

# Tree-Assisted Gossiping for Overlay Video Distribution

Ming Zhou and Jiangchuan Liu<sup>1</sup>

School of Computing Science, Simon Fraser University  
Burnaby, British Columbia, Canada

{jmzhou, jcliu}@cs.sfu.ca

**Abstract.** Given its readily deployable nature and broad applications for digital entertainment, video streaming through overlay networks has received much attention recently. While a tree topology is often advocated due to its scalability, it suffers from discontinuous playback under highly dynamic network environments. For on-demand streaming, the asynchronicity among client requests further aggravates the problem. On the other hand, gossip protocols using random message dissemination, though robust, fail to meet the real-time constraints for streaming applications. In this paper, we propose TAG, a Tree-Assisted Gossip protocol that addresses the above issues. TAG adopts a tree structure with time indexing to accommodate asynchronous requests, and an efficient pull-based gossip algorithm to mitigate the impact of network dynamicity. It seamlessly integrates these two approaches and realizes their best features, namely, low delay with a regular tree topology, and robust delivery with smart switching among multiple paths, thus making effective use of the available bandwidth in the network. We evaluate the performance of TAG under various settings, and the results demonstrate that it is quite robust in the presence of local and global bandwidth fluctuations. As compared to pure tree-based overlay VOD system, it achieves much lower and stable segment missing rates, even under highly dynamic network conditions.

**Keywords:** *Overlay networks, Video-on-demand, Multicast tree, Gossip protocol*

---

<sup>1</sup> Contact author: Tel: +1 604-291-4336 Fax: +1 604-291-3045. This research is supported by a Canadian NSERC Discovery Grant and SFU President's Research Grant.

## 1. INTRODUCTION

Recently, application-layer overlays have emerged as a readily deployable and thus promising alternative to IP multicast for multi-point video distribution [1-14]. An overlay network is built out of unicast tunnels across cooperative nodes with certain buffering capabilities. Each overlay node acts as an application-layer proxy, and caches a certain amount of the data it receives; the data are then relayed among the active nodes in the overlay to realize multicasting. As an application-layer solution, it largely avoids the known practical and political issues for IP multicast deployment.

In existing overlay construction algorithms, a tree structure is often advocated for data delivering [18-25,29,30], which originates from and works efficiently with IP multicast. For an application-level overlay with dynamic nodes, it however suffers from several severe problems. In particular, any bandwidth fluctuation or failure at a node close to the root may cause buffer underflow at a large population of downstream nodes; such situations are not uncommon as each overlay node can join or leave at will. For on-demand streaming, the asynchronicity among client requests further aggravates the above problems.

Opposite to a tree-based protocol, gossip protocols enable random data dissemination with no support from a regular overlay structure [15-17,28]. In a typical gossip process, a node randomly selects a subset of target nodes to deliver recently available data segments, and meanwhile, receives segments pushed from these nodes. It is known that gossip algorithms achieve highly robust data distribution. Nevertheless, it is not straightforward to apply gossiping in on-demand streaming, for it often fails to achieve a timely delivery. Furthermore, the push-based gossip could cause excessive data duplications, which is particularly severe for high-bandwidth videos.

In this paper, we present TAG, a Tree-Assisted Gossip protocol for on-demand media streaming. TAG constructs and maintains two overlays, namely, a tree overlay and a gossip overlay, which collectively deliver video streams to clients. We design intelligent and efficient overlay construction and data delivering algorithms for this hybrid system. They seamlessly integrate the two distinct approaches and realize their best features: low delay with a regular tree topology, and robust delivery with smart switching among multiple paths, thus making effective use of the available bandwidth in the network. We present a timing listing that accommodates the asynchronous requests in an on-demand streaming system. We also substitute the push-based delivery by a pull process, which greatly eliminates the massive redundancy due to random disseminations. Finally,

we enhance the TAG system by introducing AVL tree based indexing, which facilitates non-sequential accesses.

We evaluate the performance of TAG under various network configurations. The results demonstrate that it is highly robust when facing local and global bandwidth fluctuations. As compared a pure tree-based overlay VoD system, it achieves much lower and stable segment missing rates (<10%) under dynamic network environments. Meanwhile, its control overhead is kept at low levels, suggesting that TAG scales well to large overlay networks.

The rest of the paper is organized as follows. An overview of TAG is given in Section 2, together with detailed protocol operations presented in Section 3. In Section 4, we further enhance TAG by introducing AVL tree based indexing. The performance of TAG is evaluated in Section 5. Finally, Section 6 concludes the paper and offers some future research directions.

## 2. OVERVIEW OF TAG

A TAG system consists of a content server, which stores a repository of media files, and a set of autonomous nodes, which can join or leave the system at will. We assume that the address of the content server is publicly available through an advertising protocol, such as SAP; thus, a node can always retrieve the media stream from the server; yet a scalable solution is expected given the limited server resources. To this end, each node in the TAG system contributes a certain buffer space, which caches the recently received the data at the node, and a node thus can retrieve data not only from the server, but also from other active nodes with expected data in their buffers.

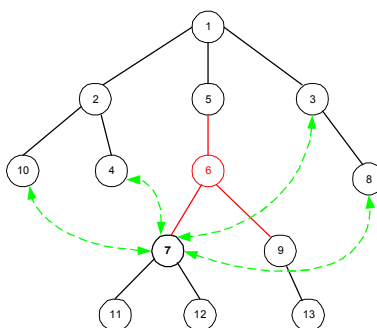


Figure 1. Tree-assisted gossip overlay.

TAG adopts a tree assisted gossip protocol to organize the nodes, locate partners with cached data, and schedule the data fetching. Fig. 1 shows such an overlay structure, where a tree organizes all the nodes, and these nodes also form gossip partners to exchange data with each other.

We divide buffer at every node  $i$  into two parts, namely, a *forward buffer* of size  $b_i^+$ , and a *backward buffer* of size  $b_i^-$ . A node stores data segments pre-fetched from its parent or partners in its forward buffer, and caches played out segments in its backward buffer, both of which can be used to supply its children or partners upon requests.

We show an example the gossip partnership for node 7 in Fig. 1, and stress three salient features of this hybrid design: (1) *Adaptive*, as a receiver can intelligently switch among multiple suppliers (parent and gossip partners), and the fanout constraint for tree nodes can be relaxed; (2) *Efficient*, as the availability at different paths/nodes can be explored; and (3) *Robust*, as the bandwidth fluctuation or node failure at a particular path has less impact.

