Streaming Live Media over Peers

Paper Id: 620, No. of pages: 14

Abstract—
The high bandwidth required by live streaming video greatly limits the number of clients that can be served by a source using unicast. An efficient solution is IP-multicast, but it suffers from poor deployment. Application-level multicast is being increasingly recognized as a viable alternative. In this work, we discuss and evaluate a tree-based overlay network called *PeerCast* that uses clients to forward the stream to their peers. PeerCast is designed as a live-media streaming solution for peer-to-peer systems that are populated by hundreds of autonomous, short-lived nodes. Further, we argue for the need to take end-host behavior into account while evaluating an application-level multicast architecture. An end-host behavior model is proposed that allows us to capture a range of realistic peer behavior. Using this model, we develop robust, yet simple, tree-maintenance policies. Through empirical runs and extensive simulations, we show that PeerCast provides good QoS, which gracefully degrades with the number of clients. We have implemented a PeerCast prototype, which is available for download.

I. Introduction

Live streaming media will form a significant fraction of the internet traffic in the near future. Recent trade reports indicate that if the current acceptance rate among end-users persists, streaming media could overtake television with respect to the size of the client base [1]. The adoption trends for streaming media are expected to be particularly remarkable in the context of peer-to-peer (P2P) systems.

Since video streams are high bandwidth applications, even a small number of clients receiving the stream by unicast are often sufficient to saturate bandwidth at the source. IP-Multicast [2] was proposed as an extension to Internet architecture to support multiple clients at network level. The deployment of IP Multicast has been slowed by difficult issues related to scalability, and support for higher layer functionality like congestion control and reliability. In the last couple of years, several research projects [3, 4, 5, 6, 7, 8, 9, 10] have argued for a multicast service at the application-level. The service is to be simulated over unicast-links between hosts, forming an overlay network over end-hosts. The application-layer assumes responsibility for providing multicast features like group management and packet replication.

The end-nodes in P2P systems form ad-hoc networks to contribute resources to the community, and in turn use resources provided by other members for a personal goal. Thus, members of a P2P system can be expected to share their bandwidth to spread a media stream to other clients. An end-host multicast solution then seems to be an ideal fit. However, a P2P system presents a challenging domain. A P2P system can have hundreds of nodes at any given instant. More significantly, nodes are autonomous, unpredictable, and may have short (of the order of minutes) lifetimes. Thus, an adopted solution must be capable of good performance over a large and dynamic system.

In this work, we propose a tree-based end-hosts multicast architecture for streaming live media that can scale to hundreds of nodes in a P2P system. The solution, called *PeerCast*, consists of a lightweight *peering* layer that runs between the application and transport layers on each client peer. The peering layers at different nodes coordinate among themselves to establish and maintain a multicast tree. The PeerCast solution can work with short-lived nodes, and allows relatively cheap deployment. Importantly, existing applications at a peer do not need to be changed to use the peering layer, and are thus conveniently enabled for such a stream distribution.

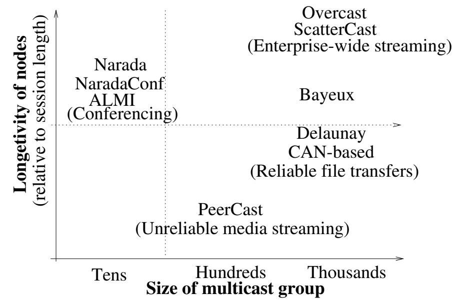

Fig. 1. Classifying different end-host multicast proposals.

To place our work in the context of proposed end-host multicast solutions, we cartooned existing proposals as in Figure 1 that illustrates the domain characteristics catered to. The horizontal axis plots the size of multicast group that can be supported. The vertical axis plots the assumed lifetime of hosts over which the overlay network is formed. The solutions indicated in Figure 1 are a representative set of wider work done in the field, and are discussed in more detail in Section VIII.

Solutions in the left portion of the grid [4, 5, 11] have been proposed for multicast over a group of tens of nodes, most of which live through the duration of the session. An example application domain is video conferencing, wherein participants are few (tens of members), and members live through most of the session. The top-right portion of the grid lists proposals [6, 7] that can scale to thousands of nodes, but assume existence of reliable, long-lived infrastructure hosts over which overlays are constructed. As such, these solutions are not adaptable to P2P systems. The middle-right portion lists proposals [8, 9, 10] that rely on primitives proposed in a P2P context to accommodate thousands of clients. These solutions have been proposed and tested in domains in which most of the members are expected to persist through the session. To the best of our knowledge, these proposals have not been proven to scale for hundreds of short-lived nodes forming a *highly dynamic* group.

The solution we propose, PeerCast, corresponds to the bottom-middle position on the grid, and is designed to scale to hundreds of short lifetime nodes that participate in long-durationed multicast session transmitted over unreliable unicast UDP/RTP. Thus, in the context of P2P systems, our contributions can be summarized as below.

- We propose an architecture for streaming media over a dynamic P2P network that uses a basic peering layer to efficiently enable a connected topology (Section III).
- As part of this work, we have implemented the peering layer which is available for download from [12].

We next discuss our contributions in the context of application-level multicast studies. Stemming from the influential work in [4], end-hosts multicast proposals have been evaluated with respect to *efficiency* metrics of *stress, relative delay penalty*, and *normalized resource usage* alone. In this work, we argue for the necessity of evaluating an end-hosts solution with respect to *end-system metrics*, that are orthogonal to efficiency penalties.

The migration of multicast functionality from network-layer to the application-layer has effects on end-application performance that go beyond packet delays. Such a migration causes the characteristics of the "infrastructure" assumed by the end-application to change. The behavior of the routing-elements (end-hosts) in application-level multicast is very different from that of routers in IP-Multicast. In particular, end-hosts are autonomous, and can withdraw their resources from public use at will. The end-applications are not easily insulated from such transience in the behavior of the infrastructure units.

To understand how the migration impacts end-application performance, let us consider streaming-media delivery via multicast. During an end-hosts multicast session, a large fraction of clients (that were acting as routers) might *unsubscribe together* in the middle of the stream, partitioning the overlay network. The time taken to repair the partitions results in loss of packets transmitted during the transience period. Such an event, while frequent in end-host multicast, is in contrast to network-level multicast where clients only occur as leaves in the multicast tree built on routers. Router failures are not as frequent, and the chances of simultaneous failures of a large number of routers, occurring regularly over the stream duration, is small. Thus, while the end-system metric of *packet loss* due to node failures is not a useful metric to evaluate network-level multicast, it is essential to characterize an end-hosts multicast solution.

The efficiency metrics mentioned above are designed to characterize the effects of the functionality-migration on the performance returned by the lower (network) layer alone. It is our thesis that apart from studying such efficiency penalties, the *end-system performance penalties* for these architectures must also be evaluated. There is a need to to characterize the behavior of application-routers, and use such models to evaluate the proposals on end-system metrics. In this work, we propose such a model that enables us to evaluate our architecture for the live streaming media application. As we show later, PeerCast insulates end-applications from router transience, thus enabling good end-system performance that degrades gracefully with increasing numbers and decreasing lifetimes of clients. To summarize, the contributions of this work are as follows.

- We propose a *peer behavior model* that seeks to capture the characteristics of a client seeking a live stream (Section VI). Correspondingly, it models the behavior of nodes (application routers) that form the P2P network.
- We evaluate the feasibility of a family of tree-based architectures with respect to the *end-system performance* metrics measured through experiments (Section V), and through extensive simulations (Section VII).
- We deduce a distributed tree-based protocol that is more robust than existing tree-based protocols (Section IV).

We conclude this paper with a discussion of related work in Section VIII, and a summary in Section IX.

II. THE PEERCAST MODEL AND ARCHITECTURE

In this section, we give an overview of the PeerCast architecture, and then point out issues that must be resolved for such an architecture to return good performance.

A media stream is a time ordered sequence of packets that is logically composed of two channels: *data* (served using unreliable RTP/UDP [13]) and *control* (sent using reliable RTSP/TCP [14]) channels. A *live* stream has the important property of being *history-agnostic*: the group-member is only interested in the stream from the instant of its subscription onwards.

PeerCast provides the multicast service by organizing the group members into a self-organized, source-specific, spanning tree that is maintained as nodes join and leave. The group members are arranged at different levels of a multicast tree rooted at the source s. Each node n (including source s) forwards the stream to all its immediate children (if any). Effectively each node acts as a multicast router, replicating and forwarding packets. Note that the resulting topology is an overlay tree: the actual forwarding paths in the underlying physical network may not be a tree, as shown in Figure 2.

