Using Available Client Bandwidth to Reduce the Distribution Costs of Video-on-Demand Services (Extended Abstract)

Jehan-François Pâris
Department of Computer Science
University of Houston
Houston, TX 77204-3010
paris@cs.uh.edu

1. Introduction

One of the most important characteristics of video-on-demand (VOD) services is their very high bandwidth requirements. Assuming that the videos are in MPEG-2 format, each user request will require the delivery of approximately six megabits of data per second. Hence, a video server allocating a separate stream of data to each request would need an aggregate bandwidth of six gigabits per second to accommodate one thousand overlapping requests. Servers capable of handling such bandwidths require a costly infrastructure, typically consisting of a large number of computing nodes linked by sophisticated interconnection network.

This situation has led to numerous proposals aimed at reducing the bandwidth requirements of VOD services. These proposals can be broadly classified into two groups. Proposals in the first group are said to be *proactive* because they distribute each video according to a fixed schedule that is not affected by the presence–or the absence–of requests for that video. They are also known as *broadcasting* protocols. Other solutions are purely *reactive*: they only transmit data in response to a specific customer request. Unlike proactive protocols, reactive protocols do not consume bandwidth in the absence of customer requests.

Nearly all these proposals assume a clear separation of functions between the server, which distributes the video, and the customers, who watch it on their personal computers or on their television sets. They are well suited to commercial environments where the respective roles of the service provider and its customers are well defined. However they do not address the case of collaborative video-on-demand services where customers could be expected to contribute to the distribution of the video. Similar arrangements already exist in peer-to-peer file distribution systems. For instance, the BitTorrent system [4] penalizes customers who are not willing to redistribute the data they have already received.

Involving clients in the distribution process raises two issues. First, most clients will only be willing to participate in the video distribution process while they themselves are watching that video. Second, home-based clients typically have much lower upstream than downstream bandwidths: while these clients might be able to download video data at twice their video consumption rate, they might only be able to forward video data at one fourth to one half that rate.

The video distribution protocol we present here addresses these two issues: first it is a purely reactive stream tapping protocol; second, it does not require clients to be able to broadcast video data at the video consumption rate. As in conventional stream tapping, our protocol requires the server to start a new video broadcast whenever a client cannot get enough video data by "tapping" a previous broadcast of the same video. Unlike conventional stream tapping, our protocol uses the previous client's available upstream bandwidth to reduce the amount of video data that the server will still have to send to the clients that "tap" a previous broadcast of the video. Our simulations indicate that our protocol works best when clients can forward video data at least half the video consumption rate. When this is not the case, the best alternative is to involve at least two previous clients in the retransmission.

The remainder of the paper is organized as follows. Section 2 reviews previous work on reactive video distribution protocols. Section 3 introduces our stream tapping protocol and section 4 discusses its performance, while Section 5 discusses possible extensions. Finally Section 6 has our conclusions.

2. Previous Work

Two of the earliest reactive distribution protocols are batching and piggybacking. *Batching* [5] reduces the bandwidth requirements of individual user requests by multicasting one single data stream to all customers who request the same video at the same time. *Piggybacking* [9] adjusts the display rates of overlapping requests for the same video until their corresponding data streams can be merged into a single stream. Consider for instance, two requests for the same video separated by a time interval of three minutes. Increasing the display rate of the second stream by 10 percent will allow it to catch up with the first stream after 30 minutes.

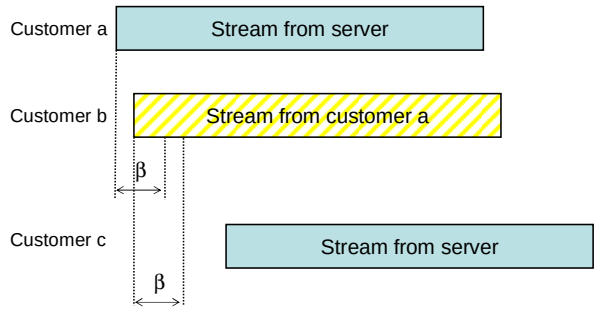

Figure 1: How chaining works

Chaining [13] improves upon batching by constructing chains of clients such that (a) the first client in the chain receives its data from the server and (b) subsequent clients in the chain receive their data from their immediate predecessor. As a result, video data are actually "pipelined" through the clients belonging to the same chain. Since chaining only requires clients to have very small data buffers, a new chain has to be restarted every time the time interval between two successive clients exceeds the capacity β of the buffer of the first client. Figure 1 shows three sample customer requests. Since customer *a* is the first customer, it will get all its data from the server. Since customer *b* arrives less than β minutes after customer *a*, it can receive all its data from customer *a*. Finally customer *c* arrives more than β minutes after customer *a* and must be serviced directly by the server.

