CoopStream: A Cooperative Cache Based Streaming Schedule Scheme for On-demand Media Services on Overlay Networks*

Baoliu Ye[+,†], Minyi Guo[†] and Jingling Xue[‡]
[+](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)
[†](Department of Computer Software, the University of Aizu, Fukushima 965-8580, Japan)
[‡](School of Computer Science and Engineering, University of New South Wales, Sydney, Australia)
yebl@dislab.nju.edu.cn, minyi@u-aizu.ac.jp, jxue@cse.unsw.edu.au

Abstract

Recently, we have witnessed a tremendous growth of interests in streaming continuous media such as video data over the Internet. However, how to provide true on-demand streaming services with VCR functionality is still a challenging task, especially when their scalability is required. In this paper, we propose CoopStream, a novel streaming scheme, to address this challenge in the context of overlay networks. At the client side, we propose to use a dual-channel cache management mechanism to dynamically adjust the cache contents based on its current playing state, so as to support VCR operations locally. On the server side, we exploit the temporal relationship among asynchronous streaming requests to schedule streams. This enables us to reduce significantly the server load by redirecting streaming requests to the clients that can serve those requests. Simulation results show that, CoopStream is capable of providing continuous streaming services that are scalable in terms of the server bandwidth consumed.

1. Introduction

Recently, Internet-oriented streaming applications have become increasingly popular. However, how to provide true on-demand streaming services with VCR functionality is still a challenging task. The true video on-demand services are distinct from the traditional real-time media streaming services in the following aspects: 1) *asynchrony* - a user may issue streaming requests at different time; 2) *nonsequentiality* - a streaming request may start from any part (or offset) of a stream; 3) *VCR support* - a user may make different kinds of VCR operations during a session. Furthermore, the behaviors of different users are unpredictable.

To address the above challenges, earlier solutions adopt IP multicast as the basic streaming sharing mechanism to serve multiple requests synchronously [3, 6, 9]. However, due to the limited deployment of IP multicast as well as other technical reasons, these solutions were not widely adopted over the Internet. Recently, *Application Layer Multicast* (ALM) has increasingly attracted enormous attention in the research community. ALM makes no special assumptions about the underlying network infrastructure and shifts the multicast functionality from the network layer to the application layer. The topology of ALM is an overlay network constructed by participating nodes. Since each node can contribute its available resources to the system, theoretically the service capacity increases as the number of nodes increases.

In this paper, we propose CoopStream, a novel streaming scheme for providing continuous streaming services with VCR functionality in the context of ALM. Our basic observation is that streaming services usually have long durations and the user requests for such services are asynchronous in nature. We can potentially provide scalable services in terms of the server bandwidth consumed by using the caching capability of end hosts and capturing the temporal correlations of asynchronous requests. We explored this possibility and addressed several associated problems by making the following contributions:

- We present a dual-channel cache management mechanism at the client side to dynamically adjust cache contents and support VCR actions locally. For a particular client, our mechanism ensures that the amount of played media objects and that of unplayed (i.e., prefetched) are balanced in local cache by using two techniques. First, once both parts become imbalanced, the cache management will temporally suspend the prefetching stream or open a second channel to compensate for the imbalance. Second, the start offset of

*This work was partially supported by the National Science Foundation of China under grant 60573106 and the National Basic Research Program of China (973) under grant 2002CB312002.

a prefetching stream is specified as an interval rather than an exact value. Thus the likelihood to find the requested objects from some existing end hosts is increased.

- We exploit for the first time the temporal correlations among asynchronous streams to allow the server to redirect streaming requests to the end hosts where the requested objects are cached. Each stream request is identified by an initial offset and a request time. These two parameters are used to define the temporal relationship among stream requests. When serving a particular request, the server first checks if the requested stream can be served by another end host. Such a streaming schedule algorithm shifts the streaming service responsibility from the server to the client side, effectively reducing the server bandwidth consumption.

To the best of our knowledge, we are the first to explore the use of ALM to provide true on-demand streaming services with VCR functionality. Our experimental results show that CoopStream is capable of providing scalable streaming services in term of the server bandwidth consumed. The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 gives an overview of CoopStream. Section 4 describes the details of our cache management mechanism. Section 5 presents our streaming schedule algorithm. Section 6 analyzes the performance results. Finally, we conclude our work in Section 7.