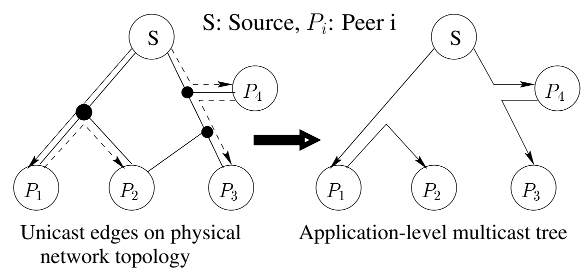

Fig. 2. An application level multicast tree built on the peers

Such an overlay network formed on a set of autonomous nodes, in which a node can simultaneously serve some nodes, while being served from another node, is a *peer-to-peer (P2P)* network. The bandwidth requirements of the source are now shared by the network of clients, providing a potentially scalable solution for media broadcast.

However, each node is autonomous, and can subscribe or unsubscribe from a stream at will, or even fail without notification. As the set of nodes $\aleph$ changes over time, the QoS delivered to current clients must not suffer. Thus, for the benefits to be realized, the following issues need to be resolved.

1. *Enabling Bootstrap*: A new member who wishes to subscribe to a stream must be able to join the multicast group. We assume that each live stream has a unique URL (Uniform Resource Locator). The URL has information about s embedded in it. A member wishing to join obtains the stream URL by an out-of-band mechanism, and sends a *join* message to s.
2. *Discovering a Server*: The multicast tree is built incrementally as nodes subscribe to the stream. New nodes are supported by nodes already subscribed to the stream. However, each peer has a certain bandwidth capacity. A peer p which is unable to serve an incoming client c with an acceptable QoS is said to be *saturated*. There needs to be a mechanism to enable c to join the multicast by identifying an existing node in the tree that is *unsaturated* and can act as a parent to c. We discuss such a mechanism in Section III.

3. *Managing Transience*: Nodes can unsubscribe from the multicast network at will. If a node opts out, all its descendants are left stranded. It is essential that such partitions be repaired for the descendants to continue receiving the feed, and that too, in a duration which has minimal effect on the QoS perceived by its descendants. We discuss recovery mechanisms in Sections III and IV.
4. *Managing Load*: The multicast tree is built over nodes whose primary intent is to act as individual clients. The resources (processing power, memory, and bandwidth capacity) of the nodes should be judiciously used while constructing the multicast tree, so as not to overload any node. Each node is aware of its own capacity, and may, for a given stream bit-rate, fix for itself a maximum limit on the number of children it will support.
5. *Ensuring efficiency*: The overlay network should incur a small efficiency penalty as compared to IP-level multicast. The accepted metrics for characterizing efficiency of an end-hosts multicast are *stress* and *resource-usage*. Policies suggested in different contexts ([4, 9, 8, 15]) have been shown to result in limited efficiency penalties when compared to IP-Multicast. The suggested heuristics need to be adapted for construction of efficient topologies with PeerCast.
6. *Ensuring end-system performance*: A streaming media application has the following characteristics:

- Timeliness constraints that require changes in membership to be accommodated quickly, and
- Sensitivity to data loss that requires minimal packet loss while accommodating changes in membership.

Correspondingly, our *end-system metrics* to characterize performance for streaming media are *response-time (time to first packet)*, *packet-loss*, and *packet-delays*. Note that the efficiency metrics of stress and resource usage are unable to describe the effects of transience on packet losses, or the time to first packet. As we argued in Section I, an end-host multicast solution necessarily affects the end-application performance due to the transience of routing-elements. An end-host multicast solution should seek to minimize the adverse effects of dynamic nodes. Section VI discusses a peer behavior model that allows us to capture the characteristics of a client. We discuss policies that minimize penalties on end-system metrics in Section IV, and evaluate the policies with respect to the model in Section VII.

III. Peering Layer: Primitives and Mechanisms

In this section, we describe the primitives supported by PeerCast, and the mechanisms used to resolve the issues described in Section II.

An instance of the data and control channels of a media stream, taken together, between a server and a client constitutes a *data-transfer session*. Such a session is distinguished from an *application session* which begins when a client subscribes to a stream from a source, and ends when the client unsubscribes. The application session can subsume several data-transfer sessions as the client changes servers from which it receives feed for a given stream.

However, current implementations of most applications assume a unicast service, and bind the two sessions irrevocably together. We introduce a basic P2P infrastructure layer between the application and the transport layers for streaming that breaks the coupling between the two sessions. All communication between application and transport layers passes through the peering layer, as shown in Figure 3.

Data transfer sessions are established between the peering layers at the two nodes. The end-points are identified by the tuple ⟨ *Server IP-address, Server port, Client IP-address, Client port, Stream URL* ⟩. The first four components identify the transport end-points involved in the session. The last component serves to identify the specific stream data is intended for.

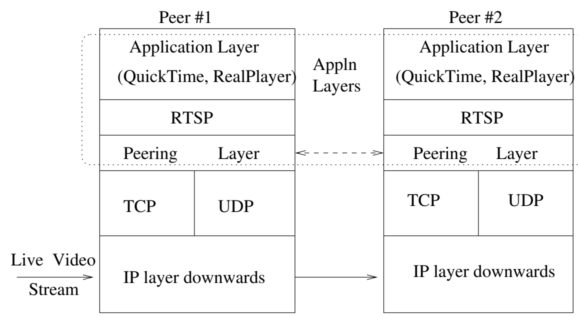

Fig. 3. A layered architecture of a peer

Application sessions are established between the peering layer and the end application. The peering layer allows the application layer above to specify the stream to obtain, through a "get-stream" interface. Given a stream URL, the peering layer locates a server which can provide the stream, and establishes a data-transfer session with the server. In the event of a data-transfer session termination, it locates a new server for data feed, and restarts the flow of data. If an alternate server cannot be found, an error is flagged to the application above. The application session survives these changes, leaving the end application *unaware* and, indeed, *unconcerned*, of shifts in the underlying multicast topology, as it should be. As we show in an extended version [16] of this paper, the code of existing applications does *not* need to be changed to enable the use of our peering layer.

The guarantees provided by the peering layer reflect those of the transport layer protocol used in the data-transfer session (for which it has to maintain state). In addition, it guarantees that the data feed to the application above will be maintained as long as an unsaturated server can be found.

A. The Redirect Primitive

The peering layer supports a simple, lightweight redirect primitive used to effect changes in the topology. The primitive proves to be a good building block for topology-maintaining algorithms. The peering layer uses redirects as hints, to enable discovery of an unsaturated node in the network.

As shown in Figure 4, a redirect message is sent by a peer p to another peer c which is either opening a data-transfer session with p, or has a session already open. The message specifies a target peer t. On receipt of the redirect message, c closes its data-transfer session with p, and tries to establish a data-transfer session with t, for the same stream URL. Note that such redirect messages are produced and used at the peering layer, and the application above is unaware of such messages. Thus, the

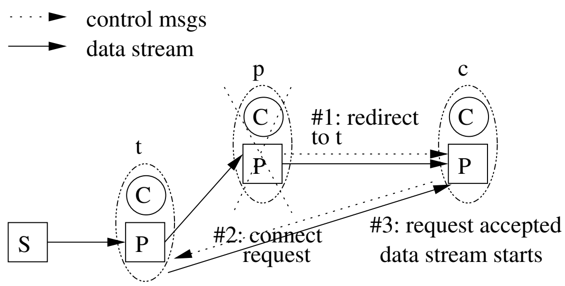

Fig. 4. An example of the redirect primitive in use when p leaves

application session persists despite changes in the data-transfer sessions.

B. Discovering an Unsaturated Server

A new node n seeking the live stream needs to be able to discover an unsaturated node in the multicast group. The node n contacts the source s of the stream at the known URL. If s is unsaturated, it accepts n as its child and establishes a data-transfer session with n. Otherwise, s redirects n to one of its immediate children c. Then, n attempts to setup a data-transfer session with c. The process continues iteratively, until n gets accommodated. If n is unable to find an unsaturated node within some specified number of tries, the peering layer flags a resource unavailable error to the upper application-layer.

