Quality Adaptation for Congestion Controlled Video Playback over the Internet *

Reza Rejaie
Information Sciences Institute
University of Southern California
reza@isi.edu

Mark Handley
AT&T Center of Internet Research
The International Computer Science Institute
mjh@aciri.org

Deborah Estrin
Information Sciences Institute
University of Southern California
estrin@isi.edu

Abstract

Streaming audio and video applications are becoming increasingly popular on the Internet, and the lack of effective congestion control in such applications is now a cause for significant concern. The problem is one of adapting the compression without requiring video-servers to re-encode the data, and fitting the resulting stream into the rapidly varying available bandwidth. At the same time, rapid fluctuations in quality will be disturbing to the users and should be avoided.

In this paper we present a mechanism for using layered video in the context of unicast congestion control. This quality adaptation mechanism adds and drops layers of the video stream to perform long-term coarse-grain adaptation, while using a TCP-friendly congestion control mechanism to react to congestion on very short timescales. The mismatches between the two timescales are absorbed using buffering at the receiver. We present an efficient scheme for the distribution of buffering among the active layers. Our scheme allows the server to trade short-term improvement for long-term smoothing of quality. We discuss the issues involved in implementing and tuning such a mechanism, and present our simulation results.

1 Introduction

The Internet has been experiencing explosive growth of audio and video streaming. Most current applications involve web-based audio and video playback[6, 14] where stored video is streamed from the server to a client upon request. This growth is expected to continue, and such semi-realtime traffic will form a higher portion of the Internet load. Thus the overall behavior of these applications will have a significant impact on the Internet traffic.

Since the Internet is a shared environment and does not currently micro-manage utilization of its resources, end systems are expected to be cooperative by reacting to congestion properly and promptly[5]. Deploying end-to-end congestion control results in higher overall utilization of the network and improves inter-protocol fairness. A congestion control mechanism determines the available bandwidth based on the state of the network, and the application should then use this bandwidth efficiently to maximize the quality of the delivered service to the user.

Currently, many of the commercial streaming applications do not perform end-to-end congestion control. This is mainly because stored video has an intrinsic transmission rate. These rate-based applications either transmit data with a near-constant rate or loosely adjust their transmission rates on long timescales since the required rate adaptation for effective congestion control is not compatible with their nature. Large scale deployment of these applications could result in severe inter-protocol unfairness against TCP-based traffic and possibly even congestion collapse.

This paper is not about congestion control mechanisms, but about a complementary mechanism to adapt the quality of streaming video playback while performing congestion control. However, to design an effective quality adaptation scheme, we need to know the properties of the deployed congestion control mechanism. Our primary assumption is that the congestion control mechanism employs an additive increase, multiplicative decrease (AIMD) algorithm.

We previously designed a simple TCP-friendly congestion control mechanism, the Rate Adaptation Protocol (RAP)[17]. RAP is a rate-based congestion control mechanism and employs an AIMD algorithm in a manner similar to TCP. We assume RAP as the underlying congestion control mechanism because it's properties are relatively simple to predict. However, our proposed mechanisms can be applied with any congestion control scheme that deploys an AIMD algorithm.

Figure 1 shows the transmission rate of a RAP source over time. Similar to TCP, it hunts around for a fair share of the bandwidth. However unlike TCP, RAP is not ACK-clocked and variations of transmission rate have a more regular sawtooth shape. Bandwidth increases linearly for a period of time, then a packet is lost, and an exponential backoff occurs, and the cycle repeats.

1.1 Target Environment

Our target environment is a video server that simultaneously plays back different video streams on demand for many heterogeneous clients. As with current Internet video streaming, we expect the length of such streams to range from 30 second clips to full-length movies. The server and clients are connected through the Internet where the dominant competing traffic is TCP-based. Clients have

*This work was supported by DARPA under contract No. DABT63-95-C0095 and DABT63-96-C-0054 as part of SPT and VINT projects.

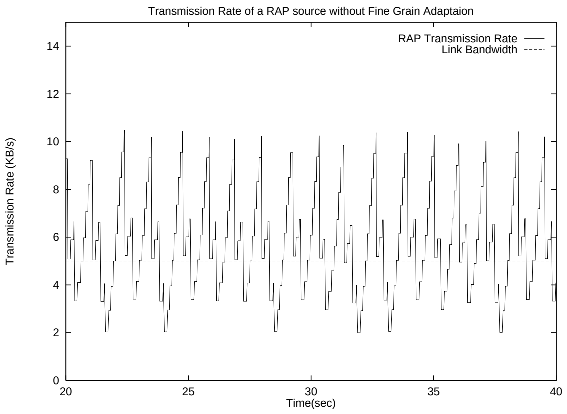

Figure 1: Transmission rate of a single RAP flow

heterogeneous network capacity and processing power. Users expect startup playback latency to be low, especially for shorter clips played back as part of web surfing. Thus pre-fetching an entire stream before starting its playback is not an option. We believe that this scenario reasonably represents many current and anticipated Internet streaming applications.

1.2 Motivation

If video for playback is stored at a single lowest - common - denominator encoding on the server, high-bandwidth clients will receive poor quality despite availability of a large amount of bandwidth. However, if the video is stored at a single higher quality encoding (and hence higher data rate) on the server, there will be many low-bandwidth clients that can not play back this stream. In the past, we have often seen RealVideo streams available at 14.4 Kb/s and 28.8 Kb/s, where the user can choose their connection speed. However, with the advent of ISDN, ADSL, and cable modems to the home, and faster access rates to businesses, the Internet is becoming much more heterogeneous. Customers with higher speed connections feel frustrated to be restricted to modem-speed playback. Moreover, the network bottleneck may be in the backbone, such as at provider interconnects or links to the server itself. In this case, the user can not know the congestion level and congestion control mechanisms for streaming video playback are critical.

Given a time varying bandwidth channel due to congestion control, the server should be able to adjust the quality of the stream it plays back so that the perceived quality is as high as the available network bandwidth will permit. We term this *quality adaptation*.

1.3 Quality Adaptation Mechanisms

There are several ways to adjust the quality of a pre-encoded stored stream, including: adaptive encoding, switching among multiple pre-encoded versions, and hierarchical encoding.

One may re-quantize stored encodings on-the-fly based on network feedback[1, 15, 20]. However, since encoding is CPU - intensive, servers are unlikely to be able to do this for large numbers of clients. Furthermore, once the original data has been stored compressed, the output rate of most encoders can not be changed over a wide range.

In an alternative approach, the server keeps several versions of each stream with different qualities. As available bandwidth changes, the server plays back streams of higher or lower quality as appropriate.

With hierarchical encoding[8, 10, 12, 21], the server maintains a layered encoded version of each stream. As more bandwidth becomes available, more layers of the encoding are delivered. If the average bandwidth decreases, the server may then drop some of the layers being transmitted. Layered approaches usually have the decoding constraint that a particular enhancement layer can only be decoded if all the lower quality layers have been received.

There is a duality between adding or dropping of layers in the layered approach and switching streams in the multiply-encoded approach. However the layered approach is more suitable for caching by a proxy for heterogeneous clients[18]. In addition, it requires less storage at the server, and it provides an opportunity for selective retransmission of the more important information. The design of a layered approach for quality adaptation primarily entails the design of an efficient add and drop mechanism that maximizes quality while minimizing the probability of base-layer buffer underflow.

The rest of this paper is organized as follows: first we provide an overview of the layered approach to quality adaptation and then explain coarse-grain adding and dropping mechanisms in section 2. We also discuss fine-grain inter-layer bandwidth allocation for a single backoff scenario. Section 3 motivates the need for smoothing in the presence of real loss patterns and discusses two possible approaches. In section 4, we sketch an efficient filling and draining mechanism that not only achieves smoothing but is also able to cope efficiently with various patterns of losses. We evaluate our mechanism through simulation in section 5. Section 6 briefly reviews related work. Finally, section 7 concludes the paper and addresses some of our future plans.

