Thin Streams: An Architecture for Multicasting Layered Video

Linda Wu lxwu@cs.cornell.edu
Rosen Sharma sharma@cs.cornell.edu
Brian Smith bsmith@cs.cornell.edu
Department of Computer Science
Cornell University
Ithaca, New York, 14853

Abstract[*]

Multicast is a common method for distributing audio and video over the Internet. Since receivers are heterogeneous in processing capability, network bandwidth, and requirements for video quality, a single multicast stream is usually insufficient. A common strategy is to use layered video coding with multiple multicast groups. In this scheme, a receiver adjusts its video quality by selecting the number of multicast groups, and thereby video layers, it receives. Implementing this scheme requires the receivers to decide when to join a new group or leave a subscribed group.

This paper presents a new solution to the join/leave problem using *ThinStreams*. In ThinStreams, a single video layer is multicast over several multicast groups, each with identical bandwidth. ThinStreams separates the coding scheme (i.e., the video layers) from control (i.e., the multicast groups), helping to bound network oscillations caused by receivers joining and leaving high bandwidth multicast groups.

This work evaluates the join/leave algorithms used in ThinStreams using simulations and preliminary experiments on the MBONE. It also addresses fairness among independent video broadcasts and shows how to prevent interference between them.

1. Introduction

The use of multicast for transmitting video over the Internet is well known. If the receivers of a multicast video are heterogeneous in their computational ability, network connectivity, and need for video quality, multicasting a single stream is undesirable. Schemes that require feedback from the receivers do not scale well, as a "feedback implosion" can occur if all receivers send feedback to the sender. Bolot suggested a scaleable feedback scheme where the minimum of the bandwidth reported by the receivers, or some percentile of the minimum (e.g., the bandwidth that 80% of the receivers can support) is used as the sending rate [2]. For video transmission, such constant bandwidth schemes usually waste bandwidth on some channels or cause congestion on others (or both!).

A better solution to the heterogeneous receiver problem is to use layered video coding and multiple multicast groups (*MMG*). A layered video stream consists of a base layer and several enhancement layers. By transmitting the layers on different multicast groups, receivers can adjust the quality of the displayed video, and the associated computation and network requirements, by joining or leaving multicast groups. More layers leads to a better video quality while fewer layers lead to reduced bandwidth requirements. To implement this scheme, the receiver must decide when to join or leave multicast groups.

McCanne proposed a solution to the join/leave problem called Receiver-driven Layered Multicast (RLM) [6]. RLM assumes that the network provides best effort service and supports IP multicast, and uses the term *session* to denote the set of groups used by a source to send layered video. Receivers in a session drop layers when they detect congestion and add layers when spare capacity is available. RLM detects congestion by measuring packet loss and uses *join-experiments* to detect spare capacity. In a join-experiment, a receiver joins a group and measures the loss rate over a time interval called the *decision-time*. If the loss is too high, the receiver leaves the group.

[*] This work is supported by funding from the US Air Force, under contract F49620-94-1-0198

The use of join-experiments can cause large oscillations in network congestion because most video compression schemes create high-bandwidth layers. For example, suppose a receiver subscribes to a group on which the source is sending data at rate *R*, exceeding the capacity of an internal router. After time *T*, the receiver detects excess loss and drops the group. In the worst case, the buffer occupancy (*B*) in the router is *R*T*, where *T* is the sum of the join latency (t_{join}), the leave latency (t_{leave}), and the measurement interval (*I*). *T* is bounded by the properties of the Internet Group Membership Protocol (IGMP) and, depending on the version of IGMP, can be between a few hundred milliseconds and a few minutes. Thus, if *R* is large (as is the case in video transmission), excess congestion will occur in the routers, leading to large oscillations in network congestion.

The real problem here is that the video codec determines the bandwidth used for the join-experiments, whereas it should be related to the network parameters. In our solution, we divide a *thick stream* (a video layer) into many *thin streams*, each of which is sent on a separate multicast group. The ThinStreams architecture reduces *R*, avoiding excess oscillations in congestion typically caused by a join experiment.

Using *MMG* for video transmission raises other issues. For example, when a network link is shared by independent sessions, the link bandwidth should be fairly distributed among the session. Our join-leave rules achieve link sharing by making the receivers that have joined few groups more aggressive in join experiments than the receivers that have joined many groups.

If two receivers that share a link conduct join experiments at the same time, they will skew each other's results. However, receivers on the same session should conduct join experiments at the same time, so they do not overload the network for excessively long periods. ThinStreams achieves these goals by sending a clock signal in the base layer of the video stream. The clock sequence is a pseudo-noise sequence with a long period, and receivers only join groups at a clock transition. This solution allows receivers in the same session to conduct their join experiments in synchrony, but prevents receivers in different sessions from conducting their experiments simultaneously, with high probability.

