DagStream: Locality Aware and Failure Resilient Peer-to-Peer Streaming *

Jin Liang and Klara Nahrstedt
University of Illinois at Urbana-Champaign
{jinliang, klara}@cs.uiuc.edu

ABSTRACT

Live peer to peer (P2P) media streaming faces many challenges such as peer unreliability and bandwidth heterogeneity. To effectively address these challenges, general "mesh" based P2P streaming architectures have recently been adopted. Mesh-based systems allow peers to aggregate bandwidth from multiple neighbors, and dynamically adapt to changing network conditions and neighbor failures. However, a drawback of mesh-based overlays is that it is difficult to guarantee network connectivity in a distributed fashion, especially when network locality needs to be optimized. This paper introduces a new P2P streaming framework called DagStream, which (1) organizes peers into a directed acyclic graph (DAG) where each node maintains at least k parents, thus has provable network connectivity (and hence failure resilience), and (2) enables peers to quickly achieve locality awareness in a distributed fashion, thus ensures efficient network resource usage. Our experiment results in both simulation and wide area environment show that with our DagStream protocol, peers can quickly self-organize into a locality aware DAG. Further, by selecting additional parents as needed, peers can achieve good streaming quality commensurate with their downlink bandwidth.

1. INTRODUCTION

Peer to peer (P2P) streaming has become a promising approach to broadcasting non-interactive live media from one source to a large number of receivers. By utilizing the residue bandwidth of participating peers to forward media data for each other, P2P streaming mitigates the load on the streaming server. This is especially attractive for live media broadcasting such as Internet TV and conference broadcasting, which would otherwise require huge amount of bandwidth from the broadcasting servers. Designing a live P2P streaming system, however, faces many challenges. First, peers are unreliable, which means they may join or leave the system at any time. Second, unlike dedicated servers, peers are limited in their uplink bandwidth, thus one supplying peer may or may not be able to forward a full video stream. In addition, peers have widely different downlink bandwidth. Therefore, it would be desirable if different peers can receive different streaming quality according to their downlink bandwidth.

Early P2P streaming systems[1–3] are designed as alternatives to IP multicast. Therefore, they all attempt to build application level multicast trees. However, a tree structure is unsuited in a P2P environment. First, a tree is vulnerable to node failures. If an interior node in the tree fails or leaves the system, the subtree rooted at the node will be affected. Second, the streaming rate of a peer cannot exceed that of its parent. This means if there is a bottleneck link higher in the tree, the bandwidth of all the downstream peers will be limited.

To address the problem of tree based architectures, several recent systems[4–7] have adopted a *multi-parent, receiver-driven* approach for P2P streaming. The idea is to allow each peer to stream media data from multiple neighbor peers. As a result, the streaming rate of a peer is the aggregate rate provided by its neighbors. To coordinate the streaming from multiple sources, a pull-based approach is used. Instead of the sending peer pushing data to the receiving peer, the receiving peer collects data availability from its neighbors, and request different data blocks from different neighbors. This allows a peer to dynamically adapt to the network conditions and neighbor failures.

While the multi-parent, receiver-driven approach offers great flexibility in dealing with peer and network dynamics, a remaining question is how should peers select their neighbors and what does the resulting overlay

*This work was in part supported by NSF ANI grant 03-23434.

look like? In all existing systems,[4, 6, 7] peers select neighbors based on their locally perceived performance, and the resulting overlays are often characterized as unstructured, mesh-based networks. No attempt has been made in these systems to ensure good overall connectivity. Unfortunately, without explicit mechanisms, overlays with poor connectivity can be formed. For example, none of the existing systems can prevent the formation of an overlay network as shown in Figure 1, where peer p_2 (perhaps a powerful peer) is a single point of failure, even though each peer has at least two neighbors. If p_2 fails, peers p_4 and p_5 would be disconnected from the network. Although network partition can be detected once it occurs, by observing that no new data packets are coming, repairing the partition nonetheless takes time and thus affects the streaming quality.

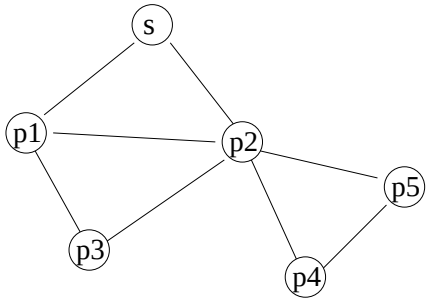


Figure 1. Overlay network with poor connectivity (failure resilience)

The network connectivity problem becomes more serious, if network locality needs to be optimized. This is because there could be a group of peers that are close to each other, but are far from other peers in the system. If they preferably connect to each other, there could easily be a network partition. Locality awareness is an important performance metric for P2P streaming systems. Without network locality, high bandwidth media streams would go back and forth across the wide area network, which not only increases the delay in receiving the media data, but could easily lead to network congestion. *For efficient network resource usage, we would like peers to stream media data from nearby peers as much as possible, and to stream from remote peers only when necessary.*

In this paper, we present a new peer to peer streaming framework called DagStream. Similar to mesh-based systems, DagStream allows multi-parent, receiver-driven streaming, thus can effectively deal with peer and network dynamics. Different from mesh-based systems, however, DagStream organizes peers into a directed acyclic graph (DAG) where each peer (except the source and its direct children) maintains at least k parents. As a result, DagStream has the property that the departure of any fewer than k peers will not cause any remaining peers to be disconnected from the source.

Based on the connectivity property, we design DagStream protocols that enable peers to improve locality awareness in a distributed fashion, while at the same time maintain good network connectivity. We have implemented our DagStream protocols and experimented in both simulation and wide area (PlanetLab[8]) environment. Our experiment results show that by exploring different parent discovery techniques, including a QoS aware membership service, DagStream enables peers to self-organize into a locality aware DAG in a quick fashion. Further, by allowing peers to select additional parents as needed, peers can achieve good streaming quality that is commensurate with their downlink bandwidth.

In the rest of the paper, we first discuss background and related work in Section 2. We then describe our system model and problem statement in Section 3. Section 4 presents the detailed design of our DagStream protocol, and Section 5 provides our evaluation results. Finally Section 6 is the conclusion.

2. BACKGROUND AND RELATED WORK

2.1. P2P Streaming Architectures

Early P2P streaming systems such as ESM,[1] NICE[2] and Zigzag[3] build application level multicast trees for streaming. ESM first builds a densely connected mesh among the peers, a per source multicast tree is then

constructed on top of the mesh. NICE and Zigzag organize the peers into a hierarchy of clusters to reduce the control overhead. In NICE, the multicast tree is implicitly defined by the control hierarchy, while in Zigzag, a set of rules is used to define the multicast tree.

