Range Multicast for Video On Demand

Kien A. Hua *Duc A. Tran*
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816-2362, U.S.A.
Email: {kienhua,dtran}@cs.ucf.edu

Abstract

We explore a communication paradigm for video on demand, called Range Multicast. This scheme is a shift from the conventional thinking about multicast where every receiver must obtain the same data packet at any time. A range multicast allows new members to join at their specified time and still receive the entire video stream without consuming additional server bandwidth. Clients enjoy better service latency since they can join an existing multicast instead of waiting for the next available server stream. We also present techniques to support video-cassette-recorder-like interactivity in this environment. Unlike existing methods which require clients to cache data in a private buffer, the Range Multicast solution utilizes the shared network storage to make more efficient and cost-effective use of the caching space. Furthermore, since a range multicast can accommodate clients with different play points in the video, a client has a better chance to join an on-going multicast for normal playback after finishing a VCR operation. This strategy avoids the need for a new server stream, and thus further alleviates the server load. Our simulation results confirm the aforementioned benefits.

Keywords: Multimedia Communications, Video on Demand, VCR-like Interactivity, Overlay Multicast, Caching.

1 Introduction

The ability to interact with video using VCR-like operations such as fast-forward and rewind is highly desirable for *video-on-demand* (VOD) applications. Currently, this feature is implemented in two ways: *server-controlled* or *client-controlled*. The server-controlled approach [16] relies on the server to selectively send the video frames at possibly higher, or lower, transmission rates. These solutions usually focus on data placement strategies to minimize the costs of retrieving video frames from the storage subsystem [42, 9, 38]. In the client-controlled approach [8, 10], each client station is equipped with a buffer for caching recently displayed video frames[1]. When a rewind is invoked, the video player renders the video frames in the buffer in reverse order at a higher frame rate. Other VCR operations are also supported, though, limitedly.

The advantages of the server-controlled approach are the simplicity of the client software and the ability to provide unrestricted VCR services such as fast-forwarding the entire video. This approach, however, is not scalable since the server must handle VCR requests for all the clients. With rapid drop in the costs of disk drives, the client-controlled approach has become the preferred technique to achieve better scalability. This method also allows the VOD system to leverage multicast technologies to reduce the demand on server bandwidth [13, 3, 28, 1, 17]. When a client wishes to resume the normal play after a VCR action, it attempts to join an existing multicast group whose play point is near the resumption point. Only if this join fails, the server needs to allocate a new stream to provide the service, or a patching stream as in the Patching technique [20, 7, 36] to help the client catch up with an on-going multicast. An efficient implementation of such an environment requires addressing two issues: (1) The Internet is based on IP Unicast. We need to extend it to provide multicast services; and (2) The server has to allocate many patching streams for those clients who resume their normal play after VCR operations. We need a new mechanism to significantly reduce the number of these server streams.

The second issue has not been addressed to date. To address the first issue, one can install software routers at strategic locations on the Internet to create an overlay structure [18, 33, 25, 11, 4]. These routers can realize various multicast trees, at the application level, by relaying data from the server toward multicast members. In this paper, we extend this approach by introducing a new multicast concept called *Range Multicast* (RM). RM provides the following new features to better support video-on-demand applications:

- RM enables clients to join a multicast at their specified time and still see the entire video *without* consuming any server bandwidth. The motivation is twofold. First, since clients do not have to wait for the next multicast, truly on-demand services can be achieved. Second, since the server does not need to multicast frequently to keep the service latency low, the

[1]TiVo (www.tivo.com), a VCR-enabled commercial product for live TV, could be considered falling in this approach.

demand on server bandwidth is less.

- Unlike the conventional multicast which requires all members to share a common play point at all time, RM can support a range of different play points simultaneously. Since it is much easier for a client to find a multicast range (as opposed to a multicast point) to resume a normal playback, RM significantly increases the chance for a client to join an existing multicast after a VCR-like operation.
- RM directly supports VCR-like operations. Client buffers are not necessary. This feature is particularly important to wireless mobile applications.

We note that Patching [20, 7, 36] also allows a new client to join an on-going multicast and still receive the entire video stream. This solution however requires the server to send a new patching stream for each new client. As this client displays the first part of the video arriving on the patching stream, it caches the multicast stream in a buffer. When the patching stream terminates, the client switches to playback the data prefetched in this local buffer while the multicast stream continues to arrive. It has been recognized that standard multicast is inadequate for VOD applications. Patching addresses this drawback by supplementing each multicast stream with many patching streams. In this paper, we address the same issue by changing the multicast concept itself, namely introducing Range Multicast. It is clear that Patching is significantly more demanding on server bandwidth than Range Multicast is.

Periodic Broadcast (e.g., [21, 31, 26]) is another efficient solution for video applications. This scheme fragments each video into data segments, each broadcast repeatedly by the server. To receive a video, a client tunes to the appropriate broadcasts to download the data. The communication protocol ensures that the broadcast of the next video segment is available to the client before the playback of the current segment runs out. To provide VCR-like services, the client-controlled approach can be used as in [17, 31]. Alternatively, periodic broadcast can also be applied to a compressed version of the same video to support VCR-like interactions [39]. Although periodic broadcast is highly efficient, it is only suitable for very popular videos. In this paper, we address general applications with video data of diverse popularity.

The remainder of this paper is organized as follows. Section 2 describes the basic concepts of Range Multicast. Section 3 presents the algorithms for satisfying VCR interaction requests. Section 4 reports the performance results from our simulation study. Finally, Section 5 concludes this paper with our remarks.

2 The Range Multicast Approach

The *Range Multicast* (RM) concept can be implemented at the network layer. We presented the RM router and the network-layer protocol in [22]. To adapt this communication paradigm for the

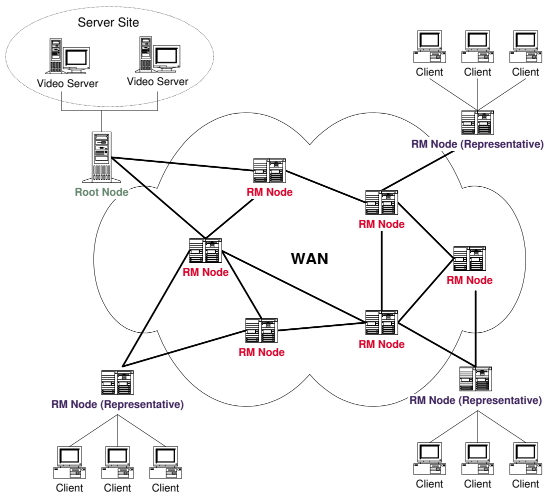

Figure 1: Range Multicast Overlay

Internet, we propose in this section an application-level implementation of RM. We also extend it to support VCR-like operations.

2.1 Network Architecture

In a Range Multicast environment, RM-enabled nodes are placed on the Internet, and interconnected using unicast paths to implement the range-multicast layer. In other words, these RM-enabled nodes act as software routers for the overlay structure. For ease of exposition, we refer to these RM-enabled nodes simply as "nodes" unless otherwise specified. They are classified into two types: *root* and *non-root*. The root node is the front-end node for the video server to communicate with the rest of the overlay network. A non-root node may or may not be representative for a local community of clients. When representing a community, a non-root node is referred to as the *representative* for each of the community members. To illustrate this RM architecture, Fig. 1 shows an overlay topology with a root node and nine non-roots, three being representative nodes. The links are logical and represent how data will be forwarded on the overlay network. The path for a video stream will be a path on this topology.

The motivation for using the overlay architecture is twofold. First, it has been shown to be an efficient solution to enabling multicast services on the Internet [25]. Second, extensions can easily be made to the overlay to support new needs. In this paper, we exploit it to support VOD services with VCR-like functionality. A problem arising is how to determine the overlay topology and how to reduce the costs of maintaining it under network dynamics such as node failures. For failures, we consider two kinds of impact: on the overlay topology and on the on-going services. Several solutions including [18, 33, 25, 11] were proposed for constructing fault-tolerant overlays, whose topology is re-configurable on failures, either statically or dynamically. Such a solution can be appropriately adopted to build and maintain our RM overlay topology. For on-going services interrupted due to a node failure, all the adjacent nodes that have been receiving data from this failed node temporarily reconnect to the root node to receive the remaining data. Even though possibly sub-optimal, this solution offers a simple and quick way to guarantee the continuity in the client playback. Since techniques addressing the above issues can be employed in our scheme, without loss of generality, we assume that the RM overlay has a predefined topology and is failure-free.

2.2 Range Multicast

We assume that a video is transmitted as a sequence of equal-sized blocks. The transmission time for each block is one time unit. A block may contain many video frames (e.g., I, P, or B frames in MPEG format). On receipt of these blocks, the client is responsible for decoding and rendering them onto the screen.

Range Multicast is realized by caching data in the overlay nodes. As a video stream passes through a sequence of nodes on the delivery path, each caches the video data into a fixed-size FIFO buffer. As long as the buffer is not full, it can be used to provide the entire video stream to subsequent clients requesting the same video. A nice property of this strategy is that all the buffers fed by a single server stream form a continuous video segment, which gets larger as more clients join. This video segment including past and future frames enables existing members to do VCR interactions. The wide range of available data also improves the chance for clients to join the multicast if their resumption or starting point is within the range. Unlike the client-controlled approach, clients share buffers in Range Multicast allowing a more efficient use of the caching space. We discuss the Range Multicast technique in more detail in the following subsections.