Our experimental results suggest that most of these features are enabled by the gossip algorithm; yet the tree structure is indispensable to meet the real-time constraints. It is, however, not straightforward to employ a tree structure or a gossip algorithm for on-demand streaming, not to mention integrating them. There are several challenges to be addressed, in particular:

1. How is a newly joined node inserted to the tree and assigned with gossip partners? Note that the nodes are with asynchronous join times and limited buffer spaces. Similar issues have to be addressed when node fail or leave the system.
2. For each expected data segment, where and when to fetch it? There are multiple suppliers with non-uniform bandwidth and data availability, and the playback deadline has to be met.

We detail the TAG operations in the next two sections, which offer efficient solutions to the above issues in this hybrid system.

### 3. PROTOCOL OPERATIONS

For ease of exposition, we focus on the distribution of a single video stream only, and the solution can be easily extended to the multi-stream case. We assume that the stream is divided into equal-sized data segments, each with a unit playback time. The buffer size is measured as the total number of segments it can accommodate. We also assume that each segment has a sequence number, and video playback at a node always starts from the first segment. Extensions to support non-sequential accesses will be addressed in the next section.

#### A. Timing Condition and List

Due to data asynchronicity in on-demand streaming, a parent-child relationship or gossip partnership cannot be directly set up between any two nodes, even without the outbound bandwidth constraint. We now derive the conditions for two nodes to form a parent-child or gossip relation, which will serve as a foundation for overlay construction and maintenance.

Fig. 2 depicts a snapshot of the buffers at nodes  $i$  and  $j$ , respectively, at time  $t$ . Suppose  $t - t_i$  is the currently played segment for node  $i$ , which joins the system at time  $t_i$ ; the maximum sequence number of the data segments in its buffer is thus  $t - t_i - b_i^+$ , and the minimum one is  $t - t_i - b_i^-$ ; so is node  $j$ .

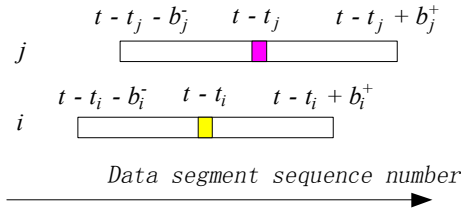


Figure 2. Buffer status at nodes  $j$  and  $i$  at time  $t$ .

From Fig. 2, the necessary condition for  $j$  being the parent of node  $i$  should be

$$\begin{cases} t - t_i < t - t_j \\ t - t_i > t - t_j - b_j^- \end{cases}, \quad (1)$$

which is equivalent to

$$t_i - b_j^- < t_j < t_i. \quad (2)$$

That is, the join time of node  $j$  should be earlier than that of node  $i$ , and their difference should be less than the draining time of the backward buffer of node  $j$ .

Opposite to the parent-child relation, data delivery is bidirectional with a gossip partnership. From Fig. 2, for node  $i$  to forward data to node  $j$ , the following condition should be met:

$$i \rightarrow j : \begin{cases} t - t_i - b_i^- < t - t_j + b_j^+ \\ t - t_i + b_i^+ > t - t_j \end{cases} \quad (3)$$

which basically states that at least part of the buffer (backward buffer plus forward buffer) of node  $i$  should overlap with the forward buffer of node  $j$ . Similarly, the condition for node  $j$  to forward data to  $i$  is

$$j \rightarrow i : \begin{cases} t - t_j - b_j^- < t - t_j + b_i^+ \\ t - t_j + b_j^+ > t - t_i \end{cases} \quad (4)$$

Combining Eq. (3) and (4), we have

$$j \leftrightarrow i : \begin{cases} t_i < t_j + b_j^- + b_i^+ \\ t_i > t_j - b_j^+ \\ t_i > t_j - b_j^+ - b_i^- \\ t_i < t_j + b_i^+ \end{cases} \quad (5)$$

which follows that

$$t_i - b_i^+ < t_j < t_i + b_j^+ \quad (6)$$

To efficiently examine the above timing conditions in TAG, we link all the active nodes into a *timing list*, sorted according to their joining times. In this list, node  $j$  is the *predecessor* of node  $i$  if node  $j$  joined system immediately before node  $i$ , and, accordingly,  $i$  is referred to as  $j$ 's *successor*. A bidirectional link is then added between the predecessor and the successor. Fig. 3 depicts such a timing list structure for the nodes.

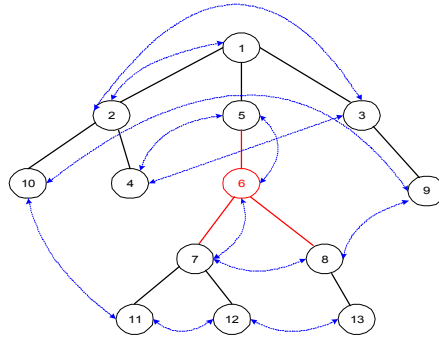


Figure 3. An illustration of the index list structure (dashed line), which facilitates the construction of the delivery tree (solid line) and gossip partnerships.

Note that, to construct and maintain the timing list, the content server needs to keep track of the latest joined node only. In the bootstrapping stage, the content server itself is such a node. Each newly joined node first contacts the content server, which then redirects the node to the existing latest joined node, and a predecessor and successor relation can then be formed, as will be detailed next.

## B. Construction of TAG Overlay

Table 1. Fields in each node for overlay construction.

<i>Parent</i>	Parent of a node in the delivery tree
<i>ChildrenList</i>	List of children in the delivery tree
<i>Predecessor</i>	Predecessor in the timing list
<i>Pre-Predecessor</i>	Predecessor of predecessor in the timing list
<i>Successor</i>	Successor in the timing list
<i>Suc-Successor</i>	Successor of successor in the timing list
<i>PartnerList</i>	List of gossip partners

A TAG system is constructed with nodes joining the overlay asynchronously. To facilitate the join process, each node maintains a set of status information, as shown in Tab.1, and a new node  $i$  performs the following join operations:

- 1) Node  $i$  sends message  $Join<i>$  to the content server;
- 2) The content server records the join time of node  $i$ , and redirects it to nodes  $L$ , which is the latest joined node so far, i.e., the one immediate before node  $i$ ;
- 3) Node  $L$  sets node  $i$  as its successor, and node  $i$  sets node  $L$  as its predecessor. The predecessor's predecessor and successor's successor relation is also set between node  $i$  and the predecessor of node  $L$ ;
- 4) Node  $i$  invokes a parent search and a partner search algorithm to locate its parent and gossip partners, and then sets the corresponding relations. Both algorithms rely on the timing list to check the timing conditions, as shown in Figs. 4 and 5, respectively.

