Proxy-based Asynchronous Multicast for Efficient On-demand Media Distribution

Yi Cui, Klara Nahrstedt[a]

[a]Department of Computer Science, University of Illinois at Urbana-Champaign, USA

ABSTRACT

To achieve scalable and efficient on-demand media distribution, existing solutions mainly make use of multicast as underlying data delivery support. However, due to the intrinsic conflict between the synchronous multicast transmission and the asynchronous nature of on-demand media delivery, these solutions either suffer from large playback delay or require clients to be capable of receiving multiple streams simultaneously and buffering large amount of data. Moreover, the limited and slow deployment of IP multicast hinders their application on the Internet.

To address these problems, we propose asynchronous multicast, which is able to directly support on-demand data delivery. Asynchronous multicast is an application level solution. When it is deployed on a proxy network, stable and scalable media distribution can be achieved. In this paper, we focus on the problem of efficient media distribution. We first propose a temporal dependency model to formalize the temporal relations among asynchronous media requests. Based on this model, we propose the concept of Media Distribution Graph (MDG), which represents the dependencies among all asynchronous requests in the proxy network. Then we formulate the problem of efficient media distribution as finding Media Distribution Tree (MDT), which is the minimal spanning tree on MDG. Finally, we present our algorithm for MDT construction/maintenance. Through theoretical analysis and experimental study, we claim that our solution can meet the goals of scalability, efficiency and low access latency at the same time.

Keywords: on-demand media distribution, asynchronous multicast, proxy

1. INTRODUCTION

On-demand media distribution has gained intensive consideration recently due to its promising usage in a rich set of Internet-based services such as media distribution, distance education, video on demand, etc. The primary challenge here is to enable efficient, scalable and on-demand media delivery. By *efficient*, we mean that the system should make minimum use of media server and network resources. By *scalable*, we mean that the media distribution system should accommodate a large number of clients dispersed in the wide area. By *on-demand*, we mean that the client is free to access any portion of the media stream (e.g., fast forwarding and rewind in VoD) and the access latency must be bounded by a small delay time. An ideal media distribution system should meet all the three requirements.

To achieve the goal of efficiency, a general approach is multicast. The nature of multicast is synchronous, i.e., data is sent to all receivers *simultaneously.* However, on-demand media distribution requires data to be delivered in an *asynchronous* manner, i.e., users want to receive data at different times. To apply multicast in the context of on-demand media distribution, different solutions have been proposed[12345].[6] In batching,[1] asynchronous requests are aggregated into one multicast session to save server and network resources. However, the users have to suffer long playback delay since their requests are enforced to be synchronized. Patching[23] tries to address this problem by allowing the client to “catch up” with an on-going multicast session and patch the missing starting portion through server unicast. In merging,[4] a client can repeatedly merge into larger and larger multicast sessions using the same way as patching. However, in order to ensure smooth playback, these two approaches need the client to be capable of receiving multiple streams simultaneously and buffering large amount of data. In periodic broadcasting[5],[6] the server separates a media stream into segments and periodically broadcasts them through different multicast channels, from which a client chooses to join. Although more efficient in saving server bandwidth, it shares the same limitations as previous approaches.

Furthermore, all these solutions assume that users always request a stream from its beginning. How the existing techniques will perform, if the user is allowed to access a random portion of the stream, is still an open question. We argue that all these limitations originate from the intrinsic conflict between the synchronous multicast transmission and the asynchronous requirement of on-demand media delivery.

Besides the above limitations, these solutions face another problem when deployed in the real world: they all assume the multicast support from underlying network. However, the deployment of IP multicast in the Internet has been slow and severely limited. Recently, application-layer multicast[78] has been proposed to compensate for the absence of IP multicast. The main idea is to build an overlay network among end hosts. To enable multicast functionalities, end hosts cooperate with each other to relay data. Current research in this area has not considered applying application-layer multicast in the context of on-demand media distribution yet. Even if this approach is feasible, the intrinsic conflict between multicast and on-demand delivery still remain unresolved.

In fact, the rise of application-layer overlay gives us a chance to avoid the above conflict by altering the nature of multicast, which is unachievable in the context of IP multicast. To this end, we propose a new mechanism, *Asynchronous Multicast*, to directly support on-demand data delivery. Operating on application-layer overlay network, asynchronous multicast relies on peer end hosts to relay data for each other. The key to *asynchronous data transmission* is *cooperative buffering.* By enabling data buffering on the relaying nodes, requests at different times can be satisfied by the same stream, achieving efficient asynchronous media delivery. While impossible in IP multicast since routers can buffer very limited amount of data, it can be realized on the application layer, where relaying nodes are end hosts or proxies with strong buffering capabilities.

Asynchronous multicast is superior to the existing media distribution techniques in the following aspects: (1) With the aid of buffering, clients with asynchronous requests can reuse the same media stream. (2) The client request is immediately satisfied by the intermediate node, which keeps the requested data in its buffer. (3) The client is not required to receive multiple streams simultaneously and buffer them to ensure smooth playback. Instead, it only receives one stream as requested. (4) The client can request a stream from any starting point, instead of always from beginning. (5) It is an application-layer solution which requires no change to the underlying network.

In this paper, we present a proxy-based overlay network to support asynchronous multicast. In particular, we design and evaluate an on-demand media distribution system on top of this proxy overlay network. Our solution is summarized as follows. (1) We propose a *temporal dependency model* to model temporal relations among asynchronous requests. We reveal that for two asynchronous requests R_1 and R_2 for the same media stream, if R_1 receives data earlier than R_2, then R_1 has the potential to benefit R_2 , in that R_2 could reuse the same stream from R_1. (2) We deploy our model in a proxy-based framework. In this framework, proxies serve as the intermediate buffering nodes in media delivery. (3) Under the temporal dependency model, we formulate the problem of efficient on-demand media distribution into a graph problem. In particular, we propose two concepts: (a) *Media Distribution Graph* (MDG), which represents the dependencies among all asynchronous requests in the proxy network; (b) *Media Distribution Tree* (MDT), which is a spanning tree on MDG. The optimal solution for efficient data distribution is to construct and maintain the optimal MDT for a given MDG, which is a *Minimal Spanning Tree.* (4) We further refine this problem into two subproblems: (a) how to construct and maintain the optimal MDT; (b) how to acquire necessary knowledge, which will facilitate the construction/maintenance of MDT. For the first problem, we present a distributed algorithm for MDT construction/maintenance. The algorithm is proved to be correct and optimal. For the second problem, we design an efficient content discovery scheme to facilitate the acquisition of necessary knowledge information. Through theoretical analysis and experimental study, we claim that our solution can meet the *scalable*, *efficient* and *on-demand* requirements at the same time.

The rest of this paper is organized as follows. Section 2 introduces the temporal dependency model and presents its deployment framework. Section 3 formulates the problem. Section 4 presents the algorithm for MDT construction/maintenance and content discovery. Section 5 presents the performance evaluation. Section 6 discusses the related work. Section 7 concludes the paper.