Stream tapping [2, 3], also known as *patching* [11], assumes that each customer set-top box has a buffer capable of storing at least 10 minutes of video data. This buffer will allow the set-top box to "tap" into streams of data on the server originally created for other clients, and then store these data until they are needed. In the best case, clients can get most of their data from an existing stream.

In particular, stream tapping defines three types of streams. *Complete streams* read out of a video in its entirety. These are the streams clients typically tap from. *Full tap streams* can be used if a complete stream for the same video started $\Delta \leq \beta$ minutes in the past, where β is the size of the client buffer, measured in minutes of video data. In this case, the client can begin receiving the complete stream right away, storing the data in its buffer. Simultaneously, it can receive the full tap stream and use it to display the first Δ minutes of the video. After that, the client can consume directly from its buffer, which will then always contain a moving Δ-minute window of the video. Stream tapping also defines *partial tap streams*, which can be used when $\Delta > \beta$. In this case clients must go through cycles of filling up and then emptying their buffer since the buffer is not large enough to account for the complete difference in video position.

To use tap streams, clients need only receive at most two streams at any one time. If they can actually handle a higher

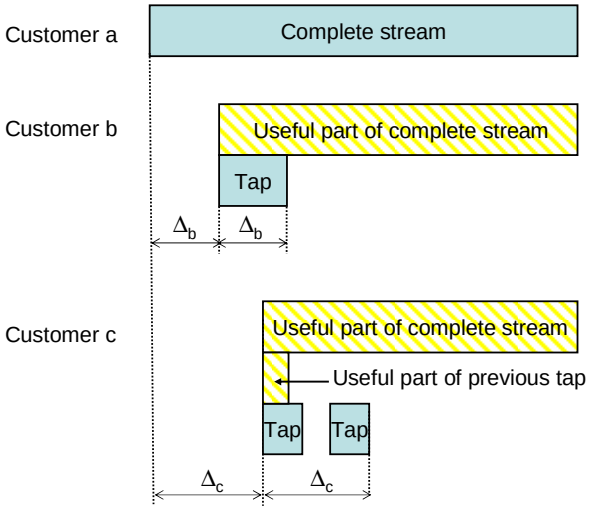

Figure 2: How stream tapping works

bandwidth than this, they can use an option of the protocol called *extra tapping*. Extra tapping allows clients to tap data from any stream on the VOD server, and not just from complete streams. Figure 2 shows some sample customer requests. Since customer *a* is the first customer, it is serviced by a complete stream, whose duration is equal to the duration *D* of the video. Since customer *b* arrives Δ_b minutes after customer *a*, it can share $D - \Delta_b$ minutes of the complete stream and only requires a full tap of duration Δ_b minutes. Finally customer *c* can use extra tapping to tap data from both the complete stream and the previous full tap, and so its service time is smaller than Δ_c.

Eager and Vernon's *dynamic skyscraper broadcasting* (DSB) [6] is another reactive protocol based on Hua and Sheu's *skyscraper broadcasting* protocol [10]. Like skyscraper broadcasting, it never requires the STB to receive more than two streams at the same time. Their more recent *hierarchical multicast stream merging* (HMSM) protocol requires less server bandwidth than DSB to handle the same request arrival rate. Its bandwidth requirements are indeed very close to the upper bound of the minimum bandwidth for a reactive protocol that does not require the STB to receive more than two streams at the same time, that is,

$$\eta_2 \ln\left(1+\frac{N_i}{\eta_2}\right)$$

where $\eta_2 = (1+\sqrt{5})/2$ and N_i is the request arrival rate.

Selective catching [8] combines both reactive and proactive approaches. It dedicates a certain number of channels for periodic broadcasts of videos while using the other channels to allow incoming requests to catch up with the current broadcast cycle. As a result, its bandwidth requirements are $O(\log(\lambda_t L_i))$ where λ_t is the request arrival rate and L_t the duration of the video.