2. Related Work

There has been a lot of work on providing continuous on-demand streaming services using IP multicast. To account for the synchronous nature of IP multicast and avoid missing a certain portion of a video requested, batching [3] tries to aggregate asynchronous requests into one multicast session at the cost of incurring a start-up delay while patching [6] allows nodes to join an ongoing multicast session by means of patching the missed portion via a unicast channel to the server. With merging [9], a node continuously attempts to catch up with the nearest predecessor streaming until it eventually merges into a largest multicast session. There has been relatively little work on supporting VCR operations using IP multicast. ABM [4] is one such a client-side buffer management scheme. In this scheme, stream segments are staggered and periodically broadcasted in different channels. Their cache management needs to receive data from these channels simultaneously. A client is required to maintain three concurrent connections in a single streaming session. In contrast, each client in CoopStream uses at most two concurrent channels, one of which is only temporarily opened to prefetch future frames.

In the case of overlay networks, although some solutions

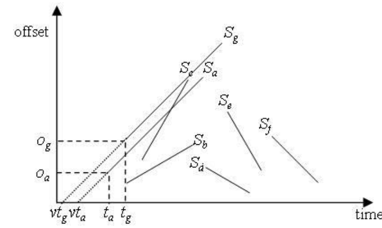

Figure 1. An example of different VCR streams.

using IP multicast have been extended to work over overlay networks [5, 8], we are not aware of any previous work on providing VCR functionality. Some other efforts using peer-to-peer network techniques can be found in [2, 8]. However, all these solutions ignore the VCR functionality, which is one of the most useful features in real-world streaming applications.

3. Overview of CoopStream

3.1. VCR Operations

Generally, a VCR operation could be described by a pair $(\Delta t, \Delta l)$, where Δt is the duration of the operation, and Δl is the moving length. Assuming that the playback rate of a streaming is a Constant-Bit-Rate (CBR) r, we give the formal definition of VCR actions as follows:

1. *Jump Forward/Backward* (*JF/JB*): $\Delta t = 0, \Delta l \neq 0$.
2. *Pause*: $\Delta t > 0, \Delta l = 0$.
3. *Fast Forward/Backward* (*FF/FB*): $|\Delta l/\Delta t| > r$, $\Delta t > 0$.
4. *Slow Forward/Backward* (*SF/SB*): $|\Delta l/\Delta t| < r$, $\Delta t > 0$.
5. *Normal Play/Play Backward* (*NP/PB*): $|\Delta l/\Delta t| = r$, $\Delta t > 0$.

Let β denote the ratio between the playback rate of a VCR operation and the normal play. Obviously, $\beta = \frac{\Delta l/\Delta t}{r}$. If $|\beta| > 1$, it is a *fast play*; If $|\beta| < 1$ and $\beta \neq 0$, it is a *slow play*. $\beta = 0$ means a *pause operation*. $\beta > 0$ means a *forward action* while $\beta < 0$ indicates *backward action*.

Fig.1 shows an example with different VCR streams. In this figure, S_a and S_g are two *NP* streams, S_b an *SF* stream, S_c an *FF* stream, S_d an SB stream, S_e an *FB* stream, and S_f an *PB* stream.

3.2. An Overview

CoopStream is an overlay networks-based on-demand streaming service scheme. The streaming server acts as the seed of stream files and accepts streaming requests. Each

node devotes a fix-sized storage space capable of buffering W units of normal playback time to cache the most recent received stream objects. The server maintains an information table (see Table 1) to monitor the streaming status of each node. CoopStream mainly consists of two components, the cache management mechanism and the streaming schedule algorithm. The former is responsible for adjusting cache contents while the latter takes care of scheduling appropriate streaming for requests.

Table 1. Status information table.

Field	Description
IP	The IP address of a client
name	The name of requested stream file
statime	The stat time of a streaming request
os	The original offset of a stream request
op	the initial prefetch offset of a request
type	The VCR type of a stream

We outline the routine of CoopStream as follows. Under *NP* status, each node simultaneously keeps a certain length of played frames and unplayed frames in its cache. The cache buffers stream objects in a round-robin manner. When a VCR action is issued, the node first tries to serve the request with local cache. If this fails, it sends a VCR request to the server. Once returning to the normal play status, the cache management adjusts its content, with the intent to support later VCR operations. Upon receiving a streaming request, the server first searches for the nodes containing the requested object. If no node is found, the server opens a new streaming channel. Otherwise, it sends the search result to the requestor. The requestor in turn selects a node to act as its "streaming server". When a node wants to leave the system, it must first informs its children to seek for a new parent. In CoopStream, an abrupt failure could be detected by the suddenly increased packet loss ratio. During the repairing period, its children can continue their services by consuming the pre-cached objects.