2 Layered Quality Adaptation

Hierarchical encoding provides an effective way that a video playback server can coarsely adjust the quality of a video stream without transcoding the stored data. However, it does not provide fine-grained control over bandwidth, i.e. bandwidth changes at the granularity of a layer. Furthermore, there needs to be a quality adaptation mechanism to smoothly adjust the quality (i.e. number of layer) as bandwidth changes. Users will tolerate poor quality video, but rapid variations in quality are disturbing.

Hierarchical encoding allows video quality adjustment over long periods of time, whereas congestion control changes the transmission rate rapidly over short time intervals (several round-trip times,(RTTs)). The mismatch between the two timescales is made up for by buffering data at the receiver to smooth the rapid variations in available bandwidth and allow a near constant number of layers to be played.

Figure 2 graphs a simple simulation of a quality adaptation mechanism in action. The top graph shows the available network bandwidth and the consumption rate at the receiver with no layers being consumed at startup, then one layer, and finally two layers. During the simulation, two packets are dropped and cause congestion control backoffs, when the transmission rate drops below the consumption rate for a period of time. The lower graph shows the playout sequence numbers of the actual packets against time. The horizontal lines show the period between arrival time and playout time of a packet. Thus it indicates the total amount of buffering for each layer. This simulation shows more buffered data for Layer 0 (the base layer) than for Layer 1 (the enhancement layer). After the first backoff, the length of these lines decreases indicating

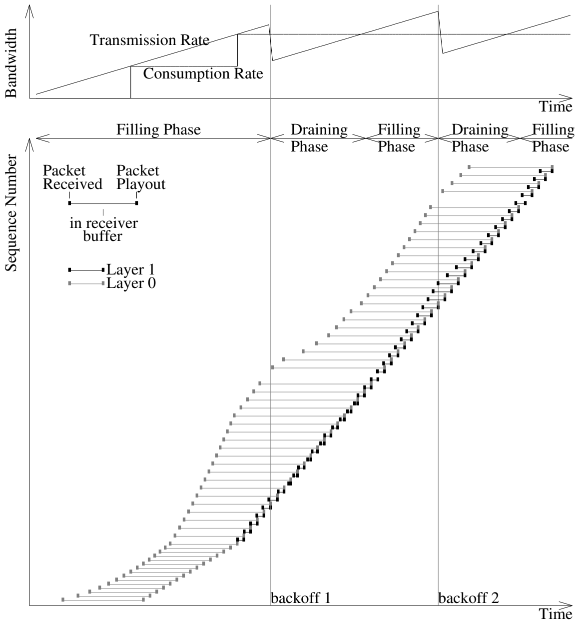

Figure 2: Layered encoding with receiver buffering

buffered data from Layer 0 is being used to compensate for the lack of available bandwidth. At the time of the second backoff, a little data has been buffered for Layer 1 in addition to the large amount for Layer 0. Thus data is drawn from both buffers properly to compensate for the lack of available bandwidth.

The congestion control mechanism dictates the available bandwidth [1]. We can not send more than this amount, and do not wish to send less[2]. In a real network even the average bandwidth of a congestion controlled flow changes over the session lifetime. Thus a quality adaptation mechanism must continuously evaluate the available bandwidth and adjust the number of active layers accordingly.

In this paper we assume that the layers are linearly spaced - that is each layer has the same bandwidth. This simplifies the analysis, but is not a requirement. In addition, we assume each layer has a constant consumption rate over time. In practice this is unlikely in a real codec, but to a first approximation it is reasonable. It can be ignored by slightly increasing the amount of receiver buffering for all layers to absorb variations in consumption rate.

Figure 3 shows a single cycle of the congestion control mechanism. The sawtooth waveform is the instantaneous transmission rate. There are n_a active layers, each of which has a consumption rate of C. In the left hand side of the figure, the transmission rate is higher than the consumption rate, and this data will be stored temporarily in the receiver's buffer. The total amount of stored data is equal to the area of triangle abc. Such a period of time is known as a *filling phase*. Then, at time t_b, a packet is lost and the transmit rate is reduced multiplicatively. To continue playing out n_a layers when the transmission rate drops below the consumption rate, some data must be drawn from the receiver buffer until the transmission rate reaches the consumption rate again. The amount of data drawn from the buffer is shown in this figure as triangle

[1] Available bandwidth and transmission rate are used inter-changeably throughout this paper.

[2] For simplicity we ignore flow control issues in this paper but implementations should not. However our final solutions generally require so little receiver buffering that this is not often an issue.

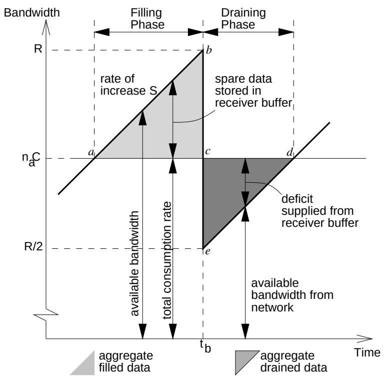

Figure 3: Filling and draining phase

cde. Such a period of time is known as a *draining phase*.

Note that the quality adaptation mechanism can *only* adjust the number of active layers and their bandwidth share. This paper attempts to derive efficient behavior for these two key mechanisms:

- A *coarse-grain* mechanism for adding and dropping layers. By changing the number of active layers, the server can perform coarse-grain adjustment on the total amount of receiver-buffered data.
- A *fine-grain* inter-layer bandwidth allocation mechanism among the active layers. If there is receiver-buffered data available for a layer, we can temporarily allocate less bandwidth than is being consumed while taking the remainder from the buffer. This smoothes out reductions in the available bandwidth. When spare bandwidth is available, we can send data for a layer at a rate higher than its consumption rate, and increase the data buffered for that layer at the receiver.

In the next section, we present coarse-grain adding and dropping mechanisms as well as their relation to the fine-grain bandwidth allocation. Then we discuss the fine-grin bandwidth allocation in the subsequent sections.

2.1 Adding a Layer

A new layer can be added as soon as the instantaneous available bandwidth exceeds the consumption rate (in the decoder) of the existing layers. The excess bandwidth could then be used to start buffering a new layer. However, this would be problematic as without knowing future available bandwidth we can not decide when it will first be possible to start decoding the layer. The new layer's *playout* is decided by the inter-layer timing dependency between its data and that in the base layer. Therefore we can not make a reasoned decision about which data from the new layer to actually send [3].

A more practical approach is to start sending a new layer when the instantaneous bandwidth exceeds the consumption rate of the existing layers plus the new layer. In this approach the layer can start to play out immediately. In this case there is some excess bandwidth from the time the available bandwidth exceeds the consumption rate of the existing layers until the new layer is added. This excess bandwidth can be used to buffer data for existing layers at the receiver.

[3] Note that once the inter-layer timing for a new layer is adjusted, it is maintained as long as the buffer does not dry out.

In practice, this bandwidth constraint for adding is still not conservative enough, as it may result in several layers being added and dropped with each cycle of the congestion control sawtooth. Such rapid changes in quality would be disconcerting for the viewer. One way to prevent rapid changes in quality is to add a buffering condition such that adding a new layer does not endanger existing layers. Thus, the server may add a new layer when:

1. The instantaneous available bandwidth is greater than the consumption rate of the existing layers plus the new layer, and,
2. There is sufficient total buffering at the receiver to survive an immediate backoff and continue playing all the existing layers plus the new layer.

To satisfy the second condition we assume (for now) that no additional backoff will occur during the draining phase, and the slope of linear increase can be properly estimated.