- 
- 1) Traverse the timing list, starting from the predecessor of node  $i$ ;
  - 2) Test condition (2) for each encountered node, until the second node violating the condition is found, or  $K$  nodes have been visited;
  - 3) For all the nodes that satisfy the condition, select the one with the maximum bandwidth to node  $i$  as its parent.
- 

Figure 4. Parent search algorithm for node  $i$ .

- 
- 1) Traverse the timing list, starting from the predecessor of node  $i$ ;
  - 2) Test condition (6) for each encountered node, until the second node violating the condition is found, or  $K$  nodes are visited;

- 4) Among all the nodes that satisfy the condition, randomly select  $k$  nodes as gossip partners.
- 

Figure 5. Gossip partner search algorithm for node  $i$ .

For both search algorithms, the number of nodes involved is bounded by  $O(K)$ , and we will show through experiments that a relatively small  $K$  (say less than 12) is enough in most cases. The number of gossip partners,  $k$ , is also an important factor, whose impact will be investigated in our experiments as well. Note that we also make each node linked to its predecessor's predecessor and successor's successor in the timing list, which helps with recovering from node failures. The predecessor for the list head (the content server) and the successor for the list tail (the latest joined node) are two special cases, in which the predecessor and the successor are set as the head itself and tail itself, respectively.

### C. Maintenance of TAG Overlay

We use a heartbeat protocol to maintain the parent-child and partner relationships. Each node periodically sends an *Echo* message to its related nodes, namely, parent, children, and partners, as well as successor and predecessor in the index list. The leave of a node, due either to an intended departure or abrupt failure, can thus be easily detected. The following failure recover operations will then be executed at the affected nodes:

*Predecessor/Successor:*

- 1) The *predecessor* and *successor* of the failed node contact each other and form a direct predecessor-successor relationship; this is viable because each node records its pre-predecessor and suc-successor as well;

*Parent/Gossip Partners:*

- 1) Removes the failed node from its children list or gossip partner list;

*Children:*

- 1) Each child invokes the parent search algorithm to locate a new parent. The starting node will be the predecessor of the child, or the pre-predecessor if its predecessor is just the failed node.

In a dynamic network, the above operations can as well be periodically invoked by a node to refine its parent-child relationship or gossip partnership.

### D. Data Delivering



In TAG, a data segment could be available at multiple suppliers, and a commonly used *push* mechanism for data delivering may cause excessive redundancy. We thus resort to a *pull* mechanism, in which a node with data available first sends a *Data Offer* message to a target node, namely, a child or a gossip partner. The target node will then send back a *Data Request* if it decides to fetch a data segment.



Figure 6. Fields of message *Data Offer*.

The fields included in a *Data Offer* are shown in Fig. 6. Note that their sizes are relatively small, as the availability for each segment is indicated by one bit only. To further reduce the overhead, the data offer and request can both be piggyback by the *Echo* messages, and the requests for a set of segments from the same supplier can be batched together as well.

Since a node will collect a set of *Data Offers* from its parent and gossip partners during an exchange period, a key issue is thus to decide which unavailable data segments should be fetched from which node. There are two constraints in this process: 1) each data segment should be fetched before its playback deadline; 2) the number of data segment fetched from a partner should be within its delivery capability, i.e., the outbound bandwidth.

We have designed a heuristic algorithm that follows the above constraints and tries to maximize the success ratio for segment delivering. It starts from examining the segment with the earliest deadline, and then the second earliest, and so on. In case multiple suppliers are available for a segment, the algorithm selects the supplier that offers the least number of unavailable data segments. For example, suppose the segment has two suppliers, one offers ten unavailable segments, while the other does not have any other unavailable segment but the expected segment; the latter is then selected, because the former is more flexible in supplying data and can potentially be use to fetch other unavailable segments if needed. In addition, fewer suppliers also imply that the segment could be relatively new, and thus should be gossiped as soon as possible to minimize delay.

#### 4. ENHANCEMENT WITH AVL TREE BASED INDEXING

In the basic TAG system, we assume sequential accesses that always starts playback from the initial segment of a stream. For implementing VCR-like operations, such as forward, backward, and random seek, however, non-sequential access from arbitrary starting position become necessary. In

this section, we present effective enhancement to the basic TAG system to support non-sequential accesses.

Suppose a new node  $i$  joins the overlay at time  $t_i$  with a playback offset  $o_i$ ; at time  $t$ , the node expects to play out segment  $(t - t_i + o_i)$ . Since the conditions to form parent/children and gossip partners still hold if we replace  $t_i$  by  $(t_i - o_i)$ , a naive solution is to search the timing list until candidates satisfying the revised condition are found. Unfortunately, in the worse case, this may result in a traverse across all the nodes in the sorted timing list, yielding unacceptably high cost. Earlier studies on this issue [20,24,25] have suggested that a centralized server maintains a global tree structure for both timing and data delivering. While this solution is easy to implement, it is often not scalable, and the delivery tree itself is not an ideal indexing structure given that its height is unbounded.

To this end, we introduce an AVL index tree to assist the search in the timing list. An AVL tree is a binary search tree with the following balance property: for any node in the tree, the height of the left and the right sub-tree can differ by at most 1. It is known that, for an AVL tree with  $N$  nodes, its height  $H$  satisfies  $H < 1.44\log(N + 2) - 1.328$ . Hence, the cost of locating an proper insertion point is  $O(\log N)$ , implying that the joining and failure recovery costs would be greatly reduced for non-sequential accesses. It is worth noting that the AVL indexing tree is a complement to the timing list, and is independent of the data delivery tree; hence, the list construction and maintenance, as well as the data scheduling and dissemination algorithms, remain unchanged.

We now detailed the operations of the AVL index tree for non-sequential accesses. Tab. 2 lists the related information kept at each node.

Table 2. Fields at each node for the AVL indexing tree.

