On the Tradeoff Between Playback Delay and Buffer Space in Streaming

Alix L.H. Chow[*] Leana Golubchik[†] Samir Khuller[‡] Yuan Yao[§]

Abstract

We consider the following basic question: a source node wishes to stream an ordered sequence of packets to a collection of receivers, which are in K clusters. A node may send a packet to another node in its own cluster in one time step and to a node in a different cluster in T_c time steps ($T_c > 1$). Each cluster has two special nodes. We assume that the source and the special nodes in each cluster have a higher capacity and thus can send multiple packets at each step, while all other nodes can both send and receive a packet at each step. We construct two (intra-cluster) data communication schemes, one based on multi-trees (using a collection of d-ary interior-disjoint trees) and the other based on hypercubes. The multi-tree scheme sustains streaming within a cluster with $O(d \log N)$ maximum playback delay and $O(d \log N)$ size buffers, while communicating with $O(d)$ neighbors, where N is the maximum size of any cluster. We also show that this protocol is optimal when $d = 2$ or 3. The hypercube scheme sustains streaming within a cluster, with $O(\log^2(\frac{N}{d}))$ maximum playback delay and $O(1)$ size buffers, while communicating with $O(\log(\frac{N}{d}))$ neighbors, for arbitrary N.

1 Introduction

Continuous media (CM) streaming over a variety of networks is an application which provides a rich source of interesting algorithmic problems. The specific problem we consider here is that of a source node streaming data to a collection of receiving nodes, where the receivers need to contribute to the delivery process, i.e., due to communication (bandwidth) resource limitations it is not possible for all nodes to receive the stream directly from the source. This is a standard motivation for use of peer-to-peer (P2P) systems or systems that involve application layer multicast. The streamed data can correspond to either a *live* source (i.e., the data is produced during the delivery process), or to a *pre-recorded* stream (i.e., all data is available at the beginning of the delivery process).

In a real environment, we may have nodes that are in different geographical locations and thus communication delays may be significantly different. Assume that the nodes are divided into several clusters (e.g., based on geographic proximity). Each node can transmit

[*]Department of Computer Science, USC, Los Angeles, CA 90089, lhchow@cs.usc.edu.
[†]Departments of CS & EE-Systems, USC, Los Angeles, CA 90089, leana@cs.usc.edu.
[‡]Department of Computer Science, University of Maryland, College Park, MD 20742, samir@cs.umd.edu.
[§]Department of Electrical Engineering-Systems, USC, Los Angeles, CA 90089, yuanyao@usc.edu.

one packet to any node within the same cluster in one time step. However, although packets could be sent from one node to any node in another cluster, the transmission delay across clusters is large (this is similar to the model in [9]). As a result, when streaming packets through the network, it is desirable to have a small number of transmissions across clusters. To be more specific, assume that there are K clusters, each containing a sufficiently large number of nodes. Let the delay to send a message between nodes in two different clusters be T_c. However, within a cluster we can send a message to another node in the same cluster in one time step. Our packet distribution scheme will distribute the stream of packets using a "super-tree", τ, on the clusters, constructed by selecting a special node from each cluster. Within each cluster a special (local) root node will be responsible for distributing the stream of packets to the members of that cluster. Details of this are given in Section 2.

For abstraction purposes, we use the following communication model for a cluster (with more details given in Section 2). We view the cluster (logically) as a fully connected graph. That is, any node i can send/receive packets to/from any other node j in the cluster. In a single time slot (as defined in Section 2), each node i can transmit one packet and receive one packet; a number of works use this model [6, 1, 10, 8] as a communication abstraction. The packets can arrive at a node in any order, but they must be played back in a specific order (and at a specific rate), corresponding to the original recording rate of the stream. The subset of the (fully connected) graph edges used for packet delivery form a mesh. The system's performance is a function of the algorithms used to construct and maintain this mesh as well as the algorithms used for scheduling packet delivery over the mesh. Hence, *our paper focuses on specific approaches to constructing such meshes within a single cluster.*

Our primary goal is to develop an understanding of two related quantities – playback delay and buffering requirements. As an example, one could simply chain the receiving nodes of a cluster (of size N) in a list, where the source streams packets to the first node in the list. Each node then simply forwards the packets to the next node in the list, and so on. While the buffering requirements are minimal in this approach, the playback delay is unacceptable for all but a few nodes, particularly since the cluster could be large. Another simple approach might be to arrange the nodes, for instance, in a binary in a tree, with the source being the root of that tree. Each node would then need to forward the packets to its two children. While this results in constant buffering requirements and $O(\log N)$ delay, this also requires each node to have at least twice as much upload bandwidth as bandwidth needed for downloading (i.e., streaming). (Note also that approximately half the nodes, i.e., leaves of the tree, are not contributing to the streaming process; hence the need for other nodes to make up for the lack of upload capacity in the system.) This is not a reasonable requirement as typically a node's upload bandwidth is significantly lower than its download bandwidth. Hence, better approaches to mesh construction are needed, which result in acceptable playback delay and buffering characteristics, while utilizing system resources efficiently.

To this end, in this paper we explore two approaches to mesh construction and subsequent streaming, one based on multi-trees and the other based on a hypercubes and generalization of [5] (which was designed for message broadcast). We use these approaches to explore the resulting playback delay, buffer space, and communication[1] requirements.

Specifically, we first adapt the scheme in [5] to streaming (as discussed in Section 3)

[1]By communication requirements we mean the number of neighbors with which a node needs to communicate in a particular scheme, as detailed later.

with $O(1)$ buffer space requirements and $O(\log N)$ playback delay. However, the more direct adaptation is done in the context of certain values of N (number of nodes in a cluster). In doing so, we observe that particular care must be taken in such an adaptation in order to limit the number of neighbors with which a node needs to communicate. Our motivation for limiting the number of neighbors with which a node communicates is that such communication requires protocol maintenance overhead, e.g.., due to "keep alive" messages, due to nodes joining and departing (under node churn), and so on. Thus, we extend this scheme further, such that (a) it works for *arbitrary* values of N while (b) *limiting* the number of neighbors with which each node needs to communicate to $O(\log N)$.