2.2.1 Overlay Cache Design

RM does not require every node to cache. For example, the root does not need to cache. For each non-root node caching, an amount of the local storage is reserved for caching purposes, which is organized as an array of equally sized chunks. Each of these chunks is used to cache a particular video stream currently passing through the node. The number of chunks is dependent on the number of streams the node can support simultaneously. Once this parameter has been determined, the available caching space is equally divided among the chunks.

Consider a node R having a number of chunks, each of size $\Delta(R) > 0$. When a new stream S arrives at node R at time t_0, this node finds a chunk, say chunk n, that will be used for caching S. The video-block replacement within this chunk follows the Interval Caching approach [12, 22]. Every time a new data block of S arrives, it is copied into chunk n. This chunk can be used to serve all clients requesting the same video, whose requests arrive at node R before time $t_0 + \Delta(R)$. In this case, chunk n would be used as a sliding window to hold and forward the data to those clients. At time $t_0 + \Delta(R)$, chunk n is full of data blocks. If currently not used to serve any client, it is cleaned up and returned to the free pool. Otherwise, it continues caching as usual and old frames will be replaced with newly arriving frames according to the FIFO policy. To see how this caching strategy helps reduce the server load, we consider an example where there is an overlay node R with a single chunk. Suppose that the request rate is λ requests per time unit and the probability of requesting a video V is p_V. Caching the video stream V for a request into that chunk will enable serving $\lambda \times \Delta(R) \times p_V$-1 other requests, on the average, without any server bandwidth.

Given that a node may have more than a chunk, the chunk replacement works as follows. We note that chunk replacement is different from block replacement done within a chunk. A chunk replacement is invoked when a new video stream arrives at a node and needs a chunk to be cached into. If all the chunks are currently serving, no caching is performed at the node, i.e., the node just forwards the data to the next node on the delivery path. Otherwise, to find a victim chunk to feed data in, we select the chunk that has not been used to service any downstream client for the longest time; the current caching age of this chunk is oldest among all non-serving chunks (a free chunk has an infinite age). By replacing an old caching chunk with a younger one, we widen the interval of arriving requests that can be served by the newly cached data. We can also employ another chunk replacement policy based on the popularity of video titles if that is known. In this case, the victim chunk would be currently caching the least popular video but not serving any client. The decision on which of the two chunk-replacement policies above is employed depends on the application in operation, hence left as a parameter of the overlay nodes.

2.2.2 Service Procedure

A client C requests a video V to its representative *Rep*(C). This representative "broadcasts" the request in a **find** packet to the overlay nodes. By broadcast, we mean to apply on the overlay layer only. In other words, a node forwards the packet to all adjacent-on-overlay nodes on the corresponding unicast paths. A duplicated packet arriving at a node is ignored. Since the number of overlay nodes is not large, the network load incurred by this broadcast should not affect the network traffic severely.

Upon the first arrival of the **find** packet, each receiving non-root node R checks if the first block of V is being cached in any of its chunks. If this condition holds, R stops forwarding and sends a **found** message on a direct unicast to *Rep*(C). This **found** informs that client C can download the video from R, which requires no server bandwidth as mentioned in Section 2.2.1. If no non-root node caches the first block, the **find** will eventually go to the root which also stops forwarding and sends a **found** to *Rep*(C) to notify.

There may be more than one node sending a **found** to *Rep*(C). *Rep*(C) selects the earliest informing node, say R_{serv}, by sending an **ack** message to it and sending **negative-ACK** messages (**nack**) to the other informing nodes. We pick up the "earliest" node to reduce the service start-up delay. The video data will be sent from R_{serve} on a delivery path down to the client. This delivery path is the reversal of the path on which the **find** request is sent from the client to R_{serv} which is also called the "serving node" of the client. As video blocks are sent to the client, each intermediate node on the delivery path caches them into a victim chunk if there is any available. These cached data can be used to service subsequent requests (see Section 2.2.1).

In summary, to start a service for a new client essentially requires the following steps: (1) Upon receipt of a request from a client C, its representative $Rep(C)$ broadcasts a **find** to the overlay; (2) C will get the video stream from the node R_{serv} that keeps any prefix of the video in its cache and that informs $Rep(C)$ first; and (3) The delivery path for this stream is reverse with the path of the **find** when traveling from $Rep(C)$ to R_{serv}.

It is optional but we can employ an admission control policy to prevent bottleneck at overlay nodes and congestion on overlay links. For this purpose, a threshold for each overlay link is set to control the number of streams that link can support concurrently. Any **find** will be rejected by an overlay link if its threshold is already reached. The threshold for a link can be a predefined number or dynamically computed based on the traffic status of that link. This dynamic adjustment is a popular issue mentioned extensively in overlay multicast works [18, 33, 25, 11, 23]; therefore, we omit describing it in our paper.

We note that there might be a delay between the time R_{serve} sends the **found** message to $Rep(C)$ to declare itself as a potential server and the time it receives the **ack** message from $Rep(C)$. Therefore, if R_{serv} is a non-root node, it might have dropped the first video block from the cache by the time it receives the permission. Hence, R_{serv} could not fully service client C. A way to avoid this is to take into consideration the end-to-end delay (based on the timestamp in the **find** message) when a non-root node assesses its ability to satisfy a request.

To disconnect service, client C sends a **quit** message in the reverse direction of the delivery path to its serving node. Upon receipt of this **quit**, each intermediate node is pruned off the delivery path if not currently using the data destined for C to serve any other client; then this node forwards **quit** to the next node on the reverse path. If an intermediate node uses the data destined for C to serve at least a client, this node just removes client C from its delivery schedule and does not need to forward **quit** to the upstream.

2.2.3 Example

To illustrate how the proposed mechanism works, we give an example in Fig. 2. We assume that each non-root node has only a chunk that can cache up to 100 video blocks. The overlay has a topology drawn in Fig. 2(a). All videos are assumed longer than 100 time units. The label on

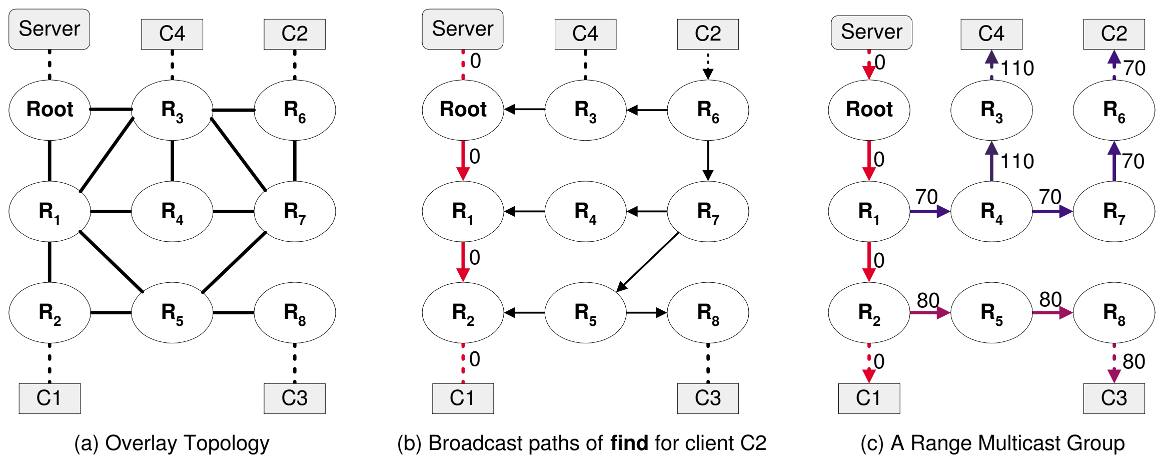
Server
C4
C2
Root
R3
R6
R1
R4
R7
R2
R5
R8
C1
C3
(a) Overlay Topology
(b) Broadcast paths of **find** for client C2
0
(c) A Range Multicast Group
110
70
80

Figure 2: Example of how a range multicast group is built

each link indicates the starting time of the service of a particular client. For instance, label "0" indicates that at time 0, the server starts delivering the video to client C_1 over the nodes *Root*, R_1, R_2. For simplicity, we assume that there is no service delay. That is, C_1 can make a request at time 0 and receive the first block of the data stream instantaneously. Fig. 2(b) illustrates the following scenario. At time 0, a node C_1 requests a video V. Since no node caches the data, the root has to allocate a new stream to serve C_1. As the data go toward C_1, all the non-root nodes along the way, R_1 and R_2, cache the data in their chunk. At time 70, client C_2 requests the same video V. As a result, its representative R_6 broadcast a **find** to the overlay. Suppose that the paths for this broadcast follow directions in Fig. 2(b) and the **find** will stop at *Root*, R_1, and R_2 since, at time 70, R_1 and R_2 have not dropped the first video block from their chunk. These three nodes can serve C_2. Suppose that R_1 informs R_6 first and therefore is selected to serve C_2. The delivery path for this client will be $R_1 \rightarrow R_4 \rightarrow R_7 \rightarrow R_6 \rightarrow C_2$ as shown in Fig. 2(c). All the nodes along this path are asked to cache the video. The subsequent events for client C_3 and client C_4 are handled similarly as illustrated in Fig. 2(c). At time 80, client C_3 also requests video V and gets the service from node R_2 which still holds the first video block. At time 100, R_1 and R_2 remove the first video block from their cache. At time 110, C_4 asks for video V and can receive the service from node R_4 which still holds the first block. In this example, we have a multicast tree including four clients C_1, C_2, C_3, and C_4 which join at different times: 0, 70, 80, and 110, respectively. Only a server stream is needed to provide data to all of them.

