such as Abacast[9], Chaincast [10], Allcast [11], vTrails [12] that form a distribution tree from the participating clients similar to CoopNet. However, there is not sufficient information about these systems for comparison. Tran et al. [13] presented a technique called ZIGZAG, for building and maintaining an efficient single-source media distribution tree. A key distinction of our work is that we present a quality adaptive delivery mechanism from multiple congestion controlled senders.

Finally, both PALS and RLM [14] are receiver-driven mechanisms that leverage layered encoding, however there are a few fundamental differences between them. In RLM, the receiver can adjust number of delivered layers by joining a different number of multicast sessions. This allows the receiver to regulate overall incoming throughput (and thus overall delivered quality) at the level that does not cause congestion in the network, *i.e.*, the receiver implements some type of congestion control mechanism by regulating incoming throughput. In the PALS framework, all flows are unicast, and a unicast congestion control mechanism is implemented at each sender. Therefore, the overall incoming throughput at the receiver is the aggregate congestion controlled bandwidth from all senders. The receiver does not control overall incoming throughput, instead it controls the quality of the stream that is being delivered through the incoming throughput from multiple senders.

3. JUSTIFYING DESIGN CHOICES

Before describing details of PALS, we need to justify our two key design choices, 1) adopting a receiver-driven adaptation and coordination, and 2) using layered (or hierarchical) encoding.

First, the coordination and adaptation machinery for cooperative playback from multiple senders can be implemented at a receiver or at the senders. We believe that the receiver-driven approach is the natural solution since the receiver is the only $permanent$ member of the many-to-one session that has complete knowledge about 1) which packets have been successfully delivered, 2) the current subset of active sender peers, and 3) the available throughput from each sender peer. This puts the receiver in a unique position to orchestrate delivery from the senders. While the receiver cannot predict the future available throughput for each sender, it should be able to leverage a degree of multiplexing among senders to its advantage. Another advantage of the receiver-driven approach is that it does not require a significant processing overhead at the senders, since coordination and adaptation is primarily done by the receiver.

The receiver-driven approach introduces a network overhead for coordination messages that should be periodically sent from the receiver to all active senders. However, this overhead should not be higher than associated overhead for any conceivable sender-driven coordination approach that uses unicast messaging [1]. The receiver-driven approach may require the availability of some detailed meta-data about each requested stream (*e.g.*, variations of stream bandwidth for VBR streams). This information could be provided by either a central server or by the sender peers, and can be used to perform encoding-specific packet scheduling (*e.g.*, [15, 16, 17]). Obtaining this information before content delivery starts leads to a small overhead with respect to sender-driven approaches, but we believe that the benefits of using a receiver-driven approach outweigh this drawback.

Second, a layered organization of data streams presents a useful structure for cooperative playback from multiple senders. Both layer encoding and Multiple Description encoding provide such organization. We selected layered encoding since it often achieves a higher compression efficiency than MD. Layer encodings are less resilient to losses in lower layers (specially in the base layer) in comparing to MD. However, as we describe in the following section, in PALS lost packets can be retransmitted within a window of time. Note that PALS can accommodate both types of data representations. For example, in a purely layered representation a specific packet is present in all servers but is only requested from one. If additional robustness is desired, however, it would be possible to request the same packet from more than one server. Taking this approach one step further it is possible to incorporate MD formats as well, where the receiver can request two redundant (but not identical) versions of a stream from different servers.

4. P2P ADAPTIVE LAYERED STREAMING (PALS)

In this section, we describe the PALS framework in more detail. After presenting our target environment and assumptions in subsection 4.1, we provide an overview of the PALS framework and identify three key components in subsection 4.2. Then, design issues for these key components along with their sample solutions are discussed in subsections 4.3, 4.4 and 4.5.

4.1 Target Environment

Figure 1 shows our target environment, where several sender peers (SPAL) across the Internet cooperatively playback a requested stream to a receiver peer (RPAL). Each peer is expected to perform TCP-friendly congestion control. Therefore, throughput from each sender peer can co-exist in a fair way with other outgoing traffic from that sender (*e.g.*, providing different contents for other peers). Sender peers are potentially scattered across the Internet, therefore connection to one sender peer could exhibit significantly different characteristics (*i.e.*, bandwidth and RTT) from connections to other senders. PALS should be able to work with any arbitrary set of sender peers across the network as long as the overall throughput of all senders is higher than the bandwidth of a single layer.

We currently assume that a requested stream is entirely available at each sender peer. For clarity of the discussion, we also assume that all layers have the same constant-bit-rate (*i.e.*, C). However, neither of these assumptions is required for PALS. General information about a requested stream (*e.g.*, number of layers, layer bandwidth distribution, stream length) can be provided to the receiver along with the initial list of sender peers by the original server (or

[1]Multicast messaging may reduce coordination overhead in a sender-driven approach but in our judgment other benefits of the receiver-driven approach still outweighs this potential benefit.