Some systems such as CoopNet[9] and SplitStream[10] build multiple trees for streaming. The goal is to balance the forwarding load on different peers. Each peer is an interior node in only one tree, thus contributes bandwidth in only that tree. Coupled with scalable source encoding such as layered encoding or multiple description coding, multi-tree solutions can also address the bandwidth heterogeneity problem. Each peer can join as many trees as its downlink bandwidth allows. In this regard, DagStream is complementary to the multi-tree approach. If the source media is encoded into multiple layers or descriptions, we can easily build multiple DAGs for the streaming, one for each layer or description. Each individual DAG still allows the flexibility of multi-parent, receiver-driven streaming.

Many recent systems such as DONet,[4] PRO[6] and Chainsaw[7] build general unstructured meshs for P2P streaming. Mesh-based architectures allow the multi-parent, receiver-driven approach for P2P streaming, which is necessary in dealing with peer dynamics and bandwidth heterogeneity. However, none of these systems provide explicit mechanisms to achieve both good network connectivity and locality awareness.

We are not the first to consider DAG for P2P streaming. In particular, the Dagster system[11] has proposed to organize peers into a DAG for streaming. However, the focus of Dagster is to design incentive schemes that motivate peers to donate their bandwidth. Network locality was not a goal of Dagster. Also, the target application of Dagster is relatively small, and centralized algorithms are used for DAG construction. In contrast, the goal of DagStream is to design distributed protocols that enable peers to quickly self-organize into a locality aware DAG, while at the same time ensure good network connectivity.

2.2. Membership Management for Overlay Construction

Overlay network construction is largely dependent on the membership information, i.e., which peers are currently in the system, and what are their characteristics such as inter-node delay and available bandwidth. Some systems such as ESM,[1] CoopNet[9] and Dagster[11] build their overlays based on global membership information. Using global information has the benefit that efficient overlays can be built. However, maintaining global membership information is not scalable for live P2P streaming, which may involve large number of peers.

In other systems such as NICE[2] and Zigzag,[3] membership information is embedded in the overlay structure. Whenever a peer joins, it probes existing peers in the overlay, starting from a shared contact point, until it finds its best position in the overlay. One drawback of this approach is that when the number of peers is large, a joining peer may need to probe many peers before locating its position in the overlay, which causes long delays in joining the system.

Many recent systems such as DONet,[4] PRO,[6] and later version of ESM[12] have adopted a gossip-style protocol for membership dissemination. The idea is that each peer only maintains a small list of other members in the system. The list is periodically updated by exchanging some entries with randomly selected peers. When the upper layer needs to select a neighbor, a random peer is returned from the local list.

Although gossip style protocol achieves great scalability in membership management, it may not be suitable for QoS aware neighbor selection. This is because a randomly selected peer in a large system is unlikely to have the desired QoS characteristics (e.g., delay and bandwidth). For the purpose of QoS aware neighbor selection, it is desirable if potential good neighbors are selected with higher probabilities.

The quality of neighbor selection can be improved by clustering peers based on their QoS characteristics, and by selecting neighbors from specific QoS clusters. This idea has been explored in previous work.[13] However, in that work, peers are divided into different clusters based on their network proximity, and one peer in each cluster is designated as the membership server for the cluster. This adds additional complexity and overhead to the application peers, especially when the membership servers are themselves unreliable. In our recent work, we have built a membership service called RandPeer.[14] RandPeer manages membership information on behalf of P2P applications, and allows the lookup of peers based on their QoS characteristics, thus greatly facilitates the development of P2P applications.

Table 1. Notations

k	minimum number of parents that a peer should maintain
P_{max}	maximum number of parents that a peer can select
C_{max}	maximum number of children that a peer can accept
l_p	level of peer p
l'_p	min level of peer p
d	delay threshold
T_R	refresh period
T_E	evolvement period

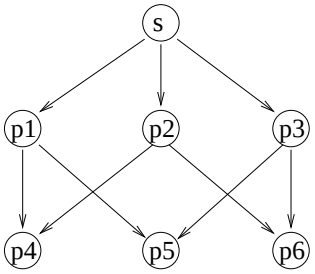


Figure 2. DagStream model

3. MODEL AND PROBLEM STATEMENT

3.1. DagStream Model and Connectivity Property

We model a P2P live streaming system as a directed acyclic graph (DAG), as is shown in Figure 2. The broadcasting server (the source) is the root of the DAG, and each peer is a node in the DAG. An arc from node p_i to p_j means that p_j is streaming data from p_i. p_i is called the parent of p_j, and p_j is the child of p_i. Each peer can only stream data from its parents, and can only provide data to its children. We assume each peer will maintain at least k, but at most P_{max} parents, and each parent can accept at most C_{max} children. (The notations used throughout the paper are listed in Table 1.)

A DAG enforces a partial order among the peers. This partial order can be captured by the *level* of a peer. Each peer p has a level l_p. The level of the source is $l_s = 0$. For a peer p whose parent set is P, the level of p is defined as follows:

$$l_p = \max_{p_i \in P} l_{p_i} + 1.$$

I.e., the level of a peer is 1 plus the largest level of its parents. The level of a peer p has another meaning. It is the length of the longest path from the source s to p. To describe the shortest path from s to p, we can similarly define the *min level* l'_p of a peer p as follows:

$$l'_p = \min_{p_i \in P} l'_{p_i} + 1.$$

In general mesh-based systems, the overlay network is often modeled as an undirected graph, each peer can send data to, and receive data from any other peer. Although mesh-based overlays provide great flexibility in P2P streaming, they make it difficult to ensure good network connectivity. In contrast, DagStream enforces a partial order among the peers and forbids loops. As a result, it has the following property.

DAG Connectivity Property: In a directed acyclic graph with one source, if each node (except the source and its direct children) has at least k parents, then the removal of any $k-1$ non-source nodes does not cause any remaining nodes to be disconnected from the source.

Proof: Suppose there exists a set P of $\leq k-1$ non-source nodes, and the removal of P causes the network to partition into two components. Consider the component C that does not include the source. There must exist a node p_i in this component, which does not have in degree in the component. Otherwise, if every node has at

least one in degree, there would be a cycle in the component. Since p_i doesn't have in degree in component C, in the original DAG, its parents must all be in the set P. However, set P only has $\leq k-1$ nodes, while p_i has at least k parents. This means the removal of P couldn't have partitioned the network.