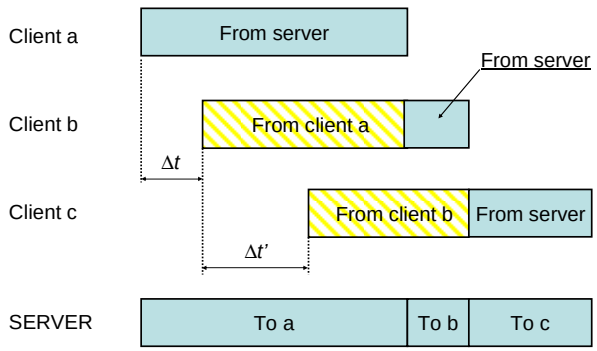

Figure 3: How the cooperative video distribution protocol works.

Finally the cooperative video distribution protocol [12] requires clients to forward the video they are watching to the next client. As shown on Figure 3, the video server will only have to distribute parts of a video that no client can forward. The protocol works best when clients have sufficient buffer capacity to store t he previously viewed portion they are watching until they are have finished watching it. As chaining, it assumes that clients can retransmit data at the video consumption rate.

3. Our Protocol

We wanted to develop a video distribution protocol that allowed clients to participate in the video distribution process even if they could not retransmit data at the video consumption rate. We thus assumed that:

1. Clients would be able to receive video data at twice their video consumption rate;
2. Clients would only be able to forward video data at a rate equal to fraction α of the same video consumption rate;
3. Clients would not have to forward video data after they have finished watching that video;
4. Clients should have enough buffer space to store the previously viewed portion of the video they are watching until they have finished watching it.

As we can see, our protocol makes few demands on the transmission capabilities of the client hardware. In contrast, it requires client buffers capable of storing an entire video, that is, several gigabytes of compressed video data. Two factors motivated this choice. First, the diminishing cost of every kind of storage let it be RAM, flash memory or disk drives, makes this requirement less onerous today than it would have been a few years ago. Second, we expected many clients to keep the previously viewed portion of the video they are watching in their buffer in order to provide the equivalent of a VCR rewind feature.

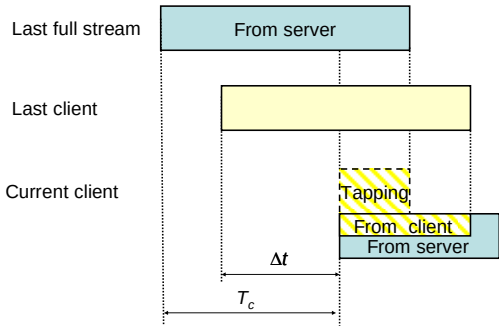

Figure 4: How a full tap streams are shared by the server and the previous customer when $T_c > D - \Delta t$ and the last client terminates before having sent its share of the tap stream of the current client.

Our protocol is a fairly straightforward implementation of stream tapping without extra tapping as it would have required clients able to receive videos at three times the video consumption rate. It only differs from the original stream tapping protocol in the way it handles tap streams. While tap streams originally were the sole responsibility of the server, this task is now shared by the server and the previous client. Consider two consecutive requests for a video of duration D. Let T_c denote the time elapsed since the start of the last complete stream and Δt represent the time interval between the two requests:

1. If $T_c \geq D$, the second client will be unable to tap any data from the last complete stream. As in the original stream tapping protocol, the server will then start a new complete stream.
2. If $T_c < D$, there is an overlap between the current request and the last complete stream. As in the original stream tapping protocol, the server will then evaluate whether it would be more advantageous to keep tapping from the last complete stream or to start a new one. If the server decides to keep tapping from the last complete stream, it will have to provide the second client with a full tap stream of duration T_c. Two alternatives must now be considered:
   a. If $T_c \leq D - \Delta t$, the previous customer will provide a fraction α of the full tap stream and the server the remaining 1 – α fraction.
   b. If $T_c > D - \Delta t$, the previous customer will finish watching the video before being able to transmit all its share of the full tap stream. As seen on Figure 4, the previous client will only be able to transmit a fraction
   $$\alpha \frac{D - \Delta t}{Tc}$$
   of the full tap stream with the server transmitting the remainder of the stream.