In nature, all participating nodes in CoopStream are organized into one ore more groups. Besides receiving media objects from the parent, each non-leaf node also forwards the cached objects to its children. In this sense, Coopstream is a variation of ALM. However, unlike previous ALM schemes, the streams delivered over different links within a tree are asynchronous, i.e., a non-leaf node simultaneously forwards different media objects to different children. Fig. 2 illustrates a typical scenario of CoopStream.

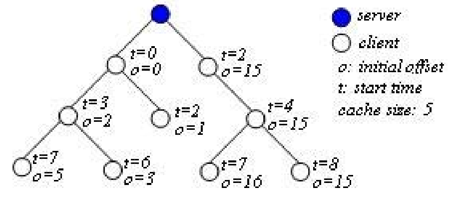

Figure 2. An typical scenario of CoopStream.

4. Cache Management

4.1. Cache Structure

Each node in CoopStream contains at least a *play pointer* (*ply ptr*) pointing to the current playback position and a *prefetch pointer* (*pref ptr*) pointing to the latest caching position. We call the media objects that have already been displayed *past frames* and those that have been prefetched into the cache but not yet played *future frames*. In addition, a non-leaf node may have one or more *forward pointers* (*fwd ptr*) which are responsible for forwarding stream objects to its children (see Fig. 3(b)). For simplicity, we first define some variables related to the cache of client x at time t as follows.

Let $pr_x(t)$ denote the position of prefetch pointer, $pl_x(t)$ the position of play pointer, $lpf_x(t)$ the length of past frames, and $lff_x(t)$ the length of future frames. $opr_x(t)$ and $opl_x(t)$ correspond to the stream offsets of $pr_x(t)$ and $pl_x(t)$, respectively. Let $\Delta_x(t)$ represent the relative distance between $pr_x(t)$ and $pl_x(t)$, $odff_x(t)$ the stream distance between $opr_x(t)$ and the nearest forward pointer falling into the future frame area, $odpf_x(t)$ the streaming distance between $opr_x(t)$ and the nearest forward pointer within the past frame area. Obviously, $\Delta_x(t) = |pl_x(t) - pr_x(t)|$. If there is no forward pointer within corresponding area, then $odpd_x(t) = \infty$ and/or $odff_x(t) = \infty$. If the cache is saturated and $pr_x(t) < pl_x(t)$, then

$$\begin{cases} lpf_x(t) = \Delta_x(t) \\ lff_x(t) = W - \Delta_x(t) \end{cases} \tag{1}$$

else,

$$\begin{cases} lff_x(t) = \Delta_x(t) \\ lpf_x(t) = W - \Delta_x(t) \end{cases} \tag{2}$$

If no VCR action is issued, both $lpf_x(t)$ and $lff_x(t)$ will remain unchanged throughout the entire streaming session. However, once a VCR action is performed, these two variables as well as $\Delta_x(t)$ will be changed. In practice, it is difficult to keep the play pointer right in the middle of the cache contents. CoopStream defines a relaxed interval to constrain the offset of play pointer.

$$|\Delta_x(t) - W/2| \leq \alpha W, 0 < \alpha < 1 \tag{3}$$

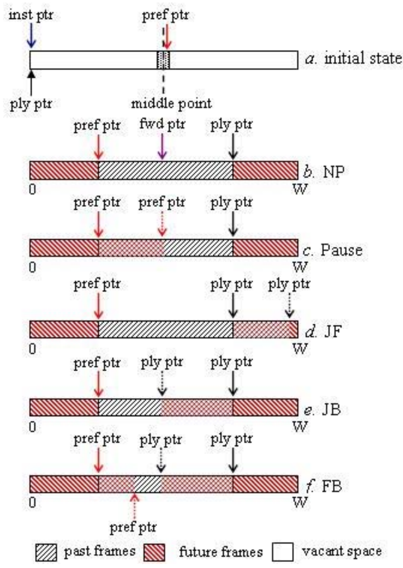

Figure 3. Cache transformation over VCR actions.

Obviously, this interval is more rigidly restricted with a smaller α.

4.2. Implementation of VCR Functionality

The VCR implementation relies fundamentally on the simultaneous availability of the past and future segments within the cache. We present a dual-channel based cache management mechanism to selectively prefetch segments according to the observation of current cache contents. Fig. 3 demonstrates the cache status transformation over different VCR operations. Note that the dotted pointer in this figure depicts the final position after a VCR action and the translucence shadow interval represents the moving length during a VCR operation. Throughout this subsection, we assume x is the client issuing a VCR action at time t_0, p is the parent node of x before the VCR action, and t_1 is the finish time of the VCR operation.

