oStream: Asynchronous Streaming Multicast in Application-Layer Overlay Networks

Yi Cui, Baochun Li, *Member, IEEE,* and Klara Nahrstedt, *Member, IEEE*

***Abstract*— Although initially proposed as the deployable alternative to IP multicast, application-layer overlay network actually revolutionizes the way network applications can be built, since each overlay node is an end host, which is able to carry out more functionalities than simply forwarding packets. This paper addresses the on-demand media distribution problem in the context of overlay network. We take advantage of the strong buffering capabilities of end hosts, and propose a novel overlay multicast strategy, *oStream*. We have performed extensive analysis and performance evaluation with respect to the scalability and the efficiency of oStream. With respect to the required server bandwidth, we show that oStream defeats the theoretical lower bound of traditional multicast-based approaches, under both sequential and non-sequential stream access patterns. oStream is also robust against bursty requests. With respect to bandwidth consumption on the backbone network, we show that the benefit introduced by oStream overshadows the topological inefficiency (e.g., link stress and stretch) introduced by using application-layer multicast.**

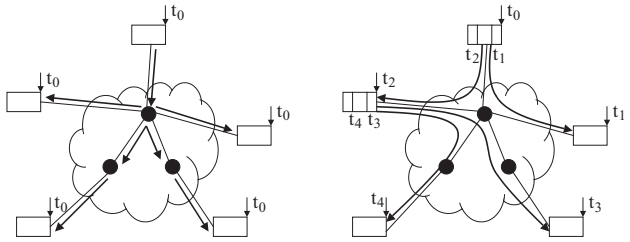
***Index Terms*— overlay, multicast, streaming**

I. Introduction

Application-layer multicast, or overlay multicast, was initially proposed to facilitate the deployment of multicast-based applications in the absence of IP multicast [1]. By organizing end hosts into an overlay network, multicast can be achieved through data relay among overlay members via unicast. Although seemingly it just elevates the multicast functionality into application layer, this approach actually revolutionizes the way network applications can be built. In IP multicast, except for nodes at the edge, the network is composed of routers, whose task is no more than forwarding packets. In contrast, each node in overlay network is an intelligent one that can contribute various resources (CPU, storage, etc.). Such great flexibilities have soon been utilized in the latest network and application designs, such as data/service indirection [2], resilient routing [3], and peer-to-peer streaming [4], etc.

This paper explores the feasibility of using an overlay-based approach to address the problem of on-demand media distribution. The fundamental challenge of this problem is the *unpredictability* of user requests in the following aspects: (1) *asynchrony*, where users may request the same media object at different times; (2) *non-sequentiality*, where users' stream access pattern is VCR-type, instead of sequential (from beginning to end); and (3) *burstiness*, where the request rate for a certain media object is highly unstable over time. The basic approach of traditional IP-multicast-based solutions [5], [6], [7], [8] is to repeat the same media content on different multicast channels over time. Clients are either enforced to be synchronized at the price of service delay, or required to participate in several multicast sessions simultaneously.

Yi Cui and Klara Nahrstedt are with the Department of Computer Science, University of Illinois at Urbana-Champaign. Their email addresses are {*yicui, klara*}*@cs.uiuc.edu*. Baochun Li is with the Department of Electrical and Computer Engineering, University of Toronto. His email address is *bli@eecg.toronto.edu*. This work was supported by NSF CISE Infrastructure grant under contract number NSF EIA 99-72884, and NSF ITR grant under contract number NSF CCR 00-86094. 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.



(a) IP Multicast (b) Asynchronous Multicast

Fig. 1. Conceptual Comparison of IP Multicast and Asynchronous Multicast

We argue that this is not necessarily the case in the context of overlay networks. In fact, we should leverage the temporal correlation of asynchronous requests and the buffering capabilities of overlay nodes to address the above challenges. As shown in Fig. 1, by enabling data buffering on the relaying nodes in an application-layer multicast tree, requests at different times can be satisfied by the same stream, thus achieving efficient media delivery. Based on this foundation, we propose *oStream*, an application-layer asynchronous streaming multicast mechanism. The main contributions of introducing oStream include the following favorable properties, supported and verified by extensive analytical and experimental results:

(1) *Scalability:* We derive the required server bandwidth in oStream, which defeats the theoretical lower bound of traditional multicast-based approaches, under both sequential and non-sequential access patterns. This may be achieved when we allow each relaying node in the multicast tree to buffer at most 10% of the media streams. Furthermore, over a certain threshold, the required server bandwidth no longer increases as the request rate grows, which suggests the robustness of oStream against "flash crowds".

(2) *Efficiency:* In previous works, server bandwidth has been used as the sole metric to evaluate system scalability and performance. However, bandwidth consumption on the backbone network for any streaming scheme has not been evaluated. This issue is of particular interest due to the following conjecture: although overlay networks inevitably introduce topological inefficiency (link stress and stretch) compared to IP multicast, the benefit introduced by asynchronous streaming multicast in

oStream may overshadow such inefficiency. Towards analyzing the efficiency of oStream, we investigate this conjecture analytically, the results of which have been confirmed by our experiments.

The remainder of the paper is organized as follows. Sec. II presents the temporal dependency model. Sec. III presents the asynchronous multicast algorithm. We extensively analyze the scalability and efficiency of oStream in Sec. IV and V, respectively. Sec. VI evaluates the performance of oStream and verify our analytical results. Sec. VII discusses the related work. Finally, Sec. VIII concludes the paper.

II. oSTREAM: PRELIMINARIES

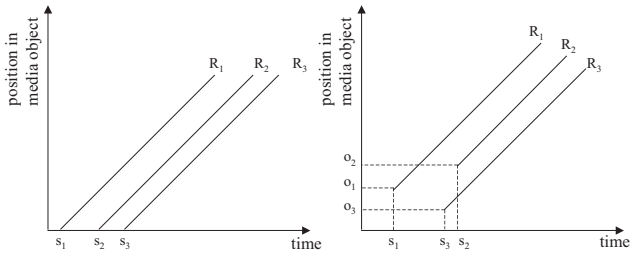
A. Temporal Dependency Model

In this paper, we consider true on-demand services, i.e., the client request is immediately processed, instead of being wait-listed. Consider two asynchronous requests R_i and R_j for the same media object. The object is played back at a constant bit rate of one byte per unit time. R_i starts at time s_i from the offset o_i. R_j starts at time s_j from the offset o_j. With respect to these *offsets*, the unit is *bytes*. Since the playback rate is one byte per unit time, the unit is equivalently *time unit* as well. This is convenient to compare these offsets with time instants (e.g., s_i and s_j). The time lengths of R_i and R_j are l_i and l_j. We define R_i as the *predecessor* of R_j, R_j as the *successor* of R_i (denoted as $R_i \prec R_j$), if the following requirements are met:

$$\begin{cases} o_i + l_i > o_j & \text{if} \quad o_i \le o_j \\ o_j + l_j > o_i & \text{if} \quad o_i > o_j \end{cases} \tag{1}$$

$$s_j - o_j > s_i - o_i \tag{2}$$

Inequality (1) demands that the media data requested by R_i and R_j must (partially) overlap. Inequality (2) means that R_i must retrieve the data before R_j does. Note that these are two parallel requirements, when both of which are met can R_i be the predecessor of R_j. Also, they only apply to the common piece of data requested by R_i and R_j.



(a) Sequential Access (b) Non-sequential Access

Fig. 2. Temporal Dependency Model

Fig. 2 (a) and (b) show examples of temporal dependencies under sequential and non-sequential access patterns. In both figures, $R_1 \prec R_2 \prec R_3$. Notice that in case of sequential access, $o_1 = o_2 = o_3 = 0$. From the definition, we may observe that R_1 has the potential to benefit R_2 in that, R_2 could (partially) reuse the stream from R_1 rather than obtaining it directly from the server. Note that in Fig. 2(b), R_2 is the predecessor of R_3, even though R_3 starts earlier than R_2. This case will not occur when all requests observe sequential access patterns (Fig. 2(a)).

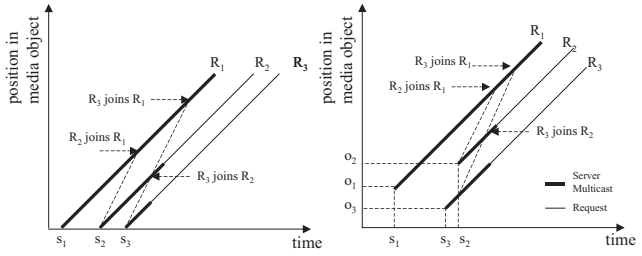
We claim that all on-demand media streaming algorithms could be described based on the temporal dependency model. In all algorithms, R_2 utilizes its predecessor/successor relation with R_1 to conserve server bandwidth, only with different ways of reusing the stream from R_1.

Notation	Definition
$R_i \prec R_j$	Request R_i is the Predecessor of Request R_j R_j is the Successor of R_i.
s_i	Starting Time of R_i
o_i	Starting Offset of R_i
l_i	Time Length (Duration) of R_i

TABLE I
NOTATIONS USED IN SEC. II

B. Hierarchical Stream Merging: a Review

We first review the hierarchical stream merging (HSM) algorithm. In [5], Eager *et al.* point out that HSM can achieve the theoretical lower bound of server bandwidth consumption for all multicast-based on-demand streaming algorithms, if the client has unlimited receiving bandwidth and buffering space. Therefore, the algorithm presented here does not belong to any particular implementation of the HSM family, but rather an ideal case, in which the theoretical lower bound in [5] can be achieved. Our purpose here is to establish a theoretical baseline, against which our proposed algorithm can be evaluated and compared.