The rest of the paper describes and evaluates ThinStreams in detail. Section 2 reviews related work. Section 3 describes our join-leave algorithm. Section 4 discusses our approach to scalability, section 5 describes how we determine IGMP leave latency (an important parameter of our architecture), section 6 reports the results of simulations and experiments that evaluate ThinStreams, section 7 discusses further issues raised by ThinStreams, and section 8 concludes the paper.

2. Related Work

Our work addresses the problem of dealing with heterogeneity among the receivers of a multicast group and also how each receiver adapts to the changing network conditions. Related work includes work on unicast and multicast video distribution with layered codecs.

Several unicast video distribution systems have studied the problems associated with storing a scaleable video stream on a server [9, 10, 11, 12]. The server receives feedback from the client or the network, and adapts the transmission rate based on a control algorithm. These algorithms must interact gracefully with other receivers that may be using the same algorithm or another congestion control algorithm (e.g., TCP).

Deering first suggested that IP multicast be used with layered video coding, mapping layers onto multiple multicast groups (*MMG*) [5]. Several researchers have presented layered compression algorithms [4,8] and suggesting using MMG, but they do not specify algorithms for joining and leaving multicast groups.

An important exception is McCanne's Receiver-driven Layered Multicast (RLM) [6], which comes closest in spirit to our work. McCanne explores algorithms to join and leave groups within a *session* (the set of groups used for by a source to send layered video). McCanne uses packet loss to detect congestion and *join-experiments* to determine when excess capacity is available. If the number of join-experiments is allowed to grow with the number of receivers, the network will be constantly overloaded.

Using a protocol like RTCP (Real Time Control Protocol) [7] reduces the number of join experiments. Although this allows the protocol to scale, it slows down the convergence of the receivers. RLM therefore uses *shared learning* to improve convergence. A receiver advertises its intention of conducting an experiment to other members of the group. Only receivers that want to conduct join-experiments for layers below or equal to the one advertised actually conduct their experiments, and receivers share the results of the experiments.

ThinStreams differs from RLM because it explicitly address link sharing and de-couples the bandwidth of the unit of network control (the multicast group) from the video encoding scheme.

3. Rules for Joining and Leaving Multicast Groups

The problem of joining and leaving groups is central to the MMG architecture. This section discusses the join-leave algorithm used in ThinStreams.

A simple join-leave algorithm is for the receiver to join a group and then leave if it detects excessive loss. This process is called a *join-experiment*. However, a failed join-experiment (i.e., one that overloads a link) will cause loss in other groups sharing the overloaded link, such as independent sessions sharing the link or the lower layers in the same session.

This problem is exasperated by the relatively high bandwidth used in most layered video coding systems. When such a group is added in a join experiment, the network buffering (about 4-8 KB for the Internet [1]) is usually insufficient to absorb the resulting congestion caused during the join experiment. For instance, if a 256 Kbps layer is added in a 2 second join experiment, the network must buffer up to 64 KB to avoid loss.

We can reduce the adverse effects of failed join experiments by making two changes to this simple algorithm. First, we must limit the bandwidth used in a join experiment so that network buffering can absorb the temporary overload induced when the experiment fails. Second, we must use a mechanism other than loss to detect network overload. These two ideas are the key insights of the ThinStreams architecture, and are elaborated in the following subsections.

3.1 Thin Stream Bandwidth

To limit the bandwidth of a layer, ThinStreams splits a video layer (a *thick stream*) into several fixed bandwidth *thin streams*. We discuss the interaction of the ThinStreams architecture and the video codec in section 7.

The thin stream bandwidth, *R*, is easily calculated. If *B* is the buffering in the network and *T* is the duration of the join experiment, then the buffering required to prevent loss is

$$B \le R \cdot T \qquad [1]$$

Using *B=4* Kbytes for the Internet (as in TCP Vegas [1]) and a conservative value for *T* (2 seconds), we get *R=16* Kbps. Since the values of *T* and *B* depend on network parameters, such as the latency between joining and leaving a multicast group and the amount of buffering in the network, *R* can be different for each receiver. Therefore the values for *T* and *R* are computed at the source based on conservative estimates and sent in the base layer. In our simulations we used 16 Kbps for *R*.

Of course, such conservative parameters may be inappropriate for some receivers. For example, if a receiver is on the same LAN as the source, it can safely conduct join experiments using much thicker streams than more remote receivers. Such nearby receivers achieve this effect by subscribing to several thin streams as part of one join experiment.