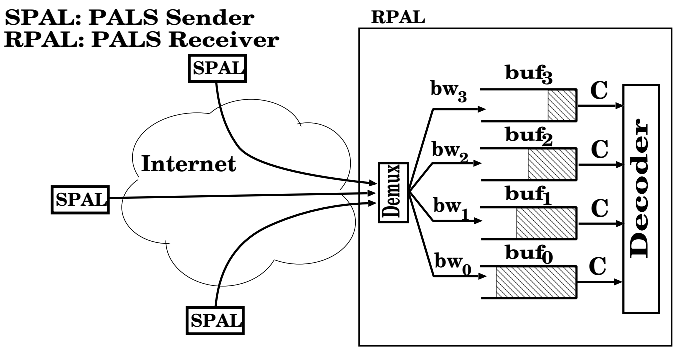

Figure 1: End-to-end Architecture in PALS

any other mechanism that is used to locate sender peers) during the initial setup phase. Table 1 summarizes our notation throughout this paper.

4.2 An Overview of PALS Framework

The primary goal of PALS is to stream the maximum quality that can be delivered by a set of available sender peers to a single receiver peer. The key challenge is that the available throughput from each sender is not known a priori, and can significantly change during a session. This implies that PALS should be able to gracefully cope with any sudden decrease in throughput while effectively leveraging sudden increases in overall throughput. The basic idea of PALS is simple and intuitive. The receiver periodically sends an $ordered$ list of packets to each sender. Each sender simply transmits the requested packets in the given order at the rate that is determined by its CC mechanism. Using ordered lists provides several benefits: 1) it allows the receiver to better control delivered packets from each sender, 2) it ensures graceful degradation in quality when throughput of a sender suddenly decreases by ordering the list based on importance of packets, and 3) it enables different receivers to implement different quality adaptation algorithms for delivery of the same stream. The requests for each sender can be piggy-backed with the CC feedbacks that are periodically sent to that sender or can be sent separately.

The machinery of PALS protocol is mostly implemented at the receiver. The receiver keeps track of the Exponentially Weighted Moving Average (EWMA) of overall throughput (T_{ewma}). At a given point in time, the receiver assumes that the current value of T_{ewma} will remain unchanged for a period Δ and estimates the total number of incoming packets (K) during this period as follows: $K = \frac{T_{ewma} * \Delta}{PktSize}$. The Quality Adaptation (QA) mechanism at the receiver distributes these K packets among *active layers* based on the overall throughput and receiver's buffer state (buf_0, buf_1, .., buf_N). For example, if the receiver expects to receive 50 packets during a period (K = 50) and four layers are currently being played, the QA mechanism may allocate k_0 = 20, k_1 = 15, k_2 = 10, k_3 = 5 packets to layer L_0 .. L_3, respectively. By controlling the distribution of incoming packets among layers, the receiver essentially allocates the distribution of overall throughput among active layers (bw_0, bw_1, .. ,bw_N) during one period, which in turn determines evolution of receiver's buffer state. Given the distribution of total packets among layers for a given period (k_0, k_1, .. k_n), the Packet Assignment (PA) mechanism at the receiver maps a subset of these packets (possibly from different layers) to each sender and sends a request that contains a list of assigned packets to each sender. This packet assignment strategy allows the receiver to loosely control allocation of each sender's throughput among

s_j	Sender j
L_i	Layer i
n	No. of Active layers
N	Max. number of layers
T_{ewma}	EWMA overall Throughput
T^j_{ewma}	EWMA Throughput from s_j
buf_i	Buffered data for L_i
bw_i	Allocated BW for L_i
$BUF[i]$	Target buffer for L_i
$PktSize$	Packet Size
$SRTT_j$	EWMA RTT from s_j
$SRTT_{max}$	Max. value among $SRTT_j$
Δ	interval between requests
K	Estimated no. of incoming Packets during Δ

Table 1: Summary of Notation

active layers. The number of assigned packets to each sender is proportional to the expected contribution of that sender's throughput to the overall throughput. For example, if sender s_i provides 50% of overall throughput, half of the overall packets during one interval are assigned to s_i.

Once the receiver initiates playback, there needs to be a mechanism at the receiver to keep senders loosely synchronized with the receiver. More specifically, playout time of the requested packets should be determined with respect to actual playout time of the ongoing session at the receiver in order to ensure "in-time" delivery of requested packets. PALS employs two complementary mechanisms to achieve this goal. First, PALS uses a Sliding Window (SW) approach that is shown in Figure 2. For each interval, the receiver only considers packets with playout time higher than t_{min} where $t_{min} = t_p + \tau$. The window should slide forward as playout time proceeds so that it always remains sufficiently ahead of the playout time. Furthermore, the SW mechanism should ensure that each sender always has outstanding packets to deliver, otherwise the sender may become idle and its throughput may drop below its congestion controlled limit, which would not be desirable. Second, any new list of requested packets from the receiver $overwrites$ any outstanding list that is being delivered by a sender, *i.e.*, when a sender receives a new list of requested packets from the receiver, it starts delivery of packets from the new list and abandons any pending packet from the previous list. This overwriting mechanism keeps slow senders loosely synchronized with receivers' playout. Finally, the receiver requires a Peer Selection (PS) mechanism in order to periodically examine other available peers and add them to the subset of active sender peers if their participation in delivery increases overall throughput of the session.