(a) Sequential Access (b) Non-sequential Access

Fig. 3. Examples of Hierarchical Stream Merging

We illustrate the algorithm using the requests shown in Fig. 2(a) and (b) as examples. In Fig. 3(a), since R_1 has no predecessor, it opens a multicast session from the server to retrieve the requested data. R_2 initiates another multicast session upon its arrival. Meanwhile, it also listens to the session of R_1 to prefetch data from R_1. R_2 stays in both sessions until the point when the prefetched data starts to get played. From this point on, R_2 continues to prefetch data from R_1's session and withdraws from its own session. We claim that R_2 "joins" R_1 at this point. Similarly, R_3 initially opens its own multicast session and retrieves data from sessions of both R_1 and R_2 at the same time. It then withdraws from its own session and joins R_2 at the point when the data prefetched from R_2 may be used. Finally, R_3 joins R_1. Within the algorithm, each request repeatedly joins the ongoing multicast

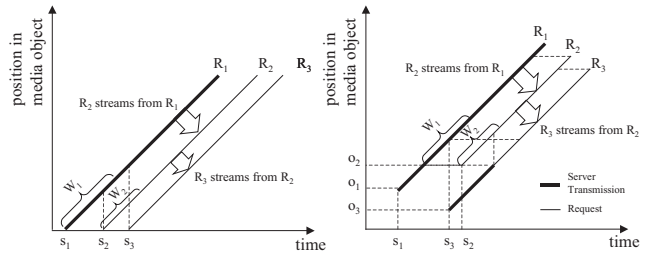
session of its closest predecessor until it catches up with the earliest one. The initial missing portion is offered by opening a "make-up" session from the server. Similar results are obtained in the non-sequential case, illustrated in Fig. 3(b).

As shown in the figures, the multicast sessions initially opened by R_2 and R_3 are terminated once all their members are merged into R_1's session. The requirement on server bandwidth is thus significantly reduced, compared to opening three separate unicast sessions to accommodate each request. Obviously, to reuse the stream from R_i, R_j must satisfy $s_i + l_i > s_j$, in addition to Inequality (1) and (2).

C. Asynchronous Multicast

We proceed to propose and illustrate the concept of *Asynchronous Multicast* (AM), again using the example in Fig. 2. We assume that each request is able to buffer the streamed data for a certain amount of time after playback. This can be achieved by using a circular buffer to cache the stream. For example, as shown in Fig. 4(a), R_1 has a buffer capable of storing data for time length W_1. In other words, any data cached in the buffer is kept for a time length of W_1, after which it is replaced by new data. Obviously, all requests that fall within this window may potentially be served by R_1. In this case, R_2 directly retrieves its stream from the buffer of R_1. R_3 is unable to stream from R_1 since it falls outside the buffer window of R_1. Instead, it streams from the buffer of R_2. The only difference in the case of non-sequential access (Fig. 4(b)) is that, R_3 needs to stream from the server for the initial portion, which is not available at R_2.

This mechanism is referred to as *asynchronous multicast*, mainly because it needs only one server stream to serve a group of requests, if any request has a predecessor in the same group, except for the earliest one. However, the difference is that members within the group receive data asynchronously. An important feature of asynchronous multicast is that it is purely end-host based. For example, in Fig. 4(a), R_1 streams from the server via unicast. R_2 also uses unicast connection to retrieve data from R_1. Another major difference is that in our approach, although a request may have multiple sources, the streaming from different sources is sequentialized. For example, in Fig. 4(b), R_3 first streams from the server, then switches to R_2. It keeps only one live connection throughout its entire session. In HSM, a request needs to stream from multiple sources in parallel.



(a) Sequential Access (b) Non-sequential Access

Fig. 4. Asynchronous Multicast

To summarize, in our algorithm, in order to reuse stream from R_i, R_j must meet the following requirement besides those listed in Inequality (1) and (2):

$$(s_j - o_j) - (s_i - o_i) < W_i \tag{3}$$

$(s_j - o_j) - (s_i - o_i)$ is also referred to as the *buffer distance* between R_i and R_j. If Inequality (3) is satisfied, we further define R_i as the *buffer-constrained predecessor* of R_j, R_j as the *buffer-constrained successor* of R_i, denoted as $R_i \overset{W_i}{\prec} R_j$. Henceforth in this paper, we simply refer to R_i as the predecessor of R_j, and R_j as the successor of R_i.

III. oSTREAM: ALGORITHMS

In this section, we present algorithms of asynchronous multicast. The algorithms are fully distributed and operate on the application-layer overlay.

Notation	Definition
$\mathbb{R}$	Request Set
$R_i \overset{W_i}{\prec} R_j$	R_i is the Buffer-Constrained Predecessor of R_j
$MDG_{\mathbb{R}} = (\mathbb{R}, E)$	Media Distribution Graph for $\mathbb{R}$
$MDT_{\mathbb{R}} = (\mathbb{R}, E^T)$	Media Distribution Tree for $MDG_{\mathbb{R}}$ ($E^T \in E$)
$c(R_i, R_j)$	Edge Weight of $(R_i, R_j) \in E$, Hop Count Between Two Hosts Carrying R_i and R_j
$MDG_{ins} = (\mathbb{R} \cup R_{ins}, E_{ins})$	Resulting Graph after R_{ins} is Added To $MDG_{\mathbb{R}}$
$MDT_{ins} = (\mathbb{R} \cup R_{ins}, E^T_{ins})$	Media Distribution Tree for MDG_{ins} ($E^T_{ins} \in E_{ins}$)
$MDG_{del} = (\mathbb{R} - R_{del}, E_{del})$	Resulting Graph after R_{del} is Deleted From $MDG_{\mathbb{R}}$
$MDT_{del} = (\mathbb{R} - R_{del}, E^T_{del})$	Media Distribution Tree for MDG_{del} ($E^T_{del} \in E_{del}$)

TABLE II
NOTATIONS USED IN SEC. III

A. Problem Formulation

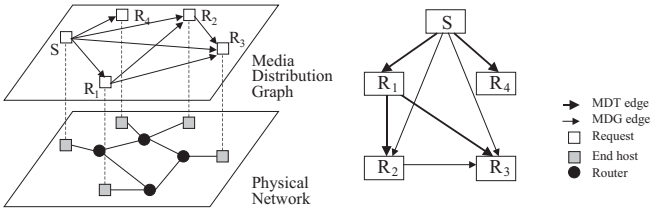
Definition 1 (Media Distribution Graph). Let $\mathbb{R}$ be a set of asynchronous requests. We define the *Media Distribution Graph* (MDG) for $\mathbb{R}$ as a directed weighted graph $MDG_{\mathbb{R}} = (\mathbb{R}, E)$, such that $E = \{(R_i, R_j) \mid R_i \overset{W_i}{\prec} R_j, R_i, R_j \in \mathbb{R}\}$. Each edge (R_i, R_j) has the weight $c(R_i, R_j)$, which is the transmission cost, i.e., hop count, between two end hosts carrying R_i and R_j, respectively.

We use an example to illustrate the concept of MDG. Consider a set of requests $\mathbb{R} = \{R_1, R_2, R_3, R_4\}$. We have $R_1 \overset{W_1}{\prec} R_2$, $R_1 \overset{W_1}{\prec} R_3$ and $R_2 \overset{W_2}{\prec} R_3$. The MDG for $\mathbb{R}$ is shown in Fig. 5(a). Each MDG node represents a request[1]. A special node S represents the server. Since the server can serve any request, it can be regarded as the predecessor of all members in $\mathbb{R}$. Thus, S has directed edges to all other nodes in the graph.

The media distribution graph changes dynamically: (1) when a new request arrives, a new node is inserted into the

[1]We use terms request and node interchangeably in the remainder of this section, depending on the context.

graph; (2) when a request is terminated, its corresponding node is removed from the graph. As such, MDG represents the dependencies among all asynchronous requests, which forms a virtual overlay on top of the physical network.



(a) Media Distribution Graph (b) Media Distribution Tree

Fig. 5. Illustration of the Media Distribution Graph

Given the definition of MDG, now we can formulate the problem of on-demand media distribution as the issue of constructing and maintaining a spanning tree on MDG, defined as the *Media Distribution Tree*.

Definition 2 (Media Distribution Tree). For a request set $\mathbb{R}$ and its MDG $MDG_{\mathbb{R}} = (\mathbb{R}, E)$, the corresponding MDT is denoted as $MDT_{\mathbb{R}} = (\mathbb{R}, E^T)(E^T \in E)$, and it is a *spanning tree* on MDG. For two nodes $R_i, R_j \in \mathbb{R}$, if $(R_i, R_j) \in E^T$, then R_i is the *parent* of R_j, R_j is the *child* of R_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. An example is shown in Fig. 5(b). In this figure, (S, R_1), (S, R_4), (R_1, R_2) and (R_1, R_3) are MDT edges.

B. Basic Algorithm: MDT Operations

We proceed to present our basic algorithm on MDT construction and maintenance. Our algorithm is fully distributed and incremental. First, we do not assume the existence of a centralized manager to control the tree construction. Second, each new request joins the tree based on its local decision, which only needs partial knowledge of the existing tree. Third, upon request departure, the tree is quickly recovered since no global reorganization is required.

The algorithm is executed each time when a new request joins the graph, or when an existing request departs. To tackle problems in both cases, our algorithm is split into two operations: **MDT-Insert** and **MDT-Delete**.

We first present **MDT-Insert**. Let $MDG_{ins} = (\mathbb{R} \cup R_{ins}, E_{ins})$ be the resulting graph after the request R_{ins} is added to $MDG_{\mathbb{R}}$, **MDT-Insert** is able to return $MDT_{ins} = (\mathbb{R} \cup R_{ins}, E^T_{ins})$ as the new MDT for MDG_{ins}.