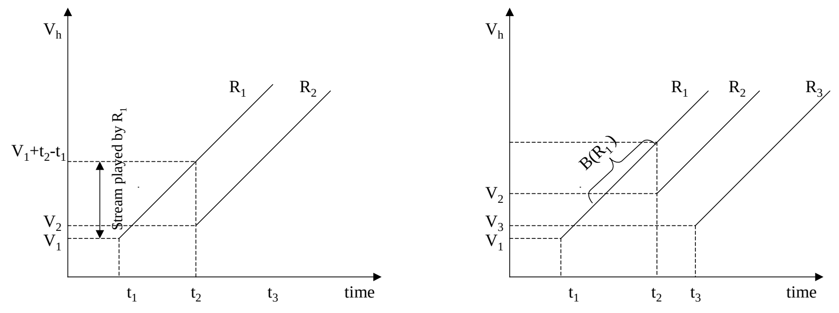
2. MODEL

2.1. Temporal Dependency Model

Consider two asynchronous requests R_i and R_j for the same video file*. The time length of the video file is V_h. R_i happens at time t_i and requests the video starting at the offset V_i. R_j happens at time t_j and requests the video starting at the offset V_j $(0 \leq V_i, V_j < V_h)$. We introduce the following definition to model their temporal dependency.

Definition 1: Given two requests R_i and R_j. If $t_i - V_i < t_j - V_j$, R_i is the *predecessor* of R_j, R_j is the *successor* of R_i, denoted as $R_i \prec R_j$.

As an example, in Figure 1 (a), $R_1 \prec R_2$. From the definition we can see, R_1 has the potential to benefit R_2 in that, R_2 could reuse the stream from R_1 instead of getting another one directly from the video server. To achieve this, we need to let the playback stream for R_1 be buffered. Clearly, the larger the buffer size, the more requests R_1 can possibly benefit. However, in practice, the buffer size needs to be bounded. Thus, only requests that fall into the buffer could be really benefited. To model such "constrained temporal dependency", we introduce the following definition:



(a) Basic Temporal Dependency (b) Buffer-based Temporal Dependency

Figure 1. Temporal Dependency Model

Definition 2: Given two requests R_i and R_j so that $R_i \prec R_j$. Let $B(R_i)$ be the buffer allocated for R_i, and $Size(B(R_i))$ be the time length of $B(R_i)$. If $t_i - V_i + Size(B(R_i)) \geq t_j - V_j$, R_i is the constrained predecessor of R_j, R_j is the constrained successor of R_i, denoted as $R_i \overset{B(R_i)}{\prec} R_j$.

As shown in Figure 1 (b), only $R_1 \overset{B(R_1)}{\prec} R_2$. So R_2 can get the stream from the host where the buffer for R_1 is located. By doing so, we can reduce the server load and save the network transmission cost (if the cost of streaming data from R_1 to R_2 is less than from the server to R_2). V_3 falls out of the buffer range, thus can not benefit from R_1. Note that our model uses the circular buffer to cache the stream. It means that any data cached in $B(R_1)$ is kept for a time length of $Size(B(R_1))$, after which it is replaced by the new data. In other words, a window of size $Size(B(R_1))$ is kept along the video playback for R_1, and all requests fall within this window can be served. Furthermore, the buffer is discarded when the streaming is over. This distinguishes our model from other prefix-caching[9] or segment-caching[10][11] schemes, where the buffer content is permanent and fixed. In this paper, we only consider buffer with fixed size†. Thus, we can simply denote $b = Size(B(R_i))$.

2.2. Model Deployment: Cooperative Proxy Overlay Network

Our temporal dependency model could be deployed in two types of frameworks: client-based or proxy-based. In client-based framework, each client buffers the stream locally and serves other clients using its buffered data. In this way, a media distribution overlay network is formed among cooperative clients. However, in practice,

*We use video distribution as an example to illustrate our framework, which can be applied to any type of streaming-based multimedia content. In the rest of this paper, we use terms video and multimedia content interchangeably.

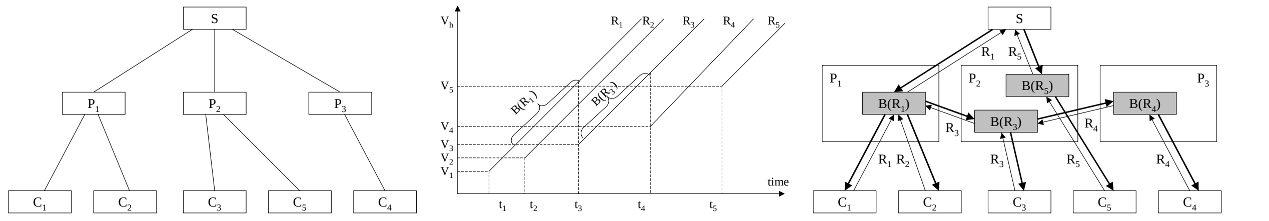
†The buffer size is a design parameter. We evaluate its impact in our simulation study.

the client-based framework is not suitable for large-scale media distribution for the following reasons: First, maintaining an overlay network scalable enough for a large number of client nodes is a difficult task; Second, the clients' heterogeneity and highly unpredictable nature (machine crash, constant leaving/joining the network) will complicate group management and make streaming between clients highly unstable.

In the proxy-based framework, proxies buffer data *on behalf of* clients. Currently in the Internet, proxies are often deployed at the head-end of the local network, serving clients within the same domain. Obviously, proxies have stronger network connectivity and buffering capabilities than clients. Their stability and availability are also much better than clients. Thus, we consider this type of framework in the paper. In a proxy-based framework, clients from different domains are organized into non-overlapping groups. Each group of clients is uniquely assigned to a proxy. Once a client issues a stream request, it is first intercepted by its proxy. If the requested data is available on the proxy, it is streamed to the client directly. Otherwise, the proxy requests it from the server or *other proxies*, then relays the incoming data to the client. At the same time, proxy buffers the playback stream on behalf of the client. The client is merely a consumer and does not buffer any data. Thus, an overlay network is formed among proxies, which cooperate with each other in media delivery. We refer this network as *cooperative proxy overlay network* in this paper.

We use a sample scenario to illustrate the on-demand media streaming in our cooperative proxy overlay network. Let us consider a simple server-proxy-client hierarchy shown in Figure 2 (a). S is the server, P_1, P_2 and P_3 are proxies serving client groups $\{C_1, C_2\}$, $\{C_3, C_5\}$ and $\{C_4\}$ respectively. The temporal dependencies among different requests are shown in Figure 2 (b). R_1 from client C_1 is intercepted by P_1. P_1 forwards R_1 to S and then relays the stream from S to C_1. P_1 also caches the stream in its local buffer $B(R_1)$ *on behalf* of C_1. Since $R_1 \overset{B(R_1)}{\prec} R_2$, P_1 can directly send data to C_2 from its local buffer $B(R_1)$, when C_2 issues R_2 to it. When R_3 from C_3 is intercepted by P_2, P_2 issues a request among proxies and servers for R_3. Since $R_1 \overset{B(R_1)}{\prec} R_3$, P_2 can retrieve the stream from $B(R_1)$ at P_1. Upon forwarding the stream to C_3, P_2 also allocates a new buffer $B(R_3)$. Similarly, R_4 from C_4 is intercepted by P_3, then satisfied by $B(R_3)$ since $R_3 \overset{B(R_3)}{\prec} R_4$. When R_5 is intercepted by P_2, P_2 issues a new request, since its existing buffer $B(R_3)$ can not satisfy R_5. This request is directly served by S since R_5 cannot benefit from any existing buffers among the proxy network. From the picture, we can see there are two types of requests: *intra-proxy request* (or *client request*) and *inter-proxy request*(or *proxy request*). If an intra-proxy request can be directly served by a local buffer of its proxy, the request is masked, such as R_2. Otherwise, an inter-proxy request is triggered, such as R_1, R_3, R_4 and R_5.