3.2 The Join-leave Algorithm

Each receiver conducts independent join experiments to add or remove thin streams. If the receiver uses loss to detect overload during a join experiment, a failed experiment may adversely affect other sessions using the shared link. To avoid such adverse effects, ThinStreams does not use loss to detect channel overload. Instead, it uses the difference between the expected and measured throughput, a solution that was inspired by TCP Vegas [3].

```
R = thin stream bandwidth (from base layer)
I = measurement interval
N = number of bytes received in measurement interval
G = number of groups joined
A = α*A  + N*(1-α)/I
E = α*E  + G*R*(1-α)
leave_threshold = G*R*exp((1-G)/8)
join_threshold = G*R*β
hold_off_time ∝ G
if ((E - A) > leave_threshold)) then leave()
if ((time elapsed since last leave >  hold_off_time ) &&
    ((E - A) < join_threshold)) then join()
```

Figure 1: The ThinStreams join-leave algorithm

TCP Vegas uses the difference between the measured throughput and the expected throughput to control the size of the TCP window. In TCP Vegas, the measured throughput is *WindowSize/RTT*, where *RTT* is the measured round trip time, and *WindowSize* is the size of the TCP window. The expected throughput is *WindowSize/BaseRTT*, where *BaseRTT* is the minimum RTT seen so far. The difference between these quantities corresponds to the amount of buffering in the network and, by adjusting *WindowSize*, is kept between 4-6 KB [1] . The advantage of this scheme is that, in the ideal case, it detects congestion before loss has occurred, and therefore causes less congestion than schemes that use loss for flow control.

Similarly, in ThinStreams the receiver uses the difference between the measured throughput and the expected throughput to make its join-leave decision. If the difference is large, then the link can not support the extra layer, and the group is dropped. If the difference is small and a join-experiment has not been conducted recently, the receiver joins a new group. This algorithm is shown in Figure 1 (the choice of `join_threshold`, `leave_threshold`, and `hold_off_time` are discussed in section 3.3).

In this algorithm, the receiver continuously monitors two quantities, the expected received bandwidth (*E*) and the actual received bandwidth (*A*). The receiver computes *A* by measuring the number of bytes it receives in a measurement interval (in our tests, it was one second) and averaging the values thus obtained to reduce measurement noise. *E* is the product of the bandwidth of each group and the number of groups joined, averaged so that *E* matches *A* if there is no loss in the network.

If the difference between *E* and *A* is greater than a threshold value (`leave_threshold`), the receiver drops the group corresponding to the highest layer. If the difference is below a threshold value (the `join_threshold`) and the receiver has not conducted a join experiment recently (within the last `hold_off_time` seconds), it joins the group corresponding to the next layer.

3.3 Fairness and Convergence

Ideally, the join-leave algorithm should achieve two goals besides avoiding excessive loss in the network during join experiments. First, it should fairly allocate bandwidth among independent sessions sharing a link. Second, it should converge quickly, but avoid rapid oscillations in the number of groups joined that may adversely affect perceived video quality. It turns out that we can achieve both goals by carefully selecting the values for `join_threshold`, `leave_threshold`, and `hold_off_time`.

Figure 2 illustrates the link-sharing problem. In

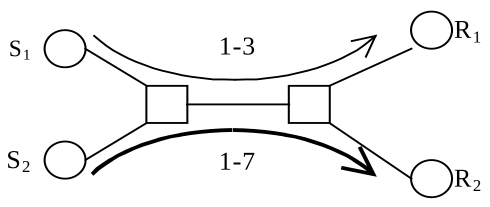

Figure 2. The link-sharing problem

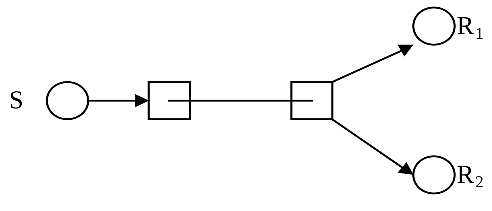

Figure 3. The Scaling Problem

this example, R_1 is subscribed to groups 1-3 of source S_1 and R_2 is subscribed to groups 1-7 of source S_2. In order for both receivers to receive equal portions of the shared link, R_2 must leave two groups and R_1 must join two groups (assuming the groups have equal bandwidth, as in ThinStreams).