These are the minimal criteria for adding a new layer. If these conditions are held a new layer can be kept for a reasonable period of time during the normal congestion control cycles. We shall show later that we normally want to be even more conservative than this. Clearly we need to have sufficient buffering at the receiver to smooth out variations in the available bandwidth so that the number of active layers does not change due to the normal hunting behavior of the congestion control mechanism.

Expressing the adding conditions more precisely:

Condition 1: $R > (n_a + 1)C$

Condition 2: $\sum_{i=0}^{n_a-1} buf_i \geq \frac{((n_a+1)C - \frac{R}{2})^2}{2S}$

where R is the current transmission rate
n_a is the number of currently active layers
buf_i is the amount of buffered data for layer i
S is the rate of linear increase in bandwidth (typically one packet per RTT)

2.2 Dropping a Layer

Once a backoff occurs, if the total amount of buffering at the receiver is less than the estimated required buffering for recovery, (i.e, the area of triangle cde in figure 3), the correct course of action is to immediately drop the highest layer. This reduces the consumption rate ($n_a C$) and hence reduces the buffer requirement for recovery. If the buffering is still insufficient, the server should iteratively drop the highest layer until the amount of buffering is sufficient. This rule clearly doesn't apply to the base layer which is always sent.

Expressing the dropping mechanism more precisely:

$$\text{WHILE } \left(n_a C > R + \sqrt{2S \sum_{i=0}^{n_a-1} buf_i}\right)$$

DO $n_a = n_a - 1$

This mechanism provides a coarse-grain criteria for dropping a layer. However, it may be insufficient to prevent buffer underflow during the draining phase for one of the following reasons:

- We may suffer a further backoff before the current draining phase completes.
- Our estimate of the slope of linear increase may be incorrect if the network RTT changes substantially.
- There may be sufficient total data buffered, but it may be allocated among the different layers in a manner that precludes its use to aid recovery.

The first two situations are due to incorrect prediction of the amount of buffered data needed to recover, and we term such an event a *critical situation*. In such events, the only appropriate course of action is to drop additional layers as soon as the critical situation is discovered.

The third situation is more problematic, and relates to the fine-grain bandwidth allocation among active layers during both filling and draining phases. We devote much of the rest of this paper to deriving and evaluating a near-optimal solution to this situation.

2.3 Inter-layer Buffer Allocation

Because of the decoding constraint in hierarchical coding, each additional layer depends on all the lower layers, and correspondingly is of decreasing value. Thus a buffer allocation mechanism should provide higher protection for lower layers by allocating a higher share of buffering for them.

The challenge of inter-layer buffer allocation is to ensure the total amount of buffering is sufficient, and that is properly distributed among active layers to effectively absorb the short-term reductions in bandwidth that might occur. The following two examples illustrate ways in which improper allocation of buffered data might fail to compensate for the lack of available bandwidth.

- **Dropping layers with buffered data:** A simple buffer allocation scheme might allocate an equal share of buffer to each layer. However, if the highest layer is dropped after a backoff, its buffered data is no longer able to assist the remaining layers in the recovery. The top layer's data will still be played out, but it is not providing buffering functionality. This implies that it is more beneficial to buffer data for lower layers.
- **Insufficient distribution of buffered data:** An equally simple buffer allocation scheme might allocate all the buffering to the base layer. Consider an example when three layers are playing, where a total consumption rate of $3C$ must be supplied for the receiver's decoder. If the transmission rate drops to C, the base layer (L_0) can be played from its buffer. Since neither L_1 nor L_2 has any buffering, they require transmission from the source. However available bandwidth is only sufficient to feed one layer. Thus L_2 must be dropped *even if the total buffering were sufficient for recovery.*

In these examples, although buffering is available, it can not be used to prevent the dropping of layers. This is *inefficient* use of the buffering. In general, we are striving for a distribution of buffering that is most *efficient* in the sense that it provides maximal protection against dropping layers for any likely pattern of short-term reduction in available bandwidth.

These examples reveal the following tradeoffs for inter-layer buffer allocations:

- Allocating more buffering for the lower layers not only improves their protection but it also increases *efficiency* of buffering.
- Buffered data for each layer can not provide more than its consumption rate(i.e. C). Thus there is a minimum number of buffering layers that are needed to cope with short-term reductions in available bandwidth for successful recovery. This minimum is directly determined by the reduction in bandwidth that we intend to absorb by buffering.

Expressing this more precisely:

$$\mathrm{n}_b = \left\lceil n_a - \frac{R}{2C} \right\rceil ; \quad n_a > \frac{R}{2C}$$

$$n_b = 0 ; \quad n_a \le \frac{R}{2C}$$

where n_b is the minimum number of buffering layers

R is the transmission rate (before a backoff)

2.4 Optimal Inter-layer Buffer Allocation

Given a draining phase following a single backoff, we can derive the optimal inter-layer buffer allocation that maximizes buffering efficiency. Figure 4 illustrates an optimal buffer allocation and its corresponding draining pattern for a draining phase. Here we assume that the total amount of buffering at the receiver at time t_b is precisely sufficient for recovery(i.e. area of triangle afg) with no spare buffering available at the end of the draining phase.

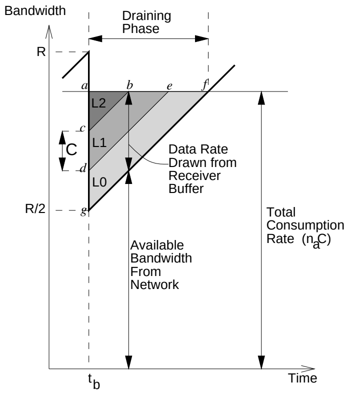

Figure 4: The optimal inter-layer buffer distribution

To justify the optimality of this buffer allocation, consider that the consumption rate of a layer must be supplied either from the network or from the buffer or a combination of the two. If it is supplied entirely from the buffer, that layer's buffer is draining at consumption rate C. The area of quadrilateral $defg$ in figure 4 shows the maximum amount of buffer that can be drained from a single layer during this draining phase. If the draining phase ends as predicted, there is no preference as to buffer distribution among active layers as long as no layer has more than $defg$ worth of buffered data. However, if the situation becomes critical due to further backoffs, layers must be dropped. Allocating area $defg$ of buffering to the base layer would ensure that the maximum amount of the buffered data is still usable for recovery, and maximizes buffering efficiency.

By similar reasoning, the next largest amount an additional layer's buffer can contribute is quadrilateral $bcde$, and this portion of buffered data should be allocated to L_1, the first enhancement layer, and so on. This approach minimizes the amount of buffered data allocated for higher layers that might be dropped in a critical situation and consequently maximizes buffering efficiency.

The optimal amount of buffering for layer i is:

$$Buf_{i,opt} = \frac{C}{2S}(C(2n_a - 2i - 1) - R) ; \quad i < n_b - 1$$

$$Buf_{i,opt} = \frac{C}{2S}(n_a C - \frac{R}{2} - iC)^2 ; \quad i = n_b - 1$$

Although we can calculate the optimal allocation of buffered data for the active layers, a backoff may occur at any random time. To tackle this problem, during the filling phase, we incrementally adjust the allocation of buffered data so that the buffer state always remains as close as possible to an optimal state.