2.2.4 Remarks

In general, by properly utilizing network caches in RM, clients have a significant chance to join an existing server stream at different times almost instantaneously and get the entire video. Several other VOD techniques [20, 35, 5, 2, 19, 27, 41, 40] have this feature however significant additional server bandwidth must be allocated in order to achieve that. In contrast, RM does not require any server bandwidth for late clients to join an existing server stream. In the case a client requests the same video very late, a new multicast has to be created as in other multicast-based VOD schemes. However, creating a new multicast in RM richens the content cached inside the network, thus increasing the chance for subsequent requests to be served by a caching node.

One might argue that forcing all intermediate nodes on a delivery path to cache a video stream is superfluous and prefer caching at several selected nodes. However, our caching policy has its advantages:

- A caching chunk is emptied when it is full and not used to serve any downstream client, thus the lifetime of the data cached in a chunk is short. As a result, caching a stream at all intermediate nodes of a delivery path does not significantly reduce the caching space for other streams. Furthermore, this does not introduce notable complexity.
- Caching a stream at all nodes of a delivery path significantly increases the chance for subsequent clients to hit the caches and decreases the cache seeking time.

We define a *multicast group* to include all the clients fed by a single server stream. This is different from the multicast group concept in IP multicast since in IP Multicast group members must join a multicast at the same time so that they do not miss any data. Consequently, they always have the same playback point. On the contrary, data delivered in our multicast group at any point of time are different and collectively form a range of consecutive frames. This multicast range is useful for VCR interactions as presented in the following section.

3 VCR Functionality using Range Multicast

In this section, we present the potential of the Range Multicast scheme in providing VCR functionality to the clients.

3.1 Preliminaries

In the system, at any point of time there are a number of on-going (range) multicast groups. Data provided to each group are delivered on a set of delivery paths, each destined for a client and emanating from its serving node. For example, in Fig. 2(c), the deliver paths for the four clients are $Root \rightarrow R_1 \rightarrow R_2 \rightarrow C_1$, $R_1 \rightarrow R_4 \rightarrow R_7 \rightarrow R_6 \rightarrow C_2$, $R_2 \rightarrow R_5 \rightarrow R_8 \rightarrow C_3$, and $R_4 \rightarrow R_3 \rightarrow C_4$, respectively. The delivery paths for clients of a group form a *delivery graph* of that group. The property of this graph is that it is directed, single-rooted at the root node, and contains a unique path to each client from its serving node. Without loss of generality, we consider only one multicast group and focus on the VCR interactivity of a participating client. Although our technique can easily be extended for working with multiple chunks, we assume for clarity that every non-root node on the delivery graph has a single caching chunk. We use the following notations:

- $Serv(C)$: The serving node of client C. E.g., in Fig. 2(b), $Serv(C_1) = Root$, $Serv(C_4) = R_4$.
- If a node R is currently caching data originally requested by a client C', the serving node of R is $Serv(R) = Serv(C')$. E.g., in Fig. 2(b), $Serv(R_4) = R_1 = Serv(C_2)$. We use the same notation *"Serv"* for both overlay nodes and clients, but the meaning depends on the parameter. That is, $Serv(X)$ alludes to the serving node of a node if X is a node, and the serving node of a client if X is a client.
- $Rep(C)$: The representative node of client C.
- $\sigma(R) = [a, b]$: The content currently cached at a non-root node R where a is the identifier of the first block cached and b is that of the last block cached. If R currently is not caching, $\sigma(R) =$ NULL. We note that this content changes over time due to the sliding nature of the caching buffer. For instance, in Fig. 2(b) of Example 2.2.3, the content cached at node R_1 at time 20 is [0, 19], at time 30 is [0, 29], at time 120 is [20, 119], at time 121 is [21, 120], etc.

The VCR functionality is enabled in the Range Multicast scheme by capitalizing a useful property stated below:

"If a node R is currently caching data arriving from its serving node $Serv(R)$, then $\sigma(Serv(R))$ holds next future blocks of $\sigma(R)$ which in turn holds recent past blocks of $\sigma(Serv(R))$."

Please refer to Fig. 3 as an example. Indeed, at time 200, $\delta(R_1) =$ [100, 199], $\delta(R_4) =$ [30, 129], $\delta(R_3) =$ [0, 89]. We know that $R_1 = Serv(R_4)$ and $R_4 = Serv(R_3)$, and clearly, the above property applies for these three nodes.

Let $\Sigma(T) = \bigcup_{nodeR \in T} \sigma(R)$, the aggregate content cached in all nodes of a delivery graph T. When a new client gets served by a node belonging to T, $\Sigma(T)$ will be augmented to the left. The meaning of this is interpreted as follows:

- For the latest client C_{latest} participating in graph T, $\Sigma(T)$ holds future blocks needed for next display by C_{latest}. These cached blocks can be used for VCR forward actions. For VCR backward actions, C_{latest} can play the blocks in its local buffer if this buffer is available to cache recently displayed blocks, or request a new stream from the server if there is no such buffer. However, for this client, forward actions are much more likely than backward actions since the play point of C_{latest} is more likely to be close to the beginning of the video. Therefore, the number of server streams required by the interactivity of C_{latest} is significantly alleviated.

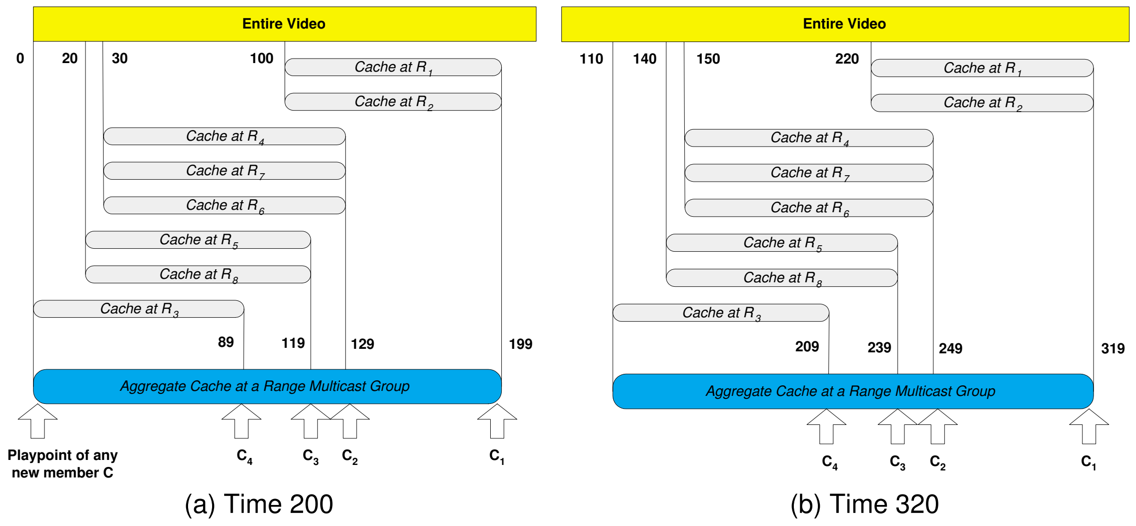

Figure 3: Aggregate Buffer of VCR Operations

- For the earliest client $C_{earliest}$ participating in graph T, $\Sigma(T)$ holds past blocks recently displayed by $C_{earliest}$. These blocks can be used for VCR backward actions. For VCR forward actions, $C_{earliest}$ requests a new stream from the server. However, for this client, backward actions are more likely than forward actions since the play point of $C_{earliest}$ is more likely to be close to the end of the video. Therefore, similar to the case of C_{latest}, the number of server streams required by the interactivity of $C_{earliest}$ is also alleviated.

- For any client C_{random} that joins in between, $\Sigma(T)$ holds both recent past and future blocks of client C_{random}, which can be used for both forward and backward actions.

Let us follow Example 2.2.3 where each node has a chunk capable of caching 100 video blocks and four clients C_1, C_2. C_3, and C_4 join the same multicast group at time 0, 70, 80, 110, respectively. Fig. 3(a) demonstrates the cached content for every node at time 200 while C_1, C_2. C_3, and C_4 are playing at positions 200, 130, 120, 90, respectively. Even though the chunk size is only 100 video blocks, the aggregate cache in this multicast group is [0, 199]. Similarly, Fig. 3(b) demonstrates the cached content for every node at time 320 while C_1, C_2. C_3, and C_4 are playing at positions 320, 250, 240, 210, respectively. The aggregate cache in this multicast group is changed to [110, 319]. This aggregate buffer can be used to fulfill VCR backward operations of C_1, both forward and backward operations of C_2, C_3, C_4, and fast-forward operations of any new member C who just joins the multicast. In this example, with only four clients, the aggregate buffer available for VCR operations is twice as large as the buffer at each node. In the general case, the number of clients should be large and we can expect that the aggregate buffer will be larger.