The above property means that so long as each peer (except the direct children of the source) maintains at least k parents, the overall network will be well connected. Thus peers can focus on improving their performance such as network locality, without worrying about being disconnected from the source.

3.2. Multi-Parent, Reciever-Driven Streaming

Since each peer in DagStream can have multiple parents, we use the multi-parent, receiver-driven approach for media streaming. Specifically, each parent will notify its children about the data blocks available at this parent, and each child will apply some scheduling algorithm and request different blocks from different parents. The data block scheduling is based on the current data availability and available bandwidth from each parent. Thus it enables peers to dynamically adapt to changing network conditions and parent failures.

The data block scheduling is an interesting problem in multi-parent, receiver-driven P2P streaming. However, in this paper, we focus on how to build the underlying DAG in a distributed fashion. We leave the study of block scheduling algorithms as our future work.

3.3. Problem Statement

The streaming quality of a peer is often determined by their receiving rate. Thus a common approach in P2P streaming is to optimize the bandwidth that a peer can receive. Focusing on bandwidth, however, might cause locality unawareness. In DagStream, we exploit the flexibility provided by multi-parent, receiver-driven streaming and take a *locality first, bandwidth second* approach. Specifically, peers will first choose nearby peers (those with smaller delays) as parents. If these parents cannot provide the necessary bandwidth, more parents (which are farther away) can be selected. In this way, peers will try to stream media data from nearby peers as much as possible. And remote peers are contacted only when necessary.

In addition to the delay to the parents, the level of the parents should also be taken into account. This is because the buffering at each overlay hop might cause some delay in the playback time. To minimize the playback delay, we would like to connect to parents with small levels.

Therefore, in this paper, we focus on designing protocols that enable peers to locate nearby and small level parents in a quick fashion, while at the same time ensure global network connectivity. Due to the dynamic nature of P2P streaming systems, we would like our protocol to be scalable and robust. In particular, our protocol should be completely distributed and involve no complex interactions among multiple peers.

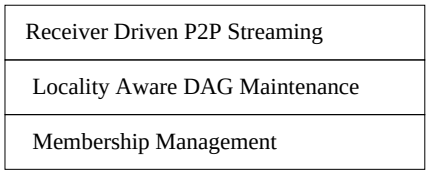
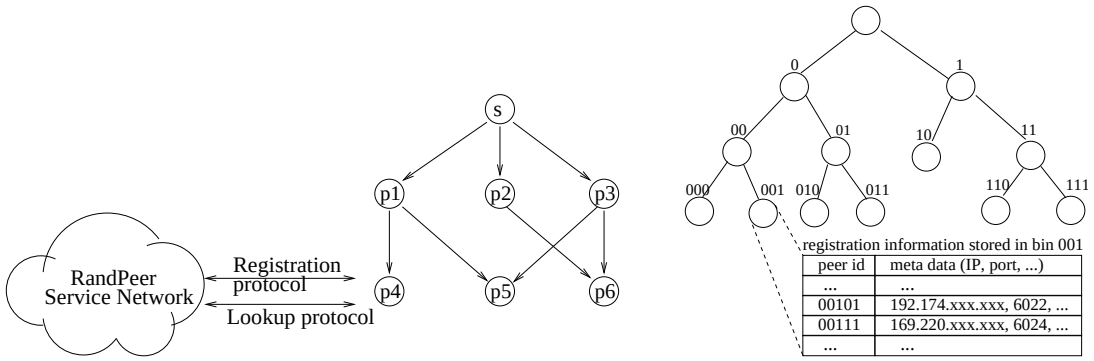


Figure 3. Layered design of DagStream peers

4. DESIGN OF DAGSTREAM

In this section we present the detailed design of DagStream. Figure 3 shows the layered architecture of each DagStream peer. There are three layers. The lowest layer is responsible for membership management. It provides candidate parents for the DAG maintenance layer. Our DagStream design makes use of the RandPeer[14] membership service. Thus the membership layer is simply some stub code for accessing the service. However, the membership layer can be replaced by other mechanisms such as gossip based protocols. The DAG maintenance layer is responsible for maintaining a locality aware DAG, where each peer (except the source and its direct children) has at least k parents. Given the DAG structure, the P2P streaming layer is responsible for advertising



(a) DagStream using RandPeer

(b) Membership trie

Figure 4. RandPeer membership service

the data availability to the children, and requesting data blocks from the parents. In the following, we first briefly describe the RandPeer membership service and how it achieves QoS aware neighbor selection, then present the detailed design of our DagStream protocols.

4.1. RandPeer Membership Service

RandPeer is a membership service that manages membership information on behalf of P2P applications. Figure 4(a) shows how DagStream makes use of the RandPeer service. Each peer in DagStream (including the source) periodically registers their membership information with the RandPeer service using a registration protocol[†]. Whenever a peer (e.g., p_4 in Figure 4(a)) needs to locate a parent, it sends a lookup request to the RandPeer service. In response, the membership information of some registered peer is returned. In its basic form, RandPeer allows the selection of a random peer from the entire system. Applications can customize the service so that peers with some desirable QoS characteristics (e.g., peers that are close by) are returned.

Internally, RandPeer uses a distributed trie data structure for membership management. Figure 4(b) shows an example membership trie. A trie is basically a tree with its nodes labeled with 0, 1 strings [‡]. The label of the root is an empty string. The label of a non-root node is the label of its parent concatenated with a 0 or 1, depending on whether it is a left or right child. The nodes in the trie are called "bins". Membership information of the peers are stored in the leaf bins.

To register its membership information, peer p_i needs to select a random 0, 1 string as its *peer id*. The peer id determines which leaf bin stores its membership entry. Specifically, the membership entry of peer p_i is stored in the leaf bin whose label is a prefix of the peer id of p_i. Each leaf bin has a capacity, if there are too many or too few membership entries in a leaf bin, it can be split or merged. The membership entries are soft state. Peers must periodically refresh their registration information. This also allows them to detect any collision in peer ids.

To lookup a candidate parent, peer p_i can generate a random *lookup key* and send a lookup message to the leaf bin whose label is a prefix of the lookup key. The leaf bin will return the membership information of the peer whose peer id immediately follows the lookup key.

