Proc. Mexican International Conference on Computer Science (ENC 2005), Puebla, Mexico, pages 240–246, Sep. 2005

A COOPERATIVE DISTRIBUTION PROTOCOL FOR VIDEO-ON-DEMAND

Jehan-François Pâris[1]
Department of Computer Science
University of Houston
Houston, TX 77204-3010

Abstract

We present a cooperative distribution protocol requiring clients that watch a video to forward it to the next client. As a result, the video server will only have to distribute parts of a video that no client can forward. Our protocol works best when clients have sufficient buffer capacity to store each video they are watching until they are done: when this is the case, the instantaneous server bandwidth never exceeds the video consumption rate.

In addition, we also show how multicasting can further reduce the server and the network bandwidth requirements of the protocol.

1. Introduction

One of the major impediments to the success of video-on-demand (VOD) services is their high bandwidth requirements. Assuming that the videos are in MPEG-2 format, each user request will require the delivery of around 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 gigabit per second to accommodate one thousand concurrent customers.

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.

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 computer or on their television set. They do not address the case of collaborative video-on-demand services where customers are 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] partitions each file to be distributed into fixed-size pieces and penalizes customers that are not willing to redistribute the file pieces they have already downloaded.

We propose to extend the same philosophy to VOD services. Essentially, we require each client watching a video to forward the video data it has received to the next customer requesting the video. We also limit this obligation to the time window during which the customer actually watch the video. As a result, some fraction of the video will still have to be distributed by the server itself.

This new organization has two major advantages. First, it considerably reduces the server workload. Second, it is surprisingly simple and does not require any multicast capability from any of the entities involved in the video distribution process.

The remainder of the paper is organized as follows. Section 2 reviews previous work on reactive video distribution protocols. Section 3 introduces our protocol and section 4 discusses its performance while Section 5 discusses a possible extension using multicast to reduce the bandwidth requirements of our protocol. 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. Some strategies even involve delaying customer requests for a short period of time in order to increase the number of customers sharing the same data stream. *Piggybacking* [9] can be used alone or in combination with batching. It 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.

[1] Supported in part by the National Science Foundation under grant CCR-9988390.

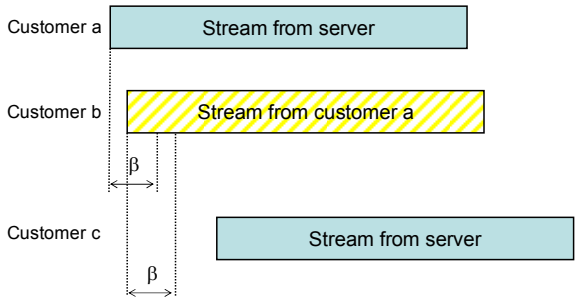

Figure 1: How chaining works

Chaining [12] 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] or *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 $\Delta >$ β. 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.

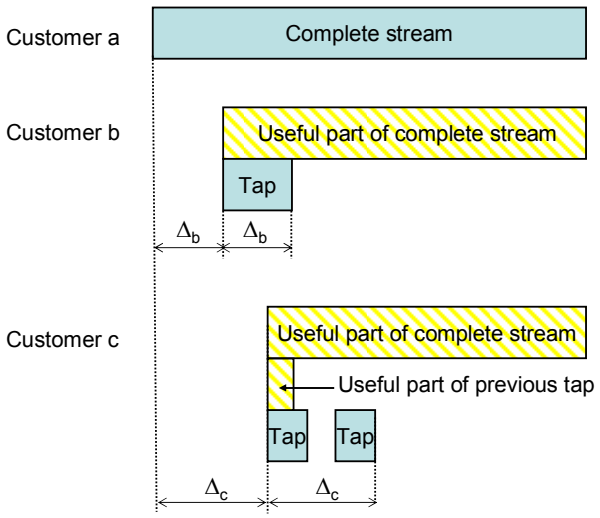

Figure 2: How stream tapping works

To use tap streams, clients only have to receive at most two streams at any one time. If they can actually handle a higher bandwidth than this, they can use an option to 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 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

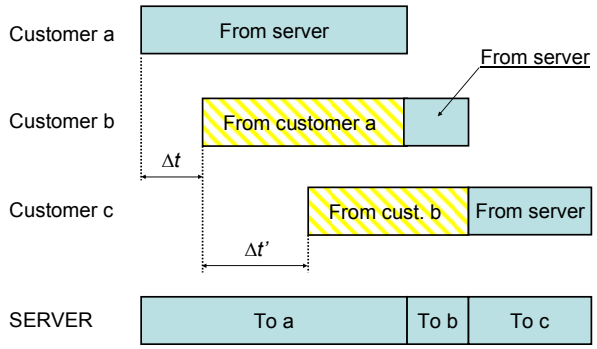

Figure 3: How our cooperative protocol works

broadcast cycle. As a result, its bandwidth requirements areO(log($\lambda_t L_i$) where λ_t is the request arrival rate and L_t the duration of the video [8].