In summary, a PALS receiver requires the following four key components: Quality Adaptation, Packet Assignment, Sliding Window, and Peer Section. We discuss the design issues for each one of these components and present some sample mechanism in the following subsections, but we primarily focus on the first three components.

4.3 Quality Adaptation

The main goal of a QA mechanism is to maximize overall delivered quality while minimizing variations in playback quality despite unpredictable changes in available bandwidth. The key design issue is to manage the receiver's buffer state in order to effectively absorb short-term mismatch between stream's consumption rate ($n*C$) and available network bandwidth. The receiver's buffer state can be controlled by proper allocation of available bandwidth among active layers. In our earlier work [5], we designed a sender-based QA mechanism for client-server streaming where the receiver reports its buffer state to the sender, and the sender regulates inter-layer bandwidth allocation on a per-packet basis to keep receiver's buffer state close to the optimal state. The optimal buffer state depends on the pattern of variations in bandwidth (*e.g.*, Additive Increase/Multiplicative Increase) which is known to the sender. We follow the same design philosophy to design QA mechanism for PALS. However there are several important differences between QA in PALS and in unicast streaming, which require a new approach to QA for PALS. First, the QA mechanism is implemented at the receiver, this adds a delay to the control loop because the QA mechanism at the receiver should determine and control inter-layer bandwidth allocation for each sender's throughput. Second, the receiver should deal with multiple independent senders with potentially different variations in bandwidth. Furthermore, the receiver does not have any knowledge about pattern of changes in overall throughput which could be used to derive optimal buffer distribution. Third, performing a receiver-based QA mechanism on a per-packet basis (*i.e.*, sending a new request for each packet) would be really expensive, therefore the QA mechanism should be invoked periodically. In the remaining of this subsection, we sketch a receiver-driven QA mechanism for PALS.

The QA mechanism has two degrees of control, 1) it can effectively control distribution of total buffered data among active layers ($buf_0, buf_1, .., buf_n$) by proper allocation of overall throughput among active layers (*i.e.*, by controlling [$bw_0, bw_1, ... , bw_n$] where $\sum_{i=0}^{n} bw_i = T_{ewma}$); 2) it can change the number of playing layers by adding/dropping the top layer (*i.e.*, adjusting quality and thus bandwidth of delivered stream). Short-term changes in throughput are absorbed by buffered data whereas long-term changes in throughput trigger adjustment in stream's quality. When overall throughput is higher than stream bandwidth ($T_{ewma} >= n*C$), receiver can utilize the excess bandwidth to fill its buffers. We call this filling phase. Once receiver's buffers are filled to the appropriate level, the receiver can increase stream's bandwidth by adding a new layer. In contrast, when overall throughput is lower than stream bandwidth ($T_{ewma} < n*C$), the receiver can drain buffered data to compensate the bandwidth deficit. If total buffered data (or its distribution) is not sufficient to absorb the bandwidth deficit during a draining phase, the receiver can drop the top layer to avoid buffer underflow for buffered data for other layers.

The key observation is that not only the total amount of buffered data but also its distribution across active layers is crucial to effectively absorb variations in throughput. To illustrate this observation, assume a scenario where 4 layers are being played in Figure 1. Note that buffered data for each layer can not be drained faster than

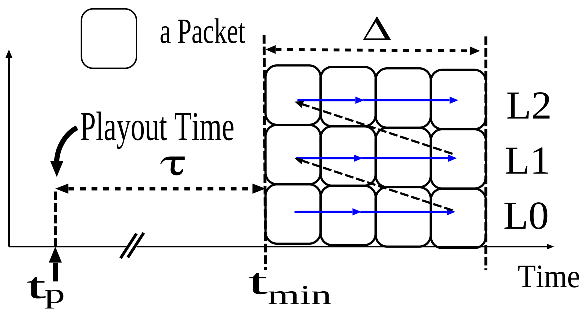

Figure 2: Sliding window scheme for loose synchronization of requested packets with playout time

its consumption rate (C). This implies that buffered data for each layer can at most compensate up to C bps deficit in throughput. Therefore, the bigger the deficit in throughput, the larger the total required buffering, and the larger the number of buffering layers that should be drained simultaneously to compensate the deficit in throughput. More specifically, the minimum number of buffering layers (n_{buf}) can be determined as $n_{buf} = \frac{n*C-T_{ewma}}{C}$. Figure 3 illustrates this requirement for two different draining scenarios with different pattern of changes in overall throughput. Although both scenarios require roughly the same amount of total buffering, scenario I requires buffered data with proper distribution across at least 3 layers whereas deficit in throughput for scenario II can be absorbed by buffering only for one layer. This observation suggests that even distribution of buffered data among active layer should be appropriate. However, we note that once a layer is dropped the amount of buffered data for that layer can not be leveraged for absorbing future variations in throughput which in turn leads to lower buffering efficiency. Therefore, given an expected bandwidth deficit in the future, the optimal buffer distribution is the one that distributes buffered data among minimum number of required buffering layers in a skewed fashion by allocating the maximum amount of data that can be drained from a buffer during a draining phase to buf_0, the next such a maximum to buf_1, and so on.