RandPeer supports QoS aware neighbor selection by clustering peers based on their QoS characteristics. Specifically, peers can map their QoS characteristics (such as geographical location) to a "QoS prefix" in their peer ids. Thus peers with the same QoS prefix will be automatically clustered under the same subtree in the membership trie. When a peer wants to lookup a neighbor with certain desired QoS characteristics, it can

[†]RandPeer is a distributed service that consists of different service nodes. We assume each peer knows about one RandPeer node via some out of band information, in a way similar to using the domain name service (DNS).

[‡]For simplicity, we only discuss binary tries.

generate a lookup key with the specific QoS prefix. The result of such a lookup is likely to be a peer with the desired characteristics. In DagStream, since network locality is our primary goal, we let each peer generate a landmark vector for itself using the landmark binning technique[15] and use the landmark vector as its QoS prefix. When a peer wants to lookup a nearby peer, it uses its own landmark vector as a prefix in the lookup key. Thus the lookup result is likely to be a peer that is nearby.

4.2. Basic DagStream Protocol

Peer join and leave: When a peer p_i initially joins the system, it selects a target peer p_j using the RandPeer service. p_i then sends a `Connect` message to p_j. If p_j currently has less than C_{max} children, it will reply with a `ConnectAck` message. Otherwise it will reply with a `ConnectRej` message. If the joining request is accepted, p_i sets its level to $l_{p_j}+1$ and begins to stream media data from p_j. At the same time it begins the failure detection and continuous evolvement process as described below. If the joining request is rejected, p_i will select a new target peer and repeat the above process. When a peer leaves the system, it just disconnects from its parents and children. A disconnection message can be optionally sent for explicitly notification.

Failure detection: During the streaming, a peer will periodically (every T_R seconds) send a `Refresh` message to its children. The `Refresh` message indicates that the parent is still alive. It also contains some state information such as the current level of the parent. If the level of the parent has changed, its children must recompute their own levels. If no `Refresh` message was received from a parent for several consecutive periods, the parent is considered dead and removed. If all parents are removed, a peer is said to be disconnected. A disconnected peer must re-join the system following the joining process described above. A child receiving the `Refresh` message should reply with a `RefreshAck` message. If no `RefreshAck` was received from a child for several periods, the child is removed.

Continuous evolvement: During the streaming, each peer p_i will periodically (every T_E seconds [§]) select a target peer p_j and send a `Probe` message to p_j. p_j should reply with a `ProbeAck` message. The `ProbeAck` message contains information about p_j, such as its current level and number of children. The `ProbeAck` also allows the probing peer p_i to measure its delay to p_j. If peer p_j is *eligible* as a parent (i.e., its level $l_{p_j} \leq l_{p_i}$ and it has a free child slot), p_i needs to decide whether to connect to p_j as a child. If currently p_i has less than k parents, then it always connects to p_j, in order to maintain the network connectivity. If p_i already has $\geq k$ parents, it will connect to p_j only if p_j is a better parent than some existing parent. This prevents frequent parent connection and disconnections. Whether p_j is a better parent is determined by the parent selection policy (in Section 4.5). Whenever peer p_i is successfully connected to a new parent, it checks if some existing parent can be removed. If it has more than k parents, then the worst parent (again determined by the parent selection policy) is removed.

4.3. DAG Evolvement

DagStream allows peers to continuously connect to new parents and disconnect from old parents in order to improve the performance of the overlay. To avoid the creation of loops, peer p_i can only choose a peer p_j as parent if $l_{p_j} \leq l_{p_i}$. Despite this constraint, peers in DagStream can still self-organize by walking "up" and "down" the DAG structure. Figure 5 illustrates the process. Suppose we have four peers as in Figure 5(a) and $k = 2$. Peers p_3 and p_4 have the same level l. If p_3 probes p_4 and finds it to be a better parent, it will connect to p_4 as a child, and adjusts its level to $l+1$. Figure 5(b) shows the scenario after p_3 finishes the adjustment. Since p_3 now has more than k parents, it will select a parent for removal. Suppose p_2 is selected, the system will be as shown in Figure 5(c). Thus p_3 has "walked down" in the DAG hierarchy. On the other hand, if the initial state is as in Figure 5(c) and p_3 finds p_2 to be a better parent than p_4, p_3 can connect to p_2 and disconnect from p_4, as a result, the level of p_3 decreases from $l+1$ to l.

[§]Apparently the evolvement period can be dynamically adjusted based on the current performance of a peer. For simplicity we just assume it is a constant.

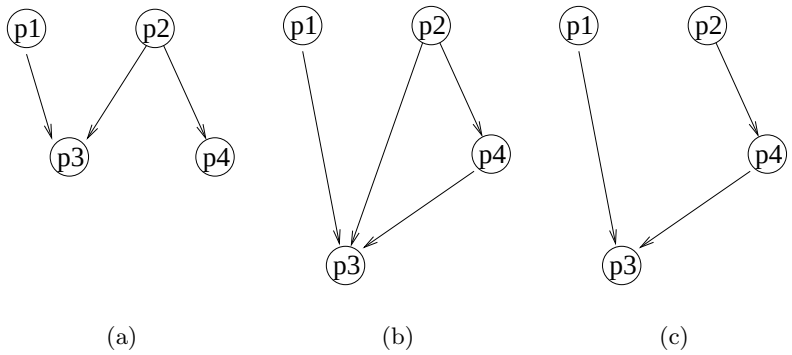


Figure 5. DAG evolvement. The level of peer p_3 increases from left to right, and decreases from right to left.

4.4. Parent Discovery Techniques

Clearly, the ability of DagStream peers to discover potential good parents is important to the achieved performance. The RandPeer service provides generic support for QoS aware neighbor selection. In our DagStream, we utilize two additional techniques to improve the quality of parent discovery.

The first technique is to explore the two hop neighbors. The motivation is that if a peer p_i has connected to a nearby neighbor p_j (parent or child), it is likely that p_i is also close to the other neighbors of p_j. For this purpose, each peer not only maintains a list of its parents and children, but also a list of its two hop neighbors. Whenever a parent needs to be selected, a peer first checks if some of its two hop neighbors is a potential good parent. If so, this two hop neighbor is probed first. Otherwise, the RandPeer service is invoked to select a peer for probing. The information about the two hop neighbors can be included in the `Refresh` and `RefreshAck` messages. Thus no additional protocol mechanisms are needed ¶.