3. Our Protocol

We wanted a protocol that would minimize the server's workload without imposing any unnecessary requirements on the service clients. We thus assumed that:

1. clients would never have to receive video data at a rate exceeding twice their video consumption rate;
2. clients would never have to forward video data at a rate exceeding that same video consumption rate;
3. clients should not have to forward video data after they have finished watching a 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. The sole notable requirement is that requires clients capable of forwarding video data at a rate equal to the video consumption rate, which excludes clients connected to the Internet through a link having insufficient upload bandwidth. In contrast, the protocol requires a rather large client buffer as storing an entire video in MPEG-2 format requires a few gigabytes. Several factors motivated this choice. First, requiring clients to store the previously viewed portion of the video they are watching is essential to guaranteeing that the instantaneous server bandwidth will never exceed the video consumption rate. Second, 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. Finally, we expected many clients to keep in their buffer the previously viewed portion of the video they are watching in order to provide the equivalent of a VCR rewind feature.

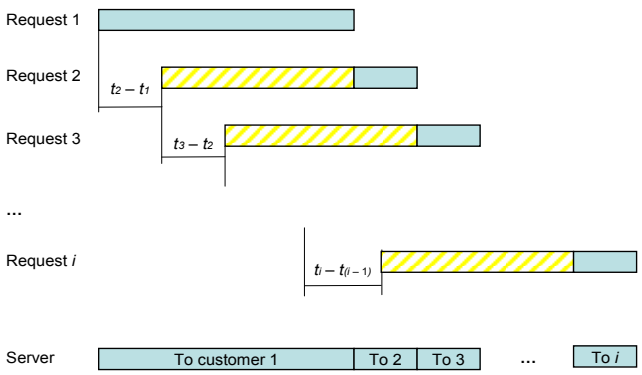

Figure 4: How the protocol handles a batch of requests

Consider now a video of duration D and a request for that video from a customer c arriving at the server at time t. LetΔt denote the time interval between that request and the last request for the same video and let b designate the customer who issued that last request. Our basic cooperative protocol will operate in the following fashion:

1. If $\Delta t \geq$ D, there is no overlap between the current request and the previous request for the same video; the server will then initiate a new transmission of the video, starting at time t and ending at time $t+D$.
2. If $\Delta t <$ D, there is an overlap between the current request and the previous request for the same video; as the servicing of that previous request will end at time $t-\Delta t+D$; the server will thus instruct client b to forward the first $D-\Delta t$ minutes of that video to client c. In addition, it will schedule a transmission of the last Δt minutes of the video to client c, starting at time $t-\Delta t+D$ and ending at time $t+D$.

Consider for instance how the protocol would handle the three requests displayed in Figure 3. The first request to the video will be entirely serviced by the server. The second request arrives before the first request is still being serviced. The server will thus instruct customer a to forward the first $D-\Delta t$ minutes of the video to customer b and schedule a transmission of the last Δt minutes of the video to the same customer. Similarly, the server instruct customer b to forward the first $D-\Delta t'$ minutes of the video to customer c and schedule a transmission of the last $\Delta t'$ minutes of the video to the same customer.

More generally, the amount of time spent by the server to service a request will always be given by max(D, Δt), where D is the duration of the video and Δt is the time interval between the request being serviced and its immediate predecessor. In addition, the service times of these requests will never overlap, which means that the server instantaneous bandwidth will never exceed the video consumption rate.

To demonstrate this property, let us consider an arbitrary batch of n consecutive requests for a given video. Let us further assume (a) that these requests overlap, that is, they arrive at arbitrary times $t_1, t_2, \ldots, t_n$ such that $t_{i+1} - t_i < D$ for $i = 2, 3, \ldots, n$ and (b) that the first request does not overlap with any previous request. As seen on Figure 4, the server will start servicing the first request at time t_1 and end it at time $t_1 + D$. The second request will get the first $D(t_2 - t_i)$ minutes of the video from the customer that issued the first request, and the remaining $t_2 - t_1$ minutes from the server, which will start its transmission at time $t_1 + D$ and end it at time $t_2 + D$. Similarly, the third request will be get the first $D - (t_3 - t_2)$ minutes of the video from the customer that issued the second request, and the remaining $t_3 - t_2$ minutes from the server, which will start its transmission at time $t_2 + D$ and end it at time $t_3 + D$. More generally, the ith request will be get the first $D - (t_i - t_{i-1})$ minutes of the video from the customer that issued the $(i-1)$th request, and the remaining $t_i - t_{i-1}$ minutes from the server, which will start its transmission at time $t_{i-1} + D$ and end it at time $t_i + D$. In other words, the server will never have to send data to a customer before it has finished sending data to the previous customer. Since the server always sends these data at the video consumption rate, its instantaneous bandwidth will never exceed that rate