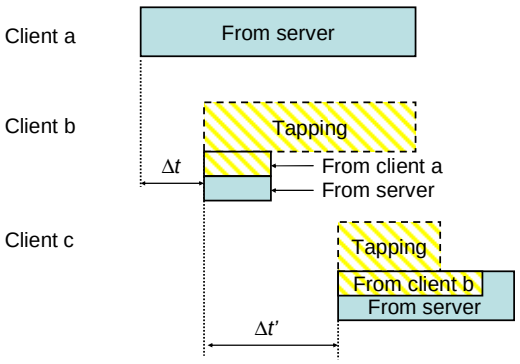

Figure 5: How the full tap streams are distributed by the server and the previous customer.

Consider for instance how the protocol would handle the three requests displayed in Figure 5. The first request to the video will be entirely serviced by a complete stream. The second request will get the last $D - \Delta t$ minutes of the video by tapping client *a*'s stream and the first Δt minutes from a full tap stream of duration Δt. A fraction α of this stream will be sent by customer *a* and the remaining $1 - \alpha$ fraction will come from the server. Assuming that the server decides not to start a new complete stream for customer *c*, that customer would get the last $D - (\Delta t + \Delta t')$ minutes of the video by tapping client *a*'s stream and the first $\Delta t + \Delta t'$ minutes from a full tap stream of the same duration. Since customer *b* will finish watching the video before the end of that stream, t will only able to send its share of the first $D - \Delta t'$ minutes of the tap stream and the server will have to pick up the rest.

One last issue to consider is when to halt tapping from the current complete stream and start a new one. Consider the group of requests sharing the same complete stream. The lowest possible server workload will be achieved by minimizing the average request service time of the group. When a group starts and only has one request, the service time of that request will be equal to the duration of a complete stream, that is, the duration *D* of the video. Adding more requests to the group will reduce the average request service time of the group as long as the full tap streams remain short. At some moment, this will not be true anymore and adding one extra request to the group could actually increase its average request service time. It will then be time to start a new group.

To implement this criterion, our protocol keeps track of the minimum average request service time of all requests sharing the same complete stream. Before adding a new request to a group, it computes what would be the new average request service time of the group if the new request was added to the group. Should this new average request service time be lesser than or equal to the minimum average request service time of the group, our protocol adds the new request to the group; otherwise, it starts a new group.

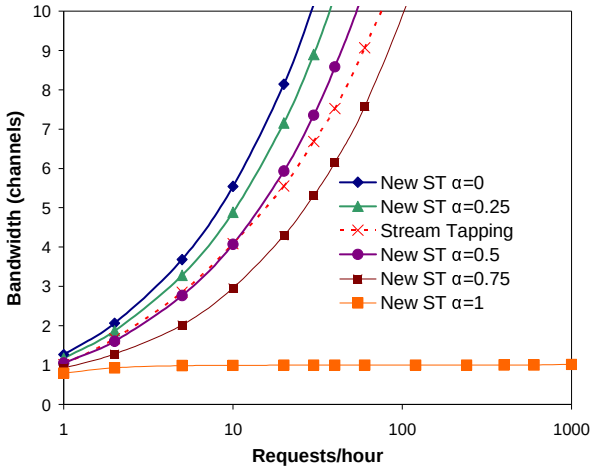

Figure 6: Server bandwidth requirements of the new stream tapping protocol. The dotted curve refers to a conventional stream tapping protocol with extra tapping.

This criterion is similar but not identical to that used by Carter and Long [2, 3].

3.1 Fault-Tolerance Issues

To operate correctly, our protocol requires all clients to forward some of the video data they have received to the next customer requesting the video. As a result, any client failure will deprive all subsequent customers from their video data. This is clearly not acceptable and requires a mechanism allowing the protocol to handle client failures either resulting from an equipment malfunction or from a voluntary disconnect.

There is a simple solution to the problem. Let us return to the scenario of Figure 5 where client *c* receives almost half of its tap stream from client *b*, who receives almost half of its tap stream from client *a*. Any failure of client *b* will immediately affect the correct flow data to client *c*. Fortunately for us, a failure of client *b* will also free client *a* from its obligation to send data to client *b* a fraction of its tap stream, thus freeing enough upstream bandwidth to allow client *a* to take over the role of client *b* and send the missing video data to client *c*. Since client *a* has no predecessors, its failure would be handled by the server alone.