In the case of multi-tree based schemes, we perform streaming on a collection of d, d-ary *interior-disjoint*, trees where we show the playback delay to be at most $d \log_d N$ for all receivers. By interior-disjoint we mean that each of the receivers does not appear as an interior node in more than one of the d trees. Unlike the hypercube-based scheme described above, the multi-tree-based schemes only require each node to communicate with at most $2d$ nodes in its cluster[2].

Our approach has *provable quality-of-service (QoS) guarantees*, and we provide analysis of corresponding performance characteristics. Specifically, in this work we focus on *playback delay* and *buffer size requirements*, as our metrics of QoS, and we study them as a function of d and N.

Note that in this paper we focus on a more "structured" approach to mesh construction, i.e., the set of edges used for delivery of packets is fixed by our algorithms. An alternate approach might be to use an "unstructured" approach – i.e., the edges used for delivery are determined on a per packet basis (essentially on the fly when the data is needed). This allows the system to more easily adapt to node churn. However, existing unstructured approaches to *streaming* are essentially "best effort", and little exists in the way of formal analysis of resulting QoS guarantees. A number of very nice works have looked at formal analysis of unstructured approaches to *file downloads*, e.g., as in [6, 1]. Here the file is decomposed into k chunks, and then distributed using a randomized gossip mechanism. However, since we are streaming a very large (*potentially infinite*) number of packets, the arrival order of packets is important, otherwise they all have to be buffered. In addition, the chunks need to be all available initially, which is not the case for *live* streaming. Techniques with provable QoS guarantees, such as ours, may be more suitable for scenarios where QoS is of importance. Independently, [12] focuses on characterizing limits of peer-assisted live streaming (for "structured" and "unstructured" systems) and gives performance bounds (on minimum source capacity and tree depth, and maximum streaming rate) using a fluid flow (rather than packet) model and under different assumptions from ours (e.g., they assume a potentially unlimited source capacity, they do not constraint trees to be interior-disjoint, etc.).

The contributions of this work are as follows:

- We provide algorithms for constructing multiple streaming trees and a corresponding transmission schedule so as to maintain QoS characteristics (see Section 2). We are also able to extend our schemes to dynamic scenarios while maintain our nice tree properties. Due to lack of space, these results are given in the Appendix [7].

[2] As shown later, smaller values of d are more desirable for playback delay and buffer space requirements, and thus should result in a smaller number of neighbors with which a node needs to communicate.

- We analyze the QoS of the multi-tree-based schemes and derive an upper bound on the delay required at each node before it can start playback, and use it to bound the required buffer size; we also prove a lower bound on the average playback delay (see Section 2.3)[3].
- Interestingly, our work establishes that it is only useful to consider degree 2 and 3 trees, if one wants to minimize worst-case delay (see Section 2.3).
- We present algorithms for extending the scheme in [5] such that it works for streaming and for arbitrary values of N with a provable limit on the number of neighbors with which a node needs to communicate (see Section 3); we refer to this as a hypercube-based scheme.
- We analyze the hypercube-based schemes to give upper bounds on worst case and average case playback delay (see Section 3.2).
- Our main schemes are presented under the model of each cluster being a fully connected graph, i.e., where we assume that all nodes in a cluster are capable of communicating with each other. The existence of two interior disjoint trees is an NP-complete problem under arbitrary graphs. Due to lack of space, the NP-completeness proof is given in the Appendix [7].

Before presenting our algorithms, we give a brief overview of related literature. The original end-system multicast approach [3] explores the use of an application level multicast tree for streaming; however, it suffers from several shortcomings, including: (i) leaf nodes contribute no resources (and hence some system resources are essentially wasted), (ii) less resilience to node failures, (iii) less flexibility in use of bandwidth (e.g., bandwidth is used in units of needed streaming rate), and (iv) internal nodes require significantly higher upload bandwidth (than the streaming rate of a stream) in order to keep a multicast tree shallow (which is desirable as a single deep tree results in long startup delays and large buffer size requirements). Existing literature explores two directions in trying to overcome these shortcomings. One direction (as in our work) is the use of multiple trees, e.g., as in [2]. However, most such works focus on providing different quality of service to users with different capabilities as well as on adapting to failures and bandwidth heterogeneity, e.g., through the use of Multiple Description Coding (MDC). Our techniques can be combined with MDC as well, but we do not rely on their use. Other works, [14], address system dynamics at the cost of not maintaining nice tree properties (e.g., balance), whereas our schemes do maintain such properties. One generalization of [2] examines stability properties of the overlay [4] and, unlike in this work, considers QoS in a probabilistic setting. *Overall, the distinction of our effort, is that we provide provable QoS guarantees (such as startup delay).*

Another direction is to abandon the use of trees in favor of unstructured peer-to-peer (P2P) systems, e.g., [15], which allows for greater flexibility in dealing with system dynamics and heterogeneity. However, to date most such solutions have been fairly heuristic in nature and (as a result) difficult to analyze, from the perspective of performance as well as QoS guarantees. One notable exception is the analysis of BitTorrent in [1]. This work gives a very nice analysis which explains BitTorrent's success for *download* applications; however it is not clear how successfully it can be applied to streaming applications, where the data needs to

[3]We also evaluated our schemes through simulations; the results are omitted here due to lack of space.

be delivered in time for playback. That is, one can view a stream as a collection of small segments and apply the analysis in [1] (or in other works, e.g., [6]) to each segment. However, given the bound in [1], keeping up with real-time constraints would require an assumption of faster packet transmission than playback, an assumption we do not make here.