In this paper, we mainly study the inter-proxy requests problem, since client requests are resolved locally. To enforce efficient media distribution, we consider the following questions: (1) Upon a proxy request, how does a proxy know which proxies buffer the stream it demands? (2) how does a proxy choose the "best buffer" to retrieve the stream from, if there exist multiple ones which can satisfy its request? These problems are formulated in the next section.



(a) Server-Proxy-Client Hierarchy (b) Request Temporal Dependencies (c) Data and Request Flow

Figure 2. Sample Scenario

3. PROBLEM FORMULATION

3.1. Temporal Dependency between Proxy Buffers

As revealed in Section 2, each proxy request triggers the allocation of a proxy buffer. To facilitate the problem formulation, we use the temporal dependency between two buffers to represent the dependency between their corresponding requests.

Definition 3: Given two proxy requests R_i and R_j, and their buffers $B(R_i)$ and $B(R_j)$. If $R_i \overset{B(R_i)}{\prec} R_j$, then $B(R_i)$ is the predecessor of $B(R_j)$, $B(R_j)$ is the successor of $B(R_i)$, denoted as $B(R_i) \prec B(R_j)$, or simply $B_i \prec B_j$.

3.2. Media Distribution Graph

Let $\mathbb{B}$ be the set which includes all proxy buffers. We define the *Media Distribution Graph* (MDG) for $\mathbb{B}$ as a directed weighted graph $MDG_{\mathbb{B}} = (\mathbb{B}, E)$, such that $E = \{(B_i, B_j) \mid B_i \prec B_j, B_i, B_j \in \mathbb{B}\}$. Each edge (B_i, B_j) has the weight $w(B_i, B_j)$, which is the transmission cost between two proxies carrying B_i and B_j respectively. As shown in Figure 3, in MDG, each node represents a buffer[‡]. An edge directed from node B_1 to B_3 means that B_1 is the predecessor of B_3, i.e., B_3 could reuse the media stream from B_1. $w(B_1, B_3)$ is the transmission cost between proxies P_1 and P_2, which hold B_1 and B_3 respectively. A special node is B_{server} representing the server. Since the server can serve any proxy request, it is the predecessor of all other buffers. Thus, B_{server} has directed edges to all other nodes in the graph. MDG is changing dynamically: (1) a new buffer is allocated when a new request arrives, thus a new node is inserted into the graph; (2) a buffer is discarded when its streaming session is over, thus the corresponding node is removed from the graph. In summary, MDG represents the dependencies among all asynchronous requests and their resulting buffers in the proxy network, which forms a virtual overlay on top of proxy overlay network. The weight of edges in MDG reflects the communication cost from the physical proxy networks.

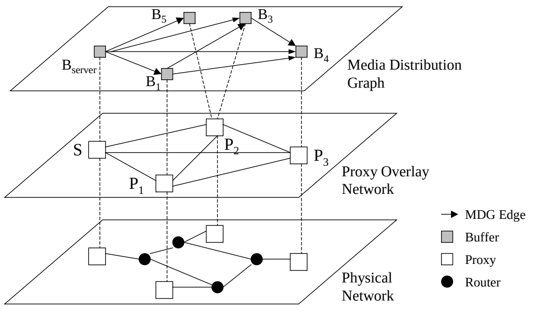


Figure 3. Illustration of Media Distribution Graph

3.3. Media Distribution Tree

Given the definition of MDG, the efficient media distribution problem can be formulated as constructing and maintaining a spanning tree on the MDG, defined as *Media Distribution Tree.* Formally, for a buffer set $\mathbb{B}$ and its MDG $MDG_{\mathbb{B}} = (\mathbb{B}, E)$, the corresponding MDT is denoted as $MDT_{\mathbb{B}} = (\mathbb{B}, E^T)(E^T \in E)$. For two nodes $B_i, B_j \in \mathbb{B}$, if $(B_i, B_j) \in E^T$, then B_i is the *parent* of B_j, B_j is the *child* of B_i. Given a MDG, the optimal solution for MDT, i.e., to minimize the overall transmission cost of media distribution, is to find the *Minimal Spanning Tree* (MST) on MDG.

Note that existing algorithms for MST can not be directly applied to our problem for the following reasons. First, our solution does not assume the existence of centralized manager, which has the global knowledge and controls the tree construction. Instead, the tree needs to be constructed in a fully distributed manner. Second, since MDG is constantly updated, the corresponding MDT maintenance must be incremental. Frequent tree

[‡]We use the terms buffer and node interchangeably in the remainder of this paper, depending on the context.

reorganization will incur unacceptable overhead. As will be presented in Section 4, our MDT algorithm satisfies the above requirements.

4. EFFICIENT MEDIA DISTRIBUTION

In this section, we present our solution on MDT construction/maintenance. We present the algorithm in Section 4.1, its analysis in Section 4.2, and its distributed implementation in Section 4.3. Given a MDG, the algorithm is able to constantly maintain its MDT as the graph is dynamically updated. To achieve so, the algorithm requires that each node in the MDG have the up-to-date knowledge about its neighbors. Section 4.4 introduces a content discovery scheme, which facilitates the acquisition of such knowledge.

4.1. Algorithm

Our algorithm makes the following assumptions. (1) For a given buffer set $\mathbb{B}$ and its MDG $MDG_{\mathbb{B}}$, each node in $MDG_{\mathbb{B}}$ has the knowledge of its in-bound and out-bound neighbors. (2) We assume that for any edge $(B_i, B_j) \in E$, $w(B_i, B_j)$ is known to B_i and B_j.

Our algorithm has the following properties: (1) It is fully distributed. The MDT construction/maintenance is achieved by local nodes. No centralized manager is required. (2) The MDT construction/maintenance is incremental. In case of node join/leave, only a portion of the tree is affected. No global tree reorganization is needed. (3) The algorithm guarantees to return the optimal result, as will be proved in Section 4.2.

The algorithm is executed each time when a new node joins the graph, or when an old node leaves. To deal with these two cases, our algorithm has two operations: **MDT-Insert** and **MDT-Delete**. Actually, MDT can be constructed incrementally by repeating **MDT-Insert**.

We first consider the case of node insertion. Let $MDG_{insert} = (\mathbb{B} \cup B_{insert}, E_{insert})$ be the resulting graph after B_{insert} and its inducing edges are added to $MDG_{\mathbb{B}}$, **MDT-Insert** is able to return $MDT_{insert} = (B \cup B_{insert}, E^T_{insert})$ as the new MDT for MDG_{insert}.

```
MDT-Insert($B_{insert}, E_{insert}, E^T, E^T_{insert}$)
   /* From all its predecessors, $B_{insert}$ finds parent $B_{min}$, whose transmission cost to $B_{insert}$ is minimal */
1  $P(B_{insert}) \leftarrow \{B_{pred} \mid (B_{pred}, B_{insert}) \in E_{insert}\}$
2  $w_{min} \leftarrow \min\{w(B_{pred}, B_{insert}) \mid B_{pred} \in P(B_{insert})\}$
3  $E^T_{insert} \leftarrow E^T \cup \{(B_{min}, B_{insert}) \mid w(B_{min}, B_{insert}) = w_{min}\}$

   /* For all its successors $B_{succ}$, $B_{insert}$ compares if the transmission cost from itself to $B_{succ}$ is less than
   from $B_{succ}$'s current parent $B_{parent}$. If so, $B_{succ}$ is asked to switch parent to $B_{insert}$. */
4  for each $B_{succ} \in \{B_{succ} \mid (B_{insert}, B_{succ}) \in E_{insert}\}$
5    if $w(B_{insert}, B_{succ}) < w(B_{parent}, B_{succ}) \in E^T$
6      do $E^T_{insert} \leftarrow (E^T_{insert} - (B_{parent}, B_{succ})) \cup \{(B_{insert}, B_{succ})\}$
```

Second, we discuss the case of node leaves. Let $MDG_{delete} = (B - B_{delete}, E_{delete})$ be the resulting graph after B_{delete} and its inducing edges are removed from MDG_B, **MDT-Delete** is able to return $MDT_{delete} = (B - B_{delete}, E^T_{delete})$ as the new MDT for MDG_{delete}.