Prior to explaining VCR actions and strategies to fulfill them, we note that it is possible for a node to recognize a frame in its cache. This is because when a video file is divided into blocks for transmission at the server sites, we add control information into each block to tell what frames belong to it and what their position in the block is. Therefore, for simplicity, we view the cache content at a node as an interval of frames (which actually are cached as blocks).

3.2 Pause (PS) and Jump Backward (JB)

By invoking a PS operation, the client pauses the playback until invoking a Normal Play (NP) operation. By invoking JB with a skip distance δ, the client would like to immediately play normally from a frame which is δ frames offset from the current play point. A common feature of PS and JB is that the new play point is always a position in the *past* of the normal playback.

As soon as a client C invokes a PS or JB action, the client immediately quits its current multicast group. When C wants to resume its normal playback at position h (i.e., the h^{th} frame), it sends a **past-play**(h, C) (meaning "play at past position h") request to node $R_0 = Rep(C)$. If this node does not currently hold h, the **past-play**(h, C) request travels up to node $Serv(C)$ on the reversed delivery path. There are two cases:

1. **past-play**$(.)$ reaches an intermediate node R_1 that is currently serving another client, say C_1, by using the cached content $\delta(R_1)$: Since there is a high chance that a node along the path from R_1 to C_1 is currently caching earlier frames, the **past-play**$(.)$ request is forwarded to nodes on this path to find one still holding frame h. If this request then reaches an intermediate node R_2 that is serving another client, say client C_2, the request again is forwarded to the path from R_2 toward C_2. This process of finding frame h repeats and eventually, the **past-play**$(.)$ request will reach a node R^* that is currently caching frame h, or the root of graph T has to intervene to provide the needed frames to client C. In the former case, C will join R^* by using a *direct* unicast to connect and download data cached there as usual as we discussed in Section 2.2.

2. **past-play**$(.)$ reaches $Serve(C)$ which is not currently serving any client by using the cached content $\delta(Serve(C))$: The root of graph T has to intervene to provide the needed frames to client C.

We note that the traveling of the **past-play**(h, C) request is not a broadcast to the entire graph, but only searches on a subset of those paths leading to the clients that are admitted to the system *after* C is. Hence, the seeking time should be short.

3.3 Jump Forward (JF)

A client currently playing normally at position $h_{current}$ would like to jump to position $h = h_{current} + \delta$ if the client invokes a JF action with skip distance δ. We present how to fulfill a JF action as follows.

As soon as a client C invokes a JF action, the client immediately quits its current multicast group. If C wants normal playback at position h, it sends a **future-play**(h, C) (meaning "play at future position h") request to node $Rep(C)$. Since $Serv(C)$ is caching the future frames of C, the representative node forwards the **future-play**(h, C) request to $Serv(C)$. If frame h is still in the caching chunk of $Serv(C)$, C will join this node and download data from its caching chunk as in Section 2.2. Otherwise, $Serv(C)$ will forward the **future-play**(h, C) message to its serving node $Serv(Serv(C))$ since $Serv(Serv(C))$ is caching frames that are future of that cached in $Serv(C)$. This process continues repeatedly until the **future-play**(h, C) request reaches a node R^* that is currently caching frame h. C will join R^* and download data cached there. We note that the traveling of the **future-play**(h, C) request is not a simple broadcast to the entire graph, but only searches on a subset of those paths leading to the clients that are admitted to the system *before* C is. If such a node R^* is not found, the root node of graph T will serve C.

3.4 Fast Forward (FF)

A client currently playing frame $h_{current}$ by invoking a FF action at rate r (a positive integer greater than 1) would like to play normally frames $h_{current}$, $h_{current} + r$, $h_{current} + 2 \times r$, .., $h_{current} + i \times r$, .., until the client requests a resumption to normal playback.

Since the client wants to play future frames that have yet to arrive at the client, we should capitalize those nodes currently caching data that are originally requested by earlier clients. The normal servicing of a client C can be modeled as a chain $Root \rightarrow R_1 \rightarrow R_2 \rightarrow ... \rightarrow R_n \rightarrow C$ where $R_n = Serv(C)$, $R_{n-1} = Serv(R_n)$, .., $R_1 = Serv(R_2)$, and $Root = Serv(R_1)$ is the root node. R_k is always caching frames that follow the frames cached by R_{k+1} if they are played in normal playback. As a result, $\sigma(R_n) \cup \sigma(R_{n-1}) \cup ... \cup \sigma(R_1)$ forms a large buffer containing the next frames needed by C. By tracing this chain from R_n back to R_1, we will actually go to the next future frames of client C. For instance, following Example 2.2.3, client C_4 fast-forwards by tracing the chain $Root \rightarrow R_1 \rightarrow R_4 \rightarrow R_3$ in reverse direction, i.e., R_3 first (please see Fig. 2).

Client C invokes an FF action by first quitting the normal service and then sending a **ff**(r, h, C) (meaning "fast-forward at rate r starting from frame h") request backward to the chain starting from R_n. Without loss of generality, suppose that we are at node R_k. Upon receipt of **ff**(r, h, C) request, R_k will transmit every r^{th} frames starting from position h in its caching chunk to client C at the normal rate[2]. If the next r^{th} frame needed, say frame $h + i \times r$, lies beyond the chunk, R_k sends a **ff**(r, $h + i \times r$, C) request to the preceding node on the chain, i.e., to node R_{k-1}. R_{k-1} will follow the same steps as R_k does and this process repeats until one the following possibilities

[2]If a video consists of dependably-coded frames of various sizes, such as in MPEG format, transmitting every r^{th} frames would slightly affect the client's playback quality because some r^{th} frames might not be decodable at the client side. One way to avoid this is counting only independently decodable frames, such as I-frames, when seeking the next r^{th} frame.

happens:

1. The root node *Root* receives the **ff**(.) request: Since at this moment the client has moved to a play position that no non-root node has reached, the root has to transmit the needed frames to C at the normal playback rate. The root will create a new stream and send every next r^{th} frame to client C until client C submits a **resume**(C) request.

2. Client C sends a **resume**(C) request: This request is sent to the node R^* that is about to transmit a frame h^* to C for FF purposes. R^* then decides not to skip frames anymore and transmits a frame at a time sequentially from the caching chunk to C. C resumes normal playback by displaying these frames as usual.

3.5 Play Backward (PB) and Rewind (RW)

A client currently playing frame $h_{current}$ by invoking a RW at rate r (a positive integer greater than 1) or a PB (where $r = 1$) would like to play normally frames $h_{current}$, $h_{current} - r$, $h_{current} - 2 \times r$, .., $h_{current} - i \times r$, .., until the client requests a resumption to normal playback.

In these cases, client C wants to play past frames which have already arrived. Therefore, we should take advantage of those nodes currently caching data originally requested by clients who are admitted to the system after C is. The idea is as follows. If there exists a node R_1 on the path from $Serv(C)$ to $Rep(C)$, that is currently serving another client, say client C_1 (i.e., $R_1 = Serv(C_1)$), the data sent from R_1 to C_1 must be the past frames of client C. On the path from R_1 to $Rep(C_1)$, if there exists a node R_2 that is currently serving another client, say client C_2, the data sent from R_2 to C_2 must be the past frames of C_1. Based on this strategy, we may be able to play past frames without server intervention by making use of data in the chunks of R_1, R_2, .., etc.

Let us consider a chain $R_1 \rightarrow R_2 \rightarrow ... \rightarrow R_n$ where:

- R_1 is the first node on the path from $Rep(C)$ to $Serv(C)$, that is serving some client C_1. If such a node does not exist, this chain consists of the root node only.
- R_{k+1} $(1 \leq k \leq n-1)$ is the first node on the path from R_k to the client C_k served by R_k, that is serving some other client R_{k+1}.
- No node on the path from R_n to C_n is currently serving any client.

Clearly, $R_{n-1} = Serv(R_n)$, .., $R_1 = Serv(R_2)$ and $\sigma(R_1) \cup \sigma(R_2) \cup ... \cup \sigma(R_n)$ always holds the recent frames for client C. To have a large size of this aggregate buffer, given node R_k, we should first find a latest client $C*$ served by R_k, then select R_{k+1} to be the first node on the path from R_k to $Rep(C*)$, that is serving at least another client. By tracing the chain from R_1 to R_n, we will actually go to the recent frames needed by client C. For instance, following Example 2.2.3, client C_1 rewinds by tracing the chain $R_1 \rightarrow R_4 \rightarrow R_3$ (Fig. 2).