<i>avlParent</i>	Parent in the AVL tree
<i>avlLeftChild</i>	Left child in the AVL tree
<i>avlRightChild</i>	Right child in the AVL tree
<i>avlLeftHeight</i>	Height of the left subtree in the AVL tree
<i>avlRightHeight</i>	Height of the right subtree in the AVL tree
<i>Virtual join time</i>	Value $(t_i - o_i)$ for node $i$
<i>avlGrandParent</i>	Parent's parent in the AVL tree

### A. Joining Operations with Playback Offset

The AVL index tree is constructed with the growth of the timing list. For a newly joined node with playback offset  $o_i$ , the following operations are performed:

- 1) Node  $i$  sends message  $Join\langle i, o_i \rangle$  to the content server;
- 2) The content server records the virtual join time  $(t_i - o_i)$  of node  $i$ , and redirects it to nodes  $R$ , which is the root of the AVL index tree;
- 3) If the virtual join time of node  $i$  is less than that of node  $R$ ,  $R$  redirects  $i$  to its left child in the AVL index tree, or otherwise to its right child. The above operations are repeated until the corresponding child is empty, and node  $i$  is then inserted to this position as a leaf node;
- 4) If  $i$  is inserted as left child of its  $avlParent$ , it will be the predecessor of  $avlParent$  in the timing list, or else its successor. Similar operations for a new node to join the timing list and data delivery tree are performed (steps 3 and 4 in Section 4.B) with this insert position;
- 5) Node  $i$  sets its height to 0, and sends a  $HeightReport$  message to its  $avlParent$ . Upon receiving the report, the parent resets its  $avlLeftHeight$  or  $avlRightHeight$ , depending on which branch the report comes from, and then calculate its own height as

$$\max(avlLeftHeight, avlRightHeight) + 1.$$

If the height is changed, the node reports as well to its own  $avlParent$  until the root of the AVL tree is reached;

- 6) If unbalance is detected after update the height, a subtree rotation should be performed, and the root of the AVL, if updated, is then reported to the content server.

Since the height of the AVL tree is  $O(\log N)$ , the cost for a joining operation is thus bounded by  $O(\log N)$ .

### B. Failure Recovery

We assume that each node also maintains its relation with its  $avlParent$ ,  $avlLeftChild$  and  $avlRightChild$  through the heartbeat protocol, and its failure can thus be detected by these nodes. The following recovery operations will then be performed (for ease of exposition, we denote the failed node as node  $F$ , and its predecessor and successor in the timing list as  $P$  and  $S$ , respectively):

- 1)  $F$ 's  $avlParent$  removes  $F$  from its children list;  $F$ 's  $avlLeftChild$  and  $avlRightChild$  respectively mark their links to  $F$  as *broken*;

- 2)  $P$  and  $S$  respectively send a probe, which is forwarded toward the root in the AVL tree, until the root or a link marked as *broken* is encountered;
- 3) Assume  $W_p$  is the last node traversed by  $P$ 's probe and  $W_s$  is that by  $S$ 's probe. There are three different cases to be addressed:

Case 1: Both probes stop after encountering a broken link.

We can prove (see Appendix) that  $W_p$  and  $W_s$  must respectively be the *avlLeftChild* and the *avlRightChild* of  $F$  in the AVL tree. Furthermore,  $S$  must be a leaf node or a node with only right child in the AVL tree. The following operations are then performed:

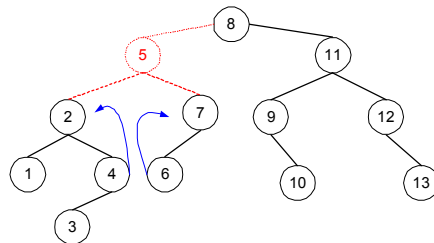
- a) If  $S$  has *avlRightChild*, it will be connected to the *avlParent* of  $S$  as a right child;
- b)  $S$  sets  $W_p$  as its *avlLeftChild*, and  $W_s$  as *avlRightChild*;
- c)  $S$  sets the *avlGrandParent* of  $W_p$  (which is  $S$ 's *avlLeftChild* now) as its own *avlParent*;

Case 2: Only one probe stops after encountering a broken link; the other stops after reaching the root, or there is no probe sent in that branch at all. We can prove (see Appendix) that  $F$  must have either *avlLeftNode* or *avlRightNode*, while not both. This child is then directly connected to its *avlGrandParent* to substitute the failed node;

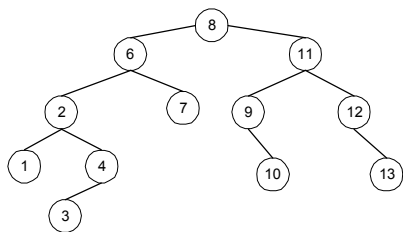
Case 3: Neither probe encounters a broken link. We can prove (see Appendix) that  $F$  in this case must be a leaf node in the AVL tree, and thus no further operations are needed;

- 4) Both the *avlLeftChild* and the *avlRightChild* of  $F$  report their tree height to their new *avlParent*, and, if necessary, perform re-balancing operations as in Step 4 of the joining process.
- 5) The timing list is recovered following the steps described in Section 4.B.

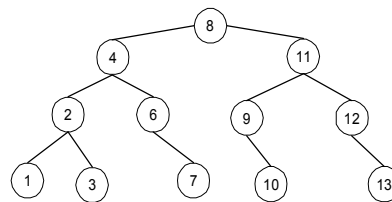
Fig. 7 shows an example of the recovery process for failed node 5. Suppose in the timing list its predecessor ( $P$ ) is node 4 and successor ( $S$ ) is node 6. According to the AVL tree construction algorithm, they should be respectively in the left subtree and the right subtree of node 5. In Step 2 of the recovery algorithm, nodes 4 and 6 probe toward the AVL root and stop nodes 2 ( $W_p$ ) and 7 ( $W_s$ ), which are respectively the *avlLeftChild* and the *avlRightChild* of failed node 5. Node 6 then serves as a substitute for node 5 (Fig. 7b), and a double rotation is then performed to re-balance the AVL index tree (Fig. 7c).



(a) AVL tree with node 5 failed;



(b) AVL tree after node substitution



(c) AVL tree after rebalancing

Figure 7. An illustration of failure recovery (failure recovery case 1).

## 5. PERFORMANCE EVALUATION

We evaluate the performance of TAG under various network settings, with a focus on the following two important measures: control overhead and streaming quality, as well as their sensitivity to parameter settings. We also compare TAG with other overlay on-demand systems, in particular, *oStream*, a pure tree-based system.

### A. System Configurations

Unless otherwise specified, the results presented in this section are based on the following default configurations; yet, similar results have been observed with other configurations, and the impact of several key parameters will be further investigated in the end of this section.

The content server has 10 videos for streaming, each with 256 Kbps rate and 2-hour length. The length of a segment (or a time unit) is 1 second, and the buffer at a node can accommodate 1080 segments, i.e., 15% of a video stream, which is equally split into the forward and backward buffers. The size of the candidate set for parent or gossip partner search is 12, and each node has 5 gossip partners.

The underlying network topology is generated using the GT-ITM package [26], which emulates the hierarchical structure of the Internet by composing interconnected transit and stub domains. The network topology for the presented results consists of 10 transit domains, each with 7 transit nodes,

and a transit node is then connected to 6 stub domains, each with 7 stub nodes. The total number of nodes is thus 3010. We assume that each node represents a local area network with plenty of bandwidth, and routing between two nodes in the network follows the shortest path. The initial bandwidth assigned to the links is as follows: 1.5 Mbps between two stub nodes, 6 Mbps between a stub node and a transit node, and 10 Mbps between two transit nodes. We will also inject cross traffic in the experiments to emulate dynamic network conditions.