The optimal buffer distribution directly depends on the pattern of variations in the overall throughput. As we mentioned earlier, this is one of the challenges in the receiver-based QA since the receiver does not have sufficient (if any) information about the pattern of changes in overall throughput. Therefore, we considered two alternative solutions to determine a proper buffer distribution: 1) the receiver should be able to use a measurement-based technique to progressively derive the pattern of changes in overall throughput from the incoming stream and determine the new optimal buffer distribution accordingly, or 2) the receiver can use a pre-specified and fixed buffer distribution. Currently, we use the second approach with a linear buffer distribution for PALS and we call this *target buffer state, i.e.*, $BUF[i] = BUF[i-1] + \alpha * SRTT_{max}$ where α is a configuration parameter that determines the slope of linear increase in buffering per layer. Given a target buffer state, the target amount of buffering for layer i when n layers are active can be specified as $buf_i = BUF[n-i-1]$. Although this approach is static and not optimal, it can still be effective. Design of measurement-based techniques for deriving pattern of changes in overall throughput remains as future work.

Given a target buffer distribution, every time the QA mechanism is invoked, it goes through the following steps to keep the buffer state as close as possible to the target buffer state. First, it compares overall throughput with stream's bandwidth to determine whether it is in a filling or a draining phase, and then implements the corresponding mechanisms as follows:

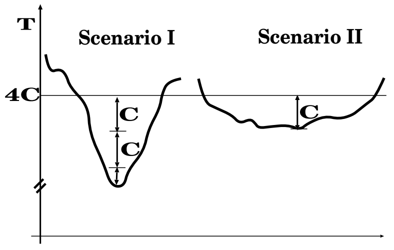

Figure 3: Impact of pattern of variations in throughput on inter-layer buffer distribution

- *Filling Phase:* If it is in a filling phase with n layers, the receiver tries to fill its buffers up to the target state with n layers. Towards this end, it starts from the base layer and sequentially determines the number of packets that each layer requires to reach its target buffer level at the end of this interval, until all packets are allocated or all layers have reach their target buffer levels. This algorithm is described in the following pseudo code:

```
$Filling\ Phase$
/* fill all layers up to target state */
$i = 0$
$WHILE\ (i < n\ AND\ K > 0)$ {
    $IF\ (BUF[n-i-1] > buf_i)$
        $buf_{deficit} = BUF[n-i-1] - buf_i$
    $ELSE$
        $buf_{deficit} = 0$
    $k_{cons} = \frac{C*\Delta}{PktSize}$
    $k_i = \frac{buf_{deficit}}{PktSize} + k_{cons}$
    $K = K - k_i$
    $i = i + 1$
}
```

  k_{cons} denotes the number of packets that are consumed by a layer during one period, and k_i is the number of allocated packets to L_i. Once buffered data for all n layers reach their target levels, if more packets are available, the QA mechanism repeats the same algorithm to fill all active layers up to their target level with (n + 1) layers. This fills buffers of all existing layers up to the point to add a new layer. Once all layers are filled up to this level and more packets are still available, a new layer can be added if available throughput is higher than stream's bandwidth with an additional layer ($(n+1) * C <= T_{ewma}$).

- *Draining Phase*: If the receiver is in a draining phase, first it determines how many active layers can be sustained during one interval with the current buffer state. If total amount of buffered data or its distribution is not sufficient to compensate the deficit in overall throughput, the QA mechanism progressively drops the top layer until the buffer state for the remaining layers is sufficient to compensate the deficit in overall throughput. Then, the QA mechanism essentially reverses its filling algorithm. More specifically, it tries to drain all layers towards the last target level starting from the top layer. If more data needs to be drained, the top layer is dropped, and the receiver repeats the above steps toward the previous target state with $n-1$ layers.

In summary, there are two key differences between the receiver-driven QA mechanism for PALS and the sender-based QA mechanism for unicast streaming. The QA mechanism for PALS should determine inter-layer bandwidth allocation for a period of time rather than on a per-packet basis. Moreover, this mechanism should somehow determine a proper client buffer distribution without any knowledge about pattern of changes in overall throughput.

We expect that the effect of any change in the number of sender peers (due to departing of an existing sender peer or availability of a new sender peer) would be similar to a sudden decrease or increase in throughput. Therefore, the QA mechanism reacts to both events the same way.