Making the protocol fault-tolerant will thus require providing each client with the addresses of the last two or three clients that have requested the video. Whenever a client detects a failure of its immediate predecessor, it will thus be able to notify its next to last predecessor and its server and request them to redirect their data flows. Once that next to last predecessor and the server have completed this task, everything will happen as if the client that failed never requested the video.

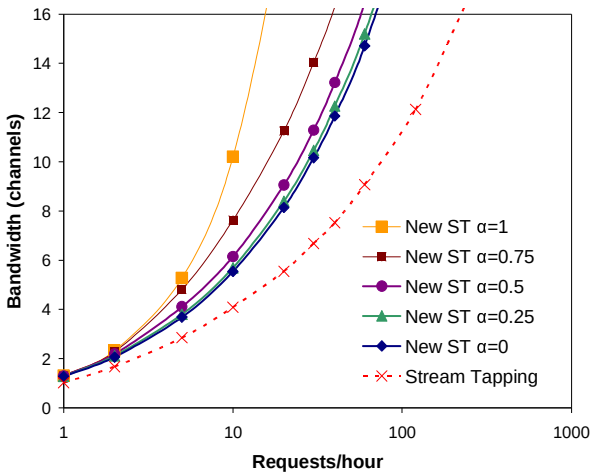

Figure 7: Network bandwidth requirements of the new stream tapping protocol.

4. Performance Evaluation

Figure 6 displays the server bandwidth requirements of our new stream tapping protocol for selected values of α and request arrival rates varying between one and one thousand requests per hour. All bandwidths are expressed in multiples of the video consumption rates. We assumed that the server was broadcasting a two-hour video and that request arrivals could be modeled by a Poisson process.

In addition, the dotted red line represents the server bandwidth requirements of the original stream tapping protocol with extra tapping. Let us note that the comparison between the two protocols is not absolutely fair since extra tapping requires clients capable of receiving video data at three times the video consumption rate, while our protocol only requires clients capable of receiving video data at two times that rate.

As we can see, our new stream tapping protocol outperforms conventional stream tapping whenever clients can forward data at more than half the video consumption rate, that is, when $\alpha > 0.5$. As can expected, the lowest server bandwidth requirements are obtained when $\alpha = 1$ because most, if not all, tap streams are then handled directly by the clients without any server intervention.

This excellent performance comes however at a stiff price. First, it requires clients capable of forwarding data at the video consumption rate, which excludes most home-based clients. Second, the very low server bandwidth requirements of the protocol are only achieved because all tap streams are now handled by the clients. Since each individual tap stream is dedicated to a single client, the network bandwidth requirements of the protocol become roughly proportional to the client request arrival rate. As seen on Figure 7, the network bandwidth requirements of our stream tapping protocol increase more rapidly than those of the original stream tapping protocol when the client request arrival rate exceeds ten requests per hour. This phenomenon can be explained in part by the fact that our protocol does not allow extra tapping. Another important factor is the way the server decides when to start a new complete stream. Recall that the server starts a new complete stream whenever adding one additional request to the group would increase the average service time of the requests in the group. Whenever α is very close to one, adding one extra request to any existing group will have a negligible impact on the server workload. As a result the server will not start a new group until it becomes physically impossible to tap data from the current complete stream. The very low server bandwidths result from the fact that the server will never start a new complete stream before the end of the previous one. Thus the average duration of a tap stream will be equal to half the duration of the video. Hence the average network bandwidth required to distribute the video will be roughly equal to one half the bandwidth required by a scheme allocating a new complete stream to each video request.

5. Possible Extensions

In this section, we present several options for improving the performance of our stream tapping protocol either by making a more efficient use of the available client bandwidth or by reducing the network bandwidth requirements of the protocol.

5.1 Involving several clients in the distribution of tap streams

In its current state, our protocol only involves the previous client in the transmission of tap streams. As a result, previous clients whose upstream bandwidth is much lower than the video consumption rate would leave most of the tap stream transmission workload to the server.

A better solution would be to involve several recent clients in the transmission of tap streams. For instance, four clients capable of transmitting data at a rate equal to one fourth of the video consumption rate could transmit the whole tap stream without any server intervention.

5.2 Allowing extra tapping

Allowing extra tapping would let several clients share a common tap stream. This would greatly reduce the network bandwidth requirements of the protocol but would require clients capable of:

1. receiving data at more than twice the video consumption rate, and
2. multicasting data to other clients, possibly through user-level multicasting [1, 14].