Second, we present **MDT-Delete**. Let $MDG_{del} = (\mathbb{R} - R_{del}, E_{del})$ be the resulting graph after R_{del} is removed from $MDG_{\mathbb{R}}$, **MDT-Delete** is able to return $MDT_{del} = (\mathbb{R} - R_{del}, E^T_{del})$ as the new MDT for MDG_{del}.

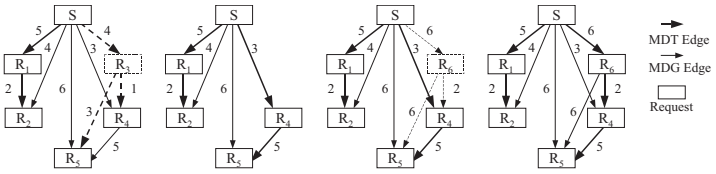
We use an example to illustrate these two algorithms. In Fig. 6 (a), when R_3 leaves, it first deletes itself from the MDT (Lines 1 to 2 in **MDT-Delete**), then notifies its children R_4 and R_5 to find their new parents as S and R_4, respectively (Lines 3 to 5 in **MDT-Delete**). In Fig. 6 (b), when a new request R_6 joins, it first finds S as its parent (Lines 1 to 2 in **MDT-Insert**), then notifies its successor R_4 to switch parent from S to R_6, since $c(R_6, R_4) < c(S, R_4)$ (Lines 3 to 5 in **MDT-Insert**).

MDT-Insert$(R_{ins}, E_{ins}, E^T, E^T_{ins})$
/* From all its predecessors, R_{ins} finds parent R_{min}, whose transmission cost to R_{ins} is minimal */
1 $R_{min} \leftarrow \arg\min\{c(R_{pre}, R_{ins}) \mid (R_{pre}, R_{ins}) \in E_{ins}\}$
2 $E^T_{ins} \leftarrow E^T \cup (R_{min}, R_{ins})$

/* For all its successors R_{suc}, R_{ins} compares if the transmission cost from itself to R_{suc} is less than from R_{suc}'s current parent R_{par}. If so, R_{suc} is asked to switch parent to R_{ins}. */
3 **for each** $R_{suc} \in \{R_{suc} \mid (R_{ins}, R_{suc}) \in E_{ins}\}$
4 **if** $c(R_{ins}, R_{suc}) < c(R_{par}, R_{suc}) \mid (R_{par}, R_{suc}) \in E^T_{ins}$
5 **do** $E^T_{ins} \leftarrow (E^T_{ins} - (R_{par}, R_{suc})) \cup \{(R_{ins}, R_{suc})\}$

MDT-Delete$(R_{del}, E_{del}, E^T, E^T_{del})$
/* R_{del} deletes the tree edge from its parent R_{par} */
1 $E^T_{del} \leftarrow E^T - \{(R_{par}, R_{del}) \mid (R_{par}, R_{del}) \in E^T\}$
2 **for each** $R_{chi} \in \{R_{chi} \mid (R_{del}, R_{chi}) \in E^T\}$ **do**
/* R_{del} deletes tree edge to each of its children R_{chi} */
3 $E^T_{del} \leftarrow E^T_{del} - (R_{del}, R_{chi})$

/* From all its predecessors, R_{chi} finds the new parent R_{min}, whose transmission cost to R_{chi} is minimal */
4 $R_{min} \leftarrow \arg\min\{c(R_{pre}, R_{chi}) \mid (R_{pre}, R_{chi}) \in E_{del}\}$
5 $E^T_{del} \leftarrow E^T_{del} \cup (R_{min}, R_{chi})$



(a) Before R_3 is Deleted (b) After R_3 is Deleted (c) Before R_5 is Inserted (d) After R_5 is Inserted

Fig. 6. Illustration of the Media Distribution Graph

Theorem 1 (Correctness): Both **MDT-Insert** and **MDT-Delete** generate loop-free spanning trees.

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

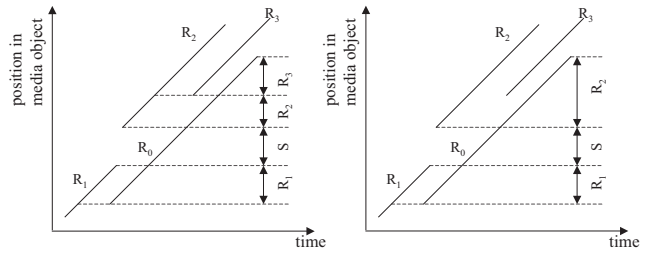
The reader is referred to [25] for the details of the proofs.

C. Practical Issues

1) Content Discovery Service: The MDT algorithms require that each request must have knowledge of all its predecessors and successors. Therefore, a publish/subscribe service needs to be in place for the purpose of information exchange and update. We have designed a distributed content discovery service to address this issue. In this approach, a number of overlay nodes are employed as discovery servers, each with a unique ID. A set of hashing functions are designed to map the timing information of a request R_i (its starting time s_i and offset o_i) to a subset of discovery servers. In this way, the registration and query operations with regard to R_i can

be directed to a subset of servers, which are responsible to keep R_i's record, until R_i leaves. Due to space constraints, the reader is referred to [25] for a the detailed presentation of the content discovery service.

2) Simplifying Session Switching: A request may have to switch its streaming session in several cases as illustrated in Fig. 7. In the figure, R_0 initially receives stream from R_1. When R_1 finishes before R_0 does, R_0 has to find a new predecessor to retrieve data from[2]. This case is unavoidable since R_0 cannot know when R_1 will finish beforehand. If no predecessor is found, R_0 directly streams data from the server S, until a new predecessor R_2 appears[3]. At this point, R_0 switches from the server to R_2. This is consistent with the primary goal of our algorithm: to maximally save the server bandwidth, *i.e.,* R_0 streams from the server only when it has no predecessor. Finally, if a new predecessor R_3 appears and the transmission cost between R_0 and R_3 is smaller than the cost between R_0 and R_2 ($c(R_0, R_3) < c(R_0, R_2)$), R_0 switches from R_2 to R_3 to save network cost. However, to achieve this goal, R_0 has to keep updated of newly arrived requests and compares if they are closer than its current predecessor (lines 3 to 5 in **MDT-Insert**). It also increases the number of session switching times. To save session switching overhead, we simplify the basic algorithm as shown in Fig. 7(b): R_0 continues to stream from R_2, without considering to switch to a closer predecessor, namely R_3. However, the price is that the basic algorithm's optimality at saving link cost is sacrificed.



(a) Basic Algorithm (b) Simplified Algorithm

Fig. 7. Simplifying Session Switching

3) Degree Constrained MDT: In the basic algorithm on MDT operations, we did not constrain the outbound degree of a tree node. Our assumption here is that each end host has unlimited outbound bandwidth to have as many children as possible. However, when we limit the outbound degree of a tree node, then regarding **Definition 2**, the MDT problem has to be modified as follows.

For a request set $\mathbb{R}$ and its MDG $MDG_{\mathbb{R}} = (\mathbb{R}, E)$, find the MDT $MDT_{\mathbb{R}} = (\mathbb{R}, E^T)(E^T \in E)$, which is the *minimal degree-constrained spanning tree* on $MDG_{\mathbb{R}}$.

This problem is NP-complete [19]. Our heuristic is as follows: for each newly arrived request, it finds one among its candidate predecessors with smallest transmission cost, whose outbound degree has not reached the constraint yet. Obviously, the heuristic algorithm is not optimal at reducing link cost. However, its optimality at conserving server bandwidth remains intact.

Theorem 3: Degree-constrained MDT algorithm consumes the same amount of server bandwidth as the basic MDT algorithm.

The reader is referred to [25] for a detailed proof of this theorem.

IV. Scalability: Server Bandwidth Savings

In this section, we analyze the scalability of HSM and asynchronous multicast with respect to conserving server bandwidth. We first introduce the analytical methodology, then derive the required server bandwidth of each approach under different stream access patterns.

A. Analytical Methodology

We consider the distribution of a single media object. The size of the object is T bytes. The object is played out at a constant bit rate of one byte per unit time. Therefore the playback time of the object is T time units. Client requests follow a Poisson process with arrival rate λ. We consider an arbitrarily small portion of the object, say, a byte. This byte is located at the offset x of the object. This byte is denoted as $\boldsymbol{x}$.

Let us assume that $\boldsymbol{x}$ is multicast by the server at time 0, we need to know till when $\boldsymbol{x}$ needs to be multicast again. Let X be the event of $\boldsymbol{x}$ being requested, Λ_X is the average arrival rate of X. We use a random variable w to denote the interarrival time of events in $\{X\}$. Let Z be the event of server multicast of $\boldsymbol{x}$. Clearly events in $\{Z\}$ are a subset of events in $\{X\}$, since not every request for $\boldsymbol{x}$ will trigger the sever multicast. We further use a random variable τ to denote the interarrival time of events in $\{Z\}$. If we know the expected value of τ, denoted as $E(\tau|x)$ (the expected value is conditional, depending on $\boldsymbol{x}$'s location in the media object, i.e., x), then with respect to the above raised question, $\boldsymbol{x}$ will be multicast again after time length $E(\tau|x)$. It means that on average, $\boldsymbol{x}$ is multicast for $\frac{1}{E(\tau|x)}$ times per unit time. Therefore, the required server bandwidth for $\boldsymbol{x}$ is $\frac{1}{E(\tau|x)}$ per unit time. Summarizing the bandwidth for all bytes in the object, the total required server bandwidth B is given by

$$B = \int_0^T \frac{\mathrm{d}x}{E(\tau|x)} \tag{4}$$

Therefore, the main goal of our analysis is to acquire $E(\tau|x)$.