Normal Play. Upon receiving an *NP* request, the server schedules two streams, named as *instant stream* and *prefetch stream*, respectively. The start offset of the instant stream equals to o_x. The start offset of the prefetch stream $opr_x(t_0)$ is decided by the streaming schedule algorithm according to (3) (see Sect. 5). Initially, the *instant pointer* (*inst ptr*) points to the beginning of the cache, while the prefetch pointer is at the position $opr_x(t_0) - o_x$. To avoid service delay, the node starts to play as soon as it receives the first frame of the instant stream. Meanwhile, the node registers its stream information to the status information table. Once having reached to position $opr_x(t_0) - o_x$, the instant streaming closes, but the prefetch stream continues (see Fig. 3(b)). Under *NP* status, the play pointer moves forward at the same speed as the prefetch pointer. Hence, $\Delta_x(t)$ remains unchanged.

Pause Operation. When a pause action is issued, the play pointer stops. For a leaf node, it only notifies its parent node to suspend the stream forwarding. However, for a non-leaf node, it has to keep on receiving media objects from its parent until the prefetch pointer meets the play pointer. In this case, a child must request for a new parent when its forward pointer reaches the prefetch pointer. When returning to the *NP* status, the cache management checks the current cache status. If (3) still holds, the streaming connection will be resumed provided that its parent contains the required object. At the same time, it updates the corresponding record in the status information table with a tri-tuple $(tx - W/2, opr_x(t_1) - W/2, opr_x(t_1))$. Otherwise, the cache management will adjust the cache contents by delaying the prefetch stream for a moment. Let χ denote the delayed time. If $\delta_x(t_1) > 0$, then this connection must be kept available after the delay. Thus, χ is computed as follows:

$$\chi = min(odpf_x(t_1), (\alpha + 1/2)W - \Delta_x(t_1), \delta(t_1))) \quad (4)$$

If $\delta_x(t_1) < 0$, node x has to find a new parent. The server searches for the nodes that may provide a stream with the offset $opr_x(t_1)$ within $odpf_x(t_1)$. The actual start time is optimized by the node with the objective of minimizing $|\Delta_x(t) - W/2|$. The node updates its stream information on the server with the tri-tuple $(t_1 + \chi - W/2, opr_x(t_1) - W/2, opr_x(t_1))$.

Jump Forward. When a jump forward action is issued, the cache management first seeks for the new playback offset within the local cache. If this fails, it notifies its children to switch to a new parent. Following that, this node unregisters its previous stream information and requests for a new *NP* stream starting with the final offset of the jump forward action; Otherwise, the play pointer directly moves to the destination and continues the playback. If (3) becomes false after this operation, then the ratio of future frames decreases (Fig. 3(d)). The cache management creates a new prefetch channel i to selectively download some future frames. The start offset of the prefetch stream $opr_i(t_1)$ should fall into the interval:$[opr_x(t_1), min(opr_x(t_1) + odpf_x(t_1), opr_x(t_1) + (\alpha + 1/2)W]$

The initial $ppr_i(t_1)$ of the new prefetch pointer equals to $ppr_x(t_1) + opr_i(t_1) - opr_x(t_1)$. The old prefetch stream channel closes when it reaches $ppr_i(t_1)$. At that time, the node renews its stream information on the server with the tri-tuple $(t_1 - W/2, opr_x(t_1) - W/2, opr_x(t_1))$.

Jump Backward. In response to this event, the cache management first locates the new playback offset from local cache. If this fails, it notifies its children to send stream re-

quests to the server with their current prefetch offset. Then the node rejoins the system with the final offset requested by the jump backward action. If it succeeds, the play pointer turns back to the destination directly. A jump backward operation may violate (3) by increasing the relative ratio of future frames (Fig. 3(e)). If so, the cache management must adjust the cache ratio by temporarily suspending the prefetch channel. The suspending time is decided according to (4).

Fast Forward. When a fast forward action is issued, the play pointer moves forward at the rate of $\beta r (\beta > 1)$. Once the play pointer catches up with the prefetch pointer, the node requests a fast forwarding stream from the server. Meanwhile, the cache management notifies all its children to find a new parent before the corresponding forward pointers are overwritten by the play pointer. In this case, the node will rejoin the system as a new node when returning to the *NP* status. If the node resumes the *NP* status before its cache contents are consumed up, all the forward pointers keep unchanged. The cache management employs the same routine as a jump forward action to check and adjust the cache contents.