Lastly, a comparison study of multi-tree-based approaches and unstructured P2P schemes is given in [13]. One interesting difference, as compared to our results, is that the results in [13] suggest the use of higher degree trees, whereas our work shows optimality of lower degree trees (refer to Section 2.3). But, the main metric in [13] is bandwidth utilization under heterogeneous conditions, whereas we focus on playback delay. Overall, a high level conclusion from [13] is that unstructured aspects of the system allow for better utilization of heterogeneous resources and better adaptivity to node churn which lead to better QoS. We note that this study is (a) done for specific multi-tree and unstructured schemes, (b) assumes the use of MDC, and (c) is simulation based, i.e., no provable QoS guarantees are given. By contrast, the work here does derive provable QoS guarantees for a class of multi-tree based schemes, and thus is more useful to scenarios where QoS is of importance.

2 Multi-Tree Construction and Transmission

Let S be the source node of the streamed data (e.g., video and/or audio), which corresponds to a (potentially infinite) sequence of data units (also referred to as "packets") being delivered to a number of receivers. We assume that the network provides sufficient bandwidth, so that a packet can be delivered within a time slot. This is essential, if we are to have a solution without the use of (potentially) unbounded buffers. (If a receiver cannot receive a packet in each time slot, then it must accumulate a lot of packets before playback can start. Even with such a scheme, when the stream is infinite, eventually the receiver will starve and cause the playback to temporarily stop when the buffer is empty.) This assumption is quite reasonable. For instance, if we stream MPEG-1 video, recorded at the rate of 1.5 Mbps using 1400 byte packets (which is quite common), then each packet would play for ≈ 7.5 msec. If we stream these packets over a 10 Mbps connection, then it would take ≈ 1.1 msec to transmit one packet. If the propagation delay is also significant – e.g., a packet sent across US might experience a one-way delay on the order of 30 msec (including propagation, queueing, and processing delays) – then, we could think of transmitting a set of packets as one "large packet", in order not to waste network resources. Given above numbers, that would be on the order of 5 packets.

2.1 Construction of tree τ

We assume that there are K clusters, with each cluster having at most N nodes. As mentioned earlier, T_c is the transmission time to send a packet from a node in one cluster to a node in another cluster. Within a cluster we assume that the nodes form a complete subgraph, with the transmission time being T_i. For clarity of presentation (and without loss of generality), we will assume in the remainder (unless otherwise stated) that T_i is 1.

The source S has $D (D \geq 3)$ times the capacity of a receiver node. Moreover, in each cluster i there are two "super nodes", S_i and S'_i, where S_i has the same capacity as the source, S, and S'_i has capacity d times the capacity of a receiver node. Under these assumptions we construct the tree τ using the following algorithm:

Step 1: Build a tree using nodes S_1 through S_K, with S being the root of this tree. The degree of S is D; other interior nodes have degree at most $D-1$. In order to keep the tree tight, at most one interior node can have degree less than $D-1$, and this node must be in the next to the last layer.

Step 2: Make S'_i the child of S_i for all i.

Step 3: Within each cluster build degree d interior-disjoint trees with S'_i as the root of the cluster.

The data is distributed (in order) from S to all the nodes S_i. Each node S_i forwards the packet it receives from its parent on to its D children ($D-1$ children in tree τ and the child S'_i). An example of the resulting tree τ using this construction is depicted in Figure 1.

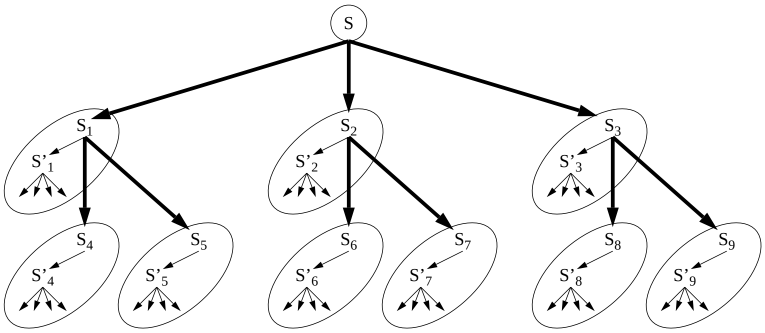


Figure 1: Cluster construction with source S, $D=3$, $d=4$: the ovals represent clusters, where thick and thin arrows represent inter- and intra-cluster transmission, respectively.

The main idea here is to use a set of "super nodes" as the "backbone" of our network. As a result, we have the following theorem:

Theorem 1 *The worst case playback delay is on the order of* $T_c \cdot \log_{D-1} K + T_i \cdot d(h-1)$, *where* h *is the maximum height of the interior-disjoint trees in all clusters.*

Proof: The $\log_{D-1} K$ term is due to the "backbone" tree, while the $d(h-1)$ term is given in Theorem 2. ⊠

What remains is to determine and evaluate tree construction and transmission schemes for each cluster. Thus (unless otherwise stated), in the remainder of the paper we focus our discussion on a *single cluster*. For clarity of presentation we use the term "source S" (or "root S") to mean the root S'_i of cluster i.

2.2 Construction of interior disjoint trees

Recall that the packets can be delivered out of order. However, since our data corresponds to continuous media (such as audio or video), the packets must be played in order *and* at the rate at which data was recorded – we assume that the playback rate is one packet per time slot. We define a time slot as the playback time of a single packet.

We also assume that the receivers are homogeneous and can transmit *and* receive one packet per time slot to other nodes in their own cluster. Similar models are used, for instance in [6, 1]. In practice, a node may send and receive more than one packet in a time slot, although still a bounded number[4]. The schemes we propose here work with either model. However, not all schemes which work under the latter model would carry over to the former. Since our goal is to develop an understanding of playback delay and buffer requirements inherent in such schemes, we use the former model.

Ideally, we would like each receiver to receive a new packet in every time slot. However, the source S does not have the capability to stream a new packet to each receiver in each time slot. We assume that S is powerful enough to send packets to up to d receivers in each time slot, where $d \geq 1$. (It is reasonable to assume that the source is a server which is slightly more powerful than the clients receiving the stream.)