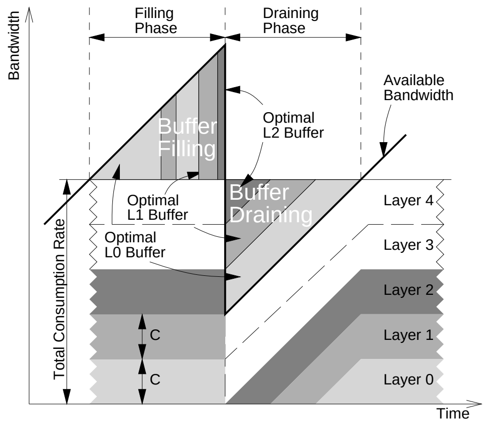

Figure 5: Optimal buffer sharing

Toward that goal, we assume that a single backoff will occur immediately, and ask the question: "if we keep only the base layer, is there sufficient buffering to survive?". If there is not sufficient buffering, then we fill up the base layer's buffer until it has enough buffering to survive a single backoff. Then we ask the question: "if we keep only two layers, is there enough buffering to survive with those buffers having optimal allocation?". If there is not enough base layer data, we fill the base layer's buffer up to the optimal level. Then we start sending L_1 data until both layers have the optimal amount of buffering to survive. We repeat this process and increase the number of expected surviving layers until all the buffering layers are filled up to an optimal level such that all active layers can survive from a single backoff. This approach results in a sequential filling pattern among buffering layers.

Figure 5 illustrates the optimal filling and draining scheme for a single backoff. If a backoff occurs exactly at time t_b, all layers can survive the backoff. Occurrence of a backoff earlier than t_b results in dropping one or more active layers. However the buffer state is always as close as possible to the optimal state without those layers. If no backoff occurs until adding conditions (section 2.1) are satisfied, a new layer is added and we repeat the sequential filling mechanism.

It is worth mentioning that the server can control the filling and draining pattern by proper fine-grain bandwidth allocation among active layers. Figure 5 illustrates that at each point of time during the draining phase, bandwidth share plus draining rate for each layer is equal to its consumption rate. Thus maximally efficient buffering results in the upper layers being supplied from the network during the draining phase while the lower layers are supplied from their buffers. For example, just after the backoff, layer 2 is supplied entirely from the buffer, but the amount supplied from the buffer decreases to zero as data supplied from the network takes over. Layers 0 and 1 are supplied from the buffer for longer periods.

3 Smoothness Constraints

In the previous section, we derived an optimal filling and draining scheme based on the assumption that we only buffer to survive a single backoff with all the layers intact. However, examination of Internet traffic indicates that real networks exhibit near-random[2] loss patterns with frequent additional backoffs during a draining phase. Thus, aiming to survive only a single backoff is too aggressive and results in frequent adding and dropping of layers.

3.1 Smoothing

To achieve reasonable smoothing of the add and drop rate, an obvious approach is to refine our adding conditions (in section 2.1) to be more conservative. We have considered the following two mechanisms to achieve smoothing:

- We may add a new layer if the *average* available bandwidth is greater than the consumption rate of the existing layers plus the new layer.
- We may add a new layer if we have sufficient amount of buffered data to survive K_{max} backoffs with existing layers, where K_{max} is a *smoothing factor* with value greater than one.

Although each one of these mechanisms results in smoothing, the latter not only allows us to directly tie the adding decision to appropriate buffer state for adding, but it can also utilize limited bandwidth links effectively. For example, if there is sufficient bandwidth across a modem link to receive 2.9 layers, the average bandwidth would never become high enough to add the third layer. In contrast, the latter mechanism would send 3 layers for 90% of the time which is more desirable. For the rest of this paper we assume that the only condition for adding a new layer is availability of optimal buffer allocation for recovery from K_{max} backoffs.

Changing K_{max} allows us to tune the balance between maximizing the short-term quality and minimizing the changes in quality. An obvious question is "What degree of smoothing is appropriate?" In the absence of a specific layered codec and user-evaluation, K_{max} can not be analytically derived. Instead it should be set based on real-world user perception experiments to determine the appropriate degree of smoothing that is not disturbing to the user. In practice, we probably also want to base K_{max} on the average bandwidth and RTT since these determine the duration of a draining phase.

3.2 Buffering Revisited

If we delay adding a new layer to achieve smoothing, this affects the way we fill and drain the buffers. Figure 6 demonstrates this issue.

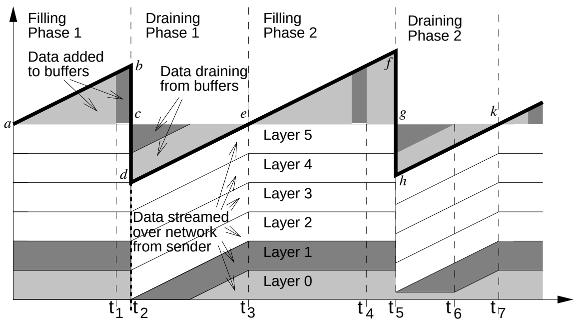

Figure 6: Revised draining phase algorithm

Up until time t_3, this is the same as figure 5. The second filling phase starts at time t_3, and at t_4 there is sufficient buffering to survive a backoff. However, for smoothing purposes, a new layer is not added at this point and we continue buffering data until a backoff occurs at t_5.

Note that as the available bandwidth increases, the total amount of buffering increases but the required buffering for recovery from a single backoff decreases. At time t_5, we have more buffering than we need to survive a single backoff, but insufficient buffering to survive a second backoff before the end of the draining phase. We need to specify how we allocate the extra buffering after time t_4, and how we drain these buffers after t_5 while maintaining efficiency.

Conceptually, during the filling phase, the server sequentially examines the following steps:

Step 1: enough buffer for one backoff with L_0 intact.
Step 2: enough buffer for one backoff with L_0 and L_1.
...
Step n_a: enough buffer for one backoff with L_0 through L_{n_a-1} intact.
Step n_a+1: enough buffer for one backoff with L_0 through layer L_{n_a-1} intact and two backoffs with L_0 intact.

At any point in the filling phase we have satisfied one step and are working towards the next step.

When a backoff occurs between steps, in this case between steps n_a and $n_a + 1$, we essentially reverse the filling process. First we identify between which two steps we're currently located. Then we traverse through the steps in the reverse order to determine which layers must be drained and by how much. In essence, during consecutive filling and draining phases, we traverse this sequence of steps (i.e. optimal buffer states) back and forth such that at any point of time the buffer state is as close to optimal as possible. In the next section, we describe this mechanism in more detail.

4 Buffer Allocation with Smoothing

To design an efficient filling and draining mechanisms in the presence of smoothing, we need to know the optimal buffer allocation among layers and the corresponding maximally efficient filling and draining patterns for multiple-backoff scenarios.

The optimal buffer allocation for a scenario with multiple backoffs is not unique because it depends on the time when the additional backoffs occur during the draining phase. If we have knowledge of future loss distribution patterns it might, in principle, be possible to calculate the optimal buffer allocation. In practice such a solution would be excessively complex for the problem it is trying to solve, and rapidly becomes intractable as the number of backoffs increases. Let us first assume that only one additional backoff occurs during the draining phase. The possible scenarios are shown in figure 7. This figure illustrates that the optimal buffer allocation for each scenario depends on the time of the second backoff, the consumption rate, and the transmission rate before the first backoff.

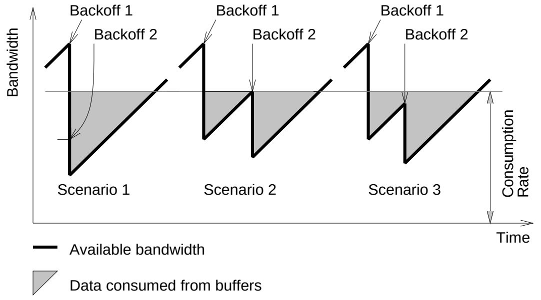

Figure 7: Possible double-backoff scenarios

We can extend the idea of optimal buffer allocation for a single backoff (section 2.4) to each individual scenario. Added complexity arises from the fact that different scenarios require different buffer allocations. For an equal amount of the total buffering needed for recovery, scenarios 1 and 2 are two extreme cases in the sense that they need the maximum and minimum number of buffering layers respectively. Thus addressing these two extreme scenarios efficiently should cover all the intermediate scenarios (e.g. scenario 3) as well.