To invoke a PB action, C quits its normal service and then sends a $\mathbf{pb}(h,\ C)$ (meaning "play backward starting from position h") request to each node on the path from $Rep(C)$ towards $Serv(C)$ until the request reaches node R_1. R_1 then forwards this request to R_2 because R_2 is caching the most recently displayed frames of C. Suppose that at this time, $\sigma(R_2) = [a,\ b]$. To support PB at client C, R_2 will send each frame at a time to client C, starting at frame b and advancing to the left of the chunk (i.e., frames b, $b-1$, $b-2$, etc. will be sent). Since $\sigma(R_2)$ also shifts its content to the left as time advances, at most half of the chunk's content will be transmitted to client C. As the next frame needed to send to C, say frame $b-i$ lies beyond the chunk of R_2, R_2 sends a $\mathbf{pb}(b-i,\ C)$ request to R_3. The same process mimics at R_3, then at R_4 and so on.

To invoke a RW action, C submits a $\mathbf{rw}(r,\ h,\ C)$ (meaning "rewind at rate r starting from position h") request. To satisfy this request is similar to the above case except that instead of transmitting consecutive frames in the chunk, each node R_k only transmits every r^{th} frame to C, starting from the right-most position of the chunk. If C continues to be in PB or RW state, C will eventually need to play a frame beyond the chunk of R_n. In this case, the root node will provide C with the rest of data needed.

To resume normal playback, client C sends a $\mathbf{resume}(C)$ request to the node R^* that is about to transmit a frame h^* to C for PB or RW purposes. R^* then decides to transmit a frame at a time starting from h^*, sequentially and forward, to C. By joining this node, C will return to normal playback by displaying the frames arrived from R^*.

3.6 Slow Forward (SF) and Slow Rewind (SR)

Suppose client C is playing frame h when C decides to play SF at rate $1/r$ (r is an integer greater than 1). C is supposed to display frame h for r time units, then frame $h+1$ for r time units, then frame $h+2$ for r time units, and so on. After r time units, frame $h+1$ would be the past frame of normal playback because if C continued playing normally, C would be playing frame $h+r$. Therefore, in order to get frame $h+1$, we should look for nodes serving clients that are admitted to the system *after* C is. The case for slow rewind (SR) is also similar. After r time units, C needs to play frame $h-1$ for r time units. This frame should also be cached at those nodes that are serving younger clients. For this reason, we use the same strategy for handling both SF and SR.

Let us consider a chain $R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_n$ that is built as in the cases of RW and PB. For SF (or SR) at rate $1/R$, C first plays the current frame h for r times. After that it sends a $\mathbf{slow}(r,\ h+1,\ C)$ (or $\mathbf{slow}(r,\ h-1,\ C)$) request to the chain starting from R_1. This request will reach the first node R_k that has frame $h+1$ (or $h-1$) in its cache chunk. R_k will send frame $h+1$ (or $h-1$) to C (which in turn will play frame $h+1$ (or $h-1$) for r times). If R_k will not drop out frame

$h + 2$ (or $h - 2$) from its chunk in r time units, R_k waits until r time units later and then sends frame $h + 2$ (or $h - 2$) to client C. In the case where R_k will drop out frame $h + 2$ (or $h - 2$) from its chunk in r time units, R_k sends a **slow**(r, $h+2$, C) (or **slow**(r, $h-2$, C)) to the next nodes in the chain until reaching R_l ($l > k$) that has frame $h + 2$ (or $h - 2$), and the same process repeats at this node. If the **slow(.)** request reaches the root node, the root will provide needed data to client C using a server stream.

To resume to normal playback, client C sends a **resume**(C) request to the node R^* that is about to transmit a frame h^* to C for SF or SR purposes. R^* then decides to transmit a frame at a time starting from frame h^*, sequentially and forward, to C. As a result, C will return to normal playback by displaying the frames arrived from R^* as usual.

3.7 Discussion

As presented in Sections 3.4, 3.5, and 3.6, there may be more than one node involving in delivering frames needed by a client VCR request of type FF, PB, RW, SF, or SR. When such a node R does not have the needed frames, it will notify the next node R_{next} to find them. A problem questionable is when R should notify. If R notifies only after encountering the situation where the next needed frame lies beyond the cache, it might introduce a delay to the requesting client since it might take some time for R_{next} to start sending the next needed frame. To avoid this, on receipt of a VCR request and if R detects that it is not able to provide all needed frames, R should notify R_{next} at some appropriate time before the missing frame is actually needed for display by the client.

If the video consists of individually-encoded frames of equal size[3], we follow the policy illustrated in Fig. 4. In this figure, f and l are the ID's of the first and last frames currently in the cache of node R, and p the current frame needed by a client VCR request. Suppose the current time is t_0 and $t_0 + i$ is the latest time that frame $p + r \times i$ needs to be transmitted to the client and still available in R's cache. To avoid delay at the client side, node R will signal R_{next} as soon as it starts sending frame p to the client to inform that R_{next} needs to send frame $p + r \times (i + 1)$ to the client at time $t_0 + i + 1$.

4 Performance Study

We study the performance of Range Multicast in providing interactive and non-interactive VOD services in this section. Specifically, we focus on two systems, both operating on an overlay network having a topology borrowed from the IBM's Global Network backbone[4]. The topology is illustrated

[3]If the video consists of dependently-encoded frames of various sizes, such as in MPEG format, the policy will be different, however with a minor change.

[4]http://www.nthelp.com/images/ibm.jpg (reachable as of January 21, 2003)

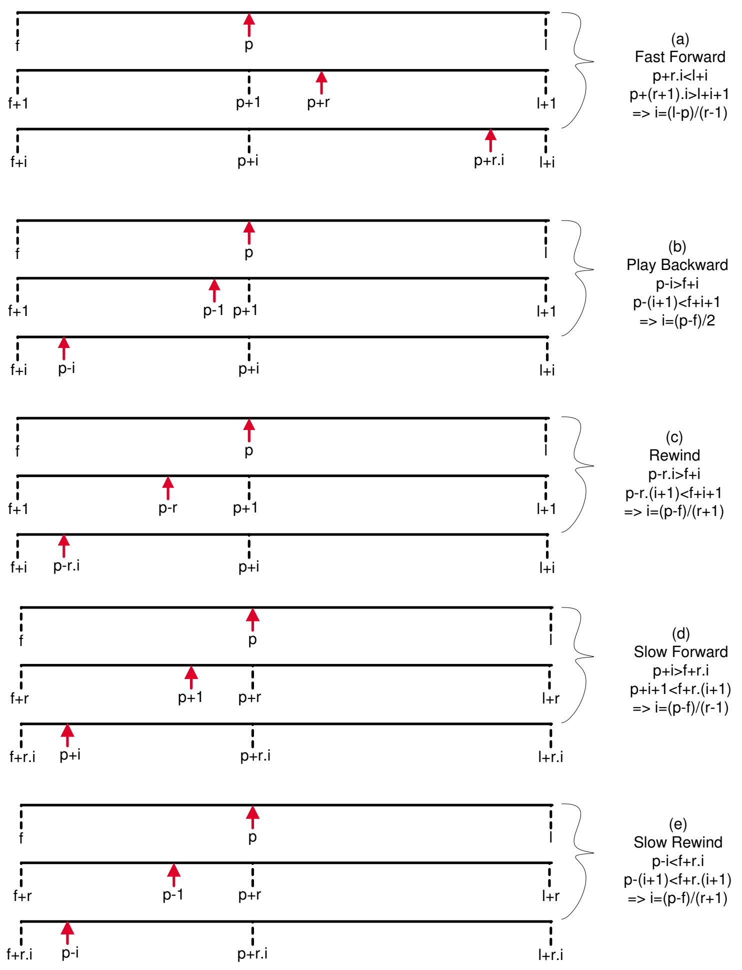

Figure 4: Determination of when to signal the next node

in Fig. 5. In the first system (which we call VCR-RM), each node runs the Range Multicast software and the Chicago node represents the root node. In the other system (which we call PCP - Prefix Caching Proxy) [19, 37, 43], each node represents a proxy server running a prefix caching algorithm. Each proxy reserves a storage space for caching purposes. This space is divided into equal-sized chunks as in a RM node. When delivered from the server to a client, a prefix of the requested video is cached at a free chunk of the corresponding proxy. If all the chunks are filled with data, the chunk replacement is based on LFU (Least Frequently Used) policy. In other words, a chunk currently storing the prefix of a less frequently requested video is more preferable to be overwritten. A client can play the data cached at the proxy instantly and download the remaining data from the server. Co-operation is also enabled; if not possible to get data from the local proxy server, the client can be satisfied by a remote proxy server if the data is available there.