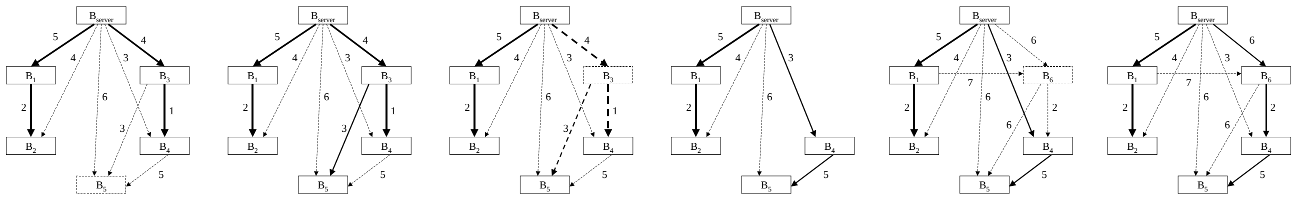
We use an example to illustrate the MDT algorithm. Initially, we have a buffer set $\mathbb{B} = \{B_{server}, B_1, B_2, B_3, B_4\}$ and the corresponding MDT. B_5 first arrives, which runs **MDT-Insert**. Then B_3 leaves the MDT, which runs **MDT-Delete**. Finally, B_6 is inserted, which runs **MDT-Insert**. The whole process is illustrated in Figure 4.

4.2. Analysis

We prove the correctness and optimality of **MDT-Insert** and **MDT-Delete** as follows. The proofs for **Lemma 1**, **Theorem 1** and **Theorem 2** are provided in our technical report.[12]

Lemma 1: MDG is Directed Acyclic Graph (DAG).

```
**MDT-Delete**$(B_{delete}, E_{delete}, E^T, E^T_{delete})$
    /* $B_{delete}$ deletes the tree edge from its parent $B_{parent}$ */
1   $E^T_{delete} \leftarrow E^T - \{(B_{parent}, B_{delete}) \mid (B_{parent}, B_{delete}) \in E^T\}$
2   **for each** $B_{child} \in \{B_{child} \mid (B_{delete}, B_{child}) \in E^T\}$
    **do**
        /* $B_{delete}$ deletes the tree edge to each of its children $B_{child}$ */
3       $E^T_{delete} \leftarrow E^T_{delete} - (B_{delete}, B_{child})$
        /* From all its predecessors, $B_{child}$ finds the new parent $B_{min}$, whose transmission cost to $B_{child}$ is
        minimal */
4       $P(B_{child}) \leftarrow \{B_{pred} \mid (B_{pred}, B_{child}) \in E_{delete}\}$
5       $w_{min} \leftarrow \min\{w(B_{pred}, B_{child}) \mid B_{pred} \in P(B_{child})\}$
6       $E^T_{delete} \leftarrow E^T_{delete} \cup \{(B_{min}, B_child) \mid w(B_{min}, B_{child}) = w_{min}\}$
```



(a) Before insertion of B_5 (b) After insertion of B_5 (c) Before deletion of B_3 (d)After deletion of B_3 (e) Before insertion of B_6 (f) After insertion of B_6

Figure 4. A example illustrating the MDT Algorithms

Theorem 1: **MDT-Insert** and **MDT-Delete** are guaranteed to return loop-free spanning trees. (*Correctness*)

Theorem 2: The trees returned by **MDT-Insert** and **MDT-Delete** are MSTs. (*Optimality*)

Note that the optimality of our algorithm is based on the assumption that for any edge $(B_i, B_j) \in E$, $w(B_i, B_j)$ does not change during the streaming session between B_i and B_j. It means that the transmission cost between the proxies carrying B_i and B_j is stable, which is impossible in realistic problem settings. Therefore, network monitoring service has to be enforced on each proxy to periodically measure the transmission cost from itself to other proxies. We argue that although the optimality of our algorithm is compromised in practice, the goal of efficient media distribution can still be achieved, given the fact that the network route between two proxies is stable or changed slowly. For the same reason, the monitoring service can operate at low frequencies or remain inactive until a streaming request comes.

4.3. Distributed Implementation

In practice, our algorithm is implemented in a fully-distributed fashion, i.e., the MDT construction/maintenance decision is made locally by each buffer.

Information Required

Each buffer $B_i \in \mathbb{B}$ carries the following information: its own parent $Parent(B_i)$ and its child list $Childlist(B_i)$. B_i also needs to know the transmission cost between itself and any other buffer. This can be achieved by asking the proxy carrying B_i to keep track of its transmission cost to all other proxies, as explained in Section 4.2. Finally, B_i needs to acquire the knowledge of its neighbors in the MDG, namely its predecessors and successors. To achieve so, *Content Discovery Service* is required. The service collects information about each buffer in the MDG and answers queries. Here we define two types of functions: *query functions* and *update functions*. Query functions include the following:

1. $GetPredecessors(B_i)$: called by B_i, the function returns all predecessors of B_i.

2. $GetSuccessors(B_i)$: called by B_i, the function returns all successors of B_i.

The update functions include the following:

1. $Register(B_i)$: B_i calls this function to register itself to the discovery service when it is newly allocated. B_i provides information such as its residing proxy, its parent $Parent(B_i)$ and the transmission cost from $Parent(B_i)$ to B_i.
2. $Deregister(B_i)$: B_i calls this function to remove itself from the discovery service when it is discarded.

The implementation of the content discovery service and the above functions will be presented in Section 4.4.

Message Exchange

During the process of **MDT-Delete**, first, B_{delete} sends out *child-leave* message to its parent $Parent(B_{delete})$. On receiving the message, $Parent(B_{delete})$ stops streaming to B_{delete} and removes it from its child list. Second, B_{delete} sends out *parent-leave* message to each buffer in $Childlist(B_{delete})$ and stops streaming to it. Then B_{delete} calls $Deregister(B_{delete})$ and leaves. For each child B_{child}, on receiving the message, it first calls the function $GetPredecessors(B_{child})$. After receiving a list of buffers which are qualified as its parent, B_{child} compares the transmission cost from each of them to itself and finds out the one with minimal cost: B_{min}. Finally, B_{child} sends out a *child-join* message to B_{min}. Upon receiving the message, B_{min} adds B_{child} into its child list and start streaming to B_{child}. Upon receiving the stream, B_{child} sets B_{min} as its new parent and calls $Register(B_{child})$. The whole process is finished after every B_{child} orphaned by B_{delete} has found its new parent.