Thus, the general scheme is then based on receivers themselves acting as senders of packets which they have just received. One approach would be to construct a single tree, with S as the root, and deliver the data along that tree, e.g., as in [3]. However, that would waste the upload capacity of the leaf nodes; e.g., in a binary tree approximately half of the upload capacity of the system is potentially wasted. This would also require internal nodes to have significantly higher upload capacity (e.g., twice the leafs' capacity in a binary tree), whereas most technologies actually provide higher download capacity. Thus, a more resource-efficient approach (e.g., as in [2]) is to construct multiple transmission trees where receivers can obtain a different fraction of the data stream from each of the trees. This leads to efficient use of resources and reduction in playback start-up delay as well as buffer space requirements. If a node splits its upload bandwidth among its d children, then in fact a node may belong to up to d trees. To maximize efficiency of resource usage, we focus on schemes where each node does belong to d trees. Thus, we choose trees so that each node is an interior node in at most one tree, in which it has exactly d children (a few dummy receivers may be added to ensure this). In the remaining $d-1$ trees it needs to be a leaf node. Constructing the d trees with this property is actually quite easy. What is surprising is that this can be done in a way that enables a schedule where in each time slot, each node will receive a packet from *exactly* one parent, with *no collisions*. Note that each node has d parents in the d trees. After receiving a packet j, each node then sends packet j to its d children in the next d time slots. Unlike the N receivers in the system, the source S distributes one packet in each time slot to a receiver in *each* of the d trees.

Thus, below we construct d trees, each being a d-ary tree, where S acts as the root in each of the trees and all N receivers appear in each tree. The data stream is then split among the d trees (e.g., the first packet might be delivered through the first tree, the second packet might be delivered through the second tree, and so on). The tree construction and transmission schemes we devise must then satisfy the following constraints: (1) each receiver node receives at most one packet in each time slot, (2) each receiver node transmits at most one packet in each time slot, (3) node S can transmit at most d packets in each time slot, and (4) after some finite amount of time, referred to as *playback delay*, each receiver node should be able to start playback and continue that playback without hiccups (i.e., without reaching a situation where the next packet to be played has not arrived yet).

[4]Essentially, this would correspond to a node splitting its bandwidth between multiple transmissions, each at a slower rate, with a longer time slot.

In what follows, we refer to receiver i, $1 \le i \le N$, as a receiver with node id i, in order to distinguish its "name" from the various positions it might occupy in the d trees. We number the positions in any tree in a breadth first order.

Intuitively, we would also like to construct these d trees in such a manner so as to reduce the playback delay. This implies that the trees should be, in some sense, balanced and that they should be interior node disjoint, i.e., that each of the N receivers should not appear as an interior node in more than one of the d trees. Let I be the number of interior nodes in a tree; then, $I = \lceil \frac{N}{d} \rceil - 1$. Let $G_0 = \{1, 2, 3, ..., I\}, G_1 = \{I+1, I+2, ..., 2I\}, \ldots, G_{d-1} = \{(d-1)I+1, (d-1)I+2, ..., dI\}, G_d = \{dI+1, dI+2, ..., N\}$. Node ids in G_0 to G_{d-1} correspond to those nodes which will appear as interior nodes in some tree. The node ids in G_d will always be leaf nodes. We refer to the j^{th} element of G_i as G_i^j.

For notational convenience, we would like to assume that each internal node in each of the d trees has exactly d children. We accomplish this by adding dummy receiver nodes. In the constructions that follow, we ensure that the dummy nodes only appear as leaf nodes in our trees (i.e., we add them into G_d). Thus, they can simply be removed in the real system. *The key point in the construction is that each node appears as the i^{th} child in only one tree.* This is what enables a transmission schedule with no collisions (see Figures 2 and 3).

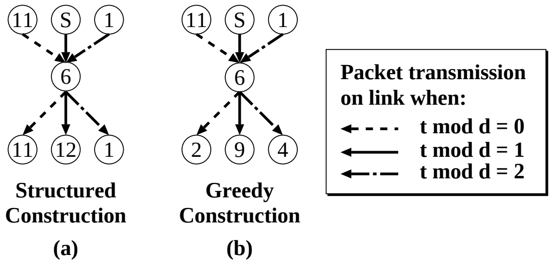


Figure 2: Receiving and sending schedules of node id 6, for example in Figure 3.

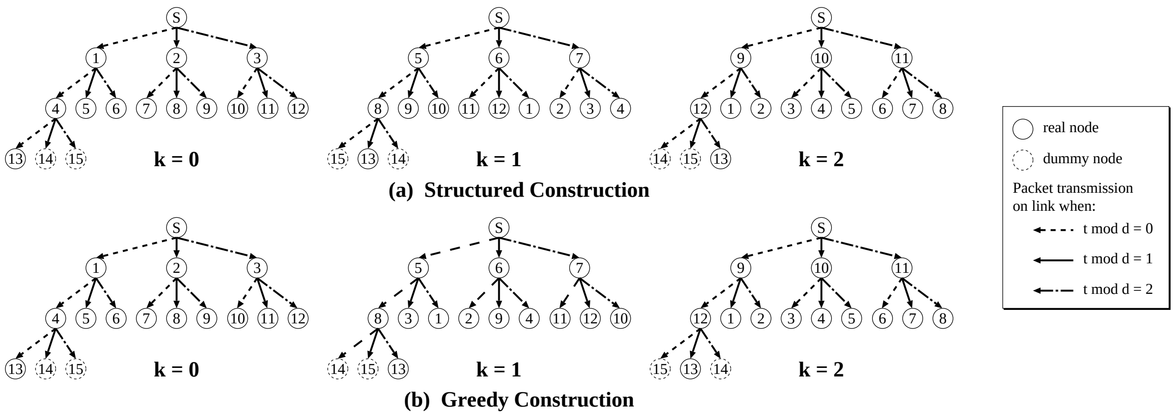


Figure 3: Example of interior disjoint tree construction using the two schemes with $N = 15, d = 3$, where $G_0 = \{1, 2, 3, 4\}$, $G_1 = \{5, 6, 7, 8\}$, $G_2 = \{9, 10, 11, 12\}$, $G_3 = \{13, 14, 15\}$

We describe two slightly different construction schemes. The essential properties achieved by both schemes are identical; however, the locations of the nodes themselves can be different. We now give tree construction and transmission schemes.