C. Managing Unsubscribe of Nodes

If a node n wishes to unsubscribe from the stream, it sends a leave message to its parent p. The parent p frees up resources dedicated to n at its side. However, the descendants of n are now disconnected from s, and experience a break in the stream. To enable a recovery following its unsubscribe, n sends redirects all its immediate children C to some target t (e.g., the source s, or the parent of n). The nodes in C then start the process of finding an unsaturated server by contacting t, as discussed above.

D. Handling Failures of Nodes

Intermediate nodes might fail, without being either able to inform their parent, or send redirect messages to their children. The overlay network needs to first detect such a failure, and then recover from it. The peering layer at a node uses a heart-beat mechanism to send alive messages to its parent, and children. If a peer detects that a child has skipped a specified number of heart-beats, it deems the child as dead, and cleans up the data-transfer session at its end. Similarly, if a peer deems its parent dead, it recovers from its transience state by redirecting itself to the source s. The unsaturated-server discovery process as mentioned in Section III-B is then followed.

IV. Policies for Topology Maintenance

As we observed in Section II, the end-system performance is characterized by packet loss, packet delay, and time to first packet. PeerCast uses an overlay spanning tree, and the shape of the tree has implications on the end-system metrics. The shape of the spanning tree is defined by the topology-maintaining policy. Hence, for optimal performance, we need to understand the effect of topology on end-system metrics.

Lemma 1: Under homogeneous unicast edge and node characteristics, an almost-complete spanning tree is the optimal overlay tree for packet loss, packet delay, and time to first packet metrics. □

Proof: Let n be a node at depth h_n in a source-rooted overlay tree. Since packet delays are additive across hops, the delays observed at n are proportional to h_n. If p_{loss} is the probability of a packet loss at a hop, then the probability of not receiving a certain packet at n is given by $1-(1-p_{loss})^{h_n} \simeq p_{loss}h_n$, for small p_{loss}. Thus, packet losses observed at n are proportional to h_n. The time to first packet for a newly joined node n is a composite of time taken to discover an unsaturated server t_{disc}, and delay for the receipt of first packet t_{delay}. The t_{disc} is proportional to the number of redirects received by n, which is equal to the node's (eventual) depth h_n. The t_{delay} is proportional to depth as indicated above. Hence, we can define the cost C of a candidate tree T as $C_T \propto \sum_{n=1}^{|\aleph|} h_n$. It is easy to see that an almost-complete tree has minimal cost. ■

Thus, an optimal policy is one that maintains an almost-complete spanning tree. In Section VII, we will compare our policies against a hypothetical optimal policy that (magically) maintains an almost-complete tree. Note that a policy defines the overlay-tree topology by specifying the choice of a target peer in a redirect message under join, leave, and failure of nodes. Such a policy can then be implemented using the mechanisms we discussed in Section III.

Given our cost function, a host of policies are feasible, ranging from light-weight to a centralized heavy-state policy that seek to build an (approximately) optimal tree. In this section, we shall discuss some of the policies, based on our simple cost function, that are appealing because of their light-weight and distributed nature. In particular, we consider policies that require each node to know only its local topology (parent and children), apart from the source s.

A. Addition of a peer (or Join Policies)

A node which is unsaturated always accepts a data-transfer session setup request. However, a saturated node n needs to forward the requesting client c to another peer in the network which is also getting the stream feed. Since a peer only knows its local topology, n can only forward c to one of n's immediate children, or its parent. Some of the options in choosing such a target are the following:

1. *Random* : The node n chooses one of its children at random as the target t, and redirects c to t. Such a policy requires minimal state at n. On an average, the tree is expected to be balanced.
2. *Round-Robin (RR)* : The node n maintains a list of its children, and forwards c to the child t at the head of the list. The child t is then moved to the end of the list. Such a policy requires some state maintenance, but is expected to keep the tree balanced.
3. *Smart-Placement (SP)* : Each node maintains the network locations of its children. The client c sends traceroute information along-with its request to n. The node n redirects c to a child that has least access latency[17] to c. Such a policy helps in creating trees taking network proximity into account. Packet losses and delays are expected to be minimized, thereby improving the

performance.

B. Deletion of a peer (or Leave/Recover Policies)

When a node n wants to unsubscribe, it needs to forward a suitable valid target t to its descendants. Each node is definitely aware of two nodes in the network which get a stream feed independent of itself: its parent, and the source. Thus, there are at least two candidate values for t. When n is unsubscribed, all of its descendants become transient. Each transient descendant can try to recover by contacting t. Alternately, only the children C of n attempt to recover by contacting t. The rest of the descendants of n, are also the descendants of C. These nodes automatically recover when the feed to C is restored. Hence, we have the following policies.

1. *Root-All (RTA)* : The node n chooses the source as target. Starting from n, a redirect message is recursively forwarded to all the descendants of n specifying s as target. The advantage of this policy is that the tree is expected to remain balanced owing to a redistribution of the affected nodes.
2. *Grandfather-All (GFA)* : The node n chooses its parent p (which is the grandfather of C) as the target. As in RTA, all the descendants of n are recursively redirected to p. The advantage of such a policy is that the effect of the unsubscription is limited to the subtree rooted at p. Moreover, the source s is protected from such requests in the event of multiple simultaneous failures. Yet the tree is expected to remain balanced as the subtree is reconstructed from the same nodes as before.
3. *Root (RT)* : The node n chooses the source as t. Only nodes in C attempt to recover by contacting t. The rest of the descendants rely on C to restore their feed. The advantage of such a policy is that the sub-trees rooted in C could be accommodated near the source, while avoiding an explosion of requests to s.
4. *Grandfather (GF)* : The node n chooses its parent p (which is the grandfather of C) as t. Only nodes in C attempt to recover by contacting t. The advantage of such a policy is that the effects of failures are localized.

Note that the policy to recover from failure is similar to leave, once a failure of the parent is detected. However, in this case, the identity of the parent of the failed node is not known to the descendants of the node. Hence, only the Root and Root-All policies are relevant here.

C. Improving Overlay Efficiency (or Adapting to Network Dynamics)

The multicast tree that has been formed may take end-host network proximity into account as in the SP join policy, but does not consider other network attributes like link latencies, congestion, or peer bandwidths. Moreover, these quantities are dynamic, and it is important that the topology is rearranged in keeping with dynamic measurements of these quantities. Indeed, a lot of the previous work [4, 5, 6, 7, 11] in end-hosts multicast has focussed on these aspects, and suggested useful heuristics that can be included in the protocol to enable low penalties in efficiency metrics. We indicate below how such heuristics can be used with our tree management policies to achieve similar efficiencies.

Nodes x and y that seek to establish a data-transfer session from x to y can compute a cost function $F(x,y)$ to characterize network proximity, unicast latency, and bandwidth capacity between them [5]. Given such a cost value, new members can join the multicast tree as suggested below.

1. *Knock-Downs* : Each node n periodically recomputes $F(n,c)$ for each of its children c. For each incoming peer x that contacts a node n in the tree, $F(n,x)$ is computed. If n is unsaturated, it accepts x as a child. If n is saturated, it compares $F(n,x)$ with the maximum $F(n,c)$ among all of its children c. If $F(n,x)$ is cheaper, then n accepts x as its child, and redirects the child with the most cost. Otherwise, x contacts each of n's children, and tries to connect to the least cost candidate node.
2. *Join-Flip*: Each internal node n periodically computes $F(p,n)$ for its parent p. A new incoming client x sends a $F(p,x)$ along with its request to n. If $F(p,x) + F(x,Y)$ is less than $F(p,n) + F(n,x)$, n closes its data transfer session with p, redirects x to p, and then itself to x. Such a policy is used in [7] to form efficient trees.

We can also define incremental optimization heuristics that rearrange an *existing* overlay to improve efficiency.

1. *Maintain-Flip*: This policy is similar to the *Join-Flip* discussed above. Instead of working with a new incoming client, existing nodes in the overlay tree are swapped. Each node n computes $F(p,n)$, $F(n,c)$ for its parent p and each of its children c periodically. If $F(p,n) + F(n,c)$ is more than $F(p,c) + F(c,n)$ for some child c, the positions of n and c are swapped. The work in [7] provides similar policies to build near-optimal trees. The work in [4, 6] uses such measures as part of their routing protocols to build a spanning tree with low efficiency penalties.
2. *Leaf-Sink*: The authors in [18] propose a centralized algorithm to sink unstable or low-bandwidth nodes to the bottom of the tree. They also propose Redundant Virtual Links in the tree, which allow nodes to receive the same packet from multiple different sources, leading to decreased packet losses.