We enforce link sharing by adjusting the `leave_threshold` based on the number of groups a receiver has joined. The more groups a receiver has joined, the lower the leave threshold. For example, if R_1's leave threshold is higher than R_2's, then R_2 will drop groups and R_1 will add groups. We compute a receiver's `leave_threshold` as an exponentially decreasing function of the number of *G*, the number of subscribed groups. Specifically, we use

$$\texttt{leave\_threshold} = G*R*e^{(1-G)/8}$$

For the first layer (*G*=1), `leave_threshold` is the bandwidth of one thin stream (*R*). It decreases exponentially as more groups are joined. The constant 8 was determined experimentally, and could be set to other values to achieve different fairness policies. For example, if we change it to the number of groups in the session, then the bandwidth is proportionally shared among independent sessions.

To improve convergence when a receiver starts up (i.e., when few layers are joined), we set `hold_off_time` proportional to the number of subscribed groups. This causes receivers that are subscribed to few groups to conduct join-experiments more often than receivers subscribed to many groups, accelerating convergence. It also causes receivers that are using little bandwidth to be more aggressive than receivers using a lot of bandwidth, promoting fairness.

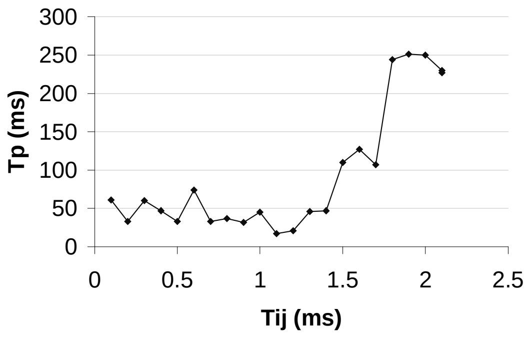

Figure 4. Estimating leave latency in IGMP

4. Synchronization among Receivers

Scalability is a problem in all systems that use join experiments. Figure 3 illustrates the problem. Receivers R_1 and R_2 are subscribed to the same session from source S, and the shared link is the bottleneck. If both receivers conduct join-experiments independently, they will overload the shared link twice as often as a single receiver. If a large number of receivers are downstream from the bottleneck link, receivers conducting join-experiments will almost always overload that link if the experiments are not coordinated in some way.

ThinStreams solves this problem by synchronizing the start of join-experiments within a session. The synchronization is achieved as follows. The source sends a clock pulse in the base layer and receivers wait for a zero to one transition of the clock to a start the join experiment. We call this clock the *ThinStreams clock*. Although the propagation delay of the ThinStreams clock may cause some skewing in the start time of the join experiments, the period of the clock and duration of the experiments are long enough (several seconds) that the skew is unimportant.

In contrast, when several independent sessions share a bottleneck link (as in Figure 2), their join experiments should *not* be synchronized. If two or more join experiments occur simultaneously, the bandwidth of the bottleneck link will increase by more than the thin stream bandwidth *R*, causing exactly the type of congestion that ThinStreams tries to avoid. To de-synchronize the join-experiments of independent sessions, the ThinStreams clock pulses are generated using a pseudo noise (PN) sequence whose seed is the IP address of the source concatenated with a random number. Since two

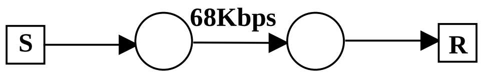

Figure 5. Topology 1

pseudo noise sequences are uncorrelated, the join experiments among receivers of different sources rarely overlap.

5. Estimation of Leave Latency

The time to join and leave multicast groups is an important parameter in ThinStreams, since it determines the bandwidth of the thin streams. In version 1.0 of *IGMP*, the leave latency could be as large as two minutes. In version 2.0 it is configurable, with a minimum of 300 ms. Our experience on the Internet showed that the latency varied considerably depending on which implementation of IGMP was being used. The minimum leave latency is a property of the router that acts as the IGMP querier on the network. It does not vary with time, thus can be determined once and given as a parameter to the receivers. To determine leave latency, we use a probing technique similar to the one used to measure the cache line size on a processor. A receiver leaves a multicast group and then joins it again after time t_{lj}. After joining it receives the first packet after time t_p. Fig 4 shows the effect of increasing t_{lj} on t_p. The knee in the curve represents the time at which the prune is sent upstream by the router. Until the prune is sent t_p remains small as the packets are already being transmitted on the local network. If the prune is sent before the receiver rejoins the group, join needs to

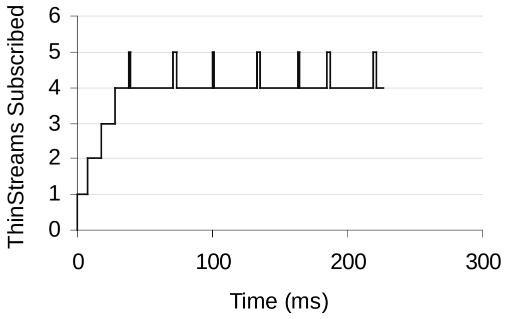

Figure 6. Simulation results for topology 1

be propagated upstream, leading to a higher value for t_p. The value of t_{lj} at the knee of the curve is used as an approximation for the leave latency.