During the process of **MDT-Insert**, first, B_{insert} finds its parent using the same way as B_{child} does in **MDT-Delete** and calls $Register(B_{insert})$. Then, B_{insert} calls the function $GetSuccessors(B_{insert})$ to find if any existing buffers are the successors of B_{insert}. This case may happen as shown in Figure 5. Although B_{insert} is allocated later than B_{succ}, it is still the predecessor of B_{succ} since V_{inser} is ahead of V_{succ}. In this case, B_{insert} first compares if the transmission cost from itself to B_{succ} is smaller than the transmission cost from $Parent(B_{succ})$ to B_{succ} (obtained from the query function $GetSuccessors(B_{insert})$). If so, B_{insert} sends a *parent-join* message to notify B_{succ} to switch to a better parent. Note that as shown in Figure 5, this message is issued at time t_{insert}, when B_{insert} is initiated. However, B_{insert} cannot benefit B_{succ} immediately until the time t_{switch}. Therefore, B_{succ} waits until the time t_{switch} to send a *child-join* message to B_{insert}. B_{insert} then adds B_{succ} to its child list and starts streaming to B_{succ}. When receiving the stream, B_{succ} updates B_{insert} as its new parent, then calls $Register(B_{succ})$ to update its information to the discovery service.

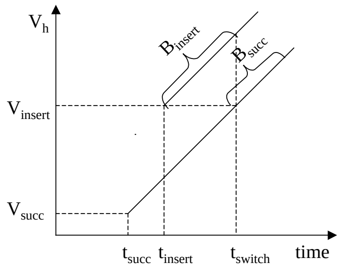


Figure 5. Temporal Dependency between B_{insert} and B_{succ}

4.4. Content Discovery Service

In Section 4.3, we stated the need for content discovery service to facilitate each buffer to find its successors or predecessors. Intuitively, we could use a centralized server to manage the information of all buffers, and accept all the update and query messages. However, this solution suffers from the drawbacks of all centralized approaches. Hierarchical static-content-based discovery solutions[13] do not fit into our case either for the following reasons. First, unlike other caching schemes, where the content of a buffer is fixed, we allow the content of a buffer to be time varying. Second, each buffer is associated with a lifetime. Therefore, events of buffer birth/death and buffer content changing will constantly invalidate the content availability information, which results into high volume of update messages.

To address these problems, we should leverage the temporal dependencies among different buffers, as defined in Section 3.1. Our solution is summarized as follows. (1) We use proxies as discovery servers. Each server has a unique ID. (2) For each buffer B_i trying to register to the discovery service by calling $Register(B_i)$, its call is directed to a subset of discovery servers, which will keep the record of B_i until B_i removes itself (by calling $Deregister(B_i)$). This subset is determined by mapping the timing information of B_i into a set of server IDs. (3) Likewise, if B_i tries to query its successors or predecessors by calling $GetSuccessors(B_i)$ or $GetPredecessors(B_i)$, its call is directed to a subset of discovery servers, which keep the records of all successors or predecessors of B_i. This subset is also determined through timing information mapping of B_i. (4) The subsets in (2) and (3) contain constant number of servers. Thus the message overhead for the above update or query operations is constantly bounded. Our solution makes the following assumptions. First, the size of each buffer is equal and fixed, which we simply denote as b throughout this section. Second, we assume the existence of uniform system time followed by all buffers accessing the same video. This can be achieved by any network synchronization solution.

Basic Model

Each buffer B_i is identified by the following information: the allocation time t_i, the starting offset V_i, and its size b. We use $N = \{h_o, h_1, \dots h_n - 1\}$ to denote all discovery servers (proxies). If B_i needs to register to the discovery service, it first calls the hashing function $r(B_i)$ to return a subset of discovery servers. Then, B_i sends update messages to these servers, which will keep its record. Similarly, function $p(B_i)$ returns a subset of servers, to which B_i sends the query message about its predecessors. Function $s(B_i)$ is designed the same way for successor queries.

Function $p(B_i)$ must ensure to return servers in N that contain the records of all predecessors of B_i. Similarly, function $s(B_i)$ must ensure to return servers in N that contain the records of all successors of B_i. Formally, we have

$$\begin{aligned} p(B_i) &\equiv \{n_k \mid n_k \in r(B_{pred}), B_{pred} \rightarrow B_i\} \\ s(B_i) &\equiv \{n_k \mid n_k \in r(B_{succ}), B_i \rightarrow B_{succ}\} \end{aligned}$$

Hashing Function Design

Let b be the buffer size, n be the number of discovery server, we design our hashing functions as below.

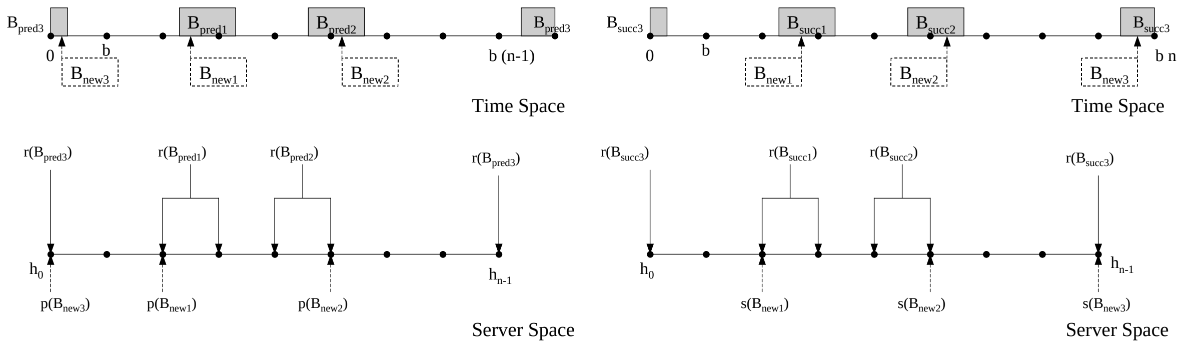
$$\begin{aligned} r(B_i) &= \{h_m, h_{(m+1) \bmod n}\} & (mb \leq (t_i - V_i) \bmod (bn) < (m+1)b) \\ p(B_i) &= \{h_m\} & (mb \leq (t_i - V_i) \bmod (bn) < (m+1)b) \\ s(B_i) &= \{h_m\} & (mb \leq (t_i - V_i + b) \bmod (bn) < (m+1)b) \end{aligned}$$

In Figure 6 (a), we illustrate the relationship between function r and p in three cases. In particular, B_{pred1} is the predecessor of the new buffer B_{new1}, B_{pred2} is the predecessor of the buffer B_{new2}, B_{pred3} is the predecessor of the buffer B_{new3}. In Figure 6 (b), we illustrate the relationship between function r and s in three cases. In particular, B_{succ1} is the successor of the new buffer B_{new1}, B_{succ2} is the successor of the buffer B_{new2}, B_{succ3} is the successor of the buffer B_{new3}.

From the picture we can see, the buffer update message is directed to two consecutive discovery servers, thus the buffer update overhead is 2. For both query operations (predecessors and successors), the message overhead is 1.

5. PERFORMANCE STUDY

In this section, we study the performance of our proxy-based cooperative media distribution system through extensive simulation.