The performance study would have been more informative had we compared RM with the client-controlled approach (CCA) [3, 28, 1, 17], however the intention behind our design is not to replace CCA. In fact, RM could be integrated with CCA to provide significantly better performance

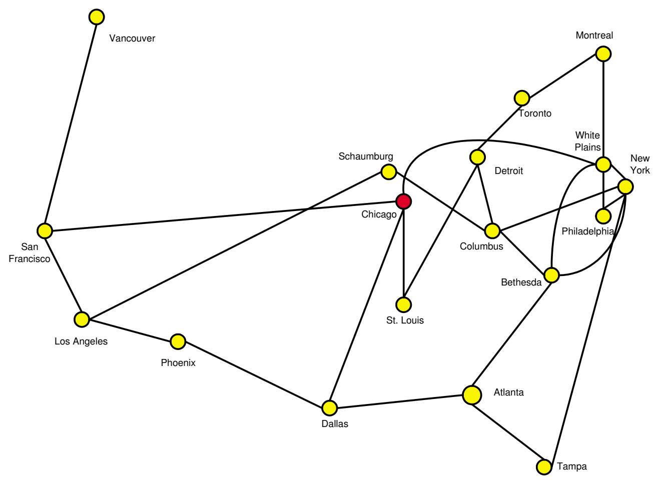

Figure 5: IBM's Global Network Topology.

than CCA alone does. Therefore, we opted to compare RM with PCP. This is due to three reasons: (1) PCP is efficient to stream video with low startup latency and reduced server bandwidth consumption; (2) Neither PCP nor RM requires a private buffer at the client; and (3) Both PCP and RM use a caching buffer at each dedicated server which is a proxy in PCP or an overlay node in RM.

Our simulation setup favored the PCP system because VCR functionality was not embedded in PCP. In spite of that, we aimed to achieve two goals:

1. When VCR functionality is disabled in RM, we aim to show that this system performs significantly better than the system running PCP.
2. When VCR functionality is enabled in RM, we aim to show that this system still outperforms the system running PCP.

Meeting these two goals substantiates that RM is really an efficient approach for developing a VCR-enabled VOD system because RM provides more features but still performs better than PCP, a well-known and efficient scheme for video streaming.

Each system consists of a video server storing $N = 50$ 90-minute videos $\{1, 2, .., N\}$. These videos are requested with frequencies following a Zipf-like distribution with skew factor z. Specifically, the probability that video i is requested is $\frac{1}{i^z \sum_{j=1}^{N} \frac{1}{j^z}}$. A high value of z implies that there are few popular videos. Zipf-like distribution is a special case of the Pareto distribution [30] and has been shown strongly applicable to Web page and video accesses [6, 14]. Our choosing 0.7 as the default value for z results in a distribution which has been used in most VOD systems [14, 3, 1, 32]. It is typically true that most users are interested in only a small number of popular videos.

For simplicity, we assume each video is a sequence of independently encoded frames of equal size, each forming a block for transmission. We assume a discrete time model where a time unit is called a "second". In such a second, the network can transmit an amount equivalent to a second of video data, which is a frame as we define above. Each node has by default five chunks of memory for caching purposes. Such a chunk can store by default 10 minutes of video (i.e., 600 blocks).

A client appeals a service by sending a request to any node (chosen in random) of the overlay. Client requests are generated in our simulation according to a Poisson process with an arrival request rate λ having the default value of 1.0 request per second. A client starts the service in normal mode (i.e., NP mode), and during its playback can be in any of the nine states presented in Section 3 with probabilities p_{NP}, p_{PB}, p_{SF}, p_{SR}, p_{FF}, p_{RW}, p_{JF}, p_{JB}, and p_{PS}, respectively. The duration of staying in any VCR mode follows a uniform distribution. This allows us to study any VCR invoking behavior of clients. For simplicity, we assume that the client quits its session only when the end of the video is reached and that $p_{NP} = 1 - p$, and $p_{PB} = p_{SF} = p_{SR} = p_{FF} = p_{RW}$ $= p_{JF} = p_{JB} = p_{PS} = p/8$ ($p \in [0, 1)$). Furthermore, the skip rate for FF and RW is twice the normal playback, and the repeat rate for SF and SR is two (i.e., each frame in the SF/SR mode is displayed twice).

4.1 Performance Metrics

An interaction is considered "cache-served" if the data currently in the aggregate buffer are sufficient to accommodate the interaction. Otherwise, the server may have to transmit the needed interactive frames to the requesting client and the interaction is called "server-served". For example, a jump to a destination point outside the aggregate buffer is server-served; a long lasting fast-forward pushing the play point off the aggregate buffer is also server-served. To measure the cache-served cases, we use **percentage of cache-served interactions** (PCSI) as the first performance metrics. PCSI is computed as the ratio of the number of VCR requests served by cache to the total number of VCR requests. For the "server-served" cases, we would like to estimate, for a VCR interaction, how much requested interactive data is provided by the aggregate cache. Therefore, we use **average server-load saving** (ASLS) as the second performance metrics. As an example, if a client wishes to rewind for 20 seconds, but forces the play point off the aggregate cache after only 15 seconds, the server load saved for this interaction is 15/20 or 75%. A caching scheme to support VCR interactions should have high values of PCSI and ASLS in order to be efficient.

When a client returns to normal playback from a VCR interaction, the client tries to join an existing multicast. If the joining fails, server bandwidth must be allocated. Additional server must also be needed when a VCR interaction needs data beyond the multicast range. If a technique results in frequent server intervention, the server bandwidth will be highly demanded and hence worsen the chance of servicing subsequent requests. To evaluate the scalability of the proposed system, we would like to estimate the **minimum server bandwidth requirement** (MinBW) needed to satisfy every request. We also take into consideration the **system throughput** (ST) for normal requests (i.e., requests to start up service) in the case where the server bandwidth is limited.

Since Range Multicast resolves the server bottleneck by capitalizing caches at intermediate overlay nodes, the bandwidth at the server is less demanded but the traffic inside the network might increase. This problem exists not only in Range Multicast but also in every overlay multicast work and it would be interesting to study the tradeoff between "link stress" (i.e. how severe links are overused) and server bandwidth consumption. We believe that using an efficient technique for building the overlay topology can reduce link stress. Range Multicast can employ such a technique, and, by having an efficient caching and delivery strategy atop the overlay topology, significantly save the server bandwidth. Since link stress was already studied in many overlay works such as [25, 11, 24, 29], we opted to report the results on server bandwidth only. It is our conviction that despite recent advances in network bandwidth technologies, that can handle increases in network traffic incurred by our scheme, server bottleneck remains a severe problem in most large-scale VOD systems nowadays.

4.2 Numerical Results

We run simulations, each lasting 24 hours, based on four values for VCR intensity: $p = 0.0$ (pure Range Multicast without VCR functionality), $p = 0.2$ (rarely frequent VCR actions), $p = 0.5$ (fairly frequent VCR actions), and $p = 0.8$ (highly frequent VCR actions). We report the simulation results in the following subsections.

4.2.1 Minimum Server Bandwidth Requirement (MinBW)

We would like to estimate MinBW in the number of streams the system creates to satisfy every request. By estimating the values for average server-load saving (ASLS) and percentage of cache-served interactions (PCSI), we would also want to estimate how VCR interactivity can take advantage of the range multicast protocol. Specifically, we ran simulations under the changes of request rate, caching space per node, and skew factor of video access pattern. Under all scenarios, as shown in Fig. 6, increasing the VCR intensity (or p value) needs the server to spend more bandwidth in order to serve all the requests, however the server bandwidth required by the RM-based systems are always less than that required by PCP. In this study, we do not only show that our proposed multicast scheme itself (VCR-RM/p=0 system) works better than the traditional proxy-based scheme, but also when integrated with VCR functionality the system is still more scalable than PCP. The gap in server bandwidth requirement is even exponentially larger as the request rate increases, as exhibited in Fig. 6(a). Moreover, according to Fig. 6(b) and Fig. 6(c), VCR-RM benefits more from enlarging caching space per node than PCP does, even though it is intuitive that more cache in the intermediate nodes will reduce the server bandwidth. An exception is when

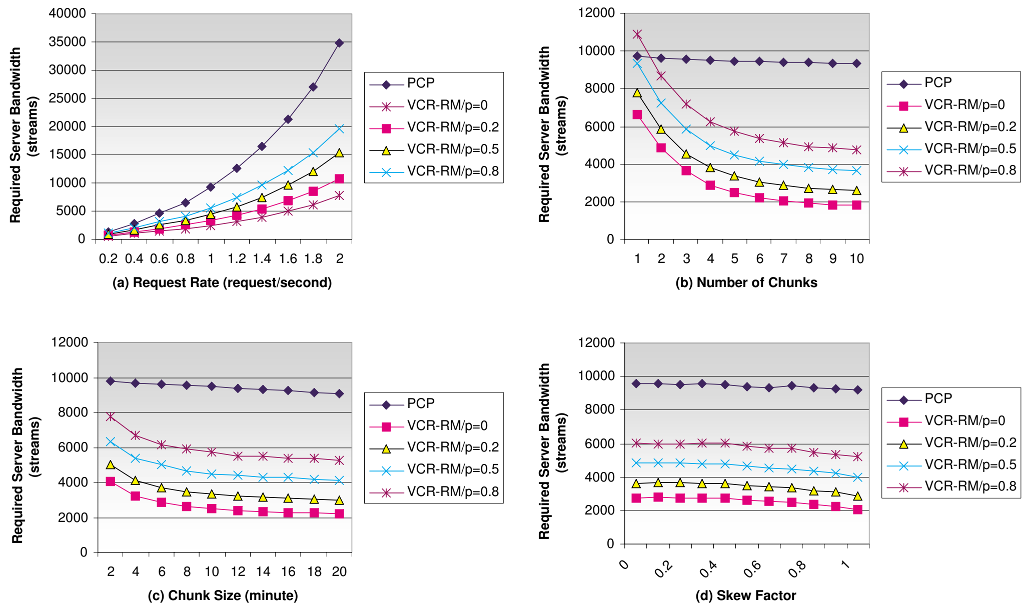

Figure 6: Required Server Bandwidth.

the number of chunks equals one, the VCR-RM/p=0.8 system requires more bandwidth than PCP. It is because the cache window for VCR interactions is too small to support too intensive VCR activity (when the client invokes VCR 80% of the play time); as a result, a lot of server bandwidth has to be allocated to satisfy those interactions not fully-servable by the aggregate cache. Another realization worth consideration is that, given the same total caching space and small chunk size, by increasing the chunk size the system gains better performance than by increasing the number of chunks. This is explained as follows. More requests will be benefited from increasing chunk size, therefore reducing server intervention. Having more caching chunks helps cache more streams into a node, however if the chunk size is small, those cached data will be obsolete for subsequent requests that come a bit late.