In this work, we do not evaluate these heuristics for the resulting overlay efficiency. We remark that such heuristics have been studied in respective projects, and shown to perform reasonably well. As we indicated above, the policies can be adapted to PeerCast, trading greater state maintenance for increased efficiency. Instead, we will focus on studying the effects of a transient infrastructure on the end-system performance metrics.

V. Experimental Evaluation

In this section, we study the performance of PeerCast. We implemented PeerCast, and performed empirical field tests. The aim of such tests was to ascertain values for basic parameters that would be used in simulations for evaluating the various policies for end-system metrics.

A. The Testbed

We used Apple's open source Darwin Streaming Server at the source to simulate a live broadcast by looping over a video clip. Apple's QuickTime was used as the application level client at the peer nodes. The peering layer was implemented in Python. The peering layer used the redirect mechanism to discover unsaturated nodes during join, leave, and failures. The join policy was set to Random, and the leave policy to Root

(RT). Failures of nodes were detected by the absence of heartbeats.

The experiments were performed on Sun Ultra 60 computers with 450MHz UltraSPARC-II processors, 256MB RAM, 1GB swap disk and 6.3GB disk running Solaris 7. The machines were connected on a 100Mbps ethernet. The video resolution was 240x180, and the sound sampling rate was 11 KHz mono. The goal of the experiments was to gather data that could be used to set parameter values in our simulations. Hence, the experiments were designed to be simple, and measured the following parameters.

1. *RTP/UDP packet delays*, defined as the difference between the time a packet was sent by the source, and the time the same packet was received by a client.

2. *Time to first packet*, defined as the time elapsed from the instant a new node sends a subscribe request to the source, to the instant the node receives the first stream packet.

3. *RTP/UDP packet drops* : We note that the encoding of a video stream has sufficient redundancy, such that a few isolated packet losses have almost no perceptible effect to human viewers. Thus, consecutive packet drops, called *lost-blocks*, are more important. The width of such lost-blocks, and the number of times they occur influences end-system metrics.

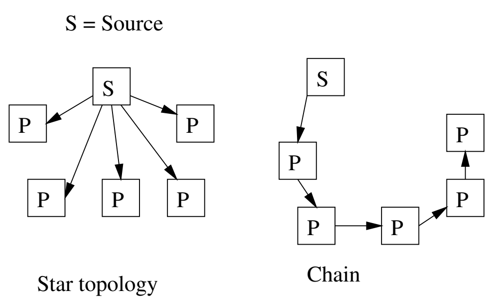

Fig. 5. Topologies used in the experiments

In one set of experiments, the source was placed in the same intranet as all the clients (peers). Measurements were then made for the peer topologies shown in Figure 5. A *Star* topology models the conventional architecture, in which all clients connect directly to a single source. Star models the best solution using a unicast service that does not use aggregation as provided by multicast. The *Chain* topology showed us the effects of increasing number of hops of a peer from the source. Note that experiment results will be biased toward peers having a high bandwidth capacity. Hence, the reader should not interpret the results of this section as absolute predictions, but rather as illustrations of performance trends. In another set of experiments, the same experiments for topologies of Figure 5 were repeated with the source placed across the Internet from the clients, all of which were in the same intranet.

B. Results from Empirical Tests

Data was collected over multiple runs for each setting, and then averaged to obtain a 90% confidence level. Our results are summarized below.

1. *Packet delays* increase linearly with the number of hops between the source and client. The average packet delay observed for a Chain topology of depth 5, when the source and the clients were in the same intranet is shown in Figure 6. Note that the order of magnitude of the time delays, even at five hops, is about $0.14s$, while QuickTime clients buffer the stream (at the application level) for upto $30s$. For the end-user, the delays due to buffering are 2 orders of magnitude greater than those due to increased hops. The linear growth implies that the height of the tree can be as high as 15, before hop delays become 1.5% of buffering delays. Thus, the number of clients supported can be in thousands (for a tree with an average degree of 3) before delays due to increasing hops become a dominating factor.

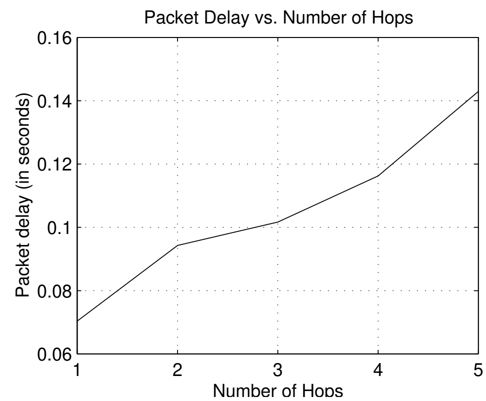

Fig. 6. Average packet delay for Chain with s and clients in the same intranet.

2. *Packet losses* : For a Chain topology of height 5, with the source placed across the internet, we observed an overall 6% packet loss rate. Moreover, the number of packets dropped across a hop within the intranet was observed to be close to half the number of packets dropped in the first hop across the internet. Thus, it is preferable for nodes to support children within their own intranet.

TABLE I
TIME TO TRAVERSE LEVELS USING REDIRECT

Number of hops	1	2	3	4	5
Time from 1^{st} hop	0	0.016	0.027	0.039	0.051

3. *Time to discover unsaturated node* : Table I shows that the measured times to traverse levels in Chain were observed to be linear in the number of hops. Notice that the time taken is two orders of magnitude smaller than the duration of QuickTime buffer. We conclude that the redirect mechanism is an efficient primitive, and can support a fast discovery protocol.

C. Inferences from Empirical Tests

The above experimental observations have the following ramifications on design rules. The $30s$ application-level buffer allows the peering layer at a node an interval of time over which packet delivery, node deletions and node failures have to be achieved. Since the buffer size was fixed for handling jitter, the topology maintenance algorithms should not use a significant portion of the buffer. A design decision might be to restrict buffer usage for topology maintenance to less than 5%. Then, the time to discover an unsaturated node, and start the stream after establishing a data-transfer session with it should be less than $1.5s$. We observed that the data-transfer session establishment between nodes across the Internet took $1.3s$. Thus, the time to discover an unsaturated node can be upto $0.2s$, which corresponds approximately to 20 redirects. Thus, we conclude that a tree of height 20 can be explored within the time bounds.

However, increasing the height of the tree implies that the deletion of a node will affect a larger number of clients. Moreover, simultaneous deletions could lead to arbitrarily bad trees, or even worse, fragmented trees. The different join and leave policies then play an important role in keeping the topology well-behaved and ensuring good end-system performance. Since it is difficult to experiment with hundreds of nodes, we performed simulations to validate the architecture for the proposed join and leave policies. The simulations are discussed in the next section.

VI. Model for End-system (peer) Behavior

In this section, we propose a model for an end-host (peer) acting as an application router. Specifically, the model is designed to enable a study of effects of router-transience on end-system metrics. Thus, the model is used in simulations to answer the following questions.

- What are the effects of the join policies on the end-system metrics? What are the parameter ranges over which the join policies can scale? Which join policy is the optimal, and under what conditions?
- What are the effects of the leave policies on the shape of the multicast tree? Are the leave policies successful in keeping the tree connected, compact, and stable?
- Which combinations of join and leave policies perform the best with respect to the packet loss metrics? Are the observed packet losses comparable with the Star topology?
- How do the policies compare against Star on the response time metric of time to first packet? Does the architecture scale gracefully with increase in the size of the client-base?
- Live streaming media often gives rise to flash crowds at the source, as the clients attempt to subscribe within a short interval of the start of the event. How does the architecture respond to such flash crowds?
- The clients receiving a stream have unpredictable lifetimes. How does the architecture behave under a range of expected lifetimes of the clients? Does the tree remain stable and connected for small lifetimes of clients? What is the effect of the average lifetime of nodes on the end-system performance enabled?
- The clients may have a range of bandwidth capacities, from T3 to symmetric DSL. What is the effect of available uplink bandwidth capacity at a node on the end-system metrics? How much upstream bandwidth is required at the nodes for the solution to be feasible?

A. The Model

We start with a discussion of a model for all clients present in the same intranet, and then generalize the model to clients spread over the internet.

A stream comprises of a sequence of packets over time. Time is modeled as a discrete, monotonically increasing quantity. There is a source s of the stream which sends out one packet every t_s time steps. The source s can support a maximum of c_{max} children (except for Star, when it can support *all* clients). The length of the stream is t_{len} time steps. A stream packet can be lost on a hop with probability p_{loss}, independent of other packets.