To mitigate randomness, each result presented in this section is the average over 10 runs of an experiment.

## B. Overhead of Join and Failure Recovery

We first consider the control overhead of TAG, in particular, the overhead for node joining, leaving, or failing in a dynamic overlay. We are interested in both local and global overheads and thus adopt two measures: the *maximum node cost*, which represents the maximum possible overhead at each node, and the *overall cost*, which represents the total control overhead of the system per operation. The costs are measured in terms of the number of messages exchanged per operation, thus reflecting both the bandwidth consumption and the execution time.

Fig. 8 shows the *maximum node cost* for a joining operation in the three variations of TAG, namely, basic TAG with sequential accesses (TAG-S), basic TAG with non-sequential access (TAG-N), and TAG with non-sequential accesses and AVL indexing (TAG-NA). We assume that the content server is the only initial node in the system, and other nodes then join the system following a Poisson arrival with an inter-arrival time of 2 seconds. In TAG-N, the naive timing list searching algorithm is employed.

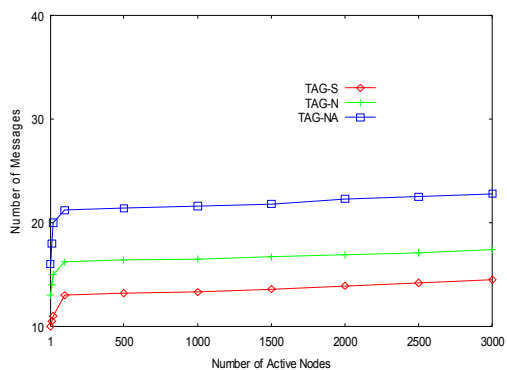


Figure 8. Maximum node cost for node join.

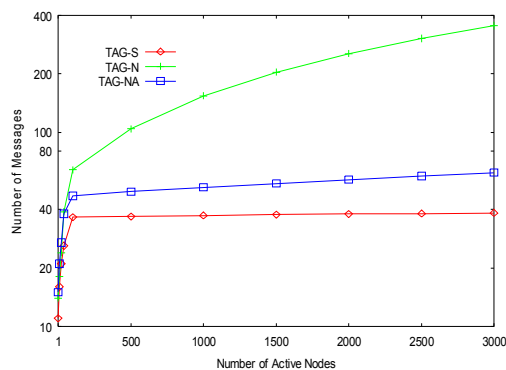


Figure 9. Overall system cost for node join.

Intuitively, the joining node itself incurs the maximum node cost, which is mainly for joining the timing list and initiating the search for parent and gossip partners. As shown in Fig. 8, the cost monotonically increases with increasing the overlay size in the initial part, and becomes almost a constant when the overlay size is greater than 100 nodes. Since TAG-NA incurs extra overhead to maintain the AVL index tree, its maximum node cost is higher than that of the other two.

Nevertheless, as shown in Fig. 9, the overall join cost of TAG-NA can be much lower than that of TAG-N. Since the overall cost is calculated across all the affected nodes in a join operation, it is related not only to the individual node cost but also the number of affected nodes. For TAG-N, the overall join cost is almost a linear function of the system size, for the number of involved nodes is proportional to the overlay size in the naive searching algorithm. For TAG-NA, this becomes a logarithmic function (note that the y-axis in Fig. 8 is log-scaled), suggesting that the joining operation with AVL indexing is scalable, and the cost for maintaining the AVL tree can be ignored for large networks. On the other hand, for TAG-S, the overhead is almost a constant, as only a limited number of tail nodes in the timing list are affected.

The maximum node costs and the overall costs for a failure recovery operation are shown in Fig. 10 and 11, respectively. The general trends are quite similar to that of joining operations, and the overall costs for failure recovery are slightly higher in all the three TAG variations. This is because more nodes are affected, in particular, all children of the failed the node have to re-locate parents.

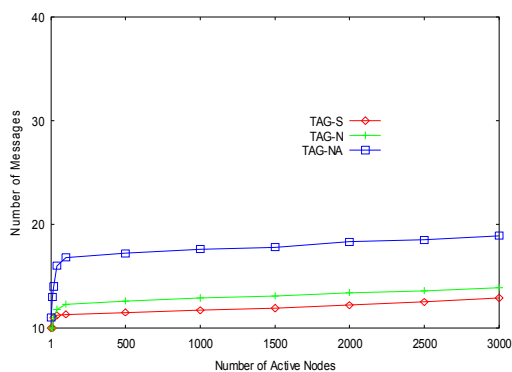


Figure 10. Maximum node cost for failure recovery

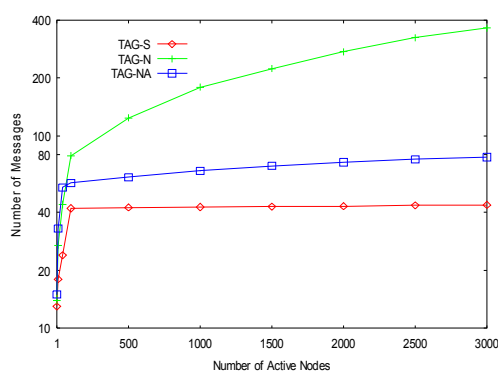


Figure 11. Overall cost for failure recovery

In summary, the joining/failure recovery operations are efficient for both TAG-S and TAG-NA, while that for TAG-N might suffer from high cost in large overlay networks. We thus focus only on the performance of TAG-S and TAG-NA in our following experiments.

### C. Streaming Quality

Given that playback continuity is critical for streaming applications, we adopt the Segment Missing Rate (SMR) as the major criterion for evaluating streaming quality. A data segment is considered missing if it is not available at a node till the play-out time, and the SMR for the whole system is the average ratio of the missed segments at all the participating nodes during the simulation time. As such, it reflects two important aspects of the system performance, namely, *delay* and *capacity*.

For comparison, we also simulate an existing on-demand overlay streaming system, *oStream*, with the same network and buffer settings. *oStream* employs a pure tree structure, in which each node caches played out data and relays to its children of asynchronous playback times. A centralized directory server is used to maintain the global information of the overlay, and facilitates node join or failure recovery. Detailed about *oStream* can found in [20].

#### C.1. Streaming Quality with Bandwidth Fluctuations

We first investigate the performance of TAG under dynamic network environments with local and global bandwidth fluctuations.

To emulate local bandwidth fluctuations, we randomly inject traffic to the network links such that the available bandwidth at each link varies over time, yet the total available bandwidth of the network remains constant, which is 0.8 of the base setting (with no cross traffic).