While our protocol is built upon Sheu, Hua and Tavanapong's chaining technique [12], it makes a much more extensive use of the wider customer buffer spaces that are available today. As a result, it requires much less server bandwidth.

We should also note that our protocol does not require clients to receive data at any rate than their video consumption rate and does not require any multicasting either at the server or at the clients.

3.1 Fault-Tolerance Issues

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

There is a simple solution to the problem. Consider for instance the scenario of Figure 3 where customer c receives most of her video data from customer b, who receives most of her video data from customer a. Hence any failure of customer b will immediately stop the flow of data to customer c. Fortunately for us, a failure of customer b will also free customer a from her obligation of forwarding her video data to customer b thus allowing to forward her video data to customer c.

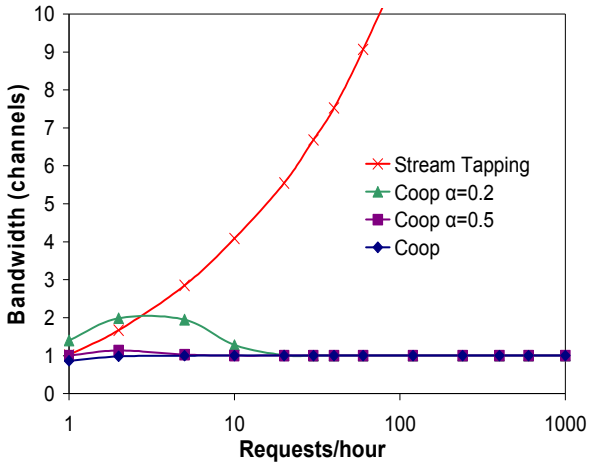

Figure 5: Server bandwidth requirements of the cooperative video distribution protocol

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

3.2 Handling Customers with Small Buffers

We have assumed so far that all customers had enough buffer space to store the previously viewed portion of the video they are watching until they have finished watching it. This will not always be possible either because the limitations of some customer equipment or the durations of some videos.

Consider for instance the case when customers can only store in their buffers a fraction α of the video they are watching. These customers will only keep in their buffers the first minutes of the video for αD minutes. After that, these data will be overwritten.

The easiest way to integrate these customers into our cooperative video distribution protocol is to modify the threshold used to decide whether the server will initiate a new transmission of the video. Our protocol now becomes:

1. If $\Delta t \geq \alpha D$, the server will initiate a new transmission of the video.
2. If $\Delta t < \alpha D$, the server will instruct the previous customer to forward the first $D - \Delta t$ minutes of the video to new customer and In addition, it will schedule at later time a transmission of the last Δt minutes of the video.

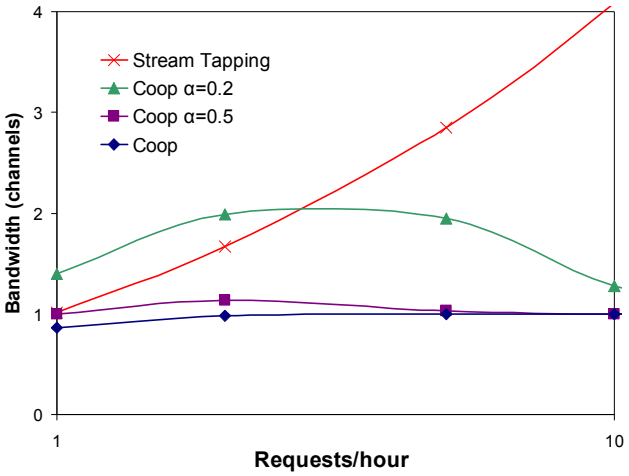

Figure 6: Server bandwidth requirements of the cooperative video distribution protocol at low arrival rates

This simple solution has only one drawback. As we will see, it increases the server workload whenever the time interval Δt is greater than αD but lesser than D.