The number of clients is given by N. For simplicity, clients are homogeneous. In addition to getting the stream for itself, each node has upstream bandwidth capacity to support c_{max} children. Nodes communicate by sending messages to each other. The delivery of a packet or message to all destination nodes in the tree is simulated in one time unit. A node sends and receives a heart-beat message from its parent, and each of its children once every t_{hb} time steps. Each node has an event queue associated with it, which stores all outstanding events the node has to act upon. Events correspond to RTSP methods: connect request (DESCRIBE), hand-shake request (SETUP), and start-stream request (PLAY). A node can only act on a single event at each time step, while other events wait in the event queue. It takes t_c, t_{hs}, and t_{ss} time steps to process each of the three events respectively.

A client can be in 3 states: *inactive*, *active*, or *transient*. A node is said to be inactive when it is not subscribed to the stream. A node is active when it is subscribed to the stream, and is getting the stream. A node is transient if it is subscribed to the stream, but is not getting the stream. The transitions between states occur as shown in Figure 7. For example, an inactive node transitions to transient state by sending a subscription request to s; an active node n goes to transient when one of its ascendants unsubscribes (or fails), cutting off the feed from s to n.

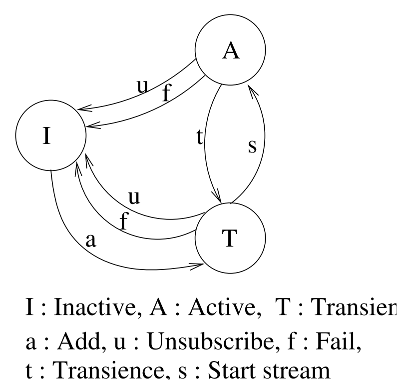

Fig. 7. State transition diagram of a node

At the start of the simulation, only the source is active, and all the clients are inactive. The source stays active for the duration of the stream. A client node can transition from one state to the other at each time step, with a probability independent of the states of other nodes. For example, an inactive node can transition to transient with a probability of p_{join}. If a node is active, it can unsubscribe and transition to inactive state with probability p_{unsub}. An active node can fail and change to inactive state with probability p_{fail}. A node in the transient state can also transition to the inactive state by unsubscribing (with a probability of p_{unsub}) or failing (with a probability of p_{fail}).

When a node changes state from inactive to active, it sends a subscribe request to s (in the form of a connect request message). The peering system then accommodates the node into the source-rooted tree as specified by the *join policy*. A node which unsubscribes from the stream, sends a redirect message to its parent, and each of its children as specified by the *leave policy* in the next time step. A node which fails, just changes its own state to inactive. The parent of such a node, and each of its children detect the failure when the next heart-beat is missed. Each of the children (or descendants) then recover as specified by the *leave policy*, starting the recovery from the time instant the failure was detected.

To generalize the above model to clients spread across the internet, we extend the single intranet model by adding the following characteristics. Each node, and the source s, can belong to one of I intranets with an equal probability. A stream packet can be lost on a hop between two nodes in the same intranet with a probability of p_{li}, and across the internet with a probability of p_{lo}. Each node can support a maximum of c_{mi} clients in the same intranet, and a maximum of c_{mo} clients across an internet. Rest of the parameters, and modeling remain unchanged from the single intranet model.

B. Candidate Parameter Values

The values for some parameters in the model were measured from empirical runs, while others are controlled variables. The parameters and their values are listed in Table II for a ready reference. We varied the guessed values in some of the simulation runs in Section VII to study their effects on end-system metrics. Unless mentioned otherwise, the simulations have parameters set to default values.

TABLE II

PARAMETERS USED AND THEIR TYPICAL VALUES.

Parameter(Symbol) for Intranet	Default Value	Nature
Packet Rate (t_s)	30 pkts/sec	Measured
Stream Duration (t_{len})	1 hour	Controlled
Number of Clients (N)	1000	Controlled
Maximum Children (c_{max})	10	Controlled
Heart-Beat Interval (t_{hb})	1 sec	Controlled
Connect Interval (t_c)	1/10 sec	Measured
Hand-Shake Interval (t_{hs})	1/2 sec	Measured
Start-Stream Interval (t_{hs})	1/30 sec	Measured
Prob. of Packet Loss (p_{loss})	1 in 100 pkts	Measured
Prob. of Addition (p_{join})	Once in 5 mins	Guessed
Prob. of Unsubscribe (p_{unsub})	Once in 30 mins	Guessed
Prob. of Failure (p_{fail})	Once in 50 mins	Guessed
Number of Intranets (I)	10	Controlled
Maximum Intranet Children (c_{mi})	10	Controlled
Maximum Internet Children (c_{mo})	5	Controlled
Pkt Loss Prob. over Intranet (p_{li})	1 in 100 pkts	Measured
Pkt Loss Prob. over Internet (p_{lo})	1 in 50 pkts	Measured

The stream length was set to an hour, which is the duration of some sporting events. The probability of failure of a peer is listed as "once in 50 mins" to give an intuition to the reader about the time scales involved. To be more precise, the phrase means that an active node can fail with $p_{fail} = 1/(50 \times 60)$ each second. The values for p_{unsub} and p_{join} are similarly described.

VII. Simulation Results and Inferences

In this section, we study the ability of PeerCast in insulating the effects of transience of application-routers from the end-application. We simulated the PeerCast policies on a set of clients following the peer behavior model above. Among the join policies, we studied Random, RR, and SP; the leave policies simulated were GF, GFA, RT, and RTA. We also studied the performance of the traditional centralized solution, modeled as a Star topology. The measurements for Star serve as a baseline against which the policies are compared. The source in a Star topology is assumed to have enough bandwidth capacity to serve all its clients simultaneously. However, the source for PeerCast is simulated as a node with similar capacity as the rest of the clients. In this respect, *the comparisons are biased toward Star*. However, as we shall see, the PeerCast policies perform comparably in most scenarios.

Statistics for each node were collected for a simulation run, and then averaged over the number of nodes. Multiple runs were performed to obtain 90% confidence levels in the values obtained. The metrics recorded were packet-losses and time to first packet (TTFP) and are defined as follows. Let N be the number of clients receiving the stream. Let P_{ia} be the number of packets lost by node i in the active state, and P_{it} be the number of packets lost by i in the transient state. Let P_i be the number of packets that are sent while i was in the active or transient states. Then, the average active-state packet-losses are computed as $(\sum_{i=1}^{i=N} P_{ia} * 100/P_i)/N$. Similarly, the average transient-state packet-losses are computed as $(\sum_{i=1}^{i=N} P_{it} * 100/P_i)/N$. The average total packet-losses are computed as $(\sum_{i=1}^{i=N} (P_{it} + P_{ia}) * 100/P_i)/N$. Let T_i be the time taken in seconds to receive the first stream packet at node i after a subscription request was sent by i. Then, the average TTFP is computed as $(\sum_{i=1}^{i=N} T_i)/N$.

A. Join/Leave Policies and Cost of the Tree

We start by evaluating the cost of the multicast trees resulting from different policies in a single intranet model. As indicated in Section IV, define the cost C of a tree T to be $C_T = \sum_{i=1}^{N} h_i$, where h_i is the depth of node i, and N is the number of nodes in T. The multicast tree maintained by a policy evolves over time as nodes join and leave. We define the cost of a policy C_P as the average of the costs of the tree instances that result over the duration of the stream. Since an almost-complete source-rooted tree minimizes C, we define an *Opt* policy as a hypothetical policy that maintains an almost-complete tree over the current set of clients.

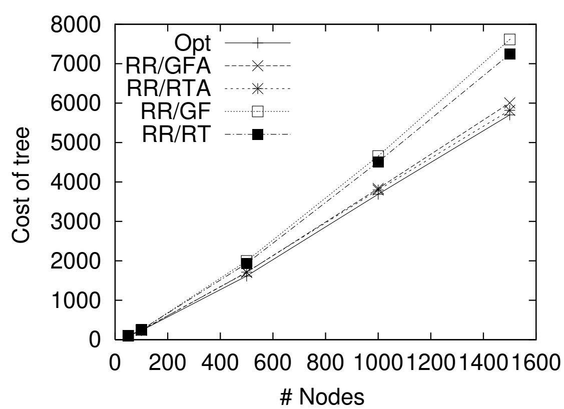

Fig. 8. Cost of policies for various sizes of client-base

Figure 8 shows the cost of different policies for different sizes of client-base, in a single intranet model with c_{max} set to 3. The curves shown are for the RR join policy, and different leave policies. The curves for Random join policy are similar, and not shown for clarity. The X-axis plots the size of the client-base. The Y-axis plots the cost of a policy. The cost of the current tree was computed once every second over the duration of the stream. The cost of a policy was taken to be the average of the costs observed at each second. We observe that RR/RTA and RR/GFA have close to optimal costs. The RR/RT and RR/GF curves diverge from the optimal as the size of the client-base increases. At $N = 1000$, the cost of RR/GF is $1.262\times$ the optimal cost. We conclude that the RR/GFA and RR/RTA leave policies result in near-optimal trees. The RR/GF and RR/RT policies are close to optimal for small client-base sizes, but degrade with increasing size of client-base. We expect the RR/GF and RR/RT policies to perform reasonably well for hundreds of nodes.