2.2.1 Structured Disjoint Tree Construction

Let $P = \frac{d}{\gcd(I,d)}$. We number the d trees $T_0, T_1, \ldots, T_{d-1}$. We construct these trees by filling in the nodes, in breadth first order, using the *current* ordering of the groups G_i, where the first group in the current order always corresponds to interior nodes. Specifically, let $\mathcal{G}$ be the concatenation ($\oplus$) of d elements.

Step 1: *Initialization:* Let $\mathcal{G} = G_0 \oplus G_1 \oplus \ldots \oplus G_{d-1}$. Construct T_0 using $\mathcal{G} \oplus G_d$. Let $k = 0$.

Step 2: *Construct group sequence for next tree:* Let $k = k + 1$. Construct a new $\mathcal{G}$ by rotating the current $\mathcal{G}$ to the left where G_i takes place of G_{i-1} and the first element of the current $\mathcal{G}$ becomes the last element of the new $\mathcal{G}$. If k mod $P \neq 0$, go to Step 4.

Step 3: *Adjust groups (after P rotations):* Construct new G_is, $0 \leq i \leq d-1$, by rotating elements of each G_i to the right, i.e., G_i^j takes place of G_i^{j+1} and the last element of the current G_i becomes the first element of the new G_i.

Step 4: *Construct next tree:* Rotate G_d to the right, i.e., G_d^j takes place of G_d^{j+1} and the last element of the current G_d becomes the first element of the new G_d. Construct T_k using $\mathcal{G} \oplus G_d$.

Step 5: *Loop:* If $k < d - 1$, go to Step 2.

Due to lack of space, we give the correctness proof for this construction in the Appendix [7]. The main idea behind the proof is to show that no node will receive more than one packet in one time slot.

2.2.2 Greedy Disjoint Tree Construction

In this scheme, for ease of exposition, we assign a parity to each receiving node, where node id i has parity $p_i = (i - 1 \bmod d)$, $i \in \{1, 2, 3, ..., N\}$. The node's parity determines which child slot the node occupies in each of the d trees. Specifically, node i with parity p_i occupies child slot $(p_i - k)$ mod d in tree k, where $0 \leq k \leq d - 1$. The scheme can then be described as follows:

Step 1: *Initialization:* Let $\mathcal{G} = G_0, G_1, \ldots, G_{d-1}$ and construct T_0 using $\mathcal{G}, G_d$. Let $k = 0$.

Step 2: *Interior node selection for next tree* Let $k = k+1$. All interior nodes of tree T_k are chosen from the set G_k where we fill in the nodes in a breadth first manner, for positions i, $i = 1, 2, \ldots, I$, by choosing the *smallest* node id j which satisfies the following conditions:

(a) $j \in G_k$,
(b) j has parity $i + k - 1$,
(c) j has not been placed in tree T_k yet.

Step 3: *Leaf node selection for next tree:* Let $\mathcal{L} = \{1, 2, 3, ..., N\}/G_k$ be the set of nodes that were not yet placed in tree T_k in Step 2. All leaf nodes are chosen from the set $\mathcal{L}$ where we fill in the nodes in a breadth first manner, for positions i, $i = I + 1, I + 2, \ldots, N$, by choosing the *smallest* node id j which satisfies the following conditions:

(a) $j \in G$,
(b) j has parity $i + k - 1$,
(c) j has not been placed in tree T_k yet.

Step 4: *Loop:* $k = k + 1$. If $k < d$ go to Step 2.

Due to lack of space, we give the correctness proof for this construction in the Appendix [7].

2.2.3 Transmission Schedule

The data streamed from S can either be produced live (e.g., as in a broadcast of a sporting event) or it can be pre-recorded (e.g., as in the case of delivery a movie). For ease of presentation, we first assume that we are delivering pre-recorded data, i.e., all packets are available at node S at time 0. It is a simple extension to make our schemes work for live streams, and we explain it briefly at the end of this section.

The transmission schedule can be obtained as follows. Let $k \in \{0, 1, 2, ..., d - 1\}$, and let t be a time slot, where $t = m \cdot d + r$ and $0 \leq r < d$. In time slot t, S transmits packet $(k + m \cdot d)$ to its r^{th} child in tree T_k. All other interior nodes in tree T_k send one packet to their r^{th} child in time slot t. (The children are numbered, left to right, from 0 to $d - 1$. Thus, the transmission essentially proceeds in a round-robin manner.) For example, in the multi-tree constructed in Figure 3, in time slot 0, S sends packet 0 to node id 1 in tree T_0, packet 1 to node 5 in tree T_1, and packet 2 to node 9 in tree T_2. Then, in time slot 1, S sends packet 0 to node 2 in tree T_0, packet 1 to node 6 in tree T_1 and packet 2 to node 10 in tree T_2. After receiving packet 0 from S in time slot 0 in tree T_0, node 1 will send packet 0 to node 5 in time slot 1, node 6 in time slot 2 and node 4 in time slot 3, etc.

In the case of live streaming, it is not possible to send packet 1 in time slot 0 because it has not been generated yet. One approach to address this problem would be to, in a sense, pipeline the packets and thus modify the schedule as follows. Let $r = (t + k) \bmod d$. Then, in time slot $t + k$, S transmits packet $(k + m \cdot d)$ to its r^{th} child in tree T_k, if $(k + m \cdot d) \leq t$. All other interior nodes in tree T_k send one packet to their r^{th} child in time slot t, as long as they have an appropriate new packet to send to that child. In this case, the transmission schedules of the different trees are not homogeneous; thus, this scheme is not easy to analyze.