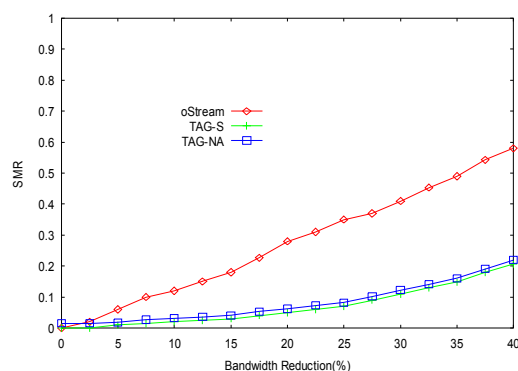
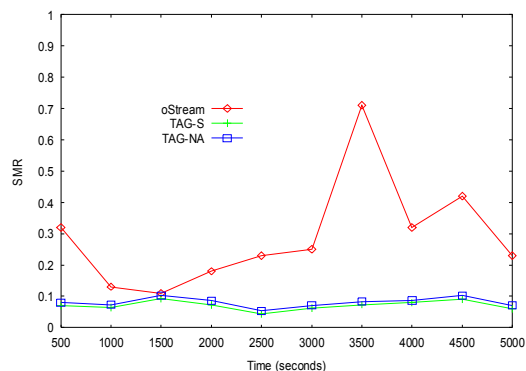


Figure 12. Quality with local bandwidth fluctuations. Figure 13. Quality with different overall network bandwidths.

Fig. 12 shows the segment loss rates (SMRs) for TAG and *oStream* over time. It can be seen that the loss rate of TAG is not only lower than *oStream*, but also quite stable, which is generally around



0.05 to 0.1. From a video decoding point of view, such a loss can be effectively masked by interleaving or error-concealment techniques. On the other hand, the loss rate of oStream greatly fluctuates over time, and the peak value can be as high as 0.7, resulting in poor video quality. This is because oStream relies on a specific tree structure for streaming, and the bandwidth reduction at an internal link of the tree, especially those close to the root, could result in the loss multiplicity problem.

It is known that not only the available bandwidth of local links dynamically changes, but also the overall available bandwidth of a network changes over time on an hour or daily basis, e.g., working and sleeping hours, working days and weekends. Hence, in the second set of experiments, we compare the performance of TAG and oStream under different global network bandwidths. Their segment loss rates are depicted in Fig. 13, where the overall available bandwidth of the network is gradually reduced from 100% to 60% of the base setting.

Not surprisingly, for both TAG and oStream, SMR increases with decreasing the overall bandwidth. However, the increasing rate for TAG is generally lower than that of oStream, especially when the reduction is less than 25%. As an example, for a reduction of 25%, the SMR of oStream has reached 0.35, or 35% of the segments are lost or missed the playback deadline; yet the SMR of TAG is still close to 0.1. This is because oStream explores the available bandwidth at a small subset of network links only, i.e., those tree links, while TAG makes more effective use of the available bandwidth across much more paths. In addition, as explained before, once a segment is lost at a high level node in an oStream tree, it will be lost at all downstream nodes. This is, however, not the case in TAG for each segment has multiple potential suppliers. As a matter of fact, we have observed that over 90% of the data segments are delivered through the gossip process in our experiments, which confirms our intuition that gossip greatly enhances the robustness of the system.

## **C.2. Streaming Quality with Node Failures**

In this set of experiments, we consider dynamic node failures. We assume that there is no global bandwidth reduction, so as to focus on the impact of node failures. Fig. 14 presents the segment missing rates as a function of node failure rate for oStream, TAG-S, and TAG-NA. It can be seen that, when there is no failed node, all the systems work well in this stable scenario. For TAG-S, the segment missing rates slightly increase with increasing the failure rate, but are generally less than

6%. The missing rate of TAG-NA is only a little higher than that of TAG-S. On the other hand, when 10% nodes fail, the segment missing rate for oStream can be as high as 25%.

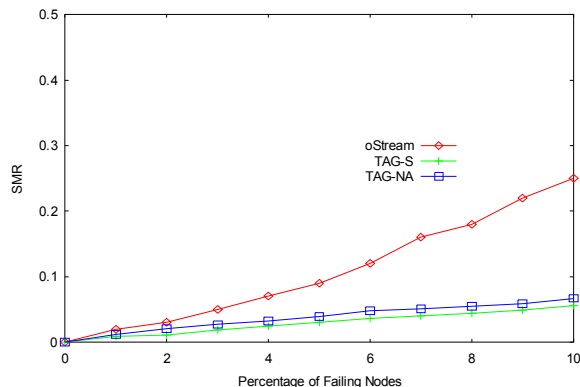


Figure 14. Segment missing rate vs. node failure rate.

We next investigate the effect of random seeking, a key operation toward supporting interactive streaming. For both oStream and TAG-NA, random seeking can be implemented by letting the node leave the system and then re-join with the new playback offset. Fig. 15 compares the streaming quality of oStream and TAG-NA in this scenario. Obviously, the tree-assisted gossip enables a quite robust delivering structure, making the re-seeking operation in TAG-NA much smoother than that in oStream. When 10% nodes perform reseeking, the SMR of TAG-NA is still lower than 10%, while that of oStream has reached 35%, which is difficult to mask at the receiver's end.

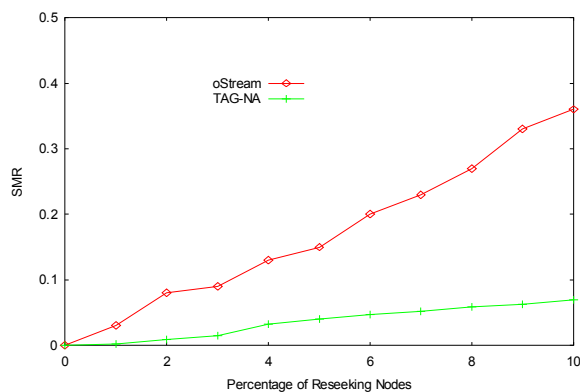


Figure 15. Streaming quality as a function of position reseeking rate.

#### D. Sensitivity to Parameter Settings

In the last set of experiments, we study the sensitivities of the key parameters in the TAG system, in particular, the number of gossip partners, the size of candidate set, and the size of buffers.