However, note that cost of a tree is an aggregate statistic over the depths of nodes. In the next section, we investigate shape of the tree to understand the tail distribution of depths of nodes.

B. Join/Leave Policies and Shape of the Tree

The cost of different policies depends on the shapes of trees that result from changes in the multicast group. In the absence of unsubscribes or failures, the resulting multicast tree would be almost-complete.

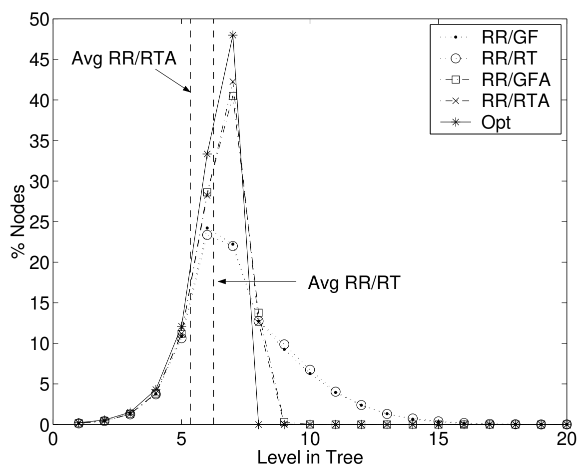

Fig. 9. Effect of leave policies on dist. of nodes across levels.

Figure 9 shows the distribution of the depth of nodes in the multicast tree for GF, RT, GFA, and RTA leave policies with the join policy set to RR, and c_{max} set to 3. The X-axis plots the level number in the tree. The Y-axis plots the average percentage of nodes subscribed to the tree at a certain level. Also indicated in the graph are average depths of a node under RT and RTA policies. We can see that RTA results in a smaller mean depth tree than RT. The GFA and RTA curves peak and fall to 0 in a small number of levels, indicating a desirable compact tree. The GF and RT curves rise with GFA and RTA, but have a smaller percent of nodes at their peak in the middle levels. Instead, the remaining percentage of nodes fall off gradually along higher levels, with less than 1.5% nodes at depths greater than 12. The reason is that in RT and GF, the failure of a node n results in each of its child rooted sub-tree moving together, causing an increase in height. In RTA, all the descendants of n independently contact s and get distributed across the tree, causing the height to remain balanced. A high depth tree is undesirable because end-system metrics increase linearly with levels. However, as we experimentally observed in Section V, the buffering at the application level ensures that cumulative delay is acceptable until approximately 15 hops. Thus, even the trees constructed by GF and RT ensure reasonable end-system performance.

C. Leave Policies and Recovery from Transience

As we saw above, GFA and RTA result in compact, and near-optimal trees. Unfortunately, the compact trees from GFA and RTA come at the price of increased effort in tree maintenance, leading to longer transience times for nodes in the overlay tree.

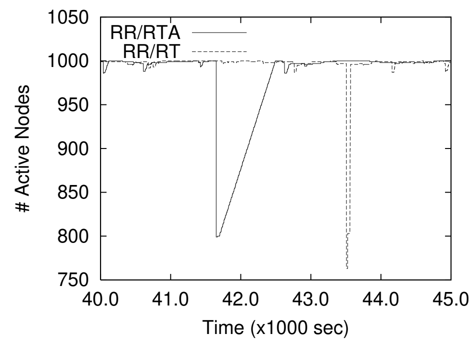

Fig. 10. Recovery times for respective dips for RTA and RT.

Figures 10 show a sample scenario for RR/RT and RR/RTA policies. The X-axis plots simulation time, while the Y-axis plots the number of active nodes in the overlay tree. The deletion of an intermediate node under both RTA and RT leave policies causes approximately 200 nodes to go into transience. The RT policy curve recovers much faster than the RTA policy as the transient nodes get assimilated faster. The reason is that, when a node unsubscribes (or fails), the target (source) in RTA is swamped by requests from *all* the descendants. A *convoy effect* sets in, causing the event queue to grow in size, as the target node processes each request in turn. As a result, an affected node spends a larger amount of time before being assimilated into the tree. The GF and RT policies avoid such a convoy effect, by having only the direct children of an unsubscribed (or failed) node recover.

We conclude that the trees formed by GFA and RTA are desirable, while those formed by RT and GF are acceptable. However, GF and RT lead to faster adjustment of transient nodes into the tree than GFA and RTA. As we shall see in Section VII-E, the latency results in an unacceptable transient state packet losses at the affected nodes for GFA and RTA, making these policies impractical for a large client-base.

D. Join Policies and End-system Metrics

We also evaluated the Random and Round-Robin join policies with respect to packet-losses and TTFP for RT leave policy on a single intranet model. The join policies were observed to perform very similar to each other on both the metrics for a wide range of client-base size. Hence, we note that the results from our simulations discussed below which use RR hold for Random as well, and vice-versa.

E. End-system metric: Packet Losses

We now study the effects of join and leave policies on the packet loss end-system metric, for the multiple intranet model. We differentiate between packet losses observed in the active state, from those observed in the transient state at a node. Active state packet losses correspond to isolated packet drops, with small expected block-length. Hence, the end user is not expected to be affected by such losses. In contrast, when a node enters the transient state, the packet losses are consecutive. For a large block-length, the disruption becomes visible to the end-user. Hence, it is also important to record the sizes of lost packet-blocks for each policy.

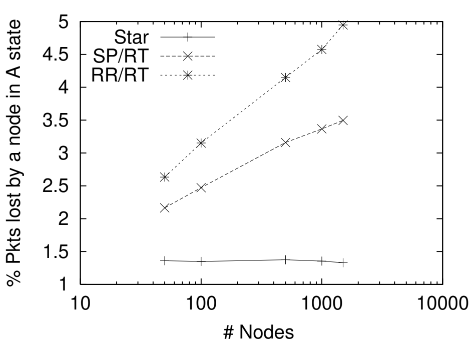

Fig. 11. Active state pkt. loss for join policies against Star

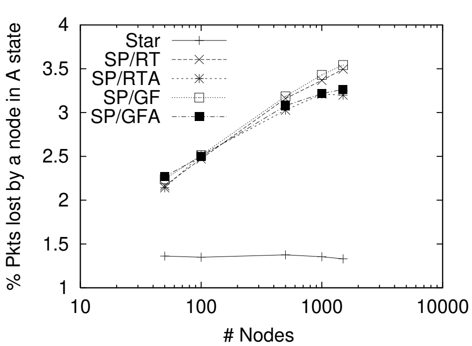

Fig. 12. Active state pkt. loss for leave policies against Star

Figures 11 and 12 show the average packet losses observed at a node in the active state as the number of clients subscribed to the stream increases. The X-axis plots the number of clients on a log scale. The Y-axis plots the percentage of packets lost in the active state by a node.

Figure 11 compares the performance of RR, and SP join policies with the RT leave policy against the Star architecture. Star performs the best on the packet loss metric. However, note that in Star, the source is supporting upto 1500 clients directly, and thus has enough bandwidth to support 1500 replicated streams. For a much reduced workload (only 10 clients per node) SP/RT and RR/RT suffer $2.65\times$ and $3.75\times$ more packet losses in the active state than Star. Among the join policies, SP consistently outperforms RR. Policy SP uses the intranet information of a node to minimize on across-intranet hops. The chances of packet loss are more for hops across an internet, than within an intranet. As a result, packet losses are fewer. We conclude that SP should be the choice of join-policy when the clients are spread across intranets.