6. Experimental Evaluation

We evaluated ThinStreams using both simulation experiments and the MBone. For the simulation experiments, we modified the REAL 4.0 simulator [15] to simulate IP-multicast. This section reports the results of those studies.

6.1 Simulation

We simulated ThinStreams on different topologies to stress different aspects of its join-leave algorithm. In all simulations, the bandwidth of each thin stream was 16 Kbps, and the packet size was 256 bytes. The averaging interval was 1 second (implying 8 packets per measurement interval), and the IGMP leave latency was set to 500 ms.

In the first topology (Figure 5), we wanted to test how the algorithms behaved in a network where only

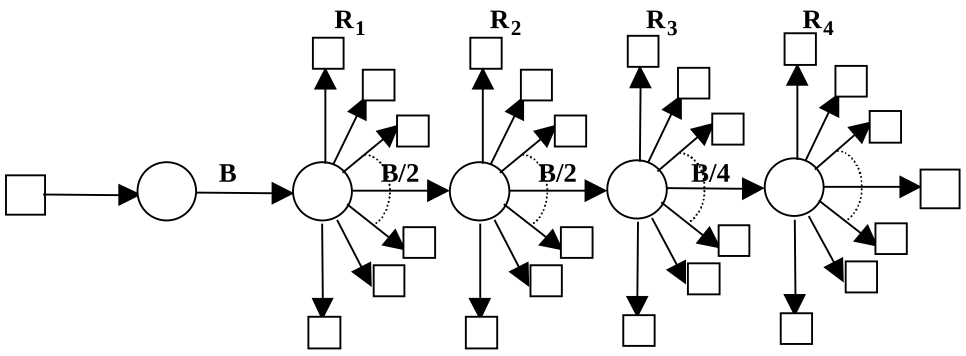

Figure 7: Topology 2 (B = 512 Kbps)

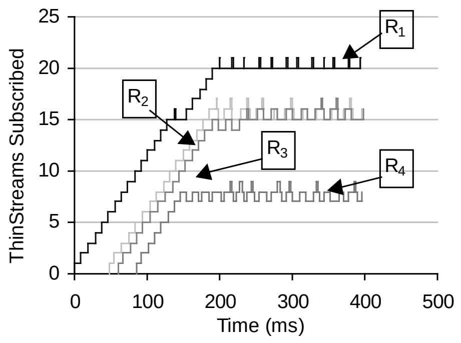

Figure 8: Results for topology 2

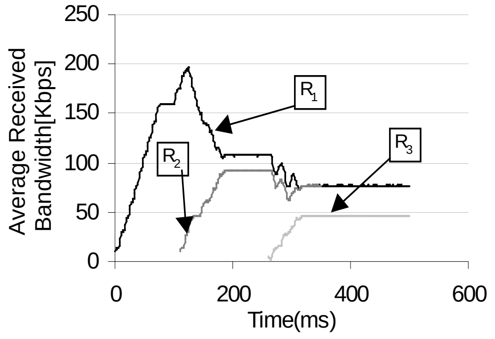

Figure 10: Results for topology 3

one receiver is connected to a bottleneck router. The bandwidth of the bottleneck link is 68 Kbps. Since each thin stream is 16 Kbps, we would expect the receiver to join 4 groups. Figure 6 plots the number of thin streams joined as a function of time. As expected, the receiver quickly subscribes to four thin streams, but there is not enough capacity to support the fifth stream. Thus, it's repeated attempts to join the fifth thin stream fail.

Our second experiment used the topology shown in Figure 7. It examined the performance of the algorithm with many receivers subscribing to the same source. The receivers have different bottleneck bandwidths and latencies to the sender. There are four clusters of receivers, each having 32 receivers, and identified by their representatives R_1, R_2, R_3 and R_4. The links between routers have a latency of 50 ms. Figure 8 shows the number of groups to which R_1, R_2, R_3 and R_4 subscribed as a function of time. The plots for other recipients in the same cluster are similar. R_1 subscribes to 20 groups as the source is sending only 20 groups. R2 and R3 both subscribe to 15 groups corresponding to 256 Kbps of bottleneck bandwidth and R4 subscribes to 7 groups. It can be also observed from the plot that the clock sequence helps to synchronize the experiments.