B. Hierarchical Stream Merging

We start with the HSM algorithm. Analytical results in this subsection have appeared in related works [5] and [9]. We therefore omit the detailed derivations. Assume that a request R_1 requests the $\boldsymbol{x}$ at time 0 and triggers a server multicast. $\boldsymbol{x}$

[2]Notice that there is a time delay from the termination of R_1 until R_0 starts streaming from its new predecessor, which is the buffer distance between R_0 and R_1. During this period, R_1 flushes the data remained in its buffer to R_0.

[3]Although R_2 arrives later than R_0, it can still be R_0's predecessor. A similar example is illustrated in Fig. 2(b)

Notation	Definition
T	Object Size in Bytes
λ	Average Request Rate
$\boldsymbol{x}$	A Byte of the Object
x	$\boldsymbol{x}$'s Location in the Object
$\{X\}$	Events of Request for $\boldsymbol{x}$
$\{Z\}$	Events of Server Multicast of $\boldsymbol{x}$
Λ_X	Average Arrival Rates of Events in $\{X\}$
Λ_Z	Average Arrival Rates of Events in $\{Z\}$
w	Interarrival Time of Events X
τ	Interarrival Time of Events Z
$E(\tau\|x)$	Expected Value of τ
B	Required Server Bandwidth
S	Request Duration in Simple Access Model
W	Buffer Size in Asynchronous Multicast

TABLE III
NOTATIONS USED IN SEC. IV

is requested later by R_2 at time t. As outlined in Sec. II-B, if R_2 starts before time 0, it can catch up the multicast of $\boldsymbol{x}$. Otherwise, it misses the multicast at time 0, and $\boldsymbol{x}$ needs to be multicast again at time t.

The answer to whether R_2 starts before time 0 varies depending on the stream access models we assume. In our analysis, we choose two models: *simple access* and *sequential access*. They are studied respectively as follows.

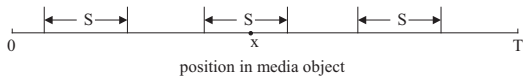


Fig. 8. Simple Access Model

1) Simple Access Model: Under simple access model, each request lasts for time length S ($S < T$). The request starts from an arbitrary offset of the object. For simplicity, we assume that the object is cyclic, which means that the access may proceed past the end of the object by cycling to the beginning of the object. As shown in Fig. 8, a request would contain $\boldsymbol{x}$ only if its starting offset is ranged within $(x-S, x)$. Assuming the starting offset of a request is uniformly distributed within $(0, T)$, then the probability that this request contains $\boldsymbol{x}$ is S/T. Consequently, the arrival rate of event X is

$$\Lambda_X = \lambda S/T \tag{5}$$

R_2's starting time s_2 is ranged within the time interval $(t-S, t)$. R_2 will trigger a new multicast of $\boldsymbol{x}$ if $s_2 > 0$. As shown in Fig. 9 (a), if $t \le S$, then $s_2 > 0$ with probability t/S. In this case, with probability t/S, an event X will trigger an event Z. If $t > S$ (Fig. 9 (b)), $s_2 > 0$ is always true. In this case, an event X will definitely trigger an event Z. Now we can derive the arrival rate of event Z as follows

$$\Lambda_Z = \begin{cases} \Lambda_X t/S & \text{if } t \le S \\ \Lambda_X & \text{if } t > S \end{cases}$$

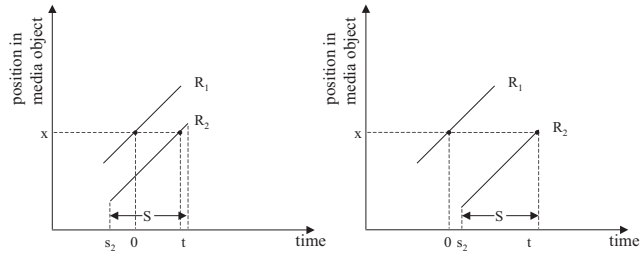


Fig. 9. Server Bandwidth Analysis of Hierarchical Stream Merging under Simple Access Model

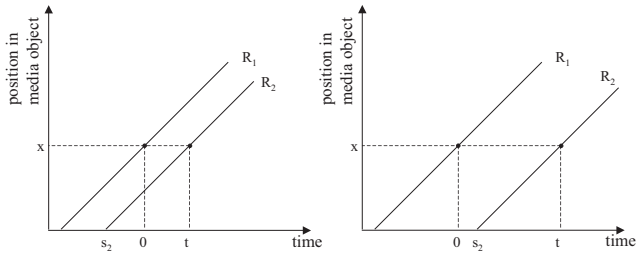


Fig. 10. Server Bandwidth Analysis of Hierarchical Stream Merging under Sequential Access Model

With Λ_Z, we can derive that $E(\tau|x) = \sqrt{\frac{\pi T}{2\lambda}}$. Based on Eq. (4), the required server bandwidth is

$$B_{HSM}^{sim} = \int_0^T \frac{\mathrm{d}x}{E(\tau|x)} \approx \sqrt{\frac{2\lambda T}{\pi}} \tag{6}$$

The detailed derivation can be found in [9].

2) Sequential Access Model: Under the sequential access model, every request contains the entire object. Then it is obvious that

$$\Lambda_X = \lambda \tag{7}$$

As shown in Fig. 10 (a), if $t \le x$, R_2 will definitely arrive no later than time 0. If $t > x$ (Fig. 10 (b)), then R_2 will never arrive before time 0. Therefore, we have

$$\Lambda_Z = \begin{cases} 0 & \text{if } t \le x \\ \lambda & \text{if } t > x \end{cases}$$

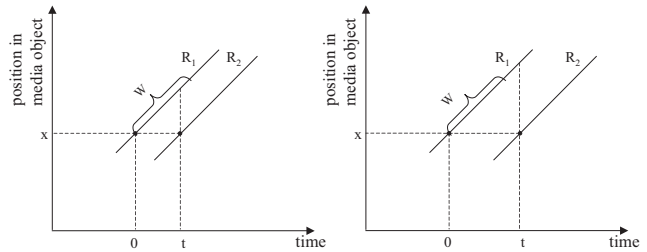
With Λ_Z, we have

$$B_{HSM}^{seq} = \int_0^T \frac{\mathrm{d}x}{E(\tau|x)} = \int_0^T \frac{\mathrm{d}x}{x + 1/\lambda} = \ln(\lambda T + 1) \tag{8}$$

The detailed derivation can be found in [5].

C. Asynchronous Multicast

We use the same analytical model to derive the required server bandwidth for asynchronous multicast. However, we rephrase event Z as the server transmission (unicast) of $\boldsymbol{x}$. For simplicity, we assume that all requests have a unified buffer size W. Suppose R_1 requested $\boldsymbol{x}$ at time 0, which triggered the server transmission of $\boldsymbol{x}$. Then $\boldsymbol{x}$ will stay in the buffer of R_1 for time W. Thus, if R_2 requests $\boldsymbol{x}$ within interval $(0, W)$ (Fig. 11(a)), $\boldsymbol{x}$ can be retrieved from the R_1 and further buffered at R_2 for time W. Otherwise (Fig. 11(b)), $\boldsymbol{x}$ has to be retransmitted by the server. In other words, a "chain" is formed among consecutive requests (X event) if



(a) $t \leq W$ (b) $t > W$

Fig. 11. Server Bandwidth Analysis of Asynchronous Multicast

their interarrival times are within W. The head request at each chain triggers the server transmission of $\boldsymbol{x}$. Therefore, $E(\tau|x)$ is the average time length of the chain. To derive $E(\tau|x)$, we first calculate the expected value of w, the interarrival time of events X. If we know Λ_X, then the expected number of events X during interval $(0,t)$ is

$$N_X = \int_0^t \Lambda_X \mathrm{d}t = t\Lambda_X$$

Then the probability of no arrival of X during interval $(0,t)$ is $P(w > t) = e^{-N_X}$, since events in $\{X\}$ are independent. Hence, the conditional distribution function of w is

$$F_w(t|x) = P(w \leq t) = 1 - e^{N_X} = 1 - e^{-t\Lambda_X}$$

Then the conditional density function can be derived as

$$f_w(t|x) = \frac{\mathrm{d}F_w(t|x)}{\mathrm{d}t} = \Lambda_X e^{-t\Lambda_X}$$

We then calculate the expected value of w when $w \leq W$. In this case, the "chain" is prolonged.

$$E_{w\leq W}(w|x) = \frac{\int_0^W t f_w(t|x)\mathrm{d}t}{P\{w \leq W\}} = \frac{\Lambda_X}{1 - e^{-\Lambda_X W}} \int_0^W t e^{-t\Lambda_X} \mathrm{d}t$$

Similarly, the expected value of w when $w > W$ is derived as follows. In this case, the "chain" is broken.

$$E_{w>W}(w|x) = \frac{\int_W^\infty t f_w(t|x)\mathrm{d}t}{P\{w > W\}} = \frac{\Lambda_X}{e^{-\Lambda_X W}} \int_W^\infty t e^{-t\Lambda_X} \mathrm{d}t$$

Let $p = P\{w \leq W\}$, we can derive $E(\tau|x)$ as

$$E(\tau|x) = \sum_{i=0}^{\infty} p^i (1-p)(iE_{w\leq W}(w|x) + E_{w>W}(w|x))$$

Based on Eq. (4), we now have the unified form of B for asynchronous multicast.

$$B_{AM} = \int_0^T \frac{\mathrm{d}x}{E(\tau|x)} = \int_0^T \frac{\Lambda_X(e^{\Lambda_X W} - 1)}{e^{2\Lambda_X W} - \Lambda_X W - 1} \mathrm{d}x \quad (9)$$

When we substitute Λ_X with Eq. (5), and (7), we can obtain the required server bandwidth under simple access and sequential access models as follows.