Figure 12 compares the performance of RT, GF, RTA, and GFA leave policies with SP join policy against the centralized Star architecture. For 1500 clients, SP/RT, and SP/RTA suffer $2.65\times$ and $2.40\times$ more packet losses in the active state than Star. The increase factor is roughly equal to the average height of a node in the tree formed for the policy, and can be explained by the cumulative packet-loss effect: a packet lost by a node higher up in the tree cannot be forwarded to any of the descendants of the node. Thus, the average packet loss increases proportionally with height of the tree.

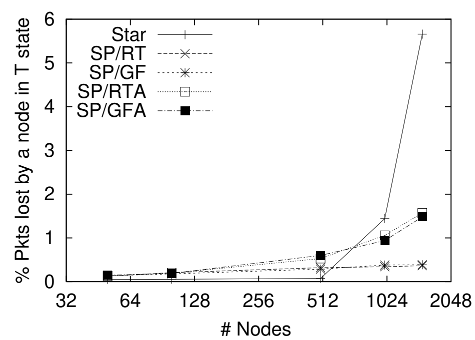

Fig. 13. Transient state pkt. loss for leave policies against Star

Figure 13 shows the average packet losses observed at a node in the transient state as the number of clients increases. The X-axis plots the number of clients on a log scale. The Y-axis plots the percentage of packets lost in the transient state by a node. The packet losses in the transient state in Star are less than SP/RT and RR/RT for small number of clients (≤ 500), but increases rapidly to nearly $4\times$ more than SP/RT for 1000 nodes, and $15.43\times$ for 1500 nodes. These observations can be explained as follows. In Star, a node is in transient state when it has sent a subscription request to s, and is waiting for the stream to start. The source s has to process each client request one at a time. As the number of nodes increases, the expected number of nodes making a subscription request increases. The client requests add up in the event queue, causing each client to spend more time in the transient state, and hence more transient state losses.

In contrast, the transient losses for the PeerCast policies of SP/RT and RR/RT increase slowly with the number of clients. The increase in subscription load is quickly spread away from a single node using the faster redirect primitive, causing smaller transient times and fewer packet drops. Thus, we observe that the fast recovery using redirect shields the user from data-transfer failures. We conclude that PeerCast gains over Star in the end-system metric of transient state packet losses. The observed behavior suggests that RT and GF leave policies are to be preferred over RTA or GFA.

As we observed above, consecutive packet losses have a

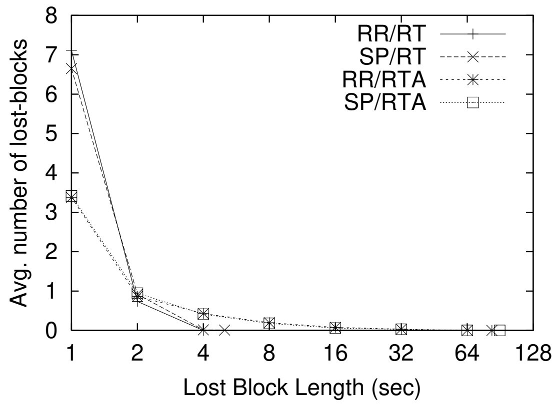

Fig. 14. Effect of policies on the avg number of outages at a node.

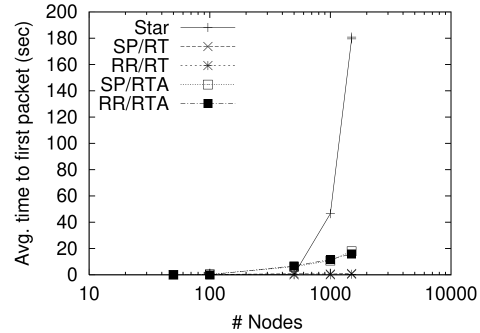

Fig. 15. Effect of size of client-base on average TTFP.

more significant effect on end-system performance than isolated packet drops. We recorded the sizes of observed lost packet-blocks at each node during its lifetime. Figure 14 shows the average number of lost-blocks observed at a node for the different policies. The X-axis plots the duration of a lost block rounded off to the next higher second, while the Y-axis plots the average number of times a node experiences such durations of lost-block. Each node was set to have a maximum intra- and extra-net degree of 3. We observe that RR/RT and SP/RT curves reduce to 0 within 5 seconds, with most of the lost-blocks in RR/RT and SP/RT being less than 1 second in duration for a 1 hour long stream. Moreover, a node is expected to have close to 7 outages of any length over a 1 hour stream for RR/RT and SP/RT. In contrast, RR/RTA and SP/RTA curves taper off smoothly reducing to 0 in almost 90 seconds, with approximately 3 outages of a duration more than 6 seconds. The reason for such increased outage lengths for RR/RTA and SP/RTA are the larger intervals of time spent by a node in the transient state. Thus, RR/RTA and SP/RTA cause more frequent disruptions in the stream, that could be visible to the end-user. We conclude that the RR/RT and SP/RT policies do cause a disruption in the stream, but the outages are both fewer and shorter. Such outages could be masked by increased redundancy in the encoding of the multimedia stream.

F. End-system metric: Time to First Packet (TTFP)

The TTFP metric is important as it records the response time of the system. A new client sends a subscribe request, and waits in the transient state until it starts receiving the stream. For good end-system performance, the response time should be small, and scale with the number of clients.

Figure 15 shows the average TTFP observed at a node as the number of clients subscribing to the stream varies for the multiple intranets model. The X-axis plots the number of nodes in the client-base, while the Y-axis plots the time in seconds on a logarithmic scale. We observe that the response time for Star increases sharply as the number of nodes increases beyond 500. Until 500, the source is fast enough to set up a stream for each client as the requests for subscription come in. However, as the number of nodes increases, the expected number of nodes making a subscription request increases. The client requests add up in the event queue at the source, causing a client to experience a larger TTFP.

The SP/RT and RR/RT policies, in contrast, scale much better with lower response times as the fast redirect primitive operates to avoid a convoy effect at a single source as in Star. An interesting point to note is that the response time increases for SP/RTA and RR/RTA as well, though more gracefully than Star. When a node unsubscribes or fails under the RTA leave policy, all the descendants of that node contact the root for adjustment in the tree. Thus, the root (source) has to process more requests than in the RT leave policy, causing the TTFP to increase for nodes in the RTA policy.

We conclude that SP/RT and RR/RT provide good scalability with respect to increasing client-base size. The response times, for 1000 nodes are $1.5s$ for SP/RT, and $1.02s$ for RR/RT. In contrast, Star causes nodes to experience $46.5s$ wait time. Thus, PeerCast gains on the time to first packet end-system metric as it distributes the processing load away from the single source.

G. End-system metrics under a flash crowd

We next studied the performance of the system for a flash crowd. For live streaming events, anecdotal evidence suggests that the source experiences a large number of subscribe requests within a short interval from the start of the stream. We simulated flash crowds by varying the probability of joining, p_{join}, of a client in the client base, while keeping the probabilities of unsubscribe p_{unsub} and the probability of failure p_f as 0. By setting p_{unsub} and p_f to 0, we ensure that a node does not unsubscribe after its first subscribe. As p_{join} increases, the expected time in which an inactive node subscribes to the stream decreases. Or, for the same time instant, the number of nodes sending a subscribe request to s increases. Thus, the size of the flash crowd can be controlled by changing p_{join}.

We compared the TTFP and the packet-losses metrics for RR/GF against Star for a single intranet model. Figure 16 plots the expected time to subscribe for a node on the X-axis, and the average TTFP on the Y-axis. As the expected time to send a subscribe request for a node decreases, the TTFP for Star increases rapidly. For $N = 1000$ nodes requesting the feed within 1 minute, Star performs $10\times$ worse than RR. Such a degradation is because s becomes the bottleneck in handling subscribe requests, and setting up a session for each client. RR with its lighter redirect, distributes load from the root quickly over the already subscribed nodes.

We observed that the packet losses in transient state follow a

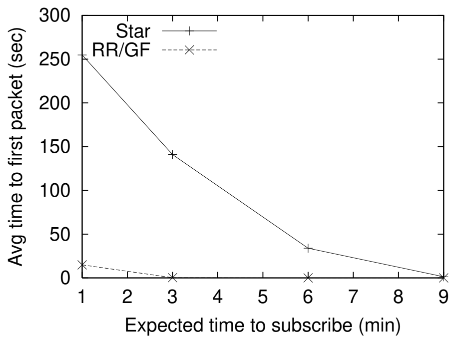

Fig. 16. Effect of flash crowds on TTFP at a node

similar trend as the TTFP for both policies, for the same reasons. The Star architecture was found to perform $15\times$ worse than RR/GF for 1000 nodes requesting the feed within 1 minute.