4.4 Sliding Window

The sliding window (SW) mechanism has two goals: 1) it should keep all senders loosely synchronized with the playout time at the receiver in order to prevent senders from sending packets whose playout time has already passed, and 2) it should ensure that all senders always have packets to send so they do not become idle. PALS achieves the first goal by using a sliding window coupled with the overwriting mechanism. As shown in Figure 2, the receiver maintains a window that is periodically slided forwarded every Δ seconds in order to stay τ seconds ahead of the playout time. If the receiver overestimates throughput from a sender, the window slides forward after Δ seconds and the receiver sends another request to the sender. This timer-driven approach (along with overwriting) prevents a sender from falling behind the ongoing session.

To achieve the second goal, the receiver keeps track of the number of delivered packets from each sender during each interval. If the number of pending packets at a sender goes below a threshold (k_{thresh}), the receiver sends another request to the sender. This mechanism reacts to a sudden increase in the throughput of a sender and triggers transmission of another request before the sender becomes idle. We call this mechanism *Reverse Flow Control* (RFC) because the receiver tries to ensure that the sender's buffer (*i.e.*, list of pending packets) does not underflow. One key parameter in the RFC mechanism is its threshold.This threshold for each sender should be specified as a portion of SRTT for that sender (δ). For example, setting δ to 0.5 for a sender means that a new request should be sent to a sender when it has less than half a SRTT worth of packets to deliver. The threshold can be translated into the maximum number of remaining packets that can trigger RFC mechanism as follows:

$$k_{rfc} = \frac{\delta * SRTT}{ipg_i} \text{ where } ipg_i = \frac{PktSize}{T^i_{ewma}}$$

Using a small threshold results in a late reaction to a sudden increase in a sender's throughput which may cause a sender to become idle. In contrast, if δ is set to a large value, RFC always sends a new request to a sender too early and frequently overwrites previous list of the sender. We call this an *overwriting effect*. The larger the threshold for RFC, the larger the portion of each list that is being overwritten, and thus the larger the number of packets that are ignored during one interval. Note that underlivered packets due to overwriting can be requested (from the same or other senders) and be delivered during the following intervals.

A key design question is *"how to couple the QA mechanism with the receiver's requests to different senders?"*. If throughput of all senders is overestimated or all senders finish close to the end of an interval, the receiver can invoke the QA mechanism once at the end of each interval to determine required packets for the next interval based on overall throughput, then send new request to all senders simultaneously at the beginning of the new interval. However, in practice, it is very likely that throughput of one sender suddenly increases during an interval, and the RFC mechanism triggers the receiver to send a new request to this fast sender. The key issue is to determine when a new request should be sent to other (slower) senders. One could devise two different approaches to address this issue as follows:

- *Synchronized Requesting*: A new request can be sent to all senders at the same time once per interval. This synchronized approach is simple because it tightly couples sliding window mechanism with QA mechanism. The window can be slided forward after Δ seconds or as soon as RFC mechanism is triggered for the fastest sender. The receiver invokes the QA mechanism once per window to determine required packets from all senders based on the overall throughput, and sends a new request to all senders simultaneously. In this approach the QA mechanism does not require to factor in throughput from individual senders.

- *Asynchronous Requesting*: The receiver can send new requests to different senders in an asynchronous fashion. A new request is sent to each fast sender as soon as its RFC mechanism is triggered whereas new request to other senders are sent at the end of one interval after sliding the window. This approach does not suffer from the overwriting effect. However, in this approach the sliding mechanism is decoupled from the QA mechanism and thus complicates coupling of the QA mechanism with outgoing requests. If the QA mechanism is invoked only once per interval to determine all packets for one interval based on the overall throughput, a request might be sent to a sender up to one interval later than the execution of the QA mechanism. Since network conditions might significantly change during one interval, this approach could lead to a poor performance (*e.g.*, frequent layer add/drop) for the QA mechanism. Alternatively, the receiver can invoke the QA mechanism before sending a request to each sender (or sending a batch of concurrent requests to a subset of senders). In this case, the QA mechanism should work in an *incremental* fashion - that means at each execution time in needs to send a request to a sender, the QA mechanism should not only consider the recently requested packets from other senders but it should also factor in variations in throughput for individual senders. This clearly adds to the complexity of the QA mechanism because it needs to cope with higher degree of network dynamics. We plan to explore new techniques for incremental QA in our future work.