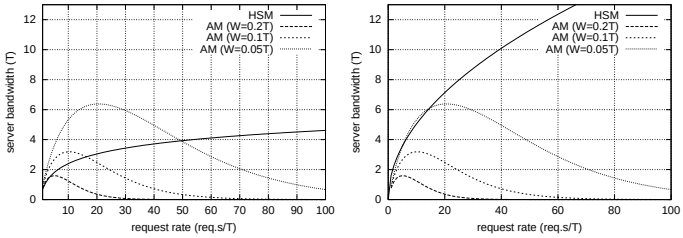
$$B_{AM}^{sim} = \frac{\lambda S(e^{\frac{\lambda SW}{T}} - 1)}{e^{\frac{2\lambda SW}{T}} - \frac{\lambda SW}{T} - 1} \quad (10)$$

$$B_{AM}^{seq} = \frac{\lambda T(e^{\lambda W} - 1)}{e^{2\lambda W} - \lambda W - 1} \quad (11)$$

D. Comparison

We plot Eq. (8) and (11) in Fig. 12(a), Eq. (6) and (10) in Fig. 12(b), as functions of the average value of Λ_X, which is calculated as

$$\Lambda_X^{avg} = \frac{1}{T} \int_0^T \Lambda_X \mathrm{d}x$$



(a) Sequential Model (b) Simple Model ($S = \frac{T}{4}$)

Fig. 12. Server Bandwidth Analysis Results

For sequential access and simple access models, Λ_X^{avg} is λ and $\lambda\frac{S}{T}$, respectively. In both figures, request rate is the number of requests issued per T time units, the object's playback duration. Server bandwidth indicates how many T bytes of data are streamed from the server during this time. As Λ_X^{avg} grows, the cost of HSM is asymptotically increasing at different speeds (logarithm growth for sequential model and square root growth for non-sequential model). The cost of asynchronous multicast is identical under both models because of its unified form in Eq. (9). The cost reaches its maximum value when $\Lambda_X = 1/W$ (the point where $\frac{\mathrm{d}B}{\mathrm{d}\Lambda_X} = 0$). The reason is that the time length of the "request chain" ($E(\tau|x)$) increases exponentially, which overcomes the linear growth of Λ_X^{avg} after this threshold. This means that B can be finite in the face of unpredictable client request rate. The maximum of B is further controllable by tuning W.

We also consider a *Random Access Model*, where the request starts and ends at arbitrary offset of the object[10]. We found it difficult to analytically derive the closed forms of B for both HSM and asynchronous multicast under this model. However, experimental results suggest that the curves of random access model are very closed to their counterparts in Fig. 12(b) (simple access model). We briefly introduce the random access model in Appendix I.

V. Efficiency: Link Bandwidth Reduction

Besides server scalability, multicast also offers network efficiency at reducing link cost. It is well observed that the network overhead increases sublinearly as the multicast group size grows [16][17], which implies that the optimal network efficiency can be achieved by maximally enlarging the multicast group. In this section, we analyze the network efficiency of HSM and asynchronous multicast. The layout of this section is the same with Sec. IV.

A. Analytical Methodology

We again use the analytical model in Sec. IV. Consider an arbitrary byte $\boldsymbol{x}$ located at offset x of the object. Suppose $\boldsymbol{x}$ is multicast at time 0, then according to our analysis in Sec. IV, $\boldsymbol{x}$ will be multicast again after $E(\tau|x)$. Therefore, all requests for x that fall within the interval $(0, E(\tau|x))$ are served by the server multicast at time 0. In other words, they are aggregated into one multicast group. We use $G(x)$ to denote the number of receivers in this group. Clearly, $G(x) = E(\tau|x) \cdot \Lambda_X$. We further use $L(n)$ to denote the link cost of a multicast group with n receivers, i.e., number of physical links this multicast group contains. Then the average per-member link cost is $\frac{L(n)}{n}$. Substituting n with $G(x)$, the average link cost per request for $\boldsymbol{x}$ is $\frac{L(G(x))}{G(x)}$. Summarizing for all bytes in the object, the total link cost per request C can be formulated as

$$C = \int_0^T \frac{L(G(x))}{G(x)} \mathrm{d}x \tag{12}$$

Since we already know $E(\tau|x)$, $G(x)$ can be easily derived. The main goal of our analysis in this section is to acquire $L(n)$.

Notation	Definition
$G(x)$	Number of Receivers Getting the Multicast of $\boldsymbol{x}$
$L(n)$	Link Cost of a Multicast Group of n Receivers
C	Link Cost per Request
$U(s)$	Reachability Function Denoting Number of Nodes s Hops Away From the Server (HSM)
$F(s)$	Probability Distribution Function of Distance s between Two Clients (Asynchronous Multicast)
D	Depth of k-ary tree

TABLE IV
NOTATIONS USED IN SEC. V

B. Hierarchical Stream Merging

The derivation of $L(n)$ varies depending on the network topology. Even within the same topology model, $L(n)$ still takes different forms, since HSM depends on IP multicast, and asynchronous multicast is end-host based. In our analysis, we use the k-ary tree model, which was also adopted in [16] and [17]. Consider a k-ary tree of depth D. Each tree node is a router. The server is attached to the root node, while clients are attached to the leaf nodes. Since we mainly care about link cost on the backbone, the cost from client to the leaf node is ignored. Therefore, only data traveled within the k-ary tree is counted.

We first derive $L(n)$ for IP multicast. For this purpose, we introduce the reachability function $U(s)$, which denotes the number of tree nodes that are exactly s hops away from the source. In k-ary tree topology, $U(s) = k^s$, which is the number of nodes at tree level s. As shown in Fig. 13(a), consider a client H_0 attached to a random leaf node. The multicast path from the server to H_0 passes tree nodes of all levels. Then for an arbitrary node N_s at level s, the probability that the path goes through N_s is $\frac{1}{U(s)} = \frac{1}{k^s}$. If there are n clients, then the probability that none of their paths goes through N_s is $(1 - \frac{1}{U(s)})^n$. Therefore, the probability that N_s belongs to the multicast delivery tree is $(1-(1-\frac{1}{U(s)})^n)$. $L(n)$ (the size of the multicast tree) is thus given by

$$L_{IP}(n) = \sum_{s=1}^{D} U(s)(1-(1-\frac{1}{U(s)})^n) \approx n(\frac{1}{\ln k} + D - \frac{\ln n}{\ln k}) \tag{13}$$

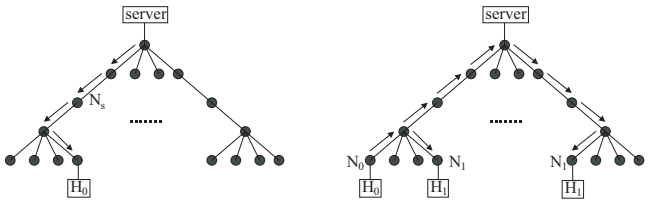
The detailed derivation of Eq. (13) can be found at [16]. Based on Eq. (12) and (13), we can derive the unified form of link cost for HSM as follows

$$C_{HSM} = \int_0^T (\frac{1}{\ln k} + D - \frac{\ln(G_{HSM}(x))}{\ln k}) \mathrm{d}x \tag{14}$$

Under sequential access model, $G^{seq}_{HSM}(x) = \lambda x + 1$ (derived from Eq. (8) and (7)). Under simple access model, $G^{sim}_{HSM}(x) = S\sqrt{\frac{\pi\lambda}{2T}}$ (derived from Eq. (6) and (5)). Now we can derive the link cost of HSM under these two models.

$$C^{seq}_{HSM} = T(\frac{1}{\ln k} + D) + \frac{\lambda T - (\lambda T + 1)\ln(\lambda T + 1)}{\lambda \ln k} \tag{15}$$

$$C^{sim}_{HSM} = T(\frac{1}{\ln k} + D - \frac{\ln(S\sqrt{\frac{\pi\lambda}{2T}})}{\ln k}) \tag{16}$$



(a) IP Multicast (b) Asynchronous Multicast

Fig. 13. k-ary Tree Topology Model

C. Asynchronous Multicast

Now we derive $L(n)$ for asynchronous multicast. Since this approach is end-host based, we need to derive the average unicast path length s between two clients. As shown in Fig. 13(b), for a given client H_0 attached to leaf router N_0, let s be the distance between H_0 and another client H_1. s is 0 if H_1 is also attached to N_0 (ignoring local link cost), which happens with probability $1/k^D$. If the router of H_1 shares the same parent with N_0 (probability k/k^D), s is either 2 or 0. In general, s is no more than $2h$ if C_0 and C_1 reside in the same subtree of height h. Therefore, we can summarize the probability distribution of s as

$$F(s) = k^{s/2-D} \tag{17}$$

If C_0 could receive data from m other clients, then the probability that none of them is within distance s to C_0 is $(1 - F(s))^n$. Then the distribution function of the distance from C_0 to the nearest one of these clients is given by

$$F_m(s) = 1 - (1 - F(s))^m$$

We can further acquire the probability density function $f_m(s) = \frac{\mathrm{d}F_m(s)}{\mathrm{d}s}$. Then the expected value of s can be derived as

$$E_m(s) = \int_0^{2D} s \cdot f_m(s) \mathrm{d}s \approx 2(D - \frac{\ln m}{\ln k}) \tag{18}$$

We illustrate the detailed derivation of Eq. (18) in Appendix II. As mentioned in Sec. IV-C, in asynchronous multicast, a "chain" is formed among consecutive requests, whose inter-arrival times are within the buffer size W. All requests on this chain form a multicast group. The header request unicasts data from the server. The link cost of this path is D. Each following request streams from one of its predecessors, whose buffer distance to itself is no more than W. The number of such candidate predecessors is $m = W\Lambda_X$. Then the link cost for a multicast group of n requests is given by

$$L_{AM}(n) = D + (n-1)E_m(s) = D + 2(n-1)(D - \frac{\ln m}{\ln k}) \quad (19)$$

Based on Eq. (12) and (19), the unified form of link cost for asynchronous multicast is

$$C_{AM} = \int_0^T \frac{D + 2(G_{AM}(x)-1)(D - \frac{\ln m}{\ln k})}{G_{AM}(x)} \mathrm{d}x \quad (20)$$

$G_{AM}(x)$ is derived from Eq. (9) as follows.