Another approach to address this problem is for S to delay the streaming until it accumulates d packets. For ease of illustration we assume that the d packets were "pre-buffered" before time 0, and S can send out d packets at time 0. Thus, all nodes experience d units of additional delay as compared to the above described approach. However, we can then assume the same transmission scheduling procedure as in the case of "pre-recorded" streaming.

2.3 Delay Analysis

In this section we first give an upper bound on the overall playback delay and buffer space requirements, given the construction and transmission schemes of Section 2. We then use this bound to determine an optimal value for the degree of our trees, i.e., for d. Finally, a lower bound on the average delay is given. We do all this in the context of pre-recorded streaming applications.

Worst-case Playback Delay: We derive the upper bound on playback delay under the assumptions that: (1) a packet playback takes one time slot; (2) all d trees are complete, i.e.,

$d+d^2+...+d^h = N$ for some integer h – here $(h+1)$ is the depth of our trees; and (3) S begins packet transmission in time slot 0. These assumptions simplify the analysis significantly. (We have also performed a simulation-based evaluation of the delay characteristics without the assumption of complete trees; this is omitted here due to lack of space.) We now state the following theorem.

Theorem 2 *Worst-case playback delay, T, satisfies the following inequality: $T \leq h \cdot d$, where $h = \lceil \log_d [N(1-\frac{1}{d})+1] \rceil$ and $h+1$ is the depth of the trees.*

Due to lack of space, we give the proof of Theorem 2 in the Appendix [7].

Given this worst-case delay, a buffer of size $h \cdot d \cdot$ (size of a packet) is sufficient at every node. Note that this is an upper bound on the buffer space requirements, i.e., not all nodes need that much buffer space. For instance, in the multi-tree system constructed in Figure 3, node 1 will receive packets $0, 1$, and 2 in time slots $0, 2$, and 1, respectively. Therefore a buffer size of 3 is sufficient for node 1.

Tree Degree Optimization: Given that we would like to minimize worst-case delay (in Theorem 2), we can state the following. Asymptotically, we can show that for large N, degree 3 trees are optimal. Moreover, for any N, either degree 2 or degree 3 trees are optimal.

Specifically, let us assume that N is large. Then, a reasonable approximation for the upper bound on playback delay is $T \approx \log_d[N(1-\frac{1}{d})] \cdot d$. Let $F(d) = \log_d[N(1-\frac{1}{d})] \cdot d$. Then, using natural logs and taking the derivative with respect to d, we obtain $\frac{dF}{dd} = \frac{(\log d - 1)[\log(d-1) + \log N] + \frac{d}{d-1}\log d}{(\log d)^2} - 1$ where d must be an integer and $d \geq 2$. Note that, when $d = 2$, $\frac{dF}{dd} = \frac{(\log 2 - 1)\log N}{(\log 2)^2} + \frac{2}{\log 2} - 1 \approx 1.89 - 0.64 \log N < 0$. And, when $d \geq 3$, $\log d - 1 > 0$, so $\frac{dF}{dd} > 0$. Consequently, an optimal value of d should always be either 2 or 3. Note that $F(2) = 2(\log_2 N - 1)$ and $F(3) = 3(\frac{\log_2 N}{\log_2 3} - \log_3(3/2))$. Thus, for sufficiently large N (and these values do not have to be very large), degree 3 trees are optimal.

We note that numerical results depicted in Figure 4 (obtained through simulations) indicate that for small and large values of N, the resulting delays for degree 2 and 3 trees are quite close, and they are better than for higher degree trees. Thus, we believe that it is reasonable to use $d = 2$ in practice.

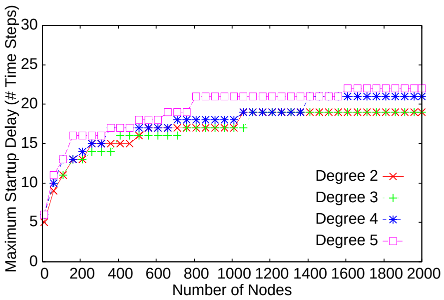


Figure 4: Worst-case delay

Average Playback Delay: In addition to the worst-case playback delay, average playback delay, $\frac{\sum_{i=1}^{N} a(i)}{N}$, is also an important metric for evaluating the performance of our scheme; here $a(i)$ corresponds to the playback delay of node id i. However, this is a difficult metric to derive analytically. Instead Theorem 3 gives a lower bound on this metric, under the same assumptions stated above (due to lack of space, the proof is given in the Appendix [7]).

Theorem 3 *The following inequality gives a lower bound on the average playback delay:* $\frac{\sum_{i=1}^{N} a(i)}{N} \geq \frac{d^h(d+1)(h-1)-d^2(h-2)-d(d+1)/2}{N(d-1)}$.