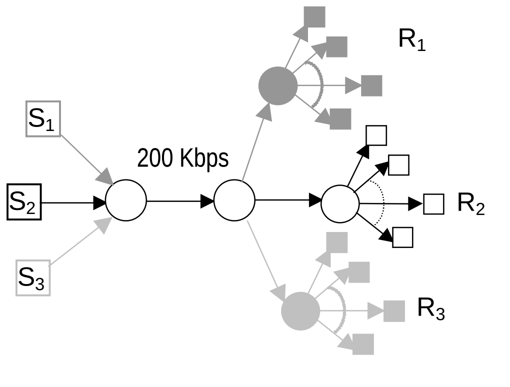

Figure 9: Topology 3

Our experiments with topology 3 (Figure 9) studied the interaction between independent sources using ThinStreams. Groups of receivers identified by R_1, R_2 and R_3 subscribe to sources S_1, S_2 and S_3 respectively. Figure 10 shows the average bandwidth received by each receiver. Again, the plots for other receivers in the same cluster are similar. R_1 initially joins sufficient groups to grab the 200 Kbps of available bandwidth. It gives up half the bandwidth for R2 when R2 subscribes to the its source. Both R_1 and R_2 give up bandwidth for R_3 when it subscribes to the S_3. The slight difference in the allocation of bandwidth is due to the fact that the receivers enter a meta-stable state where they are not getting enough bandwidth to join a higher group, but are getting enough bandwidth not to leave the present group. We encountered this problem in our simulations, but not on experiments on the MBone. We conjecture that oscillations on the MBone caused by other traffic destabilize the meta-stable state.

6.2 MBone Experiments

We ran experiments using the MBone to verify our simulation results. Our experiments were limited by the difficulty of finding hosts that had been upgraded to the latest version of the IGMP protocol. Most hosts are still running older versions detects leaves passively, thus leading to high leave latencies. The results presented here are from a series of experiments[1] done between hosts in Berkeley and Cornell. The topology of several relevant hosts in the MBone route at the time of the experiment is shown in Figure 11. We started two sources on

[1] Other experiments conducted between University of Delaware and Cornell are similar.

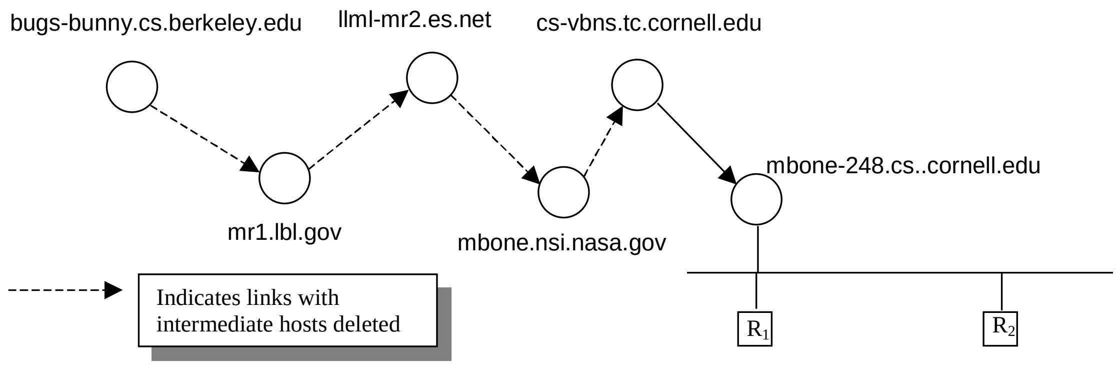
bugs-bunny.cs.berkeley.edu
llml-mr2.es.net
cs-vbns.tc.cornell.edu
mr1.lbl.gov
mbone.nsi.nasa.gov
mbone-248.cs..cornell.edu
Indicates links with intermediate hosts deleted
R1
R2

Figure 11:Topology for MBone experiments

bugs-bunny.cs.berkeley.edu and two receivers (R_1 and R_2) at Cornell. Each source produced ten thin streams to which the receivers could subscribe. We used an on/off TCP source for cross-traffic.

The results of the experiment are shown in Figure 12. Both receivers are able to share the link capacity and give up bandwidth equally when cross traffic is introduced. For the first 15 minutes of the test, the link was lightly loaded and both receivers subscribe to the maximal number of streams (10). The periodic dips in the bandwidth are due to cross traffic produced by our on/off TCP source. Also note that R_1 occasionally subscribes to more groups than R_2 and that the pattern is switches later. This behavior is due to the different clock sequences that are sent by the source. In the short term one source may gain unfairly over another, but the algorithm is fair over longer time frames. As multimedia sessions are usually long this is not a major issue.