We need to decide which scenario to consider during the filling phase. We make a key observation here. If the total amount of buffering for scenarios 1 and 2 are equal, having the optimal buffer distribution for scenario 1 is sufficient for recovery from scenario 2, although it is not maximally efficient. However, the converse is not feasible. The higher flexibility in scenario 1 comes from the fact that this scenario needs a larger number of buffering layers than does scenario 2. Thus, if we have a buffer distribution that can recover from a scenario 1, we will be able to cope with a scenario 2 that has the same total buffer requirement, but not vice versa.

This suggests that during the filling phase for the two backoff scenarios, first we consider the optimal buffer allocation for scenario 1 and fill up the buffers in a step by step sequential fashion as described in section 3.2. Once this is achieved, then we move on to consider scenario 2.

4.1 Filling Phase with Smoothing

To extend this idea to scenarios of k backoffs, we need to examine the optimal buffer allocation for scenario 1 and 2 for each successive value of k. Figure 8 illustrates the optimal buffer state, including the total buffer requirement and its optimal inter-layer allocation in scenario 1 and 2, for different values of k. Ideally, we would like to fill the buffers during the filling phase such that we traverse through these buffer states in turn. Once k exceeds K_{max} (the smoothing factor), then we add a new layer and start the process again with the new sets of optimal buffer states.

Toward this goal, we order these different buffer states in increasing value of total amount of required buffering in figure 9. Thus by traversing this sequence of buffer states, we always work towards the next optimal state that requires more buffering.

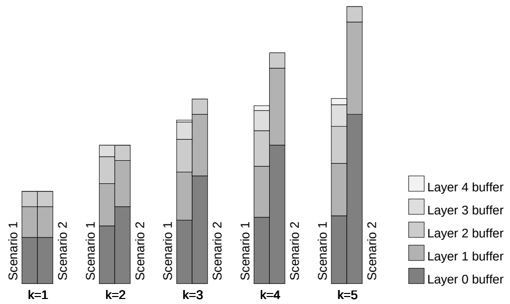

Figure 8: Buffer distributions for k backoffs

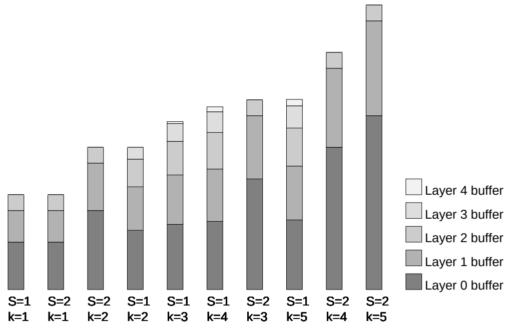

Figure 9: Distributions in increasing order of buffering

Unfortunately this requires us to occasionally drain an existing buffer in order to reach the next state[4]. Two examples of this phenomenon are visible in figure 9:

- Moving from the {scenario 2, k=2} case to the {scenario 1, k=2} case involves draining L_0's buffer.
- Moving from the {scenario 1, k=4} case to the {scenario 2, k=3} case involves draining L_3's buffer.

We do not want to drain any layer's buffer during the filling phase because that buffering provides protection for a previous scenario that we have already passed. Thus we seek the maximally efficient sequence of buffer states *that is consistent with the existing buffering*. The total amount of required buffering and the per layer buffer requirement must be monotonically increasing as we go to the next buffer state.

The key observation that we mentioned earlier allows us to calculate such a sequence. We recall that having the optimal buffer distribution for scenario 1 is sufficient for recovery from scenario 2, although it is not maximally efficient. Given this flexibility, the solution is to constrain per layer buffer allocation in each scenario-2 state to be no less than the previous scenario-1 state, and no more than the next scenario-1 state (in the sequence of states in figure 9). Figure 10 depicts a sequence of maximally efficient buffer states after applying the above constraints where each step in the filling process is numbered. By enforcing this constraint, we can traverse through the buffer states such that buffer allocation for each state satisfies the buffer requirement for all the previous states. This implies that both the total amount of buffering and the amount of per layer buffering monotonically increase. Thus the per layer buffering can always be used to aid recovery. Once we have sufficient buffering for recovery from K_{max} backoffs in both scenarios, a new layer will be added.

[4] This means that the order of these states based on increasing value of total required buffering is different from their order based on increasing value of per layer buffering.

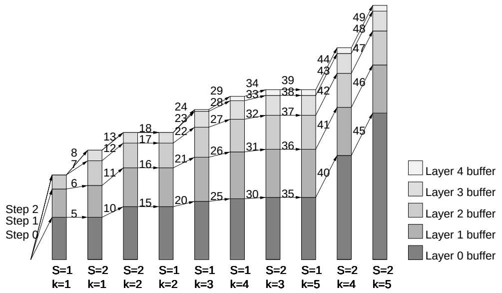

Figure 10: Step-by-step buffer filling

The following pseudo-code expresses our per-packet algorithm to ensure that buffer state remains maximally efficient during the filling phase[5]:

```
FUNCTION SendPacket

  S1Backoffs = 0; S2Backoffs = 0
  BufReq1 = 0; BufReq2 = 0

  WHILE (BufReq1 < TotBufAvailable) AND (S1Backoffs < K_max)
      INCREMENT S1Backoffs
      BufReq1 = TotalBufRequired(CurrentRate, Scenario=1,
                                      S1Backoffs, ActiveLayers)

  WHILE (BufReq2 < TotBufAvailable)
      INCREMENT S2Backoffs
      BufReq2 = TotalBufRequired(CurrentRate, Scenario=2,
                                      S2Backoffs, ActiveLayers)

  FOR Layer = 1 TO ActiveLayers
      LayerBuf1 = BufRequired(CurrentRate, Scenario=1,
                                  S1Backoffs, Layer, ActiveLayers)
      LayerBuf2 = BufRequired(CurrentRate, Scenario=2,
                                  S2Backoffs, Layer, ActiveLayers)
      IF (BufReq1 < BufReq2) AND (S1Backoffs < K_max)
                                          #We're considering scenario 1
          IF (LayerBuf1 > BufAvailable(Layer)
              SendPacketFromLayer(Layer)
              RETURN

      ELSE                                #We're considering scenario 2
          IF (LayerBuf2 > BufAvailable(Layer)) AND
             ((S1Backoffs > K_max) OR
             (LayerBuf1 < BufAvailable(Layer)))
                  SendPacketFromLayer(Layer)
                  RETURN
```

K_{max} is the smoothing factor, giving the number of backoffs for which we buffer data before adding a new layer.

The function TotalBufRequired returns the total amount of required buffering for all layers in the scenario in question, given the current sending rate, the number of active layers, and the number of backoffs being considered.

[5] The algorithm performs fine-grain bandwidth allocation by assigning the next transmitting packet to a particular layer.

TotalBufRequired()

Scenario 1

$$Buf_{total} = 0 \quad ; \quad k \leq \log_2 \frac{R}{n_a C}$$

$$Buf_{total} = \frac{1}{2S}\left(n_a C - \frac{R}{2^k}\right)^2 ; \quad k > \log_2 \frac{R}{n_a C}$$

where k is the number of backoffs being considered

Scenario 2

$$Buf_{total} = 0 \quad ; \quad k \leq \log_2 \frac{R}{n_a C}$$

$$Buf_{total} = \frac{1}{2S}\left(\left(n_a C - \frac{R}{2^{k_1}}\right)^2 + (k - k_1)\left(\frac{n_a C}{2}\right)^2\right)$$

$$k_1 = \left\lceil \log_2 \frac{R}{n_a C} \right\rceil \quad ; \quad k > \log_2 \frac{R}{n_a C}$$

The function BufRequired returns the maximally efficient amount of required buffering for a particular layer in the scenario of the state we are currently working towards. The input parameters for this function are: the layer number, the current sending rate, the number of active layers, and the number of backoffs being considered.