Slow Forward. The ratio of future frames gradually increases with a slow forward action. Once the prefetch pointer catches up with the play pointer, the prefetch stream will be suspended. During this period, its children have to find a new parent when the corresponding forward pointers reach the prefetch position. Later, if the play pointer meets the prefetch pointer again after a round, the node will request for a new stream with the offset $opr_x(t)$ from the server. There are two different cases that may cause (3) to become invalid after returning to the normal play status. If future frames are dominant in the cache, the cache management performs a similar cache adjustment routine as a jump backward action to regulate the position of the play pointer. Otherwise, it runs the same routine as a jump forward to balance the cache contents.

Play Backward/Fast Backward. The past frames are consumed at the rate of $(|\beta| + 1)r$ with a backward action. In response to this action, This node suspends the prefetch stream temporarily. Its children will request for a new parent when their forward pointers reach the prefetch pointer. Once the play pointer moves to the prefetch point, the server schedules a new *PB/FB* stream. If the VCR action ends before the play pointer reaches the prefetch position, the cache management employs the same algorithm as a pause operation to resume the prefetch stream. Otherwise, this node rejoins the system as a new node.

Slow Backward. Initially, the cache management takes no measure upon receiving a slow backward action. If this operation finishes before the player pointer and the prefetch pointer meet together, the cache management utilizes the same routine as a jump backward operation to modify the cache contents. Otherwise, the cache management requests for a slow backward stream from the sever and recovers its own streaming service. Besides, it takes the same approach as the play backward/fast backward action to maintain the streaming services of its children.

5. Streaming Schedule Algorithm

5.1. Temporal Relationship Analysis

We introduce the concept of *virtual start time* vt_x of a stream S_x to depict the absolute start time of a streaming and compare the current playback offset of asynchronous streams:

$$\begin{cases} vt_x = t_x - o_x/\beta r, if \beta > 0 \\ vt_x = t_x - (L - o_x)/\beta r, if \beta < 0 \end{cases} \quad (5)$$

where L is the time length of a media file. In the $time-offset$ (see Fig. 1) based coordinate system, the virtual start time of a forward stream is the intersection point of its extension and x-axis. The virtual start time of a backward stream is the intersection point of its extension and the L-height horizontal line. For instance, in Fig. 1 the virtual start time of stream S_a and S_g are vt_a and vt_g, respectively.

With the above definition, we define the temporal relationship among homogeneous VCR streams (i.e., the same VCR type) as follows. Suppose node x and node y send two homogeneous streams S_x and S_y for the same media file, respectively. If $vt_x < vt_y$, we call S_x the predecessor of S_y and S_y the successor of S_x (denoted as $S_x \prec S_y$). Note that the start time of a predecessor may be later than that of the streaming itself. For example, we can see from Fig. 1 that S_a and S_g satisfy with $S_g \prec S_a$, but obviously $t_g > t_a$.

Since the received objects are buffered into local cache for a certain time, it is possible to serve new streaming requests through utilizing these cached media objects. Fig. 4 illustrates a potential scenario. In this figure, S_a and S_b are two *NP* streams following $S_a \prec S_b$. Node b issues a stream request R_b with the offset o_b at time t_b. The double line along S_a indicates the caching progress of node a at time t_b. Node a contains o_b at t_b. It can serve R_b through forwarding the cached objects. Fig. 4 also shows an instance of sharing cache contents among two backward streams S_c and S_d ($S_c \prec S_d$). We may observe that, if the offset of a request falls within the current cache window of it predecessors, then the request could be served by these predecessors as long as they have enough outbound bandwidth. This is the key for our temporal relationship based streaming schedule algorithm. CoopStream should provide two kinds of streaming schedule algorithms, namely instant streaming schedule algorithm and delayed streaming schedule algorithm, respectively.

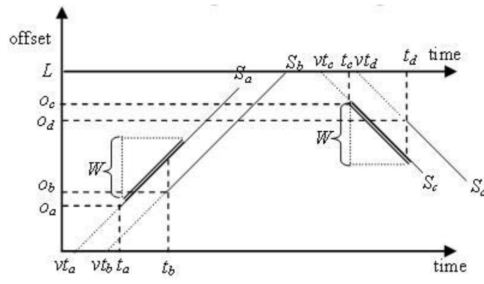

Figure 4. A scenario of cache sharing.