The basic configuration parameter for the SW mechanism is the window size (Δ). Since the dynamics of variations in throughout from a sender depend on its SRTT, the length of each interval should be a function of SRTT. In the synchronized requesting approach, since the same interval is used for all senders, it must be selected in order to accommodate all senders. Δ directly controls responsiveness of the QA mechanism as well as frequency (and thus network overhead) of receiver's requests to senders. Using a small window allows the QA to quickly react to changes in throughput and maintain the buffer state closer to the target buffer state. However, if the window is too small, it could cause the QA mechanism to oscillate due to the delay in the control loop between the receiver and each sender. Obviously, if SRTT from different senders span a wide range, it would be difficult to set the window in order to satisfy above considerations for all senders. In the asynchronous requesting approach, however, the receiver can potentially use a separate interval for each sender. This scheme allows the receiver to control each sender with a separate frequency. This in turn shifts complexity to the design of QA mechanism for the asynchronous approach as we mentioned earlier. In summary, Δ should be chosen in order to achieve a proper balance between responsiveness for the QA mechanism, and network load for receiver's requests. In the current version of PALS, we use a synchronized requesting approach and set Δ to 4 to 6 $SRTT_{max}$ where $SRTT_{max}$ is the maximum SRTT among active senders at the beginning of each window.

4.5 Adaptive Packet Assignment

If the QA mechanism determines an ordered list of packets to be requested from multiple senders (*e.g.*, it is invoked once per RTT), a packet assignment mechanism is needed to properly distribute these packets among active senders. We note that total required packets for one window may include a different number of packets from various layers. For example, the QA mechanism may require 10, 8, 5 and 6 packets for L_0, L_1, L_2 and L_3, respectively. The number of assigned packets to each sender should be proportional to its expected throughput during a window so that all senders could deliver their assigned packets at relatively the same time. To achieve this, the receiver keeps track of short-term EWMA of per-window throughput for each active sender (T^i_{ewma}) and uses this to determine the number of requested packets from each sender as follows:

$k_i = \frac{T^i_{ewma} * K}{T_{ewma}}$ where
$T_{ewma} = \sum_{i=0}^{M} T^i_{ewma}$ and $K = \sum_{i=0}^{n} k_i$.

Then, the packet assignment mechanism should assign k_i specific packets from the total list of required packets to each sender while it tries to satisfy the following goals: First, it tries to assign all required packets of a layer to a single sender whenever possible. This strategy reduces the size of request messages, and thus requires less network bandwidth for sending request messages. Second, a more important consideration is to distribute packets among senders in order to minimize any negative impact on delivered quality due to a sudden change in a sender's throughput. We discuss two basic examples to illustrate the impact of packet assignment strategies on overall behavior of PALS framework.
Coping with Slow Senders: if the receiver overestimates throughput of one or more senders, those senders can not deliver all the assigned packets within the current window. The main goal of the packet assignment mechanism is to ensure that the available throughput is used to deliver the most important packets. Since the receiver does not know a priori which sender might be slow, assigning all packets of a layer to a single sender does not achieve this goal. However, the following packet assignment strategy can cope with slow senders: once the number of packets assigned to each sender (k_i) is determined based on its throughput, the ordered list of packets is distributed among active senders in a *weighted round-robin* fashion. For example, if three senders s_0, s_1 and s_2 contribute 50%, 30% and 20% the overall throughput, the ordered list of packets is divided among them in a round-robin fashion - that means we repeatedly assign $rrbase$*0.5 packets to s_0, next $rrbase$*0.3 packets to s_1, and next $rrbase$*0.2 packets to s_2. This strategy attempts to proportionally distribute less important packets (*i.e.*, packets of higher layers with higher timestamp) at the end of all lists. These packets might have more opportunity to be delivered during the following windows if excess throughput becomes available. $rrbase$ is a configuration parameter for the packet assignment mechanism that controls size of a batch of packets that is being distributed among the senders in each round. The larger $rrbase$, the smaller the amount of control information, the smaller the number of senders that deliver packets of a given layer, and the more likely it is that the undelivered packets are the more important ones.
Limiting Overwriting Effect: The overwriting effect could affect slow senders when the synchronized requesting approach is used. If we use the above round-robin packet assignment strategy with the synchronized requesting approach, requested packets of higher layers are frequently overwritten (*i.e.*, ignored) which could lead to starvation of higher layers. To limit the negative impact of the overwriting effect, the above packet assignment strategy can be extended as follows: a window can be partitioned to two smaller windows with the length equal to $\frac{r-1}{r}$ and $\frac{1}{r}$ portions of the original window, then we can sequentially apply the weighted round-robin packet assignment strategy on the smaller windows with the order shown in Figure 4. The example in Figure 4 shows a scenario where a window of packets for all layers (*i.e.*, 10,8,5,6) was partitioned into two windows (8,6,4,5) and (2,2,1,1) where r = 0.2. In this approach the negative impact of overwriting affects all layers proportionally, thus the QA mechanism has a better control on the buffer state. r is a configuration parameter called *partitioning factor* and should be set based on the expected level of overwriting.