BufRequired()

Scenario 1

$$Buf_{i,opt} = 0 \; ; \quad k \leq \log_2 \frac{R}{n_a C}$$

$$Buf_{i,opt} = \frac{C}{2S}\left(C(2n_a - 2i - 1) - \frac{R}{2^{k-1}}\right)$$

$$k > \log_2 \frac{R}{n_a C} \; ; \quad 0 \leq i < n_b$$

Scenario 2

$$Buf_{i,opt} = 0 \; ; \quad k \leq \log_2 \frac{R}{n_a C}$$

$$Buf_{i,opt} = \frac{C}{2S}\Bigg(\left(C(2n_a - 2i - 1) - \frac{R}{2^{k_1 - 1}}\right) + (k - k_1)C(n_a - 2i - 1)\Bigg)$$

$$k > \log_2 \frac{R}{n_a C} \; ; \quad 0 \leq i < n_b$$

4.2 Draining Phase with Smoothing

As we traverse through the maximally efficient states, one or more backoffs eventually move us into a draining phase. Given that we incrementally traverse the maximally efficient *path* of buffer states during the filling phase, we would like to traverse the same path, but in the reverse direction, during the draining phase. This approach guarantees that the highest layer buffers are not drained until they are no longer required, and the lowest layer buffers are not drained too early.

At the start of each step we have an efficient amount of protective buffering for one particular state, and regressively work toward the previous maximally efficient buffer state along the maximally efficient path. However, there is an additional constraint that we can not drain a layer's buffer faster than the layer consumption rate (i.e. C).

To achieve such a draining pattern, we periodically calculate the draining pattern for a short period of time, during which we expect to drain a certain number of packets. This number is based on the current estimate of slope of linear increase and the current consumption rate. We then calculate (using an algorithm similar to the above pseudo-code) the previous optimal state along the maximally efficient path that we can achieve with the current amount of buffering. Conceptually, then we consider draining data from each layer in turn, starting from the highest layer and working downwards, such that each layer's buffering does not drop below its buffer share at the previous optimal step we are draining towards. An added constraint is that we must limit the amount of drained data from a layer to the maximum amount that can be consumed during this period. If the buffer state reaches the previous optimal state being considered before we have allocated the number of packets that must be drained in this period, then we move on to consider the previous state along the maximally efficient path and so on. We repeat this process until a sufficient number of packets for draining during this period are identified. Then we allocate the bandwidth during the period such that each active layer receives the total amount of data that it must consume during this period, minus those packets we just allocated to drain during the period.

5 Simulation

We have evaluated our quality adaptation mechanism through simulation using bandwidth traces obtained from RAP in the ns2 [3] simulator and real Internet experiments.

Figure 11 provides a detailed overview of the mechanisms in action. It shows a 40 second trace where the quality-adaptive RAP flow co-exists with 10 Sack-TCP flows and 9 additional RAP flows through an 800 KB/s bottleneck with 40ms RTT. The smoothing factor was set to 2 so that it provides enough receiver buffering for two backoffs before adding a new layer(K_{max} = 2). The consumption rate of each layer(C) is equal to 10 KB/s.

Figure 11 shows the following parameters:

- The total transmission rate, illustrating the saw-tooth output of RAP. We have also overlaid the consumption rate of the active layers over the transmission rate to demonstrate the add and drop mechanism.
- The transmission rate broken down into bandwidth per layer. This shows that most of the variation in available bandwidth is absorbed by changing the rate of the lowest layers (shown with the light-gray shading).
- The individual bandwidth share per layer. Periods when a layer is being streamed above its consumption rate to build up receiver buffering are visible as spikes in the bandwidth.
- The buffer drain rate per layer. Clearly visible are points where the buffers are used for playout because the bandwidth share is temporarily less than the layer consumption rate.
- The accumulated buffering at the receiver for each active layer.

Graphs in figure 11 demonstrate that the short-term variations in bandwidth caused by the congestion control mechanism can be effectively absorbed by receiver buffering. Furthermore, playback quality is maximized without risking complete dropouts in the playback due to buffer underflow.

Smoothing Factor

To examine the impact of smoothing factor on the behavior, we repeated the previous simulation with different values of K_{max}. Figure 12 shows the number of active layers and buffer allocation across active layers for K_{max}=2, K_{max}=3, and K_{max}=4. As expected, higher values of K_{max} reduce the number of changes in quality at the expense of increasing the time it takes to first achieve the best short-term quality. This manifests itself in two ways. As K_{max} increases, first the total amount of buffering is increased. Second, more of the buffering is allocated for higher layers to cope with the larger variations in available bandwidth as a result of successive backoffs.

Responsiveness

We have also explored the responsiveness of the quality adaptation mechanism to large step changes in available bandwidth. Figure 13 depicts a RAP trace with the same parameters as figure 11 but a CBR source with a rate equal to half of the bottleneck bandwidth is started at t=30s and stopped at t=60s and K_{max}=4. The RAP congestion control mechanism rapidly responds to these changes by adjusting the average transmission rate. The quality adaptation mechanism closely follows the changes in bandwidth. L_3 and then L_2 are dropped when bandwidth reduces and then $L2$ is added when bandwidth becomes available again. Notice that every layer's buffer is involved in this process, but the reception of the base layer is never jeopardized. Thus, we have satisfied our original design goal of providing smoothing of quality while providing protection to the most critical layers.

Efficiency

The performance of our algorithms can be examined from the efficiency of the buffer allocation. The inter-layer buffer allocation is maximally efficient if the following conditions are both satisfied: (i) no data is buffered for a layer that is dropped, and (ii) the layer is only dropped because the *total* amount of buffering is insufficient. To quantify the efficiency of our scheme, we have calculated the percentage of remaining buffer for each dropped layer as follows:

$$e = \frac{buf_{total} - buf_{drop}}{buf_{total}}$$

where buf_{total} and buf_{drop} denote the total buffering and the buffer share of the dropped layer. Then we averaged out the value of e across all drop events during the simulation and use that as an evaluation metric for efficiency.

Table 1 shows these efficiency values for different values of K_{max} during two test, T1 and T2. T1 is the 10 RAP, 10 TCP test depicted in figures 11 whereas T2 is the 10 RAP, 10 TCP test with a large CBR burst shown in figure 13. These results show that our scheme is very efficient - very little buffered data is still available in a layer that is dropped.

	K_{max}=2	K_{max}=3	K_{max}=4	K_{max}=5	K_{max}=8
T1	99.77%	99.97%	99.84%	99.85%	99.99%
T2	99.15%	99.81%	99.92%	99.80%	96.07%