Fig. 16 depicts the streaming quality as a function of number of gossip partners for TAG-S and TAG-NA under different system bandwidths. It can be seen that the segment missing rate reduces when increasing the number of gossip partners. This is consistent with our intuition that the system is more robust when increasing the number of suppliers. However, the improvement with over 5 partners is marginal. Since the computation and transmission overhead of maintaining a large number of partners can be excessive, we believe that 5 is a reasonable choice, which is used in our default setting. Similarly, from Fig. 17, we choose 12 as the default value for  $K$ , the size of the candidate set in parent or gossip partner searching. As shown in our previous results, these default settings lead to reasonably low control overhead and quite good streaming quality under various network configurations.

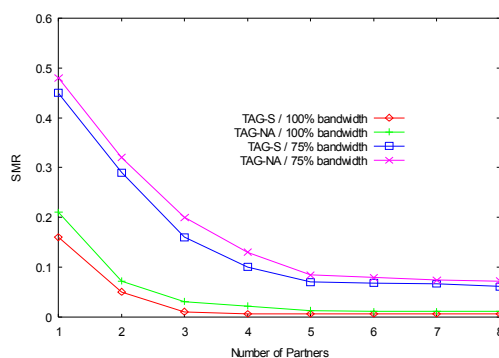


Figure 16. Segment loss rate as a function of the number of gossip partners.

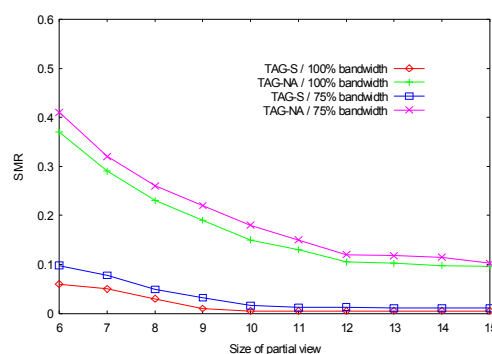


Figure 17. Segment loss rate as a function of the candidate set size.

Regarding buffer size, though it would be desirable if every overlay node caches all the video streams, it is often impractical given the large size of video streams. The choice of buffer size is also closely related to the number of active nodes in the overlay. As shown in Fig. 18, when there are enough active nodes, even a small buffer can enable reasonably good streaming quality with node collaborations. Considering these factors, we set the buffer size as 20% of the video stream size in our experiments, which is sufficient to achieve low segment loss rates and, with this setting, the computation time for the scheduling algorithm is less than 20 ms, which is suitable for real-time streaming.

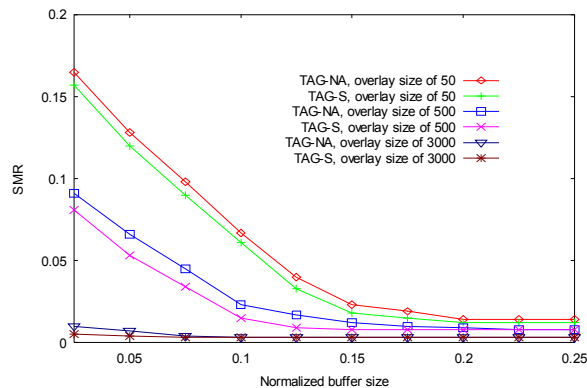


Figure 18. Segment loss rate as a function of buffer size under different overlay sizes (Normalized buffer size = buffer size / video length).

## 6. CONCLUSION AND FUTURE WORK

In this paper, we have presented TAG, a tree-assisted gossip protocol for on-demand streaming. TAG has combined the best features of tree structure and random message dissemination: low delay with a regular tree topology, and robust delivery with random switching among multiple paths, which make effective use of the available bandwidth in the network. The performance of TAG has been extensively evaluated under various network configurations. The results demonstrated that it is highly robust in the presence of local and global bandwidth fluctuations. As compared pure tree-based overlay VOD system, TAG achieves much lower and stable segment missing rates, even under highly dynamic network environments. Possible further research avenues for TAG include optimizing the scheduling algorithm and overlay organization, dealing effectively with failure of multiple related nodes, and incorporating advanced coding techniques, such as layered or multiple-description coding.

## APPENDIX

In the failure recovery algorithm for AVL index tree, assume that the predecessor and successor in the timing list for the failed node  $F$  are  $P$  and  $S$ , respectively, and  $W_P$  is the last node traversed by  $P$ 's probe and  $W_S$  is that by  $S$ 's probe. We have the following observations:

Case 1: Both probe stop after encountering a broken links. In the AVL index tree,  $W_P$  and  $W_S$  must be the *avlLeftChild* and the *avlRightChild* of  $F$ , respectively. Furthermore,  $S$  must be a leaf node or a node with only the right child in the AVL tree.

Case2: Only one probe stops after encountering a *broken* link; the other stops after reaching the root, or there is no probe message sent in that branch at all. In this case,  $F$  must have either *avlLeftChild* or *avlRightChild*, while not both;

Case 3: Neither reaches a broken link. The failed node in this case must be a leaf node in the AVL tree.

***Proof:***

**Case 1:** In this case, obviously, both the  $P$  and  $S$  are non-empty. Moreover, according to the AVL tree construction algorithm, the  $P$  must be in the left subtree of  $F$ , and  $S$  in the right subtree. It follows that, in the AVL index tree,  $W_P$  and  $W_S$  must respectively be the *avlLeftChild* and the *avlRightChild* of  $F$ , the failed node.

Suppose  $S$  has a left child, whose virtual join time should be less than that of  $S$ , but greater than that of  $F$ . That is, in the timing list, this left child should be the successor of  $F$ , which contradicts the fact that  $S$  is the successor. Hence,  $S$  must be a leaf node or a node with only the right child;

**Case 2:** We first assume that only  $P$ 's probe reaches a node with a broken link, which must be the *avlLeftChild* of the failed in the AVL tree, as proved in case 1.

In this case, if  $F$ 's successor  $S$  is empty, i.e., there is no probe sent in the right branch at all,  $F$  cannot have a right child in the AVL tree; otherwise, one of the nodes in  $F$ 's right subtree will become its successor in the timing list.

On the other hand, suppose  $S$  is non-empty and  $F$  has a right child. Since  $S$ 's probe does not reach the *avlRightChild* of  $F$ ,  $S$  cannot be in the right subtree of  $F$ . Assume  $R$  is the root of the minimum subtree that covers both  $F$  and  $S$ . Then,  $S$  must be in the left subtree of  $R$ , while  $F$  must be either  $R$  itself or a node in the right subtree of  $R$ ; otherwise,  $S$ 's probe will reach a broken link as well. It follows that the right child of  $F$  has a virtual join time greater than that of  $F$ , but less than that of  $S$ . This contradicts our assumption that  $S$  is the successor of  $F$ , and hence, the failed node  $F$  does not have right child.