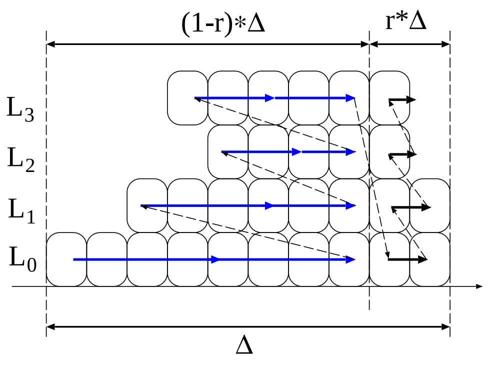

Figure 4: An example of window partitioning for packet assignment

4.6 Peer Selection

As we mentioned earlier, increasing the number of sender peers does not monotonically increase their overall effective throughput because two or more senders may share a bottleneck. PALS leverages this observation and uses a simple iterative mechanism to identify a subset of senders that maximize overall throughput as follows. The receiver starts with a randomly selected peer from the list of available peers. Then it periodically adds another random peer from the list of available peers to the subset of active senders while monitoring variations of both overall throughput and throughput of individual senders. If the overall throughput increases, the new sender is kept. Otherwise, the receiver drops the new sender and tries another random peer after a period. The impact of a new sender on overall throughput should be monitored over a sufficiently long period period in order to avoid reacting to other artifacts such as a transient congestion due to the startup phase, or similar peer selection experiments by other co-located receivers. Clearly, if the access link of the receiver is the bottleneck, changing the number of sender peers does not improve overall throughput.

5. EVALUATION

We have conducted preliminary evaluations of the PALS protocol using ns2 [18] simulation and present our initial results in this section. In our simulations, we have used a version of PALS with synchronized requesting approach and window partitioning for packet assignment. We have also simplified PALS by ignoring time-stamp of individual packets. For each period, either RFC or sliding window mechanism triggers QA to determine number of packets that should be delivered for each layer (rather than specific time-stamp) and uses the packet assignment mechanism to distribute required packets among senders. While our simulations can demonstrate dynamics of quality adaptation, packet assignment and sliding window mechanisms, they do not allow us to keep track of duplicate and late packets. Configuration parameters for PALS receiver are summarized in Table 2.

Figure 5 depicts our simulation scenario for a PALS session with

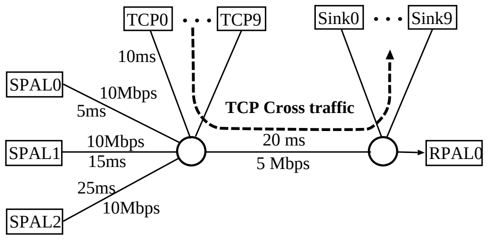

Figure 5: Simulation Setup

$EWMA\,Factor$	0.25
C	50 KByte/sec
$PktSize$	1000 Byte
Δ	4*SRTT
r	0.1
δ	0.2
α	2
$rrbase$	1000

Table 2: PALS Configuration Parameters

3 sender peers who cooperatively deliver a layered stream to a single receiver peer. Each sender uses RAP [3] for congestion control and has a different RTT to the receiver. Figure 6 shows the PALS mechanism in action when three sender peers share a bottleneck link with 10 TCP flows. The line with the most variations in Figure 6 is per-RTT average of overall throughput from all sender, a smoother line is EWMA of overall throughput and the step-wise line is the number of played back layers (*i.e.*, delivered quality) as a function of time. Furthermore, throughput of individual senders are also shown in this figure. Despite rather wide variations in per-RTT average of overall throughput, the QA mechanism manages to rapidly increase the number of playing layers up to 4 layers and then smoothly adjusts the delivered quality with variations of the overall throughput.

To examine the behavior of the QA mechanism in the presence of major changes in overall throughput, we repeated the previous simulation but added a CBR flow that starts at t=30sec and stops at t=60sec, and consumes 3 Mbps bandwidth of the bottleneck link. Figure 7 depicts the behavior of QA in PALS in the presence of the CBR source. This figure clearly shows that the throughput of sender peers (*i.e.*, bandwidth of RAP flows) quickly decreases in response to change in network condition. This in turn triggers the QA mechanism to quickly adjust the delivered quality by dropping all layers except the base layer. Once the CBR source stops and the bandwidth becomes available, the QA mechanism detects this changes and rapidly increases number of layers up to the previous level.