(a) Finding Predecessors (b) Finding Successors

Figure 6. Illustration of Hashing Functions

5.1. Simulation Setup

We choose ns-2 as our simulation tool. The primary reason for our choice is that it is a packet-level simulator which is able to provide fine-grained traffic load information on network links. We use GT-ITM transit-stub model[14] to generate five network topologies as shown in Table 1. Figure 7 shows Topology A. One proxy is placed at the head-end of each stub domain. Other nodes in the same stub domain are its clients. Each proxy is attached to one transit domain. The server is a single-node stub domain connected to a transit node. In transit-stub model, stub domains are normally regarded as local networks, while transit domains are formed via interconnection of backbone routers. Therefore, we define the *server link* as the network link between the server and the transit node that it gets attached to. *Backbone links* are the links between different transit nodes in the transit domains. Communication between two proxies or between proxy and server is routed among backbone links. *Local links* are the links among the proxy and its clients in each stub domain.

Topology Name	Server	Proxies	Clients
Topology A	1	6	96
Topology B	1	12	192
Topology C	1	18	288
Topology D	1	24	384
Topology E	1	30	480

Table 1. Experimental Network Topologies

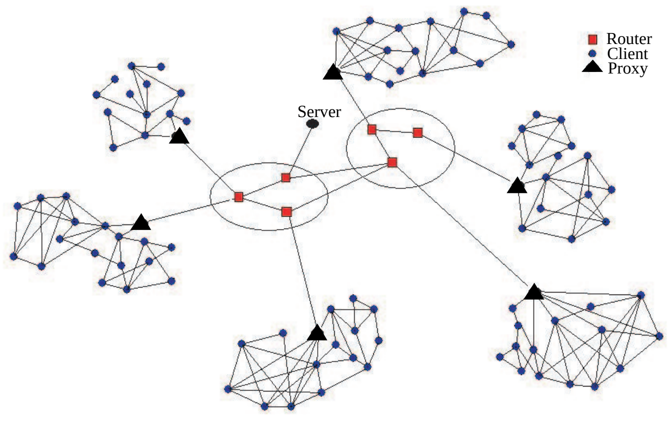


Figure 7. Topology A

We consider the case of a single CBR video distribution. The video file is 1 hour long with streaming rate of 400Kbps. During each 100-minute run of the simulation, the client requests are generated according to a Poisson process with different average request arrival rates. The session duration is exponentially distributed with an average of 30 minutes.

Throughout the simulation, we compare our scheme with two other media distribution techniques: batching[1] and proxy prefixing.[15] In batching, client requests within certain time interval are aggregated into one multicast tree rooted at the server. In proxy prefixing, each proxy stores the prefix portion of the video file in its local cache. Upon client request, the proxy first streams the cached prefix data to the client, then retrieves the remaining part of the video from the server and relays it to the client as late as possible. By this means, different client requests can be aggregated into one multicast stream, which is similar to batching. Among the three schemes, batching relies on IP multicast support, while the other two only needs unicast support.

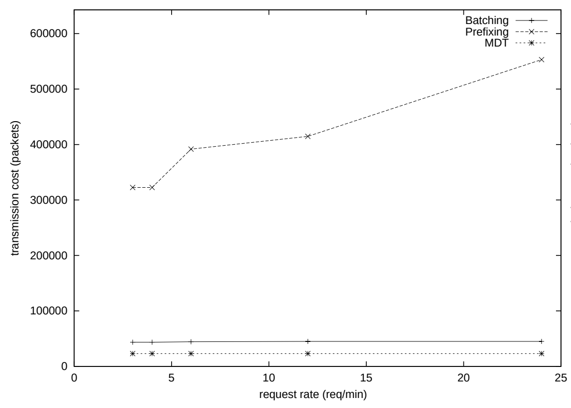
5.2. Simulation Results

In this subsection, we show representative results collected from our simulation run. The full-set results and their analysis are reported in.[12] Although our scheme allows clients to request the video at any point, batching and proxy prefixing always request the video from its beginning. For the purpose of fairness, we keep this assumption when comparing the communication cost and storage consumption of different schemes. However, when measuring the access latency of media delivery, we allow clients to request video from random starting point.

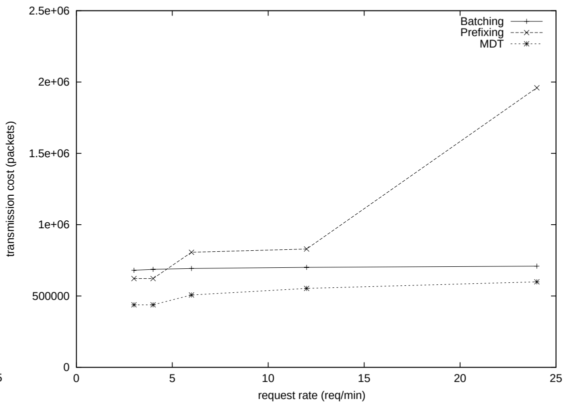
5.2.1. Communication Cost

In this set of experiments, we set the buffer size of our scheme to be 5 minutes long. The cache size of proxy prefixing is also 5 minutes. For batching, the batch interval is 15 minutes. In reality, it may cause unbearable delay for clients. However, our purpose here is to enforce a long enough interval, so that the transmission cost of batching is approximately on the same scale with our scheme. In fact, we use batching as the baseline to compare how close our scheme can perform as well as the multicast-based solutions.

Impact of Request Rate



(a) Server Communication Cost



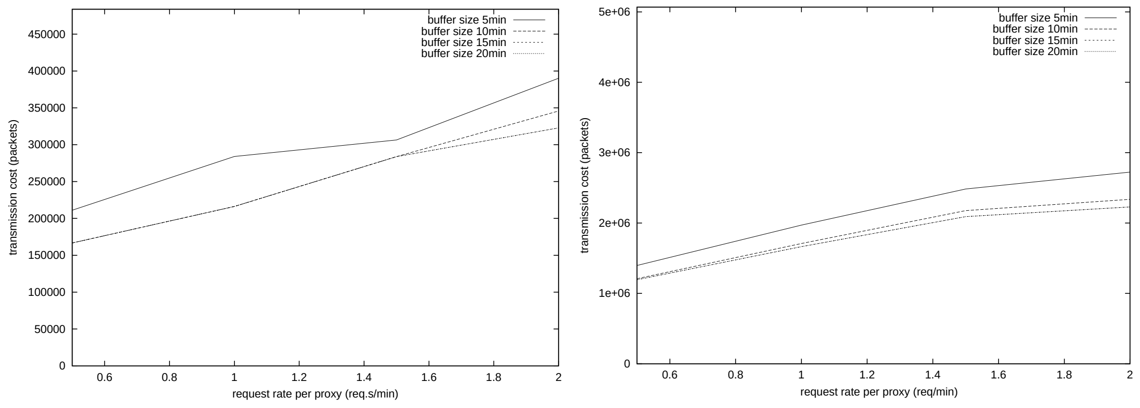
(b) Backbone Communication Cost

Figure 8. Communication Cost Comparison of Different Schemes (Topology B)

As shown in Figure 8 (a), the server communication cost of proxy prefixing does not scale well. The main reason is that all proxies request data from the server. When the request request rate grows, the server is overburdened. For batching, the server transmission is proportional to the number of multicast sessions it has opened. Our scheme outperforms batching. In fact, through the 100-minute run, the server only streams the data to the first requesting proxy. The proxy then relays the data to the next requesting proxy through buffering, and so on. Such proxy "chaining" continues unless the new proxy request cannot benefit from other proxies except server. This condition happens only when requests are far apart from each other, i.e., when the request rate is low. In Figure 8 (b), Our scheme outperforms batching in terms of the backbone communication cost. The reason is that with one media distribution tree, we can cover requests ranging in a long time interval. On the other hand, batching has to use several IP-multicast trees to serve asynchronous requests. Our finding suggests that although application-layer overlay inevitably introduces topological inefficiency compared to IP multicast, the benefit introduced by asynchronous multicast can offset such inefficiency.