Table 1: Efficiency of buffer allocation

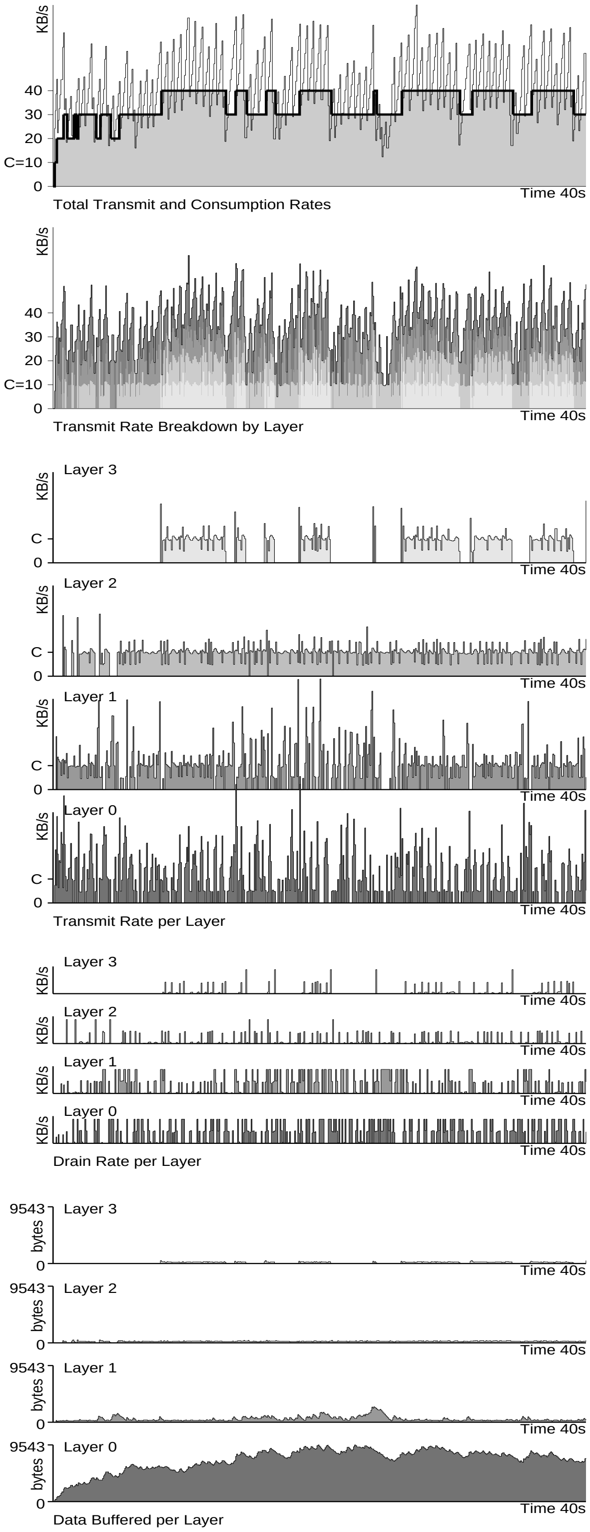

Figure 11: First 40 seconds of K_{max}=2 trace

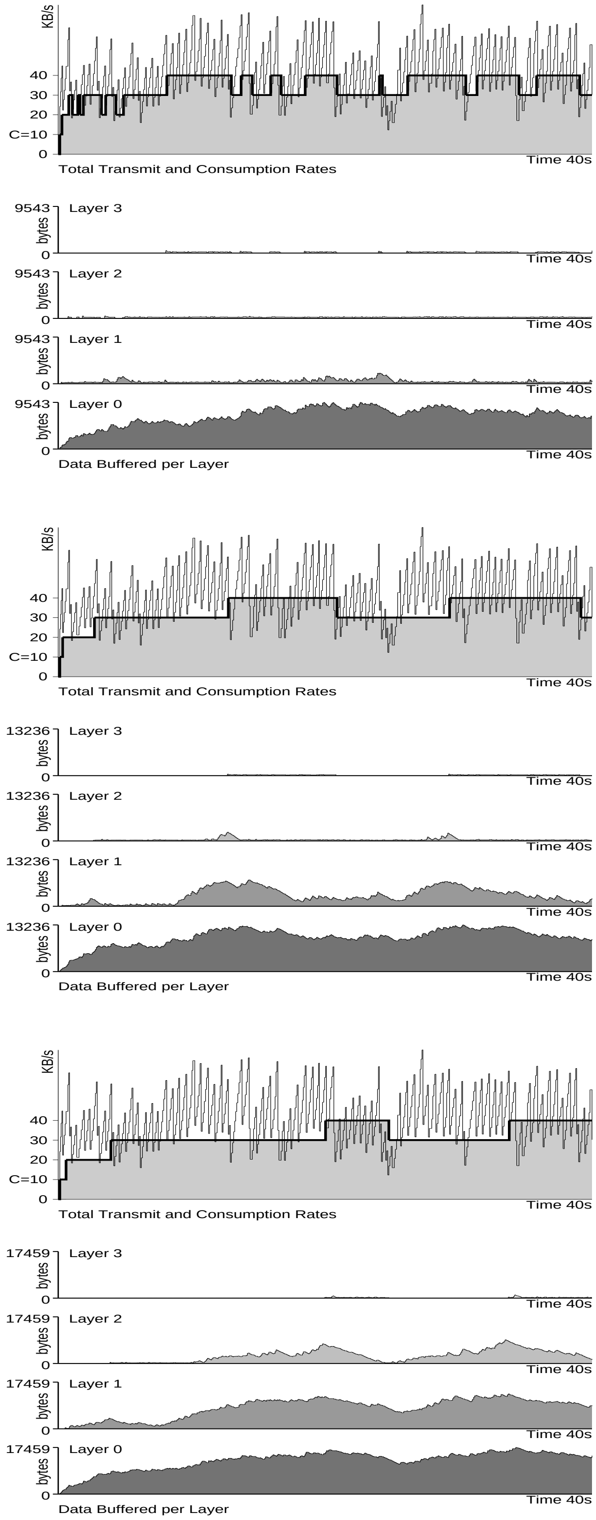

Figure 12: Effect of K_{max} on buffering and quality

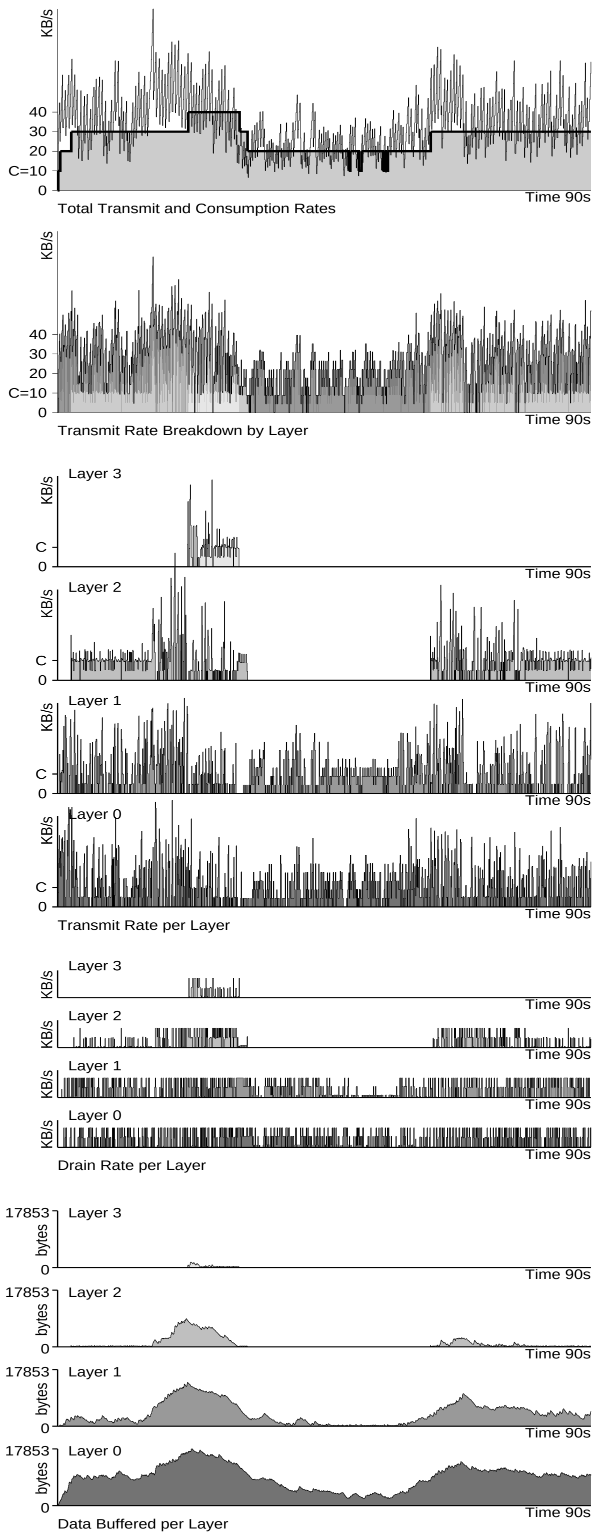

Figure 13: Effect of long-term changes in bandwidth

Table 2 shows the percentage of drops due to poor buffer distribution in test T1 and T2. These are drops that would not have happened if the amount of buffered data that was at the receiver had been distributed differently. Our mechanism is completely efficient in this respect for the T1 tests, and performs fairly well for the T2 case. Clearly the mechanism becomes less efficient as K_{max} increases. The higher the value of K_{max}, the more buffering is allocated for higher layers. Hence there is a higher probability of dropping the highest layer with some buffering particularly after sudden drops in available bandwidth such as when a CBR source appears. In essence, conservative buffering(i.e. higher K_{max}) enables the server to cope with wider variations in bandwidth. However sudden drops of bandwidth in these situations results in lower efficiency.