The second technique is called parent suggestion. When a peer p_i probes peer p_j, if p_j has good QoS characteristics (e.g., small delay) but its level is too large, p_i cannot connect to p_j as a child. However, p_i can send a `Suggest` message to p_j, and suggest itself as a parent to p_j. Of course, p_j may not decide to connect to p_i, because it may have better parents. However, the suggestion mechanism can potentially help peers to discover good parents more quickly.

4.5. Parent Selection Policies

Parent selection policy is used to decide whether a peer should connect to a potential parent, and which existing parent should be removed. Since we would like a peer to connect parents that are both close by and have small levels, in this paper we examine three different policies:

1. **delay only:** A peer always attempts to minimize the delay to its parents. If the delay to a potential parent is smaller than some existing parent, the peer will attempt to connect to the potential parent. When a parent needs to be removed, the parent with the largest delay is always removed.
2. **level only:** A peer always attempts to minimize its level. If the level of a potential parent is smaller than some existing parent, the peer will attempt to connect to the potential parent. When a parent needs to be removed, the parent with the largest level is always removed.
3. **first delay, then level (delay-level):** A peer will first attempt to minimize the delay to the parents. Once the delay to all parents are within some threshold d, the peer begins to minimize its level. To prevent a peer from over optimizing its level, if the parent that has the largest level is also the one that has the smallest delay, it will not be replaced.

¶Although not explored further in this paper, the information about two-hop neighbors can also be used to provide incentives for peers to donate their bandwidth. For example, a peer can allocate its bandwidth to each child, based on how much bandwidth the child is providing to its children.

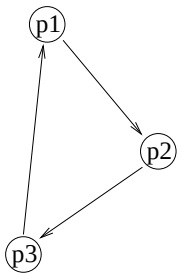


Figure 6. Loop detection in DagStream

4.6. Loop Detection

Although a peer p_i will connect to p_j as a child only when $l_{p_j} \leq l_{p_i}$, loops can form if multiple peers select parents at the same time. Therefore, DagStream must be prepared to detect and break loops. Our loop detection is based on the following observation. If a loop of size k is formed, then for each node in the loop, its level will increase by at least k after every k refresh periods. Figure 6 shows why this is the case. Suppose a loop has formed among three peers p_1, p_2 and p_3, and initially the level of p_1 is l. After at most one refresh period T_R, p_1 will send a `Refresh` message to p_2. As a result, the level of p_2 will be at least $l + 1$. After at most $2 \cdot T_R$, p_2 will send a `Refresh` message to p_3, which causes the level of p_3 to be at least $l + 2$. Therefore, after at most $3 \cdot T_R$, p_1 will receive a `Refresh` message from p_3, which causes p_1 to set its level to at least $l + 3$.

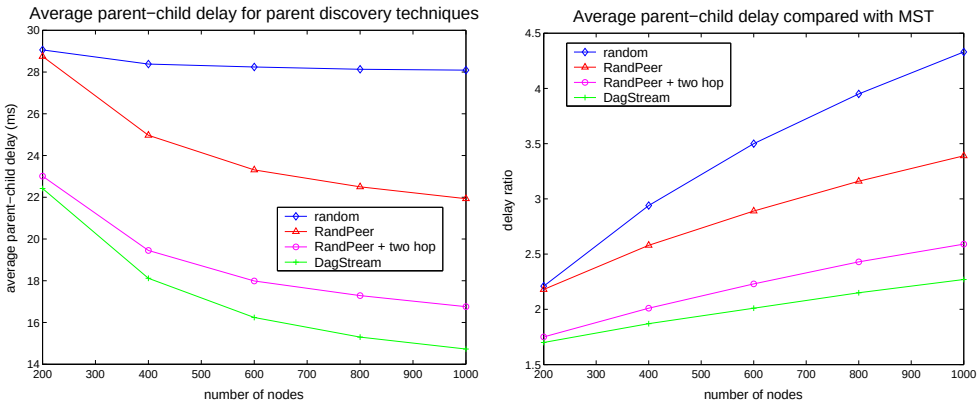
Based on the above observation, a peer sets a counter for each parent. Whenever the level of a parent increases, the counter is increased by 2. For every refresh period T_R, the counter is decreased by 1 if it is currently > 0. Thus if a loop is indeed formed, the counter of the parent will increase indefinitely. Therefore if the counter value for a parent increases to beyond some threshold, a loop is declared and the parent removed.

5. EVALUATION

We have fully implemented our DagStream protocol. Our implementation adopts an event-driven architecture, thus can work in both simulation and real world mode. In the simulation mode, messages between peers are dispatched as events, with the event firing time determined by a synthetic network topology. In the real world mode, messages are sent via UDP sockets. In this section, we mainly present our simulation results to show (1) the ability of DagStream to build locality aware DAGs, as well as the impact of different parent discovery techniques and parent selection policies; (2) the ability of DagStream to deliver good streaming quality, by allowing peers to select additional parents, as long as their streaming quality can be further improved. We also present some results from our experiment on the PlanetLab[8] testbed.

Since we want peers to preferably connect to close by parents with small levels, we evaluate the protocols using two metrics. The first is the *average parent-child delay*, which indicates on average how far is a peer from its parents. The second metric is the *max level* and *max "min level"* of a DAG overlay . The max level of a DAG overlay is the longest overlay path from the source s to any node. The max min level is the worst case shortest overlay path from s to any node. Alternatively, if the max min level of an overlay is l, it means every peer has at least one overlay path from the source that has a length $\leq l$.

In all the following simulations, we use the BRITE[16] topology generator to generate a two level hierarchical network topology. We then randomly select a subset of the nodes as peers in DagStream. We use all-pairs shortest path algorithm to compute the end-to-end delay between the peers, and normalize the delay to a maximum of 100 ms. Unless otherwise specified, we set $k = 2$, $P_{max} = 5$, $C_{max} = 5$. By default the delay-level parent selection policy is used, and the delay threshold d is set to 25 ms. T_R and T_E are both 10 seconds. When using RandPeer, each peer generates a 3-bit landmark vector[15] as its QoS prefix. Each bit encodes the delay of the peer to one landmark node.



(a) Average parent-child delay

(b) Parent-child delay compared with MST

Figure 7. Parent-child delay of different parent discovery techniques.

5.1. Impact of Parent Discovery Techniques

Figure 7 illustrates the impact of different parent discovery techniques on building locality aware DAGs. For this experiment, each time we select a specific number of nodes from the network topology as DagStream peers. Initially no peer is connected to the source. We let the peers join the system and evolve the network for 1000 seconds and measure the average parent-child delay of the DAG. Each experiment is repeated 200 times and the averages are presented. During the evolvement peers use different techniques for parent discoveries, "random" means peers select potential parents purely randomly from the entire system, and "DagStream" refers to the combination of using RandPeer, two hop neighbors and the parent suggestion techniques.