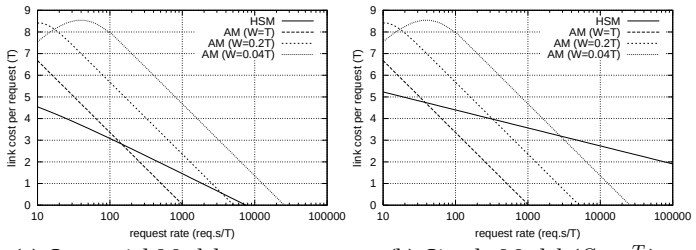
$$G_{AM}(x) = \frac{e^{2\Lambda_X W} - \Lambda_X W - 1}{e^{\Lambda_X W} - 1}$$

Substituting Λ_X with Eq. (7) and (5), we obtain the link cost under sequential and simple access model as follows.

$$C_{AM}^{seq} = T\Big[\frac{e^{\lambda W} - 1}{e^{2\lambda W} - \lambda W - 1}(\frac{2\ln \lambda W}{\ln k} - D) + 2(D - \frac{\ln \lambda W}{\ln k})\Big] \quad (21)$$

$$C_{AM}^{sim} = T\Big[\frac{e^{\frac{\lambda SW}{T}} - 1}{e^{\frac{2\lambda SW}{T}} - \frac{\lambda SW}{T} - 1}(\frac{2\ln \frac{\lambda SW}{T}}{\ln k} - D) + 2(D - \frac{\ln \frac{\lambda SW}{T}}{\ln k})\Big] \quad (22)$$

D. Comparison



(a) Sequential Model (b) Simple Model ($S = \frac{T}{4}$)

Fig. 14. Analysis of Multicast Link Cost per Request in k-ary Tree Network($k = 4$, $D = 5$)

We plot Eq. (15) and (21) in Fig. 14 (a). As shown in the figure, the first terms of Eq. (21) is negligible unless the buffer size is small (e.g., $W = 0.04T$). Furthermore, this term exponentially approaches zero as the request rate λ increases. Therefore, Eq. (21) can be simplified to $2T(D - \frac{\ln \lambda W}{\ln k})$. The scale factor here is the minus term $(-\frac{2T\ln\lambda}{\ln k})$. The remaining part of the equation is constant. Similarly, the first term of Eq. (15) is also negligible. Then its scale factor is $(-\frac{T\ln(\lambda)}{\ln k})$. The remaining part is constant. Reflected in the figure, the slope of asynchronous multicast is steeper than HSM since the scale factor of Eq. (21) is two times the scale factor of Eq. (15). Although decreasing more slowly, the cost of HSM is still the smallest, unless the buffer size W of asynchronous multicast becomes very large. Also note that increasing W for asynchronous multicast can help move down the curve, but has nothing to do with the scale factor. Furthermore, the diminishing return of increasing W is logarithmic. This is determined by the minus term $(-\frac{T\ln W}{\ln k})$ in Eq. (21). Finally, we note that the above equations become inaccurate as they approach 0. This is because $L(n)$ (Eq. (13) and (19)) is invalid when the group size n reaches its saturation point [16], *i.e.*, the number of clients exceeds the number of leaf routers ($n \geq k^D$).

Fig. 14(b) plots Eq. (16) and (22). The costs of asynchronous multicast are the same in Fig. 14 (a) and (b) because of their unified form in Eq. (20). However, the scale factor of HSM (Eq. (16)) reduces to $(-\frac{\ln\sqrt{\lambda}}{\ln k})$. Reflected in the figure, the curve of HSM decreases more slowly. This gives asynchronous multicast a better chance to outperform HSM.

The intuitive explanation for such phenomenon is that, when the stream access pattern is switched from sequential to non-sequential, HSM can aggregate fewer number of client requests into one group, while the multicast group size of asynchronous multicast stays unchanged. This observation implies the universality of cost link reduction gain of asynchronous multicast to HSM on various network topologies with the following property: the per-member link cost monotonically decreases as the multicast group grows. We reach the same conclusion for power-law topology[26], whose analysis can be found in our technical report[25].

VI. Performance Evaluation

In this section, we compare the performance of asynchronous multicast and HSM at saving server bandwidth and link cost. Note that in our experiment, asynchronous multicast is end-host based, while HSM assumes the existence of IP multicast. Furthermore, in HSM experiments, we assume that each client is able to simultaneously join unlimited number of multicast groups and calculate joining sequences offline. Therefore, the performance results of HSM is optimal but impractical. Our purpose here is to make it the theoretical baseline, along which the performance of end-host based asynchronous multicast can be evaluated.

A. Experimental Setup

We consider the case of a single CBR video distribution. The video file is one-hour long, *i.e.*, $T = 1$ hour. We do not specify the streaming rate. Instead, we use playback time to indicate the server and link bandwidth consumption. Each 12-hour run is repeated under sequential, simple and random access models. Since the results obtained from simple and random models are very closed, we only show results from the random model for space constraints. A brief introduction of the random model can be found in Appendix I.

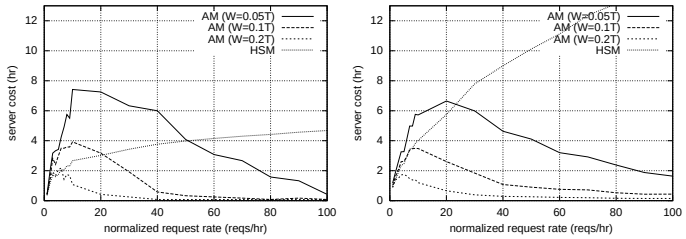
To study the impact of network topology on link cost, we run experiments on a diversified set of synthetic and real network topologies. Our selection largely falls into three categories:

1) **k-ary Tree Topology**: We choose this topology to confirm our analysis in Sec. V-D, which was conducted on the same topology. We set $k = 4$, $D = 5$, as was specified in Fig. 14. The topology size is 1365. We first experiment the case in which receivers are only located on leaf routers. In the second experiment, we allow receivers to be located on non-leaf routers as well.
2) **Router-level Topology**: We choose an Internet map (Lucent, November, 1999) [20] to represent the router-level topology. The topology size is 112969. We also use GT-ITM topology generator [21] to create a topology of 500 nodes based on transit-stub model. In this set of experiments, receivers can be located on any nodes in the topology.
3) **AS-level Topology**: We use a real AS map (March, 1997) obtained from NLANR [23]. The topology size is 6474. We also use the Inet topology generator [22] to create a topology of 6000 nodes. In both topologies, the distribution function of network distance between two nodes follows the power-law. In this set of experiments, we also allow receivers to be located on any nodes within the topology.

We assume that the IP unicast routing uses delay as its routing metric. IP multicast routing is based on shortest path tree algorithm (DVMRP [24]).

B. Server Bandwidth Consumption

We first evaluate the server bandwidth consumption (average amount of data streamed per hour). Since network topology has no impact on this metric, we only show results obtained on NLANR topology. Note that under random access model, the request rate is normalized the same way the analysis does in Sec. IV-D. The curves in Fig. 15(a) show the same growing trend as those analyzed in Fig. 12(a). The curves of both figures do not match exactly. This is mainly caused by the fact that for each specific request rate, the number of requests generated in our simulation cannot be the same as the statistical average number of requests, upon which curves in Fig. 12(a) are calculated. With regard to such statistical error, we believe that it is convincing enough that our results confirm our analysis in Sec. IV. When comparing Fig. 15(b) and Fig. 12(b), we draw the same conclusion.

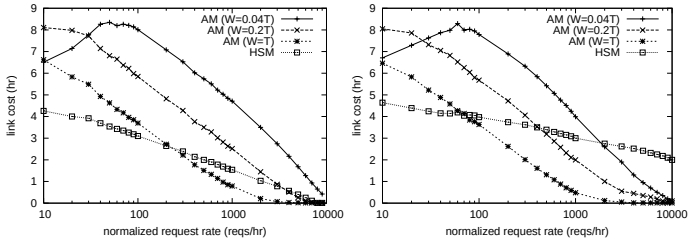


(a) Sequential Model (b) Random Model

Fig. 15. Server Bandwidth Consumption

C. Link Cost

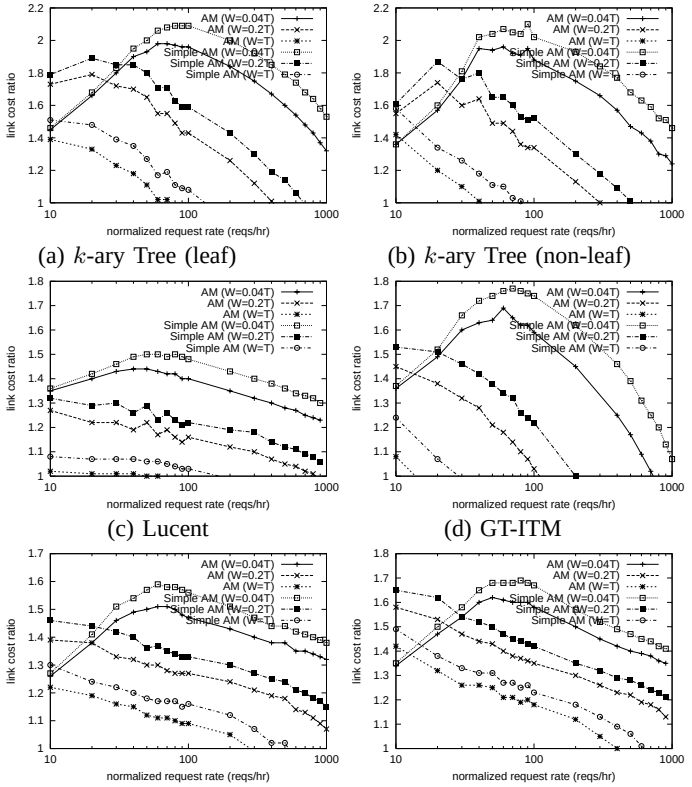
In order to validate the analysis of link bandwidth consumption in Sec. V-D, we first show the experimental results obtained from the same k-ary tree topology . The curves in Fig. 16 are nearly identical with those in Fig. 14. This experiment confirms our observation that if the stream access pattern is non-sequential, asynchronous multicast's ability of reducing link cost is stronger than HSM.