Even though changes in VCR intensity do affect the amount of server bandwidth needed, they do not affect much on the values of ASLS and PCSI (Fig. 7(a-h)). Having more VCR requests does not change much the percentage of those requests served by the aggregate cache. It only worsens the server bottleneck since more server-served requests lead to more server bandwidth being allocated. Fig. 7(a) and Fig. 7(b) experience improvements of ASLS and PCSI under the increase of request rate until some point. Afterwards, ASLS and PCSI are getting smaller. This is because when more requests arrive, but not too frequently, there will be more caches at the overlay nodes and thus there will be a higher chance to serve new requests. However, if requests arrive too frequently, the caches will be refreshed very often and therefore usually contain a short prefix of video. Consequently, the cumulative buffer window for doing VCR actions is very small and reduces the chance to support VCR activity. Nonetheless, under most simulation runs, the

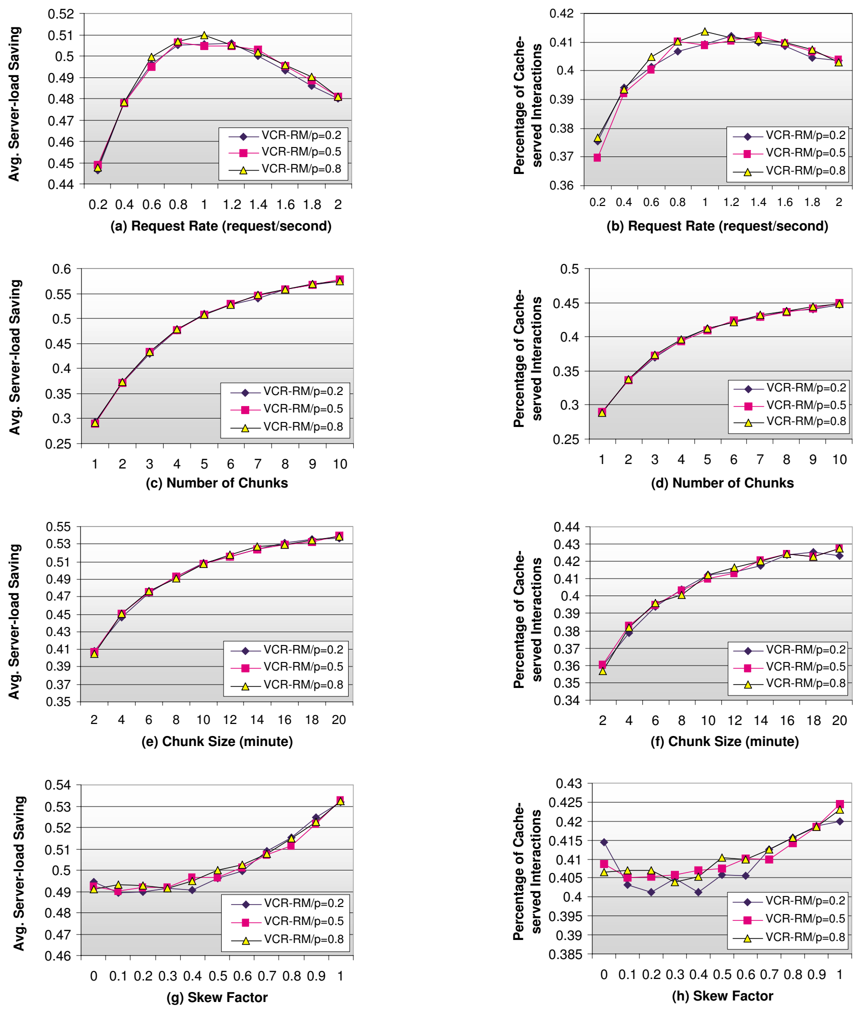

Figure 7: ASLS and PCSI: Every request is served

cache-served requests account for 30-40% among all VCR requests and server-served requests can capitalize about 30-50% of the caches.

4.2.2 Effect of Various Server Bandwidths

We carried out a study on how server bandwidth affects the performance of the VCR-RM and PCP systems. We varied this bandwidth from 400 concurrent streams to 2000 concurrent streams and plotted the results for system throughput in Fig. 8(a). We wanted to investigate how the system after embedding VCR functionality would affect the serving of requests for normal playback. The results came out promising. By using the proposed video multicast scheme, the RM-based systems outperform PCP by a factor of at least four times (Fig. 8(a)) even in the case of intense VCR interactivity (p=0.8). This demonstrates an important point that our proposed VCR supporting scheme is a lot more scalable than PCP. If we allowed PCP to embed VCR functionality, its

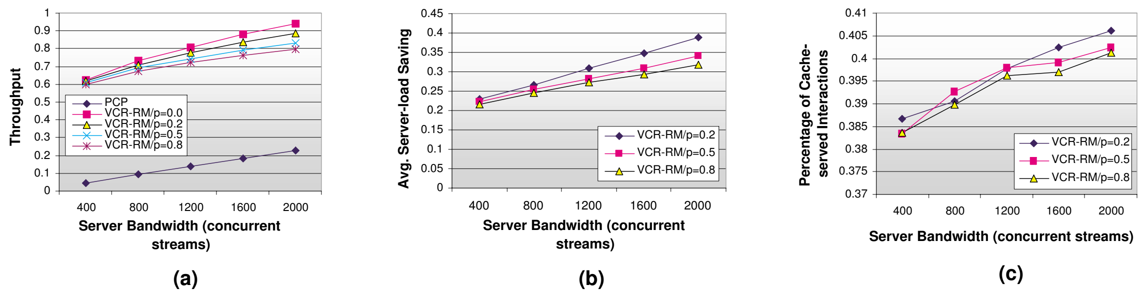

Figure 8: ST, ASLS, and PCSI: Server bandwidth is limited

performance would be even worse and consequently could not be comparable to VCR-RM.

Fig. 8(b, c) illustrates another important point. VCR-RM is quite stable under various intensity levels of VCR interactivity. This feature is not provided by existing client-controlled or server-controlled VCR supporting techniques. Indeed, the gap (ST, ASLS, or PCSI) between the performances of VCR-RM system under the cases $p = 0.2$, 0.5, and 0.8 is very narrow. There is a reason for this realization. In VCR-RM, the cache window for VCR interactivity is cumulatively formed by node caches and shared by every client in the network. Changing the client VCR invocation behavior does not change this content since when a client invokes a VCR action, it will get data cached somewhere if available, otherwise will go to the server, and this process does not force any change on the node caches. In the client-controlled approach, intense VCR interactivity would result in frequent times the play point goes beyond the client VCR buffer. On the other hand, in the server-controlled approach, intense VCR interactivity would lead to severe shortage of server bandwidth. Most of these problems are avoided in VCR-RM.

5 Conclusions

In this paper, we presented a new communication paradigm for video-on-demand applications, called Range Multicast. This scheme is a shift from conventional thinking about multicast where every receiver must obtain the same data packet at all time. In contrast, the data available from a range multicast at any time is a contiguous segment of the video. This characteristic is very important to video-on-demand applications as follows:

- *Better service latency*: Since clients can join a multicast at their specified time instead of the multicast time, they do not need to wait for the availability of the next server stream. This fact results in better service latency.

- *Less demanding on server bandwidth*: Range Multicast is less demanding on server bandwidth because many new service requests can be satisfied by having the clients to join an on-going multicast instead of demanding a new video stream from the server. Furthermore, since clients typically have a good chance to join an existing multicast after performing a VCR operation, a new video stream is not necessary to support the resume operations. This feature is particularly critical to applications with a high degree of video browsing activity.

- *Improved VCR-like interaction*: Since shared buffer space can be used more efficiently and cost-effectively than private buffers as in client-controlled techniques, each client effectively sees a larger caching space and therefore can perform longer-duration VCR actions.
- *More versatile*: Since Range Multicast does not require the clients to maintain a local buffer for VCR interactivity, this scheme is suitable for clients with limited resources such as PDA's (personal digital assistants).
- *Feasible implementation on the Internet*: Many works on VOD assume the existence of IP Multicast [15, 34]. However, IP Multicast currently is not widely available. Range Multicast is an overlay approach which implements the multicast paradigm based on IP Unicast only, and therefore easily deployable on the current Internet.