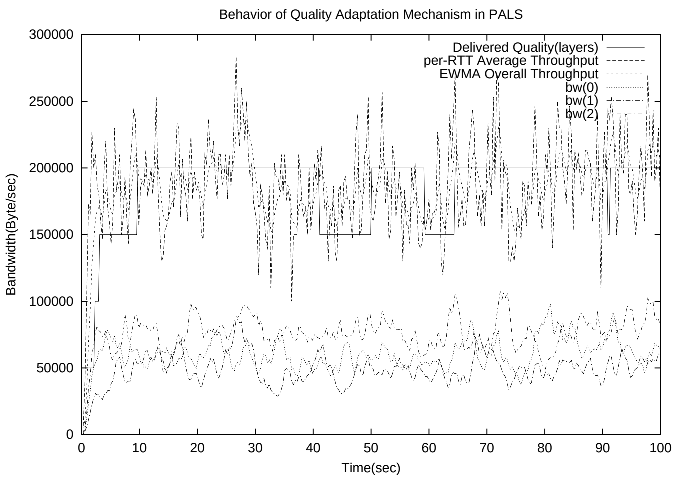

Figure 6: QA in PALS in the presence of TCP background traffic

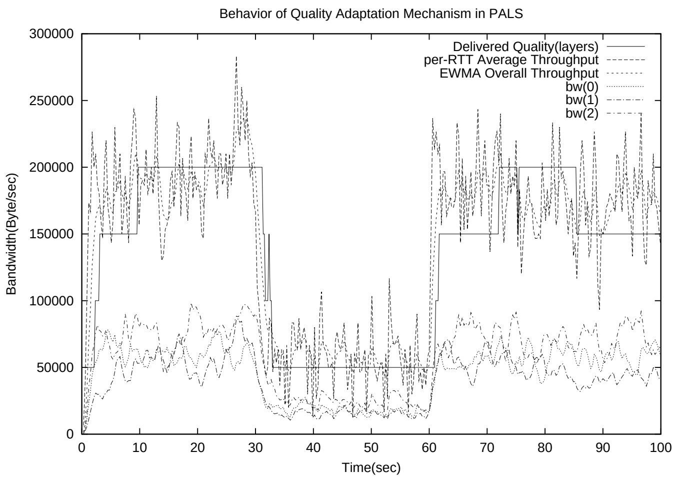

Figure 7: QA in PALS with major variations in background traffic

6. CONCLUSION AND FUTURE WORK

In this paper, we presented a new receiver-driven framework for P2P adaptive layered streaming, called PALS, where a receiver coordinates delivery of layer encoded stream from multiple senders. We described the framework, discussed the key components of the framework for coordination and adaptation, and addressed various design issues and tradeoffs. We also identified several challenging problems that arise in the design of such a receiver-driven mechanism for quality adaptive streaming.

This is obviously a starting point for our work on PALS. We plan to pursue this work in several directions. We are currently conducting extensive and detailed simulations to obtain a deeper understanding of the dynamics of different mechanisms in PALS, effect of various configuration parameters, and interactions among key components. In particular, we plan to explore various techniques to perform asynchronous requesting and incremental quality adaptation. We are also working on measurement-based techniques to derive pattern of variations in overall throughput from the incoming packets. We plan to examine our proposed peer selection mechanism. Once a peer selection mechanism is added to PALS, we can examine the impact of dynamics of peer participations on PALS performance. Finally, we plan to extend PALS to support delivery of VBR layered encoded as well as MD encoded streams.

Acknowledgments

We would like to thank the anonymous NOSSDAV reviewers for their feedback.

7. REFERENCES

[1] S. Ratnasamy, P. Francis, M. Handley, and S. Shenker, “A scalable content-addresable network,” in *Proceedings of the ACM SIGCOMM*, Aug. 2001.

[2] I. Stoica, R. Morris, D. Krager, F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *Proceedings of the ACM SIGCOMM*, Aug. 2001.

[3] R. Rejaie, M. Handley, and D. Estrin, “RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet,” in *Proceedings of the IEEE INFOCOM*, New York, NY., Mar. 1999.

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

[5] 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.

[6] V. N. Padmanabhan, H. J. Wang, P. A. Chou, and K. Sripanidkulchai, “Distributing streaming media content using cooperative networking,” in *Workshop on Network and Operating System Support for Digital Audio and Video*, Miami Beach, FL, 2002.

[7] J. Apostolopoulos, T. Wong, W-T. Tan, and S. Wee, “On multiple description streaming with content delivery networks,” in *Proceedings of the IEEE INFOCOM*, 2002.

[8] T. Nguyen and A. Zakhor, “Distributed video streaming over the internet,” in *SPIE Multimedia Computing and Networking*, Jan. 2002.

[9] “Abacast,” *http://www.abacast.com*.

[10] “chaincast,” *http://www.chaincast.com*.

[11] “allcast,” *http://www.allcast.com*.

[12] “vtrails,” *http://www.vtrails.com*.

[13] D. A. Tran, K. A. Hua, and T. Do, “Zigzag: An efficient peer-to-peer scheme for media streaming,” in *Proceedings of the IEEE INFOCOM*, 2003.

[14] S. McCanne, V. Jacobson, and M. Vettereli, “Receiver-driven layered multicast,” in *Proceedings of the ACM SIGCOMM*, Stanford, CA., Aug. 1996, pp. 117–130.

[15] Z. Miao and A. Ortega, “Expected run-time distortion based scheduling for delivery of scalable media,” in *Proc. of Packet Video Workshop 2002*, Pittsburgh, PA, Apr. 2002.

[16] H. Wang and A. Ortega, “Robust video communication by combining scalability and multiple description coding techniques,” in *EI 2003*, San Jose, CA, Jan. 2003.

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

[18] “Network simulator - ns(version 2),” *Software on-line*, 2002, http://www.isi.edu/nsnam/ns/.