7. Discussion and Future Work

One question raised by ThinStreams is which compression algorithms can exploit the ThinStreams architecture. Some existing codecs, like SPHIT [14] (an enhancement of Embedded Zero Tree Wavelets) and HVQ (Hierarchical Vector Quantization [13]), can be easily split into thin streams.

Others, such as Motion JPEG [28], MPEG [11], H.261, Haar (NV [10]) require large jumps in bandwidth to obtain significantly different end quality. When inter-frame differential coding is used (such as in H.261 or MPEG), the base layer is usually a high bandwidth stream, reducing the value of layering. JPEG can be adapted to ThinStreams by removing the inter-block dependencies (in JPEG, this corresponds to not differentially encoding the DC term) and adding a block address so the decoder knows where to place the decoded block data in the output image. In other words, the source can put, say, 10 blocks per frame in each thin stream and arrange for the blocks to be independently decodable and displayable. The grouping of blocks into layers could be based on conditional block replenishment considerations. These ideas require further research and are beyond the scope of this paper.

Splitting a thick stream into many thin steams introduces overhead at the receiver. The receiver must manage many groups and, in some cases, reassemble the thick stream. For an embedded codec assembling the thick stream is not a difficult task. If ThinStreams is implemented as a network layer, the API between the decoder and the thin stream architecture is an interesting problem. We propose that the source specify a mapping of thin streams to thick streams in the base layer. For example, it might specify that it can use thin streams grouped as 1-3,4-7,8-9,10-16. This means that although the ThinStreams layer in the receiver might have joined 4 groups, the decoder will only receive the first 3 groups (which correspond to a whole layer). Other optimizations are possible. For example, on leaving

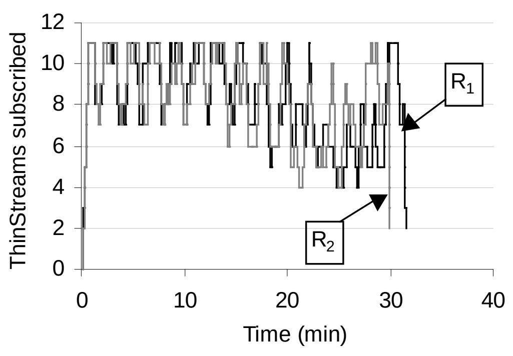

Figure 12: Experiments results for MBone

one thin stream a receiver might drop all layers of the corresponding thick stream, then add them one at a time. These issues are important and will be addressed in future work.

One of the disadvantages of using ThinStreams is slow convergence. Since ThinStreams splits a video layer among many multicast groups, each of which must be joined separately, many groups must be joined before the video quality increases significantly. Our experiences with video perception [9] show that users do not want the video quality to oscillate rapidly but prefer it to change more slowly. Thus, we conjecture that the slow rate of convergence will not be a problem in practice.

We will continue tuning the algorithm and architecture. A few enhancements are:

- If successive joins fail, the link is most likely saturated and join experiments should be done less frequently. This enhancement can be implemented by linearly increasing the `hold_off_time` parameter when a join-experiment fails, and resetting it to a base value when a join-experiment succeeds.
- Using a dynamic value of α for exponential averaging of expected bandwidth (Figure 1).
- Determine the ideal thin stream bandwidth for a group of heterogeneous receivers using feedback to the source. The feedback should specify the leave latency that a receiver has experimentally determined, and the source should set the thin stream bandwidth based on the maximum leave latency of any receiver.

In future work we will study the interaction of ThinStreams with other protocols like TCP. We will also explore the problem of designing a codec that makes efficient use of this architecture. For example, we will search for a good scheme where the relative increase in bandwidth with the addition of each thin stream is constant might correspond to a uniform increase in perception.

8. Conclusions

In this paper, we presented a new architecture for using layered video compression schemes with multiple multicast groups for scaleable video delivery. The ThinStreams architecture de-couples network control and the video codec to prevent excessive congestion in the network. The scheme scales well, and shares bandwidth among receivers within a session and between independent sessions.

References

[1] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan, “Evaluation of TCP Vegas: Emulation and Experiment,” *Proceedings of ACM Sigcomm*, 1995

[2] J. C. Bolot, T. Turletti, and I. Wakeman, “Scalable feedback control for multicast video distribution,” *Proceedings of ACM Sigcomm*, September 1994.

[3] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, “TCP Vegas: New Techniques for Congestion Detection and Avoidance,” *Proceedings of ACM Sigcomm*, September 1994

[4] N. Chaddha and A. Gupta, “A framework for live multicast of video streams over the Internet,” *Proceedings of IEEE Conference on Image Processing*, September 1996

[5] S. Deering. “Internet multicast routing. State of art and open research issues,” *MICE* Seminar, Stockholm, October 1993.