(a) Sequential Model (b) Random Model

Fig. 16. Link Cost per Request in k-ary Tree Topology($k = 4$, $D = 5$)

To simplify our illustration, we define a new metric *Link Cost Ratio*, which is the ratio of link cost of asynchronous multicast to HSM. With this metric, what we are mainly concerned about is the growing trend of the curves: if the link cost ratio drops as we increase the request rate. If the answer to this question is affirmative, our next question is when the "crossing point" (the point at which link cost ratio equals to 1) is reached.

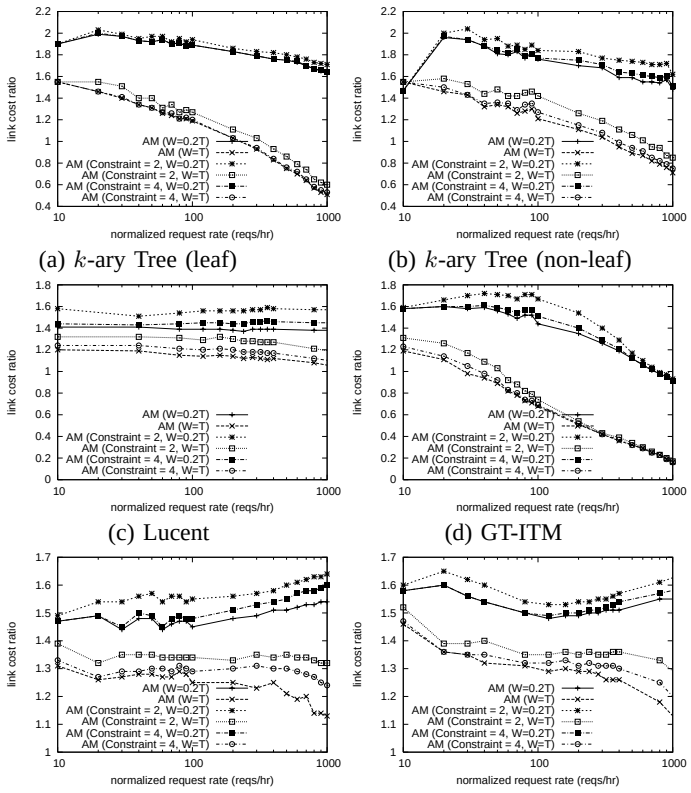


(a) k-ary Tree (leaf) (b) k-ary Tree (non-leaf)
(c) Lucent (d) GT-ITM
(e) NLANR (f) Inet

Fig. 17. Link Cost Ratio under Random Model

Fig. 17 shows the experimental results on link cost ratio under the random access model. From the results, we have the following observations. First, the link cost ratio heavily depends on the network topology. However, all curves have

negative slopes, which suggests the universality of link cost reduction gain of asynchronous multicast to HSM. The location of "crossing point" also greatly varies for different topologies. For example, in case $W = T$, the location of the point ranges from 20 reqs/hr (Fig. 17 (d)) to 300 reqs/hr (Fig. 17 (f)). Several factors may play important roles here, such as the network size, its topological properties and location of the server. Second, when we exponentially increase the buffer size (in the experiment, we set $W = 0.04T$, $0.2T$, T, respectively), the link cost reduction gain is almost linear, which confirms the same observation in our analysis (Sec. V-D). This finding suggests that small to medium sized buffers can be greatly helpful at saving link cost. Further increases with respect to the buffer size may be less beneficial. Third, the simplified algorithm (presented in Sec. III-C) increases the link cost by a fixed fraction. This impact can be offset by increasing the buffer size.



(a) k-ary Tree (leaf) (b) k-ary Tree (non-leaf)

(c) Lucent (d) GT-ITM

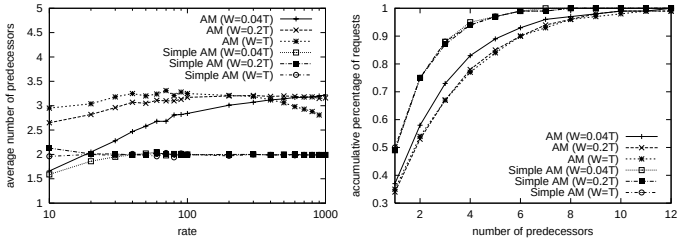
(e) NLANR (f) Inet

Fig. 18. Link Cost Ratio under Sequential Model

Fig. 18 shows the experimental results on link cost ratio under the sequential access model. A common observation is that all curves become less steep than their counterparts in Fig. 17. Plus, the "crossing point" can be hardly reached, unless the buffer size becomes large ($W = T$), or the request rate gets extremely high. Also, constraining the outbound degree of each end host does not greatly degrade the performance. When we set the constraint to 4, the curve is very similar to the one with no constraint. To summarize, this set of experiments reveal that under the sequential access model, the link cost reduction gain of asynchronous multicast to HSM is minor. The main reason is — as revealed in Sec. V-D — under the sequential access pattern, HSM is able to aggregate more requests into one multicast group than under the non-sequential access pattern.

D. Operation Complexity

Now we evaluate the operation complexity of asynchronous mulicast under non-sequential access model. Fig. 19 (a) shows the average number of predecessors a request needs to retrieve data from during its entire session. For the basic algorithm, this number is larger than 3. The simplified algorithm reduces this number to 2, which means that on average a request only needs to switch its predecessor once. Fig. 19 (b) shows the cumulative distribution of requests with different number of predecessors.



(a) Average Number of Predecessors per Request (b) Cumulative Distribution of Predecessor Number

Fig. 19. Operation Complexity of Asynchronous Multicast

VII. Related Work

A. Application-Layer Overlay

We have briefly reviewed related works [1], [2], [3], [4] in application-layer overlay at the beginning of this paper. While most current works focus on the efficient construction of an overlay network, it still remains a great challenge of how to manage the overlay members. In contrast to multicast, where relaying nodes are routers, which are assumed to be reliable and dedicated, overlay nodes are autonomous end hosts, which can easily crash, leave without notice, or refuse to share its own resources. Like most of the related works, we assume that all overlay nodes are well-behaved. However, a practical solution must be robust against unpredictable behaviors of end hosts, which constitutes possible direction of our future work.

B. Multicast-based On-Demand Media Distribution

The problem of delivering high-quality multimedia stream to asynchronous clients have been well studied. To achieve system scalability especially on the server side, IP multicast is widely adopted to serve multiple clients with one single server stream. However, the asynchrony of client requests is in conflict with the nature of multicast, which was originally designed to support applications of synchronous data transmission, such as conferencing. Various solutions have been proposed to address this conflict. In *batching* [6], client requests from different times are aggregated into one multicast session. However, users have to suffer long playback delay since their requests are enforced to be synchronized. The

approaching of *patching* [7] attempts to address this problem by allowing clients to "catch up" with an ongoing multicast session and patch the missing starting portion via server unicast. With *merging* [5], a client repeatedly merge into larger and larger multicast sessions. In periodic broadcasting [8], the server separates a media stream into segments and periodically broadcasts them through different multicast channels, from which a client chooses to join.

These solutions largely fall into two categories: true on-demand or immediate services, and near on-demand services. Solutions in the first category (patching and merging) serves the client immediately once the request is issued. For solutions in the second category (batching and periodic broadcasting), a client has to wait for a bounded delay time. In this paper, we only consider the true on-demand media distribution solutions. However, it is worth pointing out that the near on-demand solutions are in fact not superior than on-demand solutions at saving system cost. Eager *et al.* [5] reveal that using the approach of merging, the server bandwidth consumption grows at least logarithmically as the request rate increases. They also reveal that in periodic broadcasting, the server bandwidth requirement grows at least logarithmically as one tries to shorten the service delay bound. Therefore, the scale factors of both approaches are the same. Jin *et al.* [9] and Tan *et al.* [10] further confirm that this conclusion holds when the clients' media access pattern is non-sequential.

C. Media Caching and Buffering

Besides multicast, an orthogonal technique for reducing server loads is media caching. A large body of work in this area includes proxy caching. These works are similar to the proxy-based web caching in that they both use proxies to serve clients on behalf of the server, if the requested data is cached locally. However, since the media objects tend to be of large sizes, a proxy usually caches only a portion of the object. There are different ways of partial caching, such as prefix-based caching [11], [12], and segment-based caching [13]. Besides proxy caching, client-side caching is also proposed, such as chaining [14] and interval caching [15].