Figure 7(a) shows that first, compared with random neighbor selection, the QoS awareness of RandPeer can significantly improve the locality of the resulting overlay, especially for large networks. For a 1000 peer network, RandPeer alone achieves more than 20% improvement compared with random neighbor selection; second, the use of two hop neighbors can greatly improve the parent-child delay. This is because the two hop neighbors of a peer are also likely to be close neighbors of the peer. The parent suggestion mechanism also improves the locality of the the overlay. But its effect is less significant.

Figure 7(a) shows the absolute parent-child delay. Figure 7(b) compares the parent-child delay with that of the minimum spanning tree (MST). We can see that the delay ratio of DagStream increases as the network size increases. For a 1000 peer network, the average parent-child delay of DagStream is a little more than twice that of the MST. There are several reasons. First, we require each peer to have at least two parents, thus the DagStream overlay has twice as many links as the MST. Second, we limit the maximum out degree of each node to 5, while the MST has no such constraints. Third, DagStream will attempt to optimize level when the parent-child delay is within some threshold, while MST makes no attempt to reduce its level (height).

Figure 8 compares the level of MST and DagStream. The figure shows that when the network size grows, the height of the MST grows quickly. And for large networks, even the max level of DagStream is much smaller than the level of the MST. The max min level of DagStream grows slowly with the network size. For a 1000 peer system, the max min level of DagStream is 10. This is only 3 hops larger than the random parent discovery, even though the delay of DagStream at each hop is only half that of the random parent discovery.

5.2. Effect of Parent Selection Policies

Figure 9 shows the effect of different parent selection policies as introduced in Section 4.5. Figure 9(a) is the average parent-child delay of the policies. As we can see, the "level only" policy always tries to minimize the level of a peer, without considering the locality of the parents. As a result, the average parent-child delay is

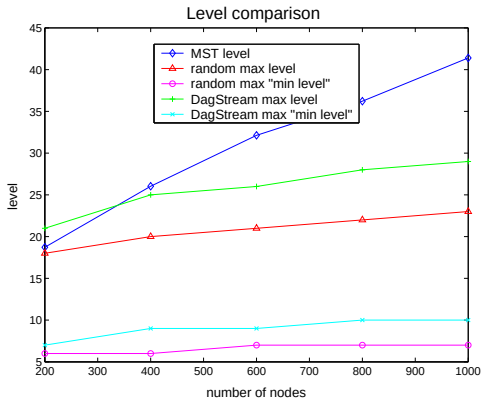
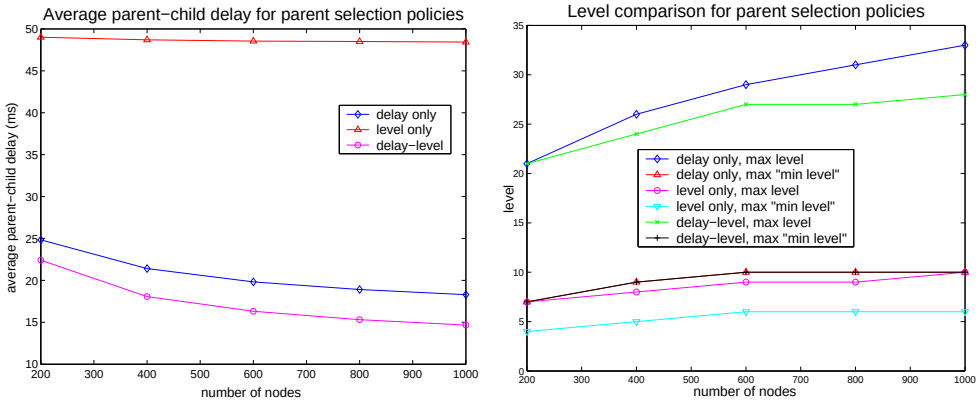


Figure 8. Level comparison with MST.



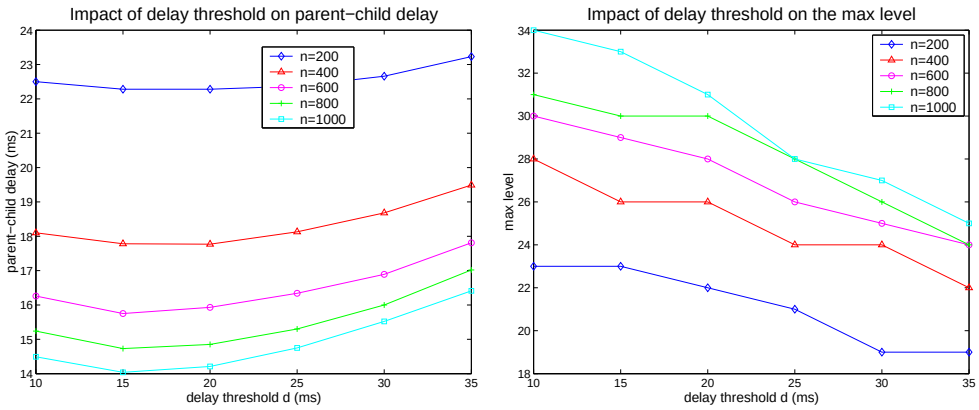
(a) Parent-child delay (b) Max level and max "min level"

Figure 9. Effect of different parent selection policies.

always about 50ms, which is the average delay between any two peers. The "delay only" policy always tries to minimize the delay to the parents. However, the achieved delay is worse than that of the "delay-level" policy. For larger networks, the average parent-child delay of the delay-level policy is about 14% smaller than that of the delay only policy. The reason is that the delay-level policy tries to reduce the level a peer once its parents are within the delay threshold d. When the level of a peer decreases, it is eligible as a parent for more peers. This increases the chances of other peers to locate good parents.

Figure 9(b) shows the max level and max min level of the different policies. Not surprisingly, the level only policy achieves the smallest max level and max min level. However, the max min level of the other two policies are only 3 or 4 hops larger than that of the level only policy. The figure shows that the max min level of the delay only and delay-level policies are the same. This is because we rounded off the average values to integers.