[6] S. McCanne, V. Jacobson, and M. Vetterli, “Receiver-driven Layered Multicast,” *Proceedings of ACM Sigcomm*, 1996

[7] H. Schulzrinne and S. Casner, “RTP: A transport Protocol for Real-Time Applications”, Internet Draft, draft-ietf-avt-rtp-04 October, 1993,

[8] S. Cheung and A. Zakhor, “Scalable Video Compression With Motion Compensated Prediction,” IEEE International Conference on Image Processing, Washington D.C., Oct 1995.

[9] M. Gilge and R. Gusella, “Motion video coding for packet switched networks -- an integrated approach,” *Proceedings of the SPIE conference on Visual Communications and Image Processing*, November, 1991.

[10] H. Kanakia, P. P. Mishra, and A. Reibman, “An Adaptive Congestion Control Scheme for Real Time Packet Video Transport,” *Proceedings of ACM SigComm*, 1993.

[11] P. Pancha and M. El Zarki, “Prioritized Transmission of Variable Bit Rate MPEG Video,” *Proceedings of IEEE Globecom*, December 1992.

[12] A. R. Reibman and A. W. Berger, “On VBR Video Teleconferencing over ATM Networks,” *Proceedings of IEEE Infocom*, 1992

[13] N. Chaddha, “Hierarchical Vector Quantization,” Data Compression Conference, 1994.

[14] A. Said and W. A. Pearlman, "A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees," IEEE Transactions on Circuits and Systems for Video Technology, vol. 6, pp. 243-250, June 1996.
[15] S. Keshav, REAL simulator (software on-line)[2]

Appendix A. Pseudo Noise Sequences

Pseudo noise sequences belong to the family of functions that can be implemented by a *linear feedback shift register (LFSR)* as shown in Figure 13. A LFSR is fully characterized by its state vector *S*, which is given by (s_{m-1} , s_{m-2} , s_{m-3} , ..., s_3, s_2, s_1). The system takes no input except for the initial state vector. At each clock cycle the LFSR generates 1 bit of output and shifts the state vector to the right. An LFSR is said to have a period *T* if its output repeats itself with after *T* clock cycles.

The characteristic function of a LFSR is defined as the binary polynomial of degree *m*

$$P(x) = x^m + a_{m-1}x^{m-1} + ... + a_2x^2 + x \quad +1$$

where the a_I's are exclusively from the binary field {0,1} and all arithmetic is modulo 2. We define the sequence generated by a binary polynomial as the sequence generated by the LFSR with this polynomial as the characteristic function.

LFSR's can be used to generate clock sequences by initializing the state vector with the address of the source. We are interested in LFSRs whose characteristic functions are *primary polynomials*. A primary polynomial is said to be a binary polynomial is a primary polynomial that cannot be expressed as a product of two or more binary polynomials of degree greater than 1 and the sequence generated by *P(x)* has a period of 2^m*-1*. Primary polynomial have several properties, including

1. The sequence generated by the polynomial has an equal number of zeros and ones.

2. Sequences generated from different initial state vectors have low correlation.

3. The number of primary polynomials of degree 2^m for a given degree *m* are given by

$$\phi\left(2^m - 1\right)/\, m$$

where ϕ is the Euler function.

Properties 1 and 2 allow us to separate the collisions between the experiments of two sources. Property 3 allows us to choose the polynomial to use at run time. In using the pseudo noise sequences as clock, we will use the edge transitions as clock pulses. It can be shown that the sequence of transitions of a pseudo noise sequence also forms a pseudo noise sequence.

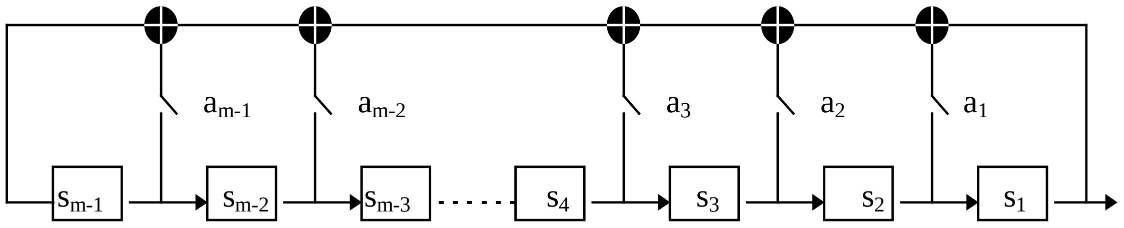

Figure 13. Linear Feedback Shift Registers (a_i . $s_i \in$ {0,1}).

[2] http://minnie.cs.adfa.oz.au/REAL/index.html