5.3 Controlling network bandwidth usage

The criterion that is now used by the server to decide when it should start a new complete stream should take into consideration the impact of its decision on the network bandwidth requirements of the protocol. We are currently investigating several possible options.

6. Conclusions

Almost all existing distribution protocols for video-on-demand assume a clear separation of functions between the server, which distributes the videos, and its clients, who watch them. Those that do not make this assumption require client machines capable of forwarding video data at the video consumption rate, which is not true for most home-based clients.

We have presented a stream tapping protocol that involves clients in the video distribution process. Our protocol is tailored to environments where client machines are able to download video data at twice the video consumption rate but can only forward video data at a fraction of that rate. As in conventional stream tapping, our protocol requires the server to start a new video broadcast whenever a client cannot get enough video data from a previous broadcast of the same video. Our protocol uses the available upstream bandwidth of the previous client to reduce the amount of video data that the server needs to send to other clients. Our simulations indicate that our protocol works best when clients can forward video data at least at one half the video consumption rate.

More work is still needed to develop techniques that would make a more efficient use of the available client bandwidth and reduce the network bandwidth requirements of the protocol. The most promising avenue seems to be involving several recent clients in the transmission of tap streams.

References

[1] Banerjee S., C. Kommareddy, K. Kar, B. Bhattacharjee, S. Khuller Construction of an efficient overlay multicast infrastructure for real-time applications. *Proc. IEEE INFOCOM Conf.*, San Francisco, pp. 1–11. April 2003.

[2] Carter, S. W. and D. D. E. Long. Improving video-on-demand server efficiency through stream tapping. *Proc. 5th Int'l. Conf. on Computer Communications and Networks*, pp. 200–207, Sep. 1997.

[3] Carter, S. W. and D. D. E. Long. Improving bandwidth efficiency on video-on-demand servers. *Computer Networks and ISDN Systems*, 30(1–2):99–111.

[4] Cohen, B. Incentive Build Robustness in Bit Torrent. *Proc. Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, 2003.

[5] Dan, A., P. Shahabuddin, D. Sitaram and D. Towsley. Channel allocation under batching and VCR control in video-on-demand systems. *Journal of Parallel and Distributed Computing*, 30(2):168–179, Nov. 1994.

[6] Eager, D. L. and M. K. Vernon. Dynamic skyscraper broadcast for video-on-demand. *Proc. 4th Int'l Workshop on Advances in Multimedia Information Systems*, pp. 18–32, Sep. 1998.

[7] Eager, D. L., M. K. Vernon and J. Zahorjan. Minimizing bandwidth requirements for on-demand data delivery. *Proc. 5th Int'l Workshop on Advances in Multimedia Information Systems*, Oct. 1999.

[8] Gao, L., Z.-L Zhang and D. Towsley. Catching and selective catching: efficient latency reduction techniques for delivering continuous multimedia streams. *Proc. of the 1999 ACM Multimedia Conf.*, pp. 203–206, Nov. 1999.

[9] Golubchik, L., J. Lui, and R. Muntz. Adaptive piggybacking: a novel technique for data sharing in video-on-demand storage servers. *ACM Multimedia Systems Journal*, 4(3):140–155, 1996.

[10] Hua, K. A. and S. Sheu. Skyscraper broadcasting: a new broadcasting scheme for metropolitan video-on-demand systems. *Proc. ACM SIGCOMM '97 Conf.*, pp. 89–100, Sept. 1997.

[11] Hua, K. A., Y. Cai, and S. Sheu. Patching: a multicast technique for true video-on-demand services. *Proc. 6th ACM Multimedia Conf.*, pp. 191–200, Sep. 1998.

[12] Pâris, J.-F. A Cooperative Distribution Protocol for Video-on-Demand. *Proc. 6th Mexican Int'l Conf. on Computer Science*, pp. 240–246., Sep. 2005

[13] Sheu, S., K. A. Hua, and W. Tavanapong. Chaining: A Generalized Batching Technique for Video-on-Demand Systems. *Proc. IEEE Int'l Conf. on Multimedia Computing and Systems*, pp. 110-117, June 1997.

[14] Xu, Z., Xu, C. Tang, S. Banerjee, and S.-J. Lee. RITA: Receiver Initiated Just-in-Time Tree Adaptation for Rich Media Distribution. *Proc. 13th Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 50–59, June 2003.