The delay-level policy uses a delay threshold d to decide when to switch from delay optimization to level optimization. Thus d may have an impact on the performance of the policy. Figure 10(a) shows the effect of the delay threshold on the parent-child delay. Each line is for a different number of peers. We can see generally when d is larger, peers will focus less on improving the parent delay. As a result, the achieved parent-child delay is larger. Thus using smaller d may improve the parent-child delay. However, when d is too small, the achieved delay will actually increase, this is due to the same reason that caused the delay only policy to perform poorly. When peers do not attempt to reduce their level, it is less likely for other peers to choose them as parents. Figure 10(b) shows the max level for different d. Indeed when d is large, peers focus more on improving their



(a) Parent-child delay

(b) Max level

Figure 10. Effect of delay threshold d.

level, thus the max level will decrease. The effect of d on the max min level is similar, although not significant.

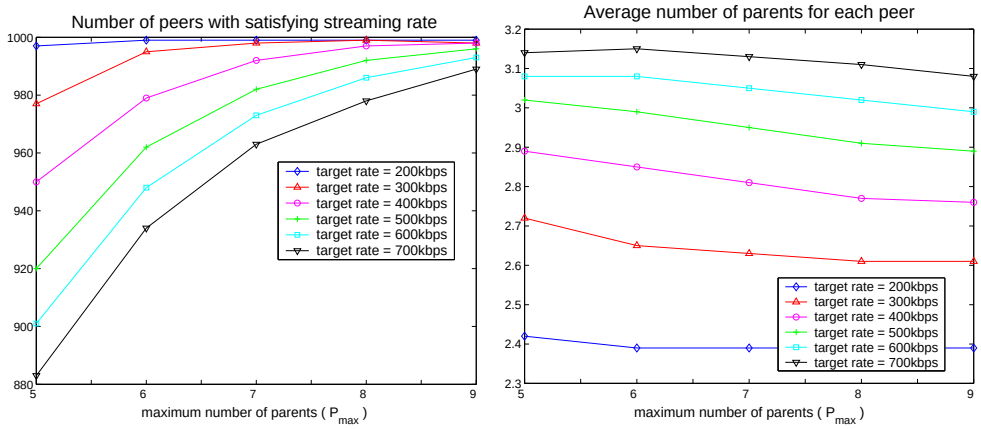
5.3. Improve Streaming Quality

DagStream focuses on improving the network locality of the underlying overlay network, and allows peers improve their streaming quality by selecting additional parents as needed. Figure 11 shows this approach can indeed deliver good streaming quality. For this experiment, we first generate the uplink and downlink bandwidth of the peers using a distribution similar to those reported in.[17] 20% of the peers are low bandwidth peers. Their downlink bandwidth is distributed in the range [384kbps, 1Mbps], and uplink bandwidth in the range [128kbps, 384kbps]. 50% of the peers are with medium bandwidth. Their downlink and uplink bandwith ranges are [1Mbps, 6Mbps] and [384kbps, 1Mbps]. Finally, 30% of the peers are with high bandwidth. Their downlink bandwith is in the range [10Mbps, 50Mbps], and their uplink bandwidth is equal to their downlink bandwidth. All the distributions are uniform. While our bandwidth distribution may not exactly match a particular P2P system, it nonetheless introduces the heterogeneity that is typical in a P2P environment. The number of peers is 1000 for this experiment.

In the experiment, peers first attempt to connect to k nearby parents. They then compute the aggregate streaming rate provided by their parents. If the aggregate rate is smaller than the target rate (the rate of the original video), and their downlink bandwidth has not been fully utilized, they will attempt to connect more parents, which are discovered and probed in the usual way. We let the system evolve for 1000 seconds, and count the number of peers with satisfying streaming quality. A peer is satisfied with its quality, if the aggregate rate provided by its parents is larger than the target rate or its own downlink bandwidth. For simplicity, we assume the uplink bandwidth provided by a peer is equally shared by its children.

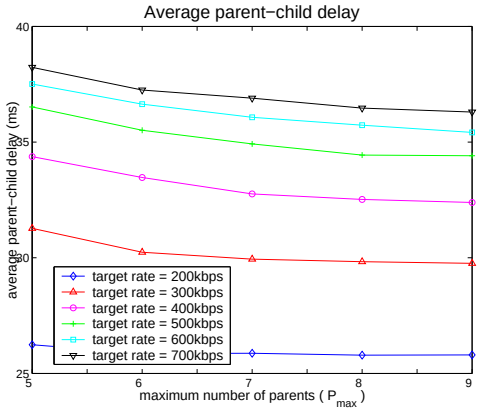
Figure 11(a) shows the number of peers that have good streaming quality (are satisfied) as a function of the maximum number of parents P_{max} that a peer is allowed to connect to. The maximum number of children that a peer can accept (C_{max}) is fixed to 12. We can see that when P_{max} is limited to 5, almost every peer will have good streaming quality when the target rate is 200kbps. But only about 88% peers have good streaming quality when the target rate increases to 700kbps. By relaxing P_{max}, however, peers can aggregate bandwidth from more parents and thus more peers will have good streaming quality. When P_{max} is 9, about 99% of the peers are satisfied even for a target rate of 700kbps. Note that in our bandwidth distribution, only about 45% peers have uplink bandwidth that is $\geq$ 700kbps.

Having $P_{max} = 9$ might seem to be too large. However, Figure 11(b) shows that even though P_{max} is set to a large value, the average number of parents for a peer is still small. For example, even when the target rate is 700kbps and P_{max} is 9, on average a peer has less than 3.1 parents. In fact, when P_{max} increases, the average



(a) Number of satisfied peers

(b) Average number of parents

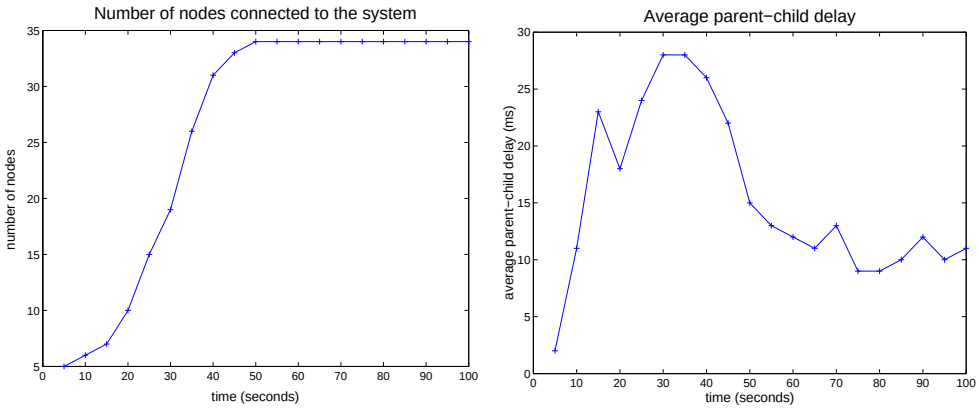


(c) Average parent-child delay

Figure 11. Streaming quality delivered by DagStream.