4. Performance Evaluation

Figure 5 compares the server bandwidth requirements of our cooperative video distribution protocol with those of the stream tapping protocol for arrival rates varying between one and one thousand customers per hour. The two curves labeled "Coop α=0.2" and "Coop α=0.5" refer to versions of the protocol accommodating customers that can only keep in their buffer a fraction α of each video, that is 20 percent of the video for α=0.2 and 50 percent for α=0.5.

Figure 6 displays in more detail the server bandwidth requirements of our protocol at arrival rates lesser than ten arrivals per hour. In both cases, we assumed that the server was broadcasting a two-hour video and that request arrivals could be modeled by a Poisson process. All bandwidths are expressed in "channels," that is, in multiples of the video consumption rate.

As we can see, our new protocol performs much better than stream tapping when customers can store at least 50 percent of the video in their local buffer. Assuming that customers can only store 20 percent of the video significantly increases the server bandwidth requirements of of the protocol for request arrival rates between 2 and 5 arrivals per hour. Even then, the server bandwidth requirements of the protocol never exceed two times the video consumption rate.

In addition, all versions of our protocol perform much better than stream tapping whenever the customer arrival rate exceeds 5 requests per hour.

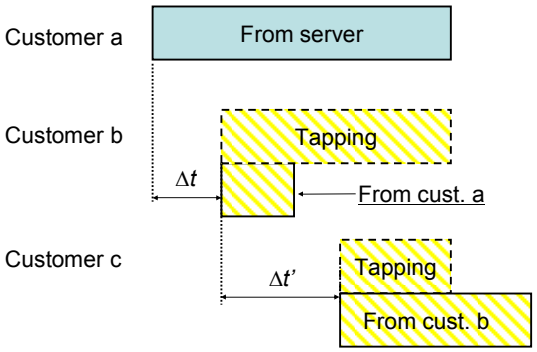

Figure 7: How clients could "tap" data from a server stream

5. A Possible Extension

A major drawback of our protocol is its heavy consumption of network bandwidth. Since it does not use multicasting the total network bandwidth B_N required by our protocol is given by

$$B_N = ADb,$$

where A is the request arrival rate, D the video duration and b the video consumption rate. Consider, for instance, the case of a two-hour video in MPEG-2 format. Its average bandwidth requirements are likely to be around six megabits per second. An arrival rate of 5 customers per hour would then result in an average network bandwidth of sixty megabits per second.

The best way to reduce this bandwidth consumption is through the use of multicast since it would allow the same data streams to be shared by many customers. Given that most of the network bandwidth required by our protocol results from client-to-client data transfers, implementing multicast at the server alone would not solve the problem. Multicast should be used instead by the server and all clients that forward data. This could be done through overlay or application-level multicast [1, 13].

Consider, for instance, the case of a customer b requesting a video just after the previous customer a has received its video data directly from the server. As shown on Figure 7, customer b could get most of its video data by "tapping" into the video stream sent by the server to customer a. This would save bandwidth because

1. customer a would now send to customer b the Δt first minutes of the video rather than the $D - \Delta t$ first minutes of the video, and
2. the server would not have to send to customer b the Δt last minutes of the video.

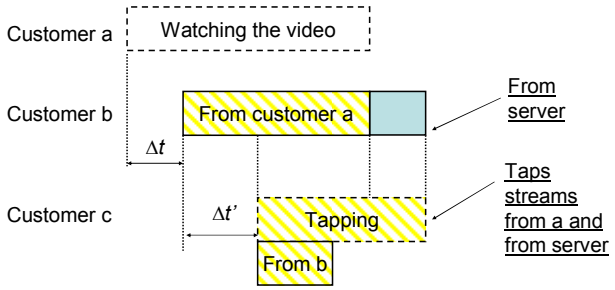

Figure 8: How clients could "tap" data from a client stream

The same approach could be followed by the next customer: it would "tap" into the video stream sent by the server to customer *a* and get from customer *b* the $\Delta t + \Delta t'$ first minutes of the video.

Another significant bandwidth savings could occur whenever a customer *c* follows a customer *b* that got the $D - \Delta t$ first minutes of the video from a previous customer *a* and the Δt last minutes of the video from the server. As shown on Figure 8, customer *c* could tap the streams respectively sent to customer *b* by customer *a* and the server. This would save bandwidth because

1. customer *b* would now send to customer *b* the $\Delta t'$ first minutes of the video rather than the $D - \Delta t'$ first minutes of the video, and
2. the server would not have to send to customer *b* the $\Delta t'$ last minutes of the video.

Both techniques assume that clients can "tap" a stream while receiving data from another stream, which means that they should be able to receive data from two different sources at a total rate equal to twice the video consumption rate.