We carried out a performance study to investigate the potential of the Range Multicast approach. In this study, Range Multicast can provide a throughput 400% higher than the conventional prefix caching proxy scheme. Under most scenarios studies, about 30-40% of the VCR requests are fully served by the aggregate cache and among those not fully served by the cache, the server load is saved by 30-50%. Range Multicast is also stable under unpredictable client interactivity behaviors. For better performance, it can be integrated with a VCR client-controlled technique taking advantage of the local buffer at the client and the properties provided by Range Multicast as listed above.

Acknowledgement

We would like to thank the US National Science Foundation for sponsoring this research under grant ANI-0088026. Especially, we are grateful to the anonymous reviewers whose constructive and interesting comments helped us improve the quality of this work.

References

[1] E. L. Abram-Profeta and K.G.Shin. Providing unrestricted vcr functions in multicast video-on-demand servers. In *Proc. IEEE Conference on Multimedia Computing and Systems*, Austin, TX, 1998.

[2] S. Acharya and B. Smith. Middleman: A video caching proxy server. In *Proc. IEEE NOSSDAV*, 2000.

[3] K. C. Almeroth and M. H. Ammar. On the use of multicast delivery to provide a scalable and interactive video-on-demand service. *IEEE Journal of Selected Areas in Communications*, 14:1110–1122, 1996.

[4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *ACM SIGCOMM*, Pittsburgh, PA, 2002.

[5] M. K. Bradshaw, B. Wang, S. Sen, L. Gao, J. Kurose, P. Shenoy, and D. Towsley. Periodic broadcast and patching services - implementation, measurement and analysis in an internet streaming video testbed. In *Proc. of ACM Conference on Multimedia*, Canada, September 2001.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOMM*, pages 126–134, 1999.

[7] S. W. Carter and D. D. E. Long. Improving bandwidth efficiency of video-on-demand servers. *Computer Networks and ISDN Systems*, 31(1):99–111, March 1999.

[8] M.-S. Chen and D. D. Kandlur. Downloading and stream conversion: Supporting interactive playout of videos in a client station. In *Proc. IEEE Conference on Multimedia Computing and Systems*, pages 73–80, Washington, DC, May 1995.

[9] M.-S. Chen, D. D. Kandlur, and P. S. Yu. Support for fully interactive playout in a disk-array-based video server. In *Proc. ACM Conference on Multimedia*, pages 391–398, Sanfrancisco, CA, OCtober 1994.

[10] W. chi Feng, F. Jahanian, and S. Sechrest. Providing vcr functionality in a constant quality video-on-demand transportation service. In *Proc. IEEE Conference on Multimedia and Computing Systems*, pages 127–135, Hiroshima, Japan, June 1996.

[11] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS*, pages 1–12, 2000.

[12] A. Dan, Y. Heights, and D. Sitaram. Generalized interval caching policy for mixed interactive and long video workloads. In *Proc. of SPIE/ACM Conf. on Multimedia Computing and Networking*, pages 344–351, San Jose, California, January 1996.

[13] A. Dan, P. Shahabuddin, D. Sitaram, and D. Towsley. Channel allocation under batching and vcr control in video-on-demand systems. *Journal of Parrallel and Distributed Computing*, 30:168–179, 1995.

[14] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic batching policies for an on-demand video server. *ACM Multimedia Systems Journal*, 4(3):112–121, June 1996.

[15] S. Deering. Host extension for ip multicasting. *RFC-1112*, August 1989.

[16] J. K. Dey-Sircar, J. D. Salehi, J. F. Kurose, and D. Towsley. Providing vcr capabilities in large-scale video servers. In *Proc. ACM Conference on Multimedia*, pages 25–52, Sanfrancisco, CA, October 1994.

[17] Z. Fei, I. Kamel, S. Mukherjee, and M. H. Ammar. Providing interactive functions for staggered multicast near video-on-demand systems. In *Proc. IEEE Conference on Multimedia and Computing Systems (ICMCS99)*, 1999.

[18] P. Francis. Yallcast: Extending the internet multicast architecture. In *http://www.yallcast.com.*, September 1999.

[19] S. Gruber, J. Rexford, and A. Basso. Protocol considerations for a prefix-caching proxy for multimedia streams. In *Proc. of the 9th International WWW Conference*, 2000.

[20] K. A. Hua, Y. Cai, and S. Sheu. Patching: A multicast technique for true video-on-demand services. In *Proc. of ACM MULTIMEDIA*, pages 191–200, Bristol, U.K., September 1998.

[21] K. A. Hua and S. Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *Proc. of the ACM SIGCOMM'97*, Cannes, France, Sepetember 1997.

[22] K. A. Hua, D. A. Tran, and R. Villafane. Caching multicast protocol for on-demand video delivery. In *Proc. of the ACM/SPIE Conference on Multimedia Computing and Networking*, pages 2–13, San Jose, USA, January 2000.

[23] K. A. Hua, D. A. Tran, and R. Villafane. Overlay multicast for video on demand on the internet. In *ACM Symposium on Applied Computing*, Melbourne, FL, USA, 2003.

[24] S. Jain, R. Mahajan, D. Wetherall, and G. Borriello. Scalable self-organizing overlays. Technical report, University of Washington, 2000.

[25] J. Jannotti, D. K. Gifford, and K. L. Johnson. Overcast: Reliable multicasting with an overlay network. In *USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.

[26] L. Juhn and L. Tseng. Harmonic broadcasting for video-on-demand service. *IEEE Transactions on Broadcasting*, 43(3):268–271, 1997.

[27] J. Jung, D. Lee, and K. Chon. Proactive web caching with cumulative prefetching for large multimedia data. In *Proc. of the 9th International WWW Conference*, 1999.

[28] W. Liao and V. O. Li. The split and merge protocol for interactive video on demand. *IEEE Multimedia*, 4:51–62, October-December 1997.

[29] J. Liebeherr and M. Nahas. Application-layer multicasting with delaunay triangulations. In *Global Internet Symposium, IEEE Globecom*, 2001.

[30] V. Pareto. Cours d'economie politique. Rouge and Cie, Lausane and Paris, 1897.

[31] J. F. Paris. An interactive broadcasting protocol for video on demand. In *Proc. 20th IEEE Performance Computing and Communications Conference*, pages 347–353, Phoenix, AZ, USA, April 2001.

[32] J. F. Paris, S. W. Carter, and D. D. E. Long. Efficient broadcasting protocols for video on demand. In *Proc. of ACM/SPIE's Conf. on Multimedia Computing and Networking (MMCN'99)*, pages 317–326, San Jose, CA, USA, January 1999.

[33] D. Pendakaris and S. Shi. ALMI: An application level multicast infrastructure. In *USENIX Symposium on Internet Technologies and Systems*, Sanfrancisco, CA, March 26-28 2001.

[34] B. Quinn and K. Almeroth. Ip multicast applications: Challenges and solutions. In *Internet Engineering Task Force (IETF) Internet Draft*, March 2001.

[35] S. Ramesh, I. Rhee, and K. Guo. Multicast with cache (mcache): An adaptive zero-delay video-on-demand service. In *Proc. of IEEE INFOCOM*, San Diego, USA, 2001.

[36] S. Sen, L. Gao, J. Rexford, and D. Towsley. Optimal patching schemes for efficient multimedia streaming. In *Proc. of IEEE NOSSDAV*, NJ, USA, June 1999.

[37] S. Sen, D. Towsley, Z.-L. Zhang, and J. K. Dey. Optimal multicast smoothing of streaming video over an internetwork. In *Proc. of IEEE INFOCOM '99*, 1999.

[38] P. J. Shenoy and H. M. Vin. Efficient support for interactive operations in multi-resolution video servers. *ACM Journal of Multimedia Systems*, 7:241–253, 1999.

[39] M. A. Tantaoui, K. A. Hua, and S. Sheu. Interaction with broadcast video. In *ACM Conference on Multimedia*, Juan Les Pins, France, December 2002.

[40] D. A. Tran, K. A. Hua, and M. A. Tantaoui. A multi-multicast sharing technique for large-scale video information systems. In *IEEE Int'l Conference on Communications*, New York, NY, April-May 2002.

[41] K.-L. Wu, P. S. Yu, and J. L. Wolf. Segment-based proxy caching of multimedia streams. In *Proc. of the 10th International WWW Conference*, Hong Kong, 2001.

[42] P. S. Yu, J. L. Wolf, and H. Shachnai. Design and analysis of a look-ahead scheduling scheme to support pause-resume for video-on-demand applications. *ACM Journal of Multimedia Systems*, 3(4):137–150, 1995.

[43] Z.-L. Zhang, Y. Wang, D. H. C. Du, and D. Su. Video staging: A proxy-server-based approach to end-to-end video delivery over wide-area networks. *IEEE/ACM Transactions on Networking*, 8(4), August 2000.