5.2. Instant Streaming Schedule Algorithm

Upon receiving an instant streaming service request, the server must schedule a stream immediately. The *Instant Streaming Schedule Algorithm* (ISSA) looks for the predecessors containing the required object. If this is successful, it returns the search result to the requestor. Otherwise, it initiates a new streaming from itself. Suppose node x sends an instant streaming request R at time t_0. Let P denote the predecessor set of x, and N the eligible candidate set. Algorithm 1 gives the pseudo-code of ISSA.

Algorithm 1: ***the instant streaming schedule algorithm***

1: **begin**
2: **while** (P $\neq$ null) **do**
3: select a member p from P and let $P = P - p$;
4: **if** R is a backward stream **then**
5: exchange the value of $p.op$ and $p.oi$;
6: **end if**
7: $\Delta o = p.op - p.oi$;
8: $\Delta t = R.statime - p.statime$;
9: **case** $\Delta o \leq W/2$:
10: **if** ($\Delta t \leq \Delta o$) **then**
11: **if** ($p.oi < R.oi < p.oi + \Delta t$ or $p.op < R.oi < p.op + \Delta t$) **then**
12: $N = N + p$;
13: **else if** ($\Delta o < \Delta t <= W - \Delta o$) **then**
14: **if** ($p.oi < R.oi < p.op + \Delta t$) **then**
15: $N = N + p$;
16: **else if** ($\Delta t > W - \Delta o$) **then**
17: **if** ($p.op + \Delta t - W < R.oi < p.op + \Delta t$) **then**
18: $N = N + p$;
19: **end if**
20: **break**;
21: **case** $\Delta o > W/2$:
22: $\Delta o = W - \Delta o$;
23: **if** ($\Delta t <= \Delta o$) **then**
24: **if**($p.oi < R.oi < p.oi + \Delta t$ or $p.op < R.oi < p.op + \Delta t$)
25: $N = N + p$;
26: **else if** ($\Delta o < \Delta t <= W - \Delta o$) **then**
27: **if** ($p.oi + \Delta t - \Delta o < R.oi < p.oi + \Delta t$ or $p.op < R.oi < p.op + \Delta t$) **then**
28: $N = N + p$;
29: **else if** ($\Delta t > W - \Delta o$) **then**
30: **if**($p.op + \Delta t - W < R.oi < p.of + \Delta t$) **then**
31: $N = N + p$;
32: **end if**
33: **break**;
34: **end while**
35: **if** ($N <> null$) **then**
36: **return** N
37: **else**
38: schedule a new stream from the server;
39: **end if**
40: **end begin**

Lines 9-20 outline the algorithm where the initial position of the prefetch pointer is at the left side of the middle line. Among them, lines 10-12 deal with the case where there is a gap between the past frames and the future frames; lines 13-15 consider the situation where the cache is consecutive but unsaturated; lines 16-19 are for the remained situation. Similarly, lines 21-33 describe the routines for the case where the start position of the prefetch pointer is at the right side of the middle line.

Algorithm 1 is useful for answering a streaming request with a predetermined offset. However, it doesn't address the cases where the start offset of a prefetch stream is an interval $[a, b]$ subject to (3). In fact, this problem is transformed to one of determining whether there exists an intersection between the cache window of a predecessor and the offset interval. The essence of ISSA is to judge whether the requested stream offset falls into the cache of a predecessor. Therefore, it could be addressed by replacing the judging condition of algorithm 1 (Line 11, 14, 17, 24, 27 and 30, correspondingly).

5.3. Delayed Streaming Schedule Algorithm

A delayed request is made up of the start offset and the maximal allowable delayed time χ. There are two kinds of nodes that may potentially answer a delayed request: (1) the nodes that already hold the requested object and (2) the nodes that will cache the requested media object within χ. The *Delayed Streaming Schedule Algorithm* (DSSA) should search for all these nodes. If no node is returned, it opens a new streaming connection from the server. Algorithm 2 gives the pseudo-code of the DSSA.

Algorithm 2: ***the Delayed streaming schedule algorithm***

1: **begin**
2: **while** (P $\neq$ null) **do**
3: select a member p from P and let $P = P - p$;
4: $\Delta o = p.op - p.oi$;
5: $\Delta t = R.statime - p.statime$;

Table 2. VCR action patterns

FB	SB	PB	PA	NP	SF	FF	JB	JF
0.05	0.01	0.01	0.7	0.04	0.05	0.05	0.05	0.05

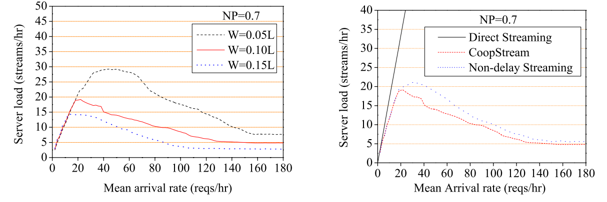

Figure 5. a) Effect of cache size; b) A comparison with non-delay scheduling policies.