It is well observed that multicast and caching can help reduce the server load in media distribution. However, their performance at reducing link cost largely remains uninvestigated. The only work we are aware of is by Jin *et al.* [18], which analyzes the link cost of a client-based caching approach and shows its scalability. In this paper, we have evaluated the performance of multicast-based and cache-based solutions with respect to reducing link cost via in-depth analysis and extensive experiments. We have also investigated the impact of user access patterns (sequential or non-sequential) on the performance of both approaches.

VIII. Concluding Remarks

In this paper, we propose the concept of *asynchronous multicast*. This approach takes advantage of the strong buffering capabilities of end hosts in application-layer overlay networks. Based on the concept, we propose a novel overlay multicast strategy, *oStream*, to address the on-demand media distribution problem. Through in-depth analysis and extensive performance evaluation, we are able to draw the following conclusions. First, the required server bandwidth of oStream defeats the theoretical lower bound of traditional multicast-based solutions. Second, with respect to bandwidth consumption on the backbone network, the benefit introduced by oStream overshadows the topological inefficiency of application overlay.

References

[1] Y. Chu, S. Rao and H. Zhang, A Case for End System Multicast, *ACM SIGMETRICS*, 2000.

[2] I. Stoica, D. Adkins, S. Zhaung, S. Shenker and S. Surana, Internet Indirection Infrastructure, *ACM SIGCOMM*, 2002.

[3] D. Andersen, H. Balakrishnan, M. Kaashoek and R. Morris, Resilient Overlay Network, *ACM SOSP*, 2001.

[4] V. Padmanabhan, H. Wang, P. Chou and K. Sripanidkulchai, Distributing Streaming Media Content Using Cooperative Networking, *NOSSDAV*, 2002.

[5] D. Eager, M. Vernon and J. Zahorjan, Minimizing Bandwidth Requirements for On-Demand Data Delivery, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 13, No. 5, 2001.

[6] L. Gao and D. Towsley, Supplying Instantaneous Video-on-Demand Services Using Controlled Multicast, *Proceedings of IEEE Multimedia*, 1999.

[7] K. Hua, Y. Cai and S. Sheu, Patching: A Multicast Technique for True On-Demand Services, *ACM Multimedia*, 1998.

[8] K. Hua and S. Sheu, Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan VoD Systems, *ACM SIGCOMM*, 1997.

[9] S. Jin and A. Bestavros, Scalability of Multicast Delivery for Non-sequential Streaming Access, *ACM SIGMETRICS*, 2002.

[10] H. Tan, D. Eager and M. Vernon, Delimiting the Effectiveness of Scalable Streaming Protocols, *International Symposium on Computer Performance Modeling and Evaluation*, 2002.

[11] S. Sen, J. Rexford and D. Towsley, Proxy Prefix Caching for Multimedia Streams, *IEEE INFOCOM*, 1999.

[12] S. Ramesh, I. Rhee and K. Guo, Multicast with Cache (MCache): An Adaptive Zero-Delay Video-on-Demand Service, *IEEE INFOCOM*, 2001.

[13] Y. Chae, K. Guo, M. Buddhikot, S. Suri and E. Zegura, Silo, Tokens, and Rainbow: Schemes for Fault Tolerant Stream Caching, *IEEE Journal of Selected Areas on Communications, Special Issue on Internet Proxy Services*, 2002.

[14] S. Sheu, K. Hua and W. Tavanapong, Chaining: a Generalized Batching Technique for Video-on-Demand Systems, *IEEE ICMCS*, 1997.

[15] A. Dan and D. Sitaram, A Generalized Interval Caching Policy for Mixed Interval and Long Video Environments, *SPIE MMCN*, 1996.

[16] G. Philips and S. Shenker, Scaling of Multicast Trees: Comments on the Chuang-Sirbu Scaling Law, *ACM SIGCOMM*, 1999.

[17] C. Adjih, L. Georgiadis, P. Jacquet and W. Szpankowski, Multicast Tree Structure and the Power law, *ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[18] S. Jin and A.Bestavros, Cache-and-Relay Streaming Media Delivery for Asynchronous Clients, *NGC*, 2002.

[19] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.

[20] USC Information Sciences Insititute, Internet maps, *http://www.isi.edu/div7/scan/mercator/maps.html*, 1999.

[21] E. Zegura, K. Calvert and S. Bhattacharjee, How to Model an Internetwork, *IEEE INFOCOM*, 1996.

[22] C. Jin, Q. Chen and S. Jamin, Inet: Internet Topology Generator, *Technical Report CSE-TR-443-00, University of Michigan*, 2000.

[23] National Laboratory for Applied Network Research, *http://moat.nlanr.net/Routing/rawdata*, 1997.

[24] S. Deering and D. Cheriton, Multicast Routing in Datagram Internetworks and Extended LANs, *ACM Transactions on Computer Systems*, vol. 8, no. 2, 1990.

[25] Y. Cui, B. Li and K. Nahrstedt, oStream: Asynchronous Streaming Multicast in Application-Layer Overlay Networks, *Technical Report UIUCDCS-2002-2289/UILU-ENG-2002-1733*, Department of Computer Science, Univeristy of Illinois at Urbana-Champaign, 2003.

[26] M. Faloutsos, P. Faloutsos and C. Faloutsos, On Power-Law Relationships of the Internet Topology, *ACM SIGCOMM*, 1999.

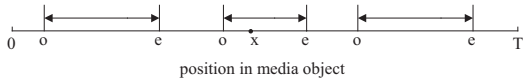


Fig. 20. Random access model

APPENDIX I

We briefly introduce the random access model, where a request starts and ends at arbitrary offsets. [10] has a detailed illustration of this model. Corresponding notations can be found at Tab. III. Suppose both the starting offset o and ending offset e of the request are uniformly distributed ($o < e$), as shown in Fig. 20. Consider an arbitrary byte $\boldsymbol{x}$ of the object located at offset x. If $o \leq x$, then in order to contain $\boldsymbol{x}$, e must satisfy that $x < e$. The probability is $\frac{T-x}{T-o}$. If $o > x$, then by no means $\boldsymbol{x}$ can be contained in (o, e). Therefore, we have

$$\Lambda_X = \frac{\int_0^x \lambda \frac{T-x}{T-o} \mathrm{d}o + \int_x^T \lambda \cdot 0 \mathrm{d}o}{T} = \lambda \frac{T-x}{T} \ln \frac{T}{T-x} \quad (23)$$

Similar to the analysis with simple access model in Sec. IV-B.1, we first derive Λ_Z. Similar to Fig. 10, if $t > x$, then R_2 can never arrive before time 0, thus will trigger the multicast of $\boldsymbol{x}$. Otherwise, the probability that R_2 arrives after time 0 is derived as follows.

$$\frac{\int_{x-t}^x \lambda \frac{T-x}{T-o} \mathrm{d}o}{T} = \lambda \frac{T-x}{T} \ln \frac{T-x+t}{T-x}$$

Therefore, the arrival rate of Z becomes

$$\Lambda_Z = \begin{cases} \lambda \frac{T-x}{T} \ln \frac{T-x+t}{T-x} & \text{if } t \leq x \\ \Lambda_X & \text{if } t > x \end{cases}$$

With Λ_X and Λ_Z, following the same procedure in Sec. IV-B and IV-C, we can get the server bandwidth for HSM and asynchronous multicast, which is hard to derive analytically. Instead, we showed our experimental results under this model in Sec. VI.

APPENDIX II

In this appendix, we derive $E_m(s)$ (Eq. (18)) in Sec. V-C.

$$E_m(s) = \int_0^{2D} s \cdot f_m(s) \mathrm{d}s = \int_0^{2D} s \cdot m(1-k^{s/2-D})^{m-1} \frac{\ln k}{2k^D} \mathrm{d}s \quad (24)$$

Let $y = 1 - k^{s/2-D}$, then Eq. (24) becomes

$$\begin{aligned} & \int_0^{1-\frac{1}{k^D}} \frac{2\ln[(1-y)k^D]}{\ln k} \cdot m y^{m-1} \mathrm{d}y \\ = \; & \frac{2\ln(k^D)}{\ln k} \int_0^{1-\frac{1}{k^D}} m y^{m-1} \mathrm{d}y + \frac{2}{\ln k} \int_0^{1-\frac{1}{k^D}} \frac{\ln(1-y)}{\ln k} m y^{m-1} \mathrm{d}y \\ = \; & [2D + \frac{2}{\ln k} \ln(\frac{1}{k^D})](1-\frac{1}{k^D})^m + \frac{2}{\ln k} \int_0^{1-\frac{1}{k^D}} \frac{y^m}{1-y} \mathrm{d}y \end{aligned}$$

Now the only unsolved integral form is $\int_0^{1-\frac{1}{k^D}} \frac{y^m}{1-y} \mathrm{d}y$. We define it as $N(m)$. then we have

$$\begin{aligned} N(m) - N(m-1) &= -\int_0^{1-\frac{1}{k^D}} y^{m-1} \mathrm{d}y = -\frac{(1-\frac{1}{k^D})^m}{m} \\ N(0) &= \int_0^{1-\frac{1}{k^D}} \frac{1}{1-y} \mathrm{d}y = -\ln(\frac{1}{k^D}) \end{aligned}$$

Then we derive $N(m)$ as follows

$$N(m) = -\ln(\frac{1}{k^D}) - \sum_{i=1}^{m} \frac{(1-\frac{1}{k^D})^i}{i} \quad (25)$$

Since k^D (total number of leaf nodes) is usually large, we can approximate $(1-\frac{1}{k^D})$ as 1. Finally, we have

$$E_m(s) \approx 2D - \frac{2}{\ln k} \sum_{i=1}^{m} \frac{1}{i} \quad (26)$$

Since $\sum_{i=1}^{m} \frac{1}{i}$ is asymptotically close to $\ln m$, we can simplify Eq. (26) as $2(D - \frac{\ln m}{\ln k})$.