Impact of Buffer Size

Figure 9 shows that both server and backbone communication cost drop significantly when the buffer size is increased from 5 minutes to 10 minutes. Then, the gain is trivial as we further increase the buffer size. Finally, the curve for buffer size of 20 minutes is identical with the one of 15 minutes. The reason is that, when buffer size is further increased, as shown in Figure 12, the hit ratio does not increase any more. Thus, the number of proxy requests can not be further reduced.



(a) Server Communication Cost (b) Backbone Communication Cost

Figure 9. Impact of Buffer Size (Topology B)

5.2.2. Storage Cost

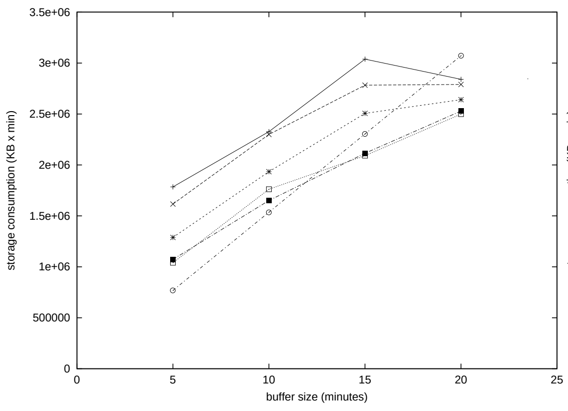
In this subsection, we compare our scheme with proxy prefixing about their efficiencies on proxy storage consumption cost. The buffer (prefix cache) size is ranged from 5 minutes to 20 minutes. Note that the cost is time-based in our scheme, where each buffer is associated with a lifetime. However, for proxy prefixing, the prefix cache allocation is static and permanent. To make it comparable to our scheme, we multiply the prefix cache size with the simulation run time, which is shown as straight lines in Figure 10. In proxy prefixing, the storage consumption grows linearly when increasing the buffer size. Our scheme, on the other hand, has a much slower growing speed. From Topology B to E, the curves of our scheme move down as a whole, which suggests better scalability as network size increases.

Also note that we do not consider the storage consumption at the client side. In proxy prefixing, the client is required to have the same caching capabilities with the proxy to ensure smooth playback. In our scheme, the client requires no buffer at all, since it only receives one stream from the proxy as requested.

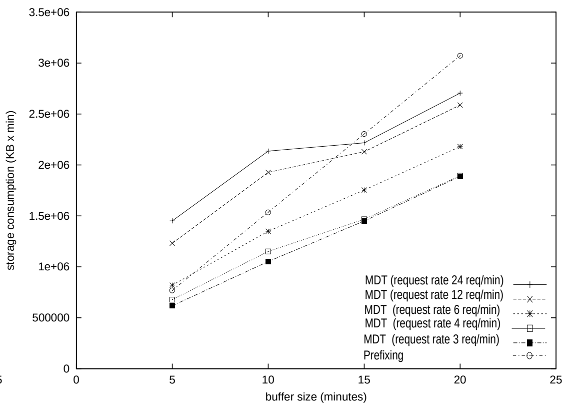
5.2.3. Access Latency

In this subsection, we evaluate the access latency. Among the three schemes, the latency of batching is far worse than the other two, where a client has to wait for at most 15 minutes. Therefore, we only compare our scheme with proxy prefixing, both of which provide true on-demand media delivery. We note that the comparison is not entirely fair. In proxy prefixing, we set the client requests to be always starting from the beginning. However, in our scheme, we allow clients to request data from any part of the video file.

As shown in Figure 11, the latency of our scheme is slightly higher than the proxy prefixing but bounded within 0.4 second. The picture also suggests that the buffer (prefix cache) size has no obvious impact to the end-to-end latency. Moreover, the average access latency drops when the request rate increases. The reason is

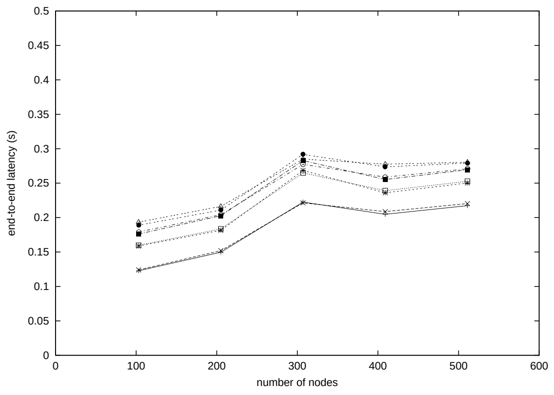


(a) Topology B

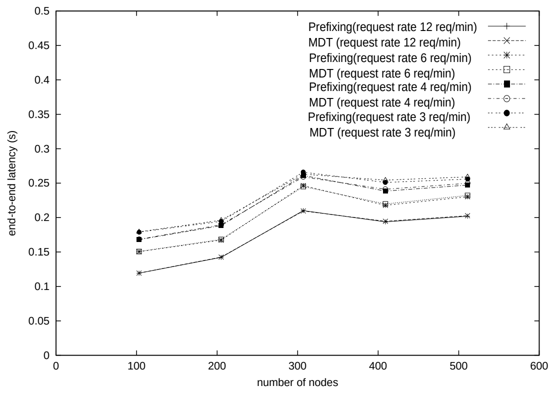


(b) Topology E

Figure 10. Timed-based Storage Consumption per Proxy under different Buffer Sizes



(a) Buffer Size 5 min



(b) Buffer Size 20 min

Figure 11. Access Latency

revealed in Figure 12. In this picture, the buffer hit ratio is defined as the percentage of client requests that are directly served by it proxy buffer. For other requests which can not be served by the proxy buffer, the proxy has to retrieve a new stream from the server or some other proxy, then relays the data to the client. Obviously, in this case the access latency will be longer. As shown in Figure 12, the buffer hit ratio grows when the client request rate increases. For the client request, its latency only involves the streaming delay from the proxy to the client. For the proxy request, it includes latencies of request interception and content discovery (performed by the proxy), and the delay of data relaying from the source to the client via the proxy.

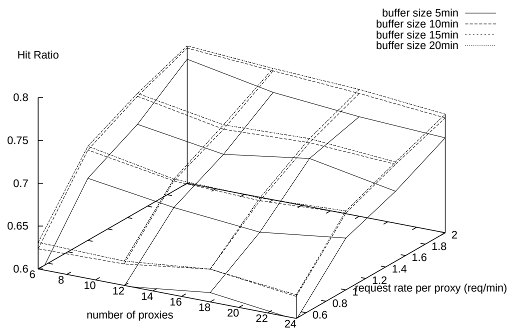


Figure 12. Buffer Hit Ratio

We summarize the latency distribution of proxy requests in Figure 13. As shown in the picture, streaming delay dominates the overall latency. Discovery latency and proxy latency only incur little additional delay.