	K_{max}=2	K_{max}=3	K_{max}=4	K_{max}=5	K_{max}=8
T1	0%	0%	0%	0%	0%
T2	2.4%	0%	4.8%	11%	-

Table 2: % drops due to poor buffer distribution

6 Related Work

Receiver-based layered transmission has been discussed in the context of multicast video[9, 11, 22] to accommodate heterogeneity while performing coarse-grain congestion control. This differs from our approach that allows fine-grain congestion control for unicast delivery with no step-function changes in transmission rate.

Merz et al. [13] present an iterative approach for sending high bandwidth video through a low bandwidth channel. They suggest segmentation methods that provide the flexibility to playback a high quality stream over several iterations, allowing the client to trade startup latency for quality.

Work in [7, 16, 19] discuss congestion control for streaming applications and focusing on rate adaptation. However, variations of transmission rate in a long-lived session could result in client buffer overflow or underflow. Quality adaptation is complementary for these scheme because it prevents buffer underflow or overflow while effectively utilizing the available bandwidth.

Feng et al. [4] propose an adaptive smoothing mechanism combining bandwidth smoothing with rate adaptation. The send rate is shaped by dropping low-priority frames based on prior knowledge of the video stream. This is meant to limit quality degradation caused by dropped frames but the quality variation cannot be predicted.

Unfortunately, technical information for evaluation of popular applications such as RealVideo G2 [14] is unavailable.

7 Conclusions and Future Work

We have presented a quality adaptation mechanism to bridge the gap between short-term changes in transmission rate caused by congestion control and the need for stable quality in streaming applications. We exploit the flexibility of layered encoding to adapt the quality along with long-term variations in available bandwidth. The key issue is appropriate buffer distribution among the active layers. We have described an efficient mechanism that dynamically adjusts the buffer distribution as the available bandwidth changes by carefully allocating the bandwidth among the active layers. Furthermore, we introduced a smoothing parameter that allows the server to trade short-term improvement for long-term smoothing of quality. The strength of our approach comes from the fact that we did not make any assumptions about loss patterns or available bandwidth. The server adaptively changes the receiver's buffer state to incrementally improve its protection against short-term drops in bandwidth in an efficient fashion. Our simulation and experimental results reveal that with a small amount of buffering the mechanism can efficiently cope with short-term changes in bandwidth due to AIMD congestion control. The mechanism can rapidly adjust the quality of the delivered stream to utilize the available bandwidth while preventing buffer overflow or underflow. Furthermore, by increasing the smoothing factor, the frequency of quality variation is effectively limited.

Given that buffer requirements for quality adaptation are not large, we believe that these mechanisms can also be deployed for non-interactive *live* sessions where the client can tolerate a short delay in delivery.

We plan to extend the idea of quality adaptation to other congestion control schemes that employ AIMD algorithms and investigate the implications of the details of rate adaption on our mechanism. We will also study quality adaptation with a non-linear distribution of bandwidth among layers. Another interesting issue is to use a measurement-based approach to adjust K_{max} on-the-fly based on the recent history.

Finally, quality adaptation provides a perfect opportunity for proxy caching of multimedia streams which we plan to examine. The proxy would cache each stream and missing pieces that are likely to be needed would be pre-fetched in a demand-driven fashion.

8 Acknowledgments

We would like to thank Sally Floyd, Ted Faber, Joe Bannister, John Heidemann, David J. Wetherall, Roger G. Kermode, Ahmed Helmy, Haobo Yu, Art Mena, Hongsuda Tangmunarunkit, Mohit Talwar and the anonymous reviewers for their thoughtful comments on drafts of this paper.

References

[1] J. Bolot and T. Turletti. A rate control mechanism for packet video in the internet. *Proc. IEEE Infocom*, pages 1216–1223, June 1994.

[2] J. C. Bolot. Characterizing end-to-end packet delay and loss in the internet. *Journal of High Speed Networks*, 2(3):289–298, September 1993.

[3] S. Bajaj et al. Improving simulation for network research. Technical Report 99-702, USC-CS, March 1999.

[4] W. Feng, M. Liu, B. Krishnaswami, and A. Prabhudev. A priority-based technique for the best-effort delivery of stored video. *Proc. of Multimedia Computing and Networking*, January 1999.

[5] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *Under submission*, February 1998. http://www-nrg.ee.lbl.gov/floyd/papers.html/end2end-paper.html.

[6] Microsoft Inc. Netshow service, streaming media for business. http://www.microsoft.com/NTServer/Basics/NetShowServices.

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

[8] Jae-Yong Lee, Tae-Hyun Kim, , and Sung-Jea Ko. Motion prediction based on temporal layering for layered video coding. *Proc. ITC-CSCC*, 1:245–248, July 1998.

[9] X. Li, M. Ammar, and S. Paul. Layered video multicast with retransmission(LVMR): Evaluation of hierarchical rate control. *Proc. IEEE Infocom*, March 1998.

[10] S. McCanne. Scalable compression and transmission of internet multicast video. *Ph.D. thesis, University of California Berkeley, UCB/CSD-96-928*, December 1996.

[11] S. McCanne, V. Jacobson, and M. Vettereli. Receiver-driven layered multicast. *Proc. ACM SIGCOMM*, August 1996.

[12] S. McCanne and M. Vetterli. Joint source/channel coding for multicast packet video. *Proc. IEEE International Conference on Image Processing*, pages 776–785, October 1995.

[13] M. Merz, K. Froitzheim, P. Schulthess, and H. Wolf. Iterative transmission of media streams. *Proc. ACM Multimedia*, November 1997.

[14] Real Networks. Http versus realaudio client-server streaming. http://www.realaudio.com/help/content/http-vs-ra.html.

[15] A. Ortega and M. Khansari. Rate control for video coding over variable bit rate channels with applications to wireless transmission. *Proc. IEEE International Conference on Image Processing*, October 1995.

[16] J. Padhye, J. Kurose, D. Towsley, and R. Koodli. TCP-friendly rate adjustment protocol for continuous media flows over best effort networks. Technical Report 98_11, UMASS CMPSCI, 1998.

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

[18] R. Rejaie, M. Handley, H. Yu, and D. Estrin. Proxy caching mechanism for multimedia playback streams in the internet. *Proc. International Web Caching Workshop*, March 1999.

[19] D. Sisalem and H. Schulzrinne. The loss-delay based adjustment algorithm: A TCP-friendly adaptation scheme. *Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[20] W. Tan and A. Zakhor. Error resilient packet video for the internet. *Proc. IEEE International Conference on Image Processing*, October 1998.

[21] M. Vishwanath and P. Chou. An efficient algorithm for hierarchical compression of video. *Proc. IEEE International Conference on Image Processing*, November 1994.

[22] L. Wu, R. Sharma, and B. Smith. Thin streams: An architecture for multicasting layered video. *Workshop on Network and Operating System Support for Digital Audio and Video*, May 1997.