3 Hypercube-based Scheme

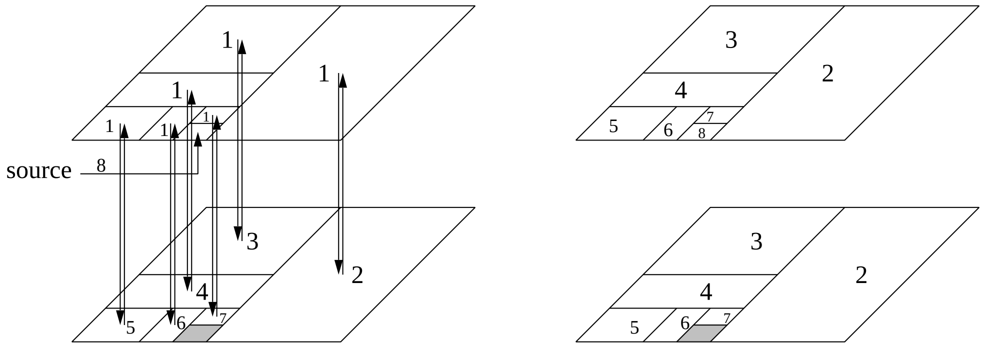
In this section we present another approach to mesh construction for streaming purposes; we refer to this scheme as *hypercube-based streaming* (for reasons made clear below). For simplicity of presentation we focus on streaming within a single cluster. However, this scheme can be easily adapted to streaming over multiple clusters, using the tree τ, as in the context of multi-trees. All assumptions remain the same as in the context of multi-trees (Section 2), except that, at first, we do not assume that the source S has a greater transmission capability than any other node. Towards the end of the section we comment on how to adjust the results if the same assumption is made as in the case of multi-trees, i.e., if the source S has d times the capacity of a receiver node.

3.1 Hypercube Streaming for Special N

For ease of illustration, we first present a streaming scheme under the assumption that the number of nodes, N, is one less than a power of two. That is, let $N = 2^k - 1$, where k is an integer. We can construct a scheme with $O(1)$ buffer size requirement and $O(k)$ playback delay (using a generalization of [5]) by reaching a state in which $\frac{N}{2^i}$ nodes have the i^{th} packet ($i = 1 \ldots k$). In the next round, we can have all (remaining) nodes receive packet 1 and at the same time double the nodes having packets i, $i = 2 \ldots k$, while the source sends out a new packet $(k+1)$. Packet 1 can now be consumed, and the protocol repeats. We take a somewhat different approach than in [5, 11] as we are designing our scheme for streaming rather than for message distributions. Moreover, in [5, 11], the number of messages is limited to a finite number, say m; thus after m time slots the source can aid in packet exchange without having to send new packets. However, in streaming (and particularly live streaming which is potentially infinite) this is often not possible as the source always has new packets to send.

Figure 5 depicts a simple example of our scheme. The main problem with this scheme, as described above, is that it can potentially involve an arbitrary communication pattern, where a node may actually need to communicate with *all* other nodes. (Due to lack of space, an illustration of such a possible communication pattern is given in the Appendix [7].)

To limit the number of neighbors with which a node communicates, we construct a specific communication pattern as follows. For simplicity let the source S have node ID 0. In order to transmit packets, in each time slot we pair up the $N+1$ nodes (i.e., including the source) and have them exchange packets as follows. Let $n = 0, 1, 2, 3, ..., j = 0, 1, 2, ..., k-1$. In time slot $kn + j$, we pair up nodes with IDs $(xx...x0xx...x)_2$ and $(xx...x1xx...x)_2$, where 0 and 1 appear in the $j + 1^{st}$ position from the right. (Here we use "$()_2$" to indicate a node ID written in binary form.) Then, the $N+1$ nodes can be viewed as vertices of a k dimensional



(a) beginning of timeslot X (b) end of timeslot X

Figure 5: (a) depicts the beginning of time slot X, where each parallelogram corresponds to a group of nodes which transmit, the packet number indicated (by the number inside the parallelogram); the arrows indicate the direction of transmission (i.e., sender/receiver relationship between groups of nodes). The shaded parallelogram indicates an absence of a node. Note that the scheme is depicted for $N = 2^k - 1$ nodes, where one node is receiving the next packet (here, packet 8) from the source. (b) depicts how many nodes have packet i, $2 \leq i \leq 8$, at the end of time slot X (each node also has packet 1, so we omit that from the figure), i.e., we have doubled the number of nodes which have packet i as compared to the beginning of the time slot.

hypercube[5], where in each time slot, communication between vertices is performed along the same dimension. For instance, suppose we have 7 nodes, plus a source, with node IDs 0 to 7. Then, (a) in time slot $3n + 1$ we pair up nodes with IDs $(xx0)_2$ and $(xx1)_2$, i.e., we pair up node IDs 0, 2, 4, and 6 with node IDs 1, 3, 5, and 7, respectively; (b) in time slot $3n + 2$ we pair up node IDs $(x0x)_2$ and $(x1x)_2$, i.e., we pair up node IDs 0, 1, 4, and 5 with node IDs 2, 3, 6, and 7, respectively; (c) in time slot $3n$ we pair up node IDs $(0xx)_2$ and $(1xx)_2$, i.e., we pair up node IDs 0, 1, 2, and 3 with node IDs 4, 5, 6, and 7, respectively, and so on. (Due to lack of space, we give a corresponding depiction in the Appendix [7].) Then, the performance of our scheme is described by Proposition 1.

Proposition 1 *Given $N = 2^k - 1$, under the hypercube streaming scheme, where nodes are arranged as vertices of a k-dimensional hypercube, each node only communicates with k other nodes and can begin playback after time slot $k+1$. Moreover, each node is only required to store 2 packets in its buffer, i.e., this scheme has $O(1)$ buffer space requirements.*

3.2 Hypercube Streaming for Arbitrary N

We now extend our hypercube-based streaming approach to arbitrary values of N, where the basic idea is to divide the N nodes into multiple hypercubes. Let $k_1 = \lfloor \log_2(N + 1) \rfloor$. Then, we construct a hypercube from the source S and $N_1 = 2^{k_1} - 1$ nodes; we refer to this hypercube as HC_1. In HC_1, in each time slot, the node receiving data from node ID 0 (i.e., the source) has nothing to send. Thus, this spare capacity can be utilized to send packets to a node in another hypercube. Specifically, let $n = 0, 1, 2, ..., j = 0, 1, 2, ..., k_1 - 1$; then

[5]We would like to thank Matt McCutchen for suggesting that we consider hypercube communication.

in time slot $nk_1 + j$, the node with ID 2^j receives a new packet from S, while consuming another packet (previously received from another node). Since this node has nothing to send to a node within HC_1 (refer to Figure 5), we let this node send the packet it just consumed to another hypercube. As a result, HC_1 as a whole can be viewed as a logical source for the remaining $N - N_1$ nodes, as it has the capacity to send one packet (in appropriate order), in every time slot. The one difference here is that this logical source begins sending packets in time slot $k_1 + 1$. Given the logical source HC_1 and $N - N_1$ nodes, we can repeat the above process until all nodes are assigned to some hypercube. The performance of the resulting scheme is described by Proposition 2.