Both techniques assume that clients can "tap" a stream while receiving data from another stream, which means that they should be able to receive data from two different sources at a total rate equal to twice the video consumption rate.

Figures 9 and 10 compare the bandwidth requirements of cooperative protocols that use and do not use multicasting to reduce network traffic. As before, we assumed that the server was broadcasting a two-hour video and that request arrivals could be modeled by a Poisson process. To avoid counterproductive tappings, we compared the bandwidth cost of each potential new tap with the bandwidth cost of all requests sharing the same stream. Tapping is then allowed if the former is less than $1 + \varepsilon$ times the latter. Otherwise, a new stream is initiated. This criterion is similar, but not identical to that used by Carter and Long in their stream tapping protocol [2, 3].

Our data show that multicasting would dramatically alter the bandwidth requirements of our protocol. First, the

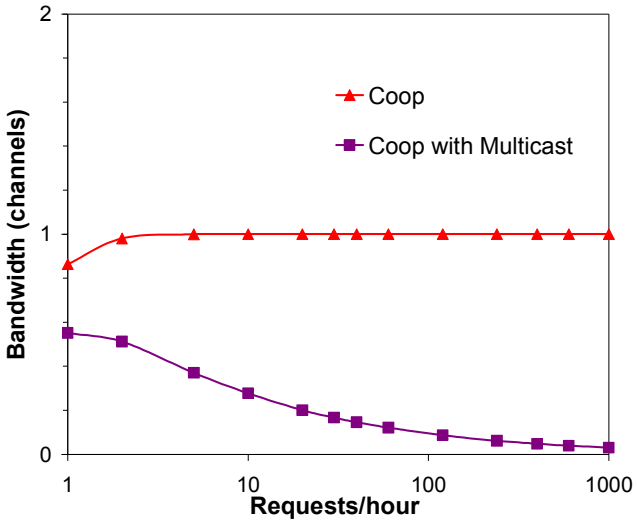

Figure 9: Server bandwidth requirements of the cooperative video distribution protocol with and without multicasting

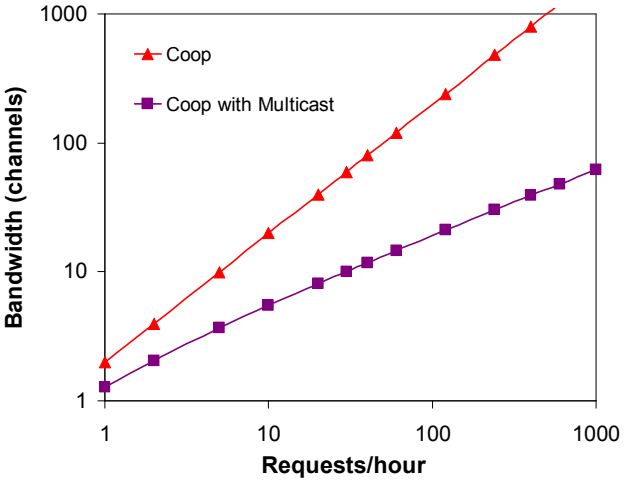

Figure 10: Total bandwidth requirements of the cooperative video distribution protocol with and without multicasting

server bandwidth requirements of the cooperative protocols actually decrease when the customer arrival rate increases above one request per hour. More importantly, the network bandwidth requirements of the protocol always remain below 63 times the video consumption rate, even at customer arrival rates as high as 1,000 requests per hour.

We should emphasize that these results are very preliminary. In particular, we did not investigate the actual limitations of user-level multicast nor did we explore the possible tradeoffs between reducing the server workload and the network bandwidth requirements of our protocol.

6. Conclusions

We have presented a cooperative distribution protocol requiring clients that watch a video to forward it to the next client. As a result, the video server will only have to distribute the parts of a video that no client can forward. Our protocol works best when clients have sufficient buffer capacity to store the previously viewed portion of the video they are watching until they have finished watching it. In that case, the server bandwidth requirements of the protocol never exceed one time the video consumption rate.

We also showed how multicasting can further reduce the server and the network bandwidth requirements of the protocol. More work is still needed to investigate the actual limitations of user-level multicasting as well as cooperative protocols taking into account the physical locations of the customers on the network.

Acknowledgements

We wish to thank Professor Ying Cai for having pointed to us the chaining method.

References

[1] Banerjee S., C. Kommareddy, K. Kar, B. Bhattacharjee, . 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*, pages 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. *Proceedings of the ACM SIGCOMM '97 Conf.,* pp. 89–100, Sept. 1997.

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

[12] Sheu, 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.

[13] 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.