6: **if** p contains stream object $R.oi$ **then**
7: $p.rt = W - (p.op + \Delta t - R.oi)$; *//the residing time*
8: $p.et = 0$; *//the enter time*
9: $N = N + p$;
10: **else if** p doesn't contain stream object $R.oi$ **then**
11: **if** $(\Delta t \leq \Delta o)$ **then**
12: **if** $(p.oi + \Delta t < R.oi \leq p.oi + \Delta t + \chi)$ **then**
13: $N = N + p$;
14: **if** $(p.oi + \Delta t < R.oi < p.op)$ **then**
15: $p.enttime = R.oi - (p.oi + \Delta t)$;
16: $p.restime = W - (p.op + \Delta t - R.oi)$
17: **else**
18: $p.enttime = R.oi - (p.oi + \Delta t)$;
19: $p.restime = W$;
20: **end if**
21: **end if**
22: **else if** $(\Delta t > \Delta off)$ **then**
23: **if** $(p.op + \Delta t < R.oi \leq p.op + \Delta t + \chi)$ **then**
24: $N = N + p$;
25: $p.enttime = R.oi - (p.op + \Delta t)$;
26: $p.restime = W$;
27: **end if**
28: **end if**
29: **end if**
30: **end while**
31: **return** N to the requestor
32: **end begin**

Obviously, we can use ISSA (lines 2-30) to search for the node currently containing the requested objects(line 6 in DSSA). Lines 10-29 present the algorithm of looking for the second kind of nodes. Lines 12-21 deal with the situation where there is a gap between the past frame and the future frame. Lines 22-28 capture the case where the cache is consecutive.

6. Simulation Results

We evaluate the performance of CoopStream using J-Sim [www.j-sim.org]. In our experimental environment, 70% of end hosts have the bandwidth capability of $10Mbps$, the remained nodes are with $100Mbps$. The network bandwidth of the streaming server is $100Mbps$. We assume the video length L to be 100 minutes, the normal playback rate r to be $1Mbps$. The start offset of a new request is uniformly distributed within (0, L). Let $|\beta| = 2$ for the fast play and $|\beta| = 0.5$ for the slow play. The arrival of requests is a Poisson process with an inter-arrival time of λ. We define the default probabilities of each VCR action in Table 2.

Fig. 5(a) shows the server load with different cache sizes. We can see that the bandwidth consumption initially increases over λ until it reaches to a peak value. Later, it gradually decreases and finally converges. The reason is as follows. A small arrival rate means that the interval between two requests is large while the system size is small. The chance of getting the required segment from end hosts is small. However, when λ increases, more nodes join the system within a shorter time. As a result, the streaming file is gradually divided and cached into different nodes by these asynchronous requests. Hence, the possibility of serving streaming requests via end hosts improves. When λ reaches a threshold, the service activity of the system is balanced. Thus, the server load is converged. From fig. 5(a) we can also see that the larger the cache size, the smaller the threshold will be. Fig. 5(b) compares the performance of CoopStream with non-delay scheduling. We set $\chi = 0$ in DSSA to simulate the non-delay scheduling where the server only searches for the nodes currently holding the requested object. As shown in Fig. 5(b), CoopStream is superior to the non-delay scheduling algorithm. The result confirms that the DSSA further improves the system performance through predicting the cache content availability using the temporal relationship model. We also depict the theoretical result of direct streaming where all the requests are served by the server directly (generated with $\lambda L/60$). We may observe that, when λ is very small, the server load of CoopStream is a little larger than that of direct streaming. CoopStream usually needs to adjust cache contents with a second channel after a VCR action. Therefore, the server has to initiate a second one from itself when λ is small.