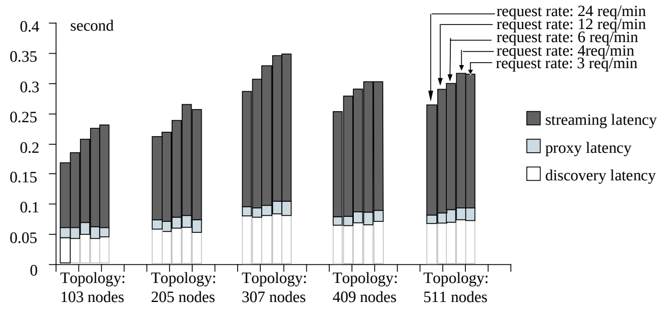


Figure 13. Access Latency Analysis (Buffer Size: 5 minutes)

6. RELATED WORK

Efficient media distribution problem has been extensively studied in the existing literature. We have discussed multicast-based solutions[5][6][1][2][3][4] in the beginning of this paper in details. Besides multicast, an orthogonal technique for reducing server loads, network traffic and access latencies is the cooperative proxy caching. Existing works in this area include prefix-based caching[16],[17] and segment-based caching[11].[10] These schemes also allow partial caching of a media object. However, the content of a segment cache remains static until it is replaced by a new one due to a cache hit miss. In our scheme, the buffer content keeps updating following the playback stream. Moreover, each buffer has a lifetime determined by the streaming session it belongs to. By using this "flowing buffer", a long-term temporal dependency can be established between two requests, e.g., a client can retrieve its data from one proxy buffer throughout the entire streaming session. On the other hand, in former schemes, a client may have to switch among proxies in order to stream from distributed caches. Despite their differences, these two types of caching/buffering schemes can be combined to further save the server and network load. For example, if a client request is unable to "catch up" with the latest "flowing proxy buffer", they can be bridged by a prefix segment already cached in the proxy, which avoids the need to start a new stream.

Before our work, the temporal locality of media distribution has been utilized in the context of VoD server architecture, such as interval caching[18] and chaining.[19] We extend the usage of temporal locality into content indexing and lookup service, which reduces signaling overhead and facilitates the efficient media distribution.

The notion of "proxy overlay network" has been proposed in[8],[20] etc. However, these works are targeted at other problems such as live media broadcasting or service path finding. Our problem is different from those problem in that: (1) our system aims at on-demand media delivery, while live media broadcasting is synchronous transmission; (2) Our design focuses on "media content", while[20] focuses on "media service". We argue that "media content" involves many fine-grained features of a media stream, such as temporal information. Our solution explores these features for efficient system design.

7. CONCLUSION

Efficient on-demand media distribution is challenging task. Existing multicast-based solutions suffer from various limitations mainly due to the intrinsic conflict between the synchronous data transmission manner of multicast and the asynchronous nature of on-demand media delivery. In this paper, we propose the concept of *Asynchronous Multicast*, which is able to resolve this conflict. Deployed in the proxy-based overlay network, asynchronous multicast is an application-layer solution. To model this new communication paradigm, we present a temporal dependency model to represent the temporal relations among asynchronous requests. Based on the model, we formulate the efficient media distribution problem into finding *Media Distribution Tree* inside the proxy network. We also present our MDT algorithm, which is able to constantly maintain the optimal MDT in a fully distributed fashion. Through theoretical analysis and experimental study, we claim that our solution can meet the goals of scalability, efficiency and low access latency at the same time.

ACKNOWLEDGMENTS

We would like to thank anonymous reviews and Dr. Milind Buddhikot, our shepherd, for their detailed and constructive comments which have helped us improve the final version of this paper. This work was supported by the National Science Foundation under contract number 9870736, the Air Force Grant under contract number F30602-97-2-0121, National Science Foundation Career Grant under contract number NSF CCR 96-23867, NSF CISE Infrastructure grant under contract number NSF EIA 99-72884, and NASA grant under contract number NASA NAG 2-1250. Views and conclusions of this paper are those of authors, which should not be interpreted as representing the official policies, either expressed or implied, of the funding agencies.

REFERENCES

1. C.C. Aggarwal, J.L. Wolf and P.S. Yu, "On Optimal Batching Policies for Video-on-Demand Storage Servers," in *IEEE International Conference on Multimedia Computing and Systems (ICMCS '96)*, 1996.
2. K.A. Hua, Y. Cai and S. Sheu, "Patching: A Multicast Technique for True On-Demand Services," in *ACM Multimedia '98*, 1998.
3. Y. Cai, K. Hua and K. Vu, "Optimzed Patching Performance," in *ACM/SPIE Multimedia Computing and Networking (MMCN '99)*, 1999.
4. Derek Eager, Mary Vernon and John Zahorjan, "Minimizing Bandwidth Requirements for On-Demand Data Delivery," *IEEE Transactions on Knowledge and Data Engineering* **13**(5), 2001.
5. K.A. Hua and S. Sheu, "Skyscraper Broadcasting: A new Broadcasting Scheme for Metropolitan VOD systems," in *ACM SIGCOMM '97*, 1997.
6. S. Viswanathan and T. Imielinski, "Metropolitan Area Video-on-Demand Service using Pyramid Broadcasting," *Multimedia Systems* **4**, 1996.
7. Yang-hua Chu, Sanjay G. Rao and Hui Zhang, "A Case for End System Multicast," in *ACM SIGMETRICS '00*, 2000.
8. Y. Chawathe, "Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service," *PhD. Dissertation, University of California at Berkeley* , 2000.
9. Bing Wang, Subhabrata Sen, Micah Adler and Don Towsley, "Optimal Proxy Cache Allocation for Efficient Streaming Media Distribution," in *INFOCOM '02*, 2002.
10. Younsu Chae, Katherine Guo, Milind Buddhikot, Subhash Suri and Ellen Zegura, "Silo, Tokens, and Rainbow: Schemes for Fault Tolerant Stream Caching," *Special Issue of IEEE JSAC on Internet Proxy Services* , 2002.
11. Soam Acharya andBrian Smith, "MiddleMan: A Video Caching Proxy Server," in *NOSSDAV '00*, 2000.
12. Yi Cui, Klara Nahrstedt, "Cooperative Proxy Overlay Network for Scalable On-Demand Media Distribution," *Technical Report No. UIUC-DCS-R-2002-2289* , 2002.
13. A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz and K. Worrell, "A Hierarchical Internet Object Cache," in *Proceedings of the USENIX Technical Conference*, 1996.
14. Ellen W. Zegura, Ken Calvert and S. Bhattacharjee, "How to Model an Internetwork," in *INFOCOM '96*, 1996.
15. Y. Guo, S. Sen and D. Towsley, "Prefix Caching assisted Periodic Broadcast:Framework and Techniques to Support Streaming for Popular Videos," *Technical Report TR 01-22, UMass CMPSCI* , 2001.
16. S. Sen, J. Rexford, and D. Towsley, "Proxy Prefix Caching for Multimedia Streams," in *IEEE INFOCOM '99*, 1999.
17. S. Ramesh, I. rhee and K. Guo, "Multicast with cache(mcache): An Adaptive Zero-delay Video-on-Demand Service," in *IEEE Infocom '01*, 2001.
18. A. Dan and D. Sitaram, "a Generalized Interval Caching Policy for Mixed Interactive and Long Video Environments," in *MMCN '96*, 1996.
19. S. Sheu, K. Hua and W. Tavanapong, "Chaining: a generalized batching technique for video-on-demand systems," in *IEEE International Conference on Multimedia Computing and Systems*, 1997.
20. Dongyan Xu and Klara Nahrstedt, "Find Service Paths in Service Proxy Network," in *MMCN '02*, 2002.