Similarly, we can prove that  $F$  does not have a left child if only  $S$ 's probe reaches a *broken* link (Note that, we can ignore the case that  $P$  is empty in the proof given that content server persists). In summary, the failed node has a single child in this case;

**Case 3:** Suppose  $F$  has a non-empty *avlRightChild*. Since the virtual join time of this *avlRightChild* is greater than that of  $F$ ,  $F$  must have a non-empty successor according to the AVL tree construction algorithm. As proved in Case 2, if  $S$  is non-empty and  $F$  has a right child,  $S$  must be in the right subtree of  $F$ . Hence,  $S$ 's probe will encounter the broken link in the right branch, which contradicts the fact that no broken link is encountered. Similarly, we can prove that  $F$  does not have a left child, and it thus must be a leaf node.

## REFERENCES

- [1] K. Sripanidkulchai, A. Ganjam, B. Maggs and H. Zhang, "The feasibility of supporting large-scale live streaming applications with dynamic application end-points," in *Proc. ACM SIGCOMM'04*, Portland, OR, USA, Aug. 2004.
- [2] Y. Chu, S. Rao, and H. Zhang, "A case for end system multicast," in *Proc. ACM SIGMETRICS'00*, Santa Clara, CA, USA, 2000.
- [3] D. Xu, M. Hefeeda, S. Hambruch, and B. Bhargava, "On peer-to-peer media streaming," in *Proc. IEEE ICDCS'02*, Wien, Austria, Jul. 2002.
- [4] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proc. ACM SIGCOMM'02*, Pittsburgh, PA, Aug. 2002.
- [5] V.N. Padamanabhan, H.J. Wang, P.A. Chou, and K. Sripanidkulchai, "Distributing streaming media content using cooperative networking," in *Proc. NOSSDAV'02*, USA, May 2002.
- [6] M. Hefeeda, A. Habib, B. Botev, D. Xu, and B. Bhargava, "PROMISE: peer-to-peer media streaming using CollectCast," in *Proc. ACM Multimedia(MM'03)*, Berkeley, CA, Nov. 2003.
- [7] M. Yang and Z. Fei, "A proactive approach to reconstructing overlay multicast tree," in *Proc. IEEE INFOCOM'04*, Hong Kong, Mar. 2004.
- [8] H. Deshpande, M. Bawa, and H. Garcia-Molina, "Streaming live media over peer-to-peer network," *Technical Report*, Stanford University, 2001.
- [9] D.A. Tran, K.A. Hua, and T.T. Do, "A peer-to-peer architecture for media streaming," in *IEEE J. Select. Areas in Comm.*, vol. 22, Jan. 2004.
- [10] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron and A. Singh, "SplitStream: high-bandwidth multicast in cooperative environment," in *Proc. ACM SOS'03*, New York, USA, Oct. 2003.
- [11] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: high bandwidth data dissemination using an overlay mesh," in *Proc. ACM SOS'03*, New York, USA, Oct. 2003.
- [12] R. Rejaie and A. Ortega, "PALS: peer to peer adaptive layered streaming," in *Proc. NOSSDAV'03*, Monterey, CA, USA, Jun. 2003.
- [13] S. Banerjee, B. Bhattacharjee, and A. Srinivasan, "Resilient multicast using overlays," in *Proc. ACM SIGMETRICS'03*, San Diego, CA, USA, Jun. 2003.
- [14] J. Liu, B. Li, and Y.-Q. Zhang, "Adaptive video multicast over the Internet," *IEEE Multimedia*, vol. 10, no. 1, pp. 22-31, Jan./Feb. 2003.

- [15] P. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, “From epidemics to distributed computing,” in *IEEE Computer*, 37(5), pages 60-67, May 2004.
- [16] A.-M. Kermarrec, L. Massoulié, and A.J. Ganesh, “Probabilistic reliable dissemination in large-scale systems,” in *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 3, pp. 248-258, Mar. 2003.
- [17] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, “CoolStreaming/DONet: a data-driven overlay network for live media streaming,” to appear in *Proc. IEEE INFOCOM’05*, Miami, FL, USA, March 2005.
- [18] Y.-H. Chu, A. Ganjam, T.S. E. Ng, S.G. Guo, K. Sripanidkulchai, J. Zhan, and H. Zhang, “Early deployment experience with an overlay based Internet broadcasting system,” in *Proc. USENIX Annual Technical Conference*, Jun. 2004.
- [19] S. Jin and A. Bestavros, “Cache-and-relay streaming media delivery for asynchronous clients,” in *Proc. International Workshop on Networked Group Communication(NGC’02)*, Boston, MA, USA, Oct. 2002.
- [20] Y. Cui, B. Li, and K. Nahrstedt, “oStream: asynchronous streaming multicast,” in *IEEE J. Select. Areas in Comm*, Jan. 2004.
- [21] Y. Guo, K. Suh, J. Kurose, and D. Towsley, “P2Cast: peer-to-peer patching scheme for VoD service,” in *Proc. WWW’03*, Budapest, Hungary, May 2003.
- [22] T. Do, K. A. Hua, and M. Tantaoui, “P2VoD: providing fault tolerant video-on-demand streaming in peer-to-peer environment,” in *Proc. IEEE ICC’04*, Paris, France, Jun. 2004.
- [23] Y. Cui and K. Nahrstedt, “Layered peer-to-peer streaming,” in *Proc. NOSSDAV’03*, Monterey, CA, USA, Jun., 2003.
- [24] Y. Guo, K. Suh, J. Kurose, and D. Towsley, “A peer-to-peer on-demand streaming service and its performance Evaluation,” in *Proc. IEEE ICME’03*, Baltimore, MD, July 2003.
- [25] M. Hefeeda, B. Bhargava, “On-demand media streaming over the Internet,” in *Proc. IEEE FTDCS’03*, San Juan, Puerto Rico, May, 2003.
- [26] E. Zegura, K. Calvert, and S. Bhattacharjee, “How to model an internetwork,” in *Proc. IEEE INFOCOMM*, San Francisco, CA, USA, Mar. 1996.
- [27] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, Second Edition, MIT press, Cambridge, MA, 2001.
- [28] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, and J.D. Kubiawicz, “Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination,” in *Proc. NOSSDAV’01*, New York, Jun. 2001.
- [29] L. Guo, S. Chen, S. Ren, X. Chen, and S. Jiang, “PROP: a scalable and reliable P2P assisted proxy streaming system,” in *Proc. IEEE ICDCS’04*, Tokyo, Japan, Mar. 2004.
- [30] S. Shen, K. Hua, W. Tavanapong, “Chaining: a generalized batching technique for video-on-demand systems,” in *Proc. IEEE ICMCS’97*, Ottawa, ON, Canada, Jun. 1997.