Fig. 6 demonstrates the performance of CoopStream under different VCR action patterns. In this experiment, when the probability of *NP* action (*NP*) decreases, the reduced value is proportionally added to the other VCR actions, and vice versa. Fig. 6(a) gives the experimental results for the

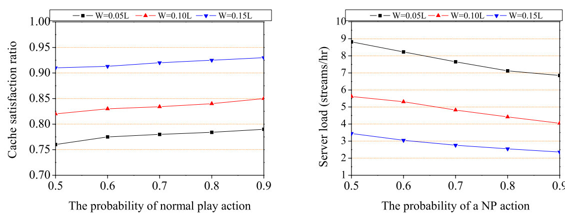

Figure 6. a) Cache satisfaction; b) Server load

cache satisfaction and Fig. 6(b) illustrates its impact on the server load. These two figures indicate that, when the probability of non-NP VCR actions increases, the cache satisfaction ratio decreases slightly while the server load augments lightly. This means that the negative impact of VCR action patterns is small. Fig. 6 further confirms that the cache size is the main factor affecting the performance of CoopStream.

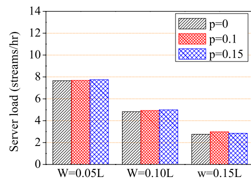

Figure 7. Effect of abnormal leaving.

We also investigate the server performance variations over different abnormal leaving probabilities p. Fig. 7 shows that the abnormal leaving has little impact. The reason is twofold. First, the node can detect this abnormal event through the observation of a suddenly heavy packet loss. Besides, the node can continue its streaming service by consuming the future frames in the cache. Second, the server in turn can try to find an eligible node from participating nodes by performing the DSSA algorithm, rather than opening a new streaming channel immediately. Thus, CoopStream is robust as well as scalable.

7. Conclusions

In this paper, we propose an overlay networks based scheme, *CoopStream*, for the true on-demand streaming service with VCR functionality. Our solution is centered around a dual-channel based cache management mechanism, and a temporal correlation based streaming schedule algorithm. To maximally reduce the bandwidth consumption on server, the dual-channel based cache management actively monitors and adjusts the cache contents based on the observation of current cache status, so as to satisfy VCR actions with the cache as possible, and the temporal relationship based streaming schedule algorithm tries to redirect the stream request to an end host prior to opening a new streaming from the server. Our study shows that CoopStream performs well in term of scalability and efficiency. It can support all kinds of VCR operations as studied in this paper without delay.

References

[1] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. of ACM SIGMETRICS 2000*, pages 1–12, Santa Clara, CA, June 2000.

[2] Y. Cui, B. Li, and K. Nahrstedt. ostream: Asynchronous streaming multicast in application layer overlay networks. *IEEE JSAC, Special Issue on Recent Advances in Service Overlays*, 22(1):91–106, January 2004.

[3] A. Dan, D. Sitaram, and P. Shahabuddin. Schedule policies for an on-demand video server with batching. In *Proc. of ACM Multimedia '94*, pages 15–23, San Francisco, CA, October 1994.

[4] Z. Fei, M. H. Ammar, I. Kamel, and S. Mukherjee. Providing interactive functions through active client buffer management in partitioned video broadcast. In *Proc. Of Networked Group Communication (NGC'99)*, pages 152–169, Pisa, Italy, November 1999.

[5] M. Guo, M. H. Ammar, and E. W. Zegura. Cooperative patching: a client based p2p architecture for supporting contimuous live video streaming. In *Proc. of ICCCN 2004*, pages 481–486, Chicago, IL, October 2004.

[6] K. A. Hua, Y. Cai, and S. Sheu. A multicast technique for true video-on-demand services. In *Proc. of ACM Multimedia '98*, pages 191–200, Bristol, UK, September 1998.

[7] J. Jannotti, K. Gifford, M. Kaashoek, and J. J. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation*, pages 197–212, San Diego, CA, October 2000.

[8] S. Jin and A. Bestavros. Cache-and-relay streaming media delivery for asynchronous clients. In *Proc. Of NGC '02*, Boston, MA, October 2002.

[9] S. W. Lau, C. S. Liu, and L. Golubchik. Merging video streams in a multimedia storage server: complexity and heuristics. *Multimedia Systems*, 6(1):29–42, January 1998.

[10] B. Ye, M. Guo, D. Chen, and L. Sang. A heuristic routing algorithm for degree-constrained minimum overall latency application layer multicast. In *Proc. of ISPA'05*, pages 320–332, Nanjing, China, November 2005.

[11] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork,. In *Proc. Of IEEE INFOCOM'96*, pages 594–602, San Francisco, CA, March 1996.

[12] R. Zhang, A. R. Butt, and Y. C. Hu. Topology-aware peer-to-peer on-demand streaming. In *Proc. Of 2005 IFIP Networking Conference*, pages 1–14, Waterloo, Ontario, Canada, May 2005.