Proposition 2 *Given an arbitrary N, in the hypercube-based scheme each node communicates with at most $O(\log N)$ other nodes. The corresponding worst case playback delay is at most $O(\log^2 N)$, where only two packets need to be stored in each node's buffer.*

In scenarios where we stream over multiple hypercubes, nodes start playback in different time slots. Thus, the average delay (average playback start time) is also of interest and is characterized in Theorem 4.

Theorem 4 *Hypercube-based streaming over N nodes (for arbitrary values of N) results in an average delay of no more than $2 \log N$.*

Due to lack of space, the proof of Theorem 4 is given in the Appendix [7].

Lastly, in Section 2, we assumed that the source can send d packets in one time slot. Under this assumption, the hypercube-based scheme can be adjusted as follows. We can divide N nodes (as evenly as possible) into d groups, and then apply hypercube-based streaming to these groups individually. Since each group has at most $\lceil \frac{N}{d} \rceil$ nodes, the worst case and average playback delay would be bounded by $O(\log^2(\frac{N}{d}))$ and $2\log\lceil \frac{N}{d} \rceil$, respectively, with each node communicating with $O(\log\lceil \frac{N}{d} \rceil)$ other nodes.

4 Summary and Future Work

We summarize the comparison between hypercube-based and multi-tree-based streaming in Table 1. As can be seen from this table, the multi-tree-based scheme provides better worst case playback delay but requires larger size buffers. Moreover, in the multi-tree-based scheme each node communicates with a constant number of nodes, while in the hypercube-based scheme communication is needed with $O(\log N)$. This is particularly important, given that small values of d are preferable in the multi-tree scheme.

Schemes	Max Delay	Ave Delay	Buffer Size	Num of Neighbors
Multi-tree	$O(d \log N)$	$O(d \log N)$	$O(d \log N)$	$O(d)$
hypercube for special N	$O(\log N)$	$O(\log N)$	$O(1)$	$O(\log N)$
hypercube for arbitrary N	$O(\log^2(\frac{N}{d}))$	$O(\log(\frac{N}{d}))$	$O(1)$	$O(\log(\frac{N}{d}))$

Table 1: Comparison between multi-tree based streaming and hypercube based streaming.

Our current and future efforts are focused on several directions. In this paper, our algorithms are given in a static context, i.e., where all N nodes are present in the system initially and for the duration of the data delivery process. However, in a real world streaming system nodes arrive and depart throughout the streaming process, i.e., there is node churn. Effective algorithms are needed for multi-tree-based and hypercube-based schemes to adjust to node dynamics with as little affect as possible on the remaining participating nodes. Due to lack of space, we give our adaptation of the multi-tree-based scheme to node dynamics in the Appendix [7]. Our ongoing efforts include constructing algorithms for dealing with node dynamics in the context the hypercube-based scheme, which would work well *for arbitrary values of* N. In addition, the maximum playback delay of the hypercube-based scheme is $O(\log^2 N)$, for arbitrary values of N. As N grows, this can be quite large. Our current efforts are focused on determining whether there exists an algorithm, such that *for arbitrary* N, it has the following characteristics: a maximum playback delay of $O(\log N)$, buffer space requirements of $O(1)$, and communication requirements of $O(\log N)$ (i.e., each node only needs to communicate with at most $O(\log N)$ other nodes in the system).

References

[1] D. Arthur and R. Panigrahy. Analyzing the efficiency of bittorrent and related peer-to-peer networks. In *ACM-SIAM SODA*, 2006.

[2] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[3] Y. Chu, S. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE JSAC*, 20(8), 2002.

[4] G. Dan, V. Fodor, and I. Chatzidrossos. On the performance of multi-tree-based peer-to-peer live streaming. In *IEEE INFOCOM*, 2007.

[5] A. M. Farley. Broadcast time in communication networks. *SIAM Journal on Applied Mathematics*, 39(2):385–390, 1980.

[6] C. Fernandess and D. Malkhi. On collaborative content distribution using multi-message gossip. In *IEEE IPDPS*, 2006.

[7] http://bourbon.usc.edu/leana/ipdps2009appendix.pdf.

[8] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *IEEE FOCS*, 2003.

[9] S. Khuller, Y. Kim, and Y-C. Wan. Broadcasting on networks of workstations. In *ACM Symp. on Parallel Algs. and Arch.*, 2005.

[10] S. Khuller, Y. Kim, and Y-C. Wan. On generalized broadcasting and gossiping. *Journal of Algorithms*, 59(2):81–106, 2006.

[11] Arthur L. Liestman, Thomas C. Shermer, and Matthew J. Suderman. Broadcasting multiple messages in hypercubes. In *ACM ISPAN*, 2000.

[12] S. Liu, R. Zhang-Shen, W. Jiang, J. Rexford, and M. Chiang. Performance bounds for peer-assisted live streaming. In *SIGMETRICS*, 2008.

[13] N. Magharei, R. Rejaie, and Y. Guo. Mesh or multiple-tree: A comparative study of live p2p streaming approaches. In *IEEE INFOCOM*, 2007.

[14] V. Venkataraman and P. Francis. Chunkyspread: Multi-tree unstructured peer-to-peer multicast. In *5t IPTPS*, 2006.

[15] X. Zhang, J. Liu, B. Li, and T. P. Yum. Coolstreaming/donet: A data-driven overlay network for efficient live media streaming. In *IEEE INFOCOM*, 2005.