number of parents may decrease slightly. The reason is that when P_{max} is small, a peer may connect to P_{max} parents that have low uplink bandwidth. In this case the peer is not allowed to try more parents, and it will not disconnect from existing parents, because this will further reduce its streaming rate. When P_{max} is large, however, the peer has the opportunity to try more parents, and as a result discover parents with large uplink bandwidth. At this time the peer is satisfied with its quality and can disconnect some parents with low uplink bandwidth. This result shows to allow peers to explore more parents when needed, P_{max} should probably not be set too small. Allowing a peer to connect to P_{max} parents doesn't necessarily mean it will always maintain P_{max} parents.

One goal of DagStream is to let peers stream preferably from closeby parents, and to connect to remote parents only when necessary. Figure 11(c) shows DagStream achieves this goal. When the target rate is 200kbps, on average the delay of a peer from its parents is only a little more than 25 ms. When the target rate increases, the average parent-child delay increases, because peers have to connect to more parents, which might be farther away. However, even when the target rate is 700kbps, the average parent-child delay is still less than 40ms, which is about 20% less than the average delay between any two peers. An interesting thing is that when P_{max} increases, the average parent-child delay actually decreases. The reason is the same as for the



(a) Number nodes connected (b) Average parent-child delay

Figure 12. Performance results on 34 PlanetLab nodes.

average number of parents. The initial parents of a peer may not be good, both in terms of uplink bandwidth and delay. By allowing peers to try more parents, they have better chance to discover parents that not only have large uplink bandwidth, but also are close by.

5.4. Performance on PlanetLab

We have experimented with our DagStream on about 34 nodes on the PlanetLab[8] testbed. Each time we start DagStream on all nodes at the same time, and let the nodes join the source (`planetlab2.cs.uiuc.edu`) and evolve the overlay structure. We repeat the experiment 20 times and show the average results. Since the network is small, we do not use RandPeer. Each node initially only knows about the source. The information about other nodes are learned by contacting the source. Figure 12(a) shows the number of nodes that joined the system as a function of time. After about 40 seconds [‖], most peers are joined in the system. Figure 12(b) shows how the average parent-child delay changes over time. Initially the delay is very small, the reason is that initially if no nodes have connected to the source, the delay is counted as 0. When most nodes first join the system (between 30 and 40 seconds), the average parent-child delay is the largest. However, peers can quickly improve the locality of the overlay, and at about 75 seconds, the delay has decreased to about 10ms. We have measured the delay between the 34 PlanetLab nodes and the average inter-node delay is about 46ms. This shows the ability of DagStream to build locality aware overlays.

6. CONCLUSION

In this paper, we have presented a new P2P streaming framework called DagStream. DagStream organizes peers into a directed acyclic graph where each peer maintains at least k parents. As a result, DagStream can achieve provable network connectivity, even when peers optimize their performance in a distributed fashion. Our DagStream protocol enables peers to quickly self-organize into a locality aware DAG, which ensures efficient network resource usage. Above this, peers can select additional parents, as long as their streaming quality can be further improved. Our experiment results in both simulation and wide area environment verifies the ability of DagStream to build locality aware DAGs and to deliver good streaming quality to the peers.

[‖]Note each peer tries a candidate parent every 10 seconds. If the candidate is full, the peer will wait for 10 seconds and try again.

REFERENCES

1. Y. hua Chu, S. G. Rao, and H. Zhang, "A case for end system multicast," in *Proceedings of ACM SIGMETRICS*, (Santa Clara, CA), June 2000.
2. S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proceedings of ACM SIGCOMM'02*, August 2002.
3. K. A. H. Duc A. Tran and T. Do, "ZIGZAG: An efficient peer-to-peer scheme for media streaming," in *Infocom'03*, 2003.
4. X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "DONet: A data-driven overlay network for efficient live media streaming," in *IEEE INFOCOM'05*, (Miami, FL), 2005.
5. M. Hefeeda, A. Habib, D. Xu, B. Bhargava, and B. Botev, "CollectCast: A peer-to-peer service for media streaming," in *ACM Multimedia'03*, 2003.
6. R. Rejaie and S. Stafford, "A framework for architecting peer-to-peer receiver-driven overlays," in *NOSSDAV'04*, 2004.
7. V. Pai, K. Tamilmani, V. Sambamurthy, K. Kumar, and A. Mohr, "Chainsaw: Eliminating Trees from Overlay Multicast," in *4rd International Workshop on Peer-to-peer Systems (IPTPS'05)*, Feburary 2005.
8. "Planetlab." http://www.planet-lab.org/.
9. H. J. W. Venkata N. Padmanabhan and P. A. Chou, "Distributing streaming media content using cooperative networking," in *NOSSDAV'02*, 2002.
10. M. Castro, P. Druschel, A.-M. Kermarrec, and A. Nandi, "Splitstream: High-bandwidth multicast in cooperative environments," in *SOSP'03*, 2003.
11. W. T. Ooi, "Dagster: Contributor-Aware End-Host Multicast for Media Streaming in Heterogeneous Environment," in *SPIE Multimedia Computing and Networking (MMCN 05)*, 2005.
12. Y. hua Chu, A. Ganjam, T. S. E. Ng, S. G. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang, "Early experience with an internet broadcast system based on overlay multicast," in *Proceedings of USENIX Annual Technical Conference*, (Boston, MA), June 2004.
13. K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang, "The feasibility of supporting large-scale live streaming applications with dynamic application end-points," in *ACM SIGCOMM'04*, August 2004.
14. J. Liang and K. Nahrstedt, "Randpeer: Membership management for QoS sensitive P2P applications," Tech. Rep. UIUCDCS-R-2005-2576, UIUC, May 2005.
15. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection," in *Proceedings of IEEE Infocom'02*, 2002.
16. A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An approach to universal topology generation," in *In Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems- MASCOTS '01*, (Cincinnati, Ohio), 2001.
17. S. Saroiu, K. P. Gummadi, and S. D. Gribble, "Measuring and analyzing the characteristics of Napster and Gnutella hosts," *Multimedia System Journal* **9**, pp. 170–184, 2003.