We conclude that PeerCast avoids flash crowds from overwhelming the single source as in Star. PeerCast returns better end-system performance than Star for a range of flash crowd sizes.

H. End-system metrics and unsubscribe rates

In PeerCast, the nodes subscribed to a stream serve as a source for other clients, and feed them the stream. If an upstream node unsubscribes or fails, the stream to their descendants in the tree is severed, and the downstream nodes are affected. However, if the affected nodes can recover quickly by discovering an alternate source, and restoring the stream feed, the failure can be made transparent to the end-user. Clearly, if such disruptions are frequent, the multicast tree will become unstable, and the end-system metrics at a node will suffer. We varied the lifetime of a node by varying its probability of unsubscribe, p_{unsub}, while keeping its probability of joining p_{join}, and probability of failure p_f, fixed. As the p_{unsub} increases, the probability that a live node unsubscribes at each time step increases, causing its lifetime to decrease. We then studied the end-system metrics for a range of node lifetimes.

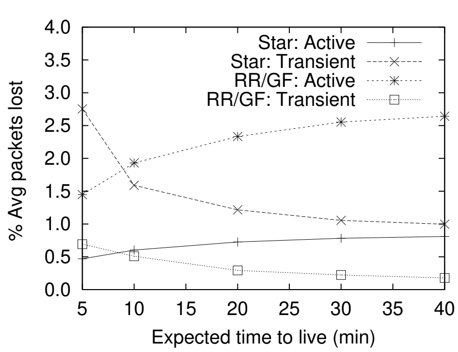

Fig. 17. Effect of unsubscribe rates on packet loss at a node

In Figure 17, the average packet losses observed at a node are plotted against the expected lifetimes of the nodes. The X-axis plots the expected lifetime in minutes. The Y axis plots the average percentage of packet losses observed, in the active and the transient states, at a node. The curves indicate the losses for RR/RT, and for Star. We first note that the active state packet losses increase for Star with increasing lifetimes. This is because a node now spends more time in the active state. The active state packet losses increase for RR/RT as well, but faster than Star. The reason is that as lifetime of nodes increase, the number of nodes subscribed to the stream increases, causing the depth of the tree to increase. As the depth increases, the number of hops traveled by a packet increases, increasing the active state packet drops observed at a node.

Figure 17 also shows that the transient state packet loss increases with decreasing lifetimes for Star. Since the nodes which had unsubscribed in the past, re-subscribe with a probability of joining p_a, the frequency of subscription requests at the source increases. The increased frequency of concurrent subscribe requests results in an increase in the TTFP, and thus higher transient state packet losses for Star. The RR/RT also witnesses an increase in transient packet losses with decreasing lifetimes. We observe that the packet losses increase to 0.7% for an expected lifetime of 5 minutes. The increased transient packet losses are a concern for PeerCast as the packets that are lost are consecutive, and the disruptions can be visible to the end-user.

We also studied the TTFP values for changing lifetimes of nodes. As expected, RR/RT policy shows an increase in TTFP with decreasing lifetimes. However, the observed values were upto an order of magnitude lesser than for Star.

We conclude that the PeerCast architecture would not be feasible for applications where clients subscribe to a stream for short durations (5 mins or less), and leave. For a large range of higher lifetime values, PeerCast performs comparably with Star, and offers a reasonable end-system performance.

I. End-system metrics and bandwidth at a client

Since clients forward the stream to their peers, the performance of the PeerCast architecture is sensitive to the bandwidth available at the nodes participating in the system. We characterize the bandwidth at a node by the number of maximum children the node can feed. Figures 18 and 19 show the effects of different bandwidth available at the client on the TTFP, and packet loss metrics. The performance of different join/leave policies is plotted against Star, in which the source is deemed to have enough bandwidth to support all the clients simultaneously. We stress again that such a comparison between Star and PeerCast is not fair because of the $1000\times$ bandwidth capacity assumed for Star, and denied to PeerCast. But it is still an interesting comparison to see how far PeerCast is from the best possible scenario.

In Figure 18, the value of maximum children, c_{max}, that can be supported at a node is plotted on the X-axis. The average active state packet losses at a node are plotted on the Y-axis. We observe that the active state packet losses increase with decreasing c_{max}. The reason is that as c_{max} increases, the depth of the resulting multicast tree decreases. Packets now have to cross fewer hops, leading to reduced losses. Moreover, as the depth of the tree decreases, the number of unsaturated nodes available at a smaller depth increases, leading to faster subscriptions. Thus, the nodes spend less time in transient state, and experience fewer transient packet losses as seen in Figure 19. We note that the transient packet losses increase with decreasing bandwidth

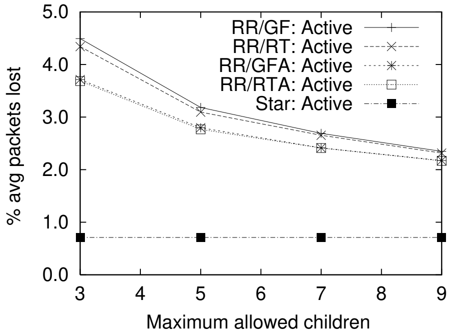

Fig. 18. Effect of bandwidth capacity at a node on active state pkt loss

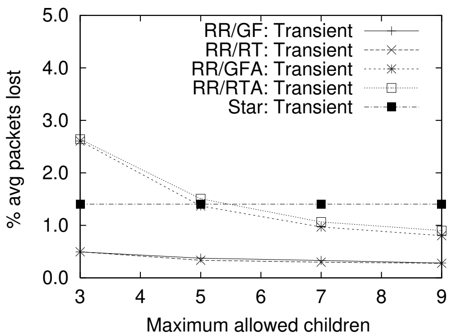

Fig. 19. Effect of bandwidth capacity at a node on transient state pkt loss

at a client, increasing rapidly for RTA and GFA to more that 1%. Since transient state packet losses are consecutive, RTA and GFA are not suitable for clients with small bandwidth capacity. On the other hand, RT and GF scale much more gracefully and experience losses less than 0.5%. Small transient state packet losses result in a good end-system performance.

We conclude that for high bandwidth clients, all the policies perform favorably with respect to Star on the end-system metrics. For low bandwidth clients, RTA and GFA degrade rapidly. The RT and GF leave policies scale gracefully across a range of bandwidth capacities at clients, and should be the preferred leave policies for PeerCast.

VIII. Related Work

The problem of distributing high-bandwidth streaming data from a single source to a large number of clients has witnessed renewed interest over the last couple of years. Several contemporary projects have argued for moving the multicast feature up from the IP-layer to the application-layer. The case was quite forcibly put forth first in [3, 4, 6]. Yoid [3] was one of the first proposals that seeks to provide an overlay toolkit for diverse applications. Since then end-hosts multicast has been the focus of some excellent work in the context of P2P systems, and overlay networks formed on proxy-hosts. The different projects can be classified as shown in Figure 1 in Section I.

The proposals mapped to the left in Figure 1 have been proposed for tens of nodes. ALMI [11] constructs and maintains a single overlay tree for packet forwarding, and has the desirable feature of being non-source specific. However, it has a centralized group management policy, that limits its scalability with respect to both the number, and small lifetimes, of nodes. End System Multicast [4] constructs a mesh that is incrementally improved. The group management protocol has been designed for small sparse groups, with each node maintaining a list of all other members in the group. The authors extend their work in [5] to provide better mesh optimization methods and present results of experiments showing the viability of End System Multicast. The authors also study the fundamental efficiency metrics for the application-level model. However, the effect on end-system metrics (apart from packet delays) due to varying end-host behavior is not studied.

The top-right of Figure 1 has Overcast [7] and ScatterCast [6]. These solutions make a distinction between end clients, and infrastructure nodes placed strategically in the network. As such they are not amenable to be used in P2P systems, where the existence of such infrastructure cannot be assumed. The nature of the problem changes significantly when application-level routers are assumed to be part of the infrastructure, rather than end-hosts with unpredictable behavior.

In the context of constructing and maintaining tree-based overlay networks, Cohen and Kaempfer [15] give theoretical results on the problem of finding a network-optimal "maximum-bottleneck" multicast tree. They show the problem to be NP-complete, even for a source-specific multicast tree with centralized group management policies. They suggest heuristics for finding near-optimal trees. However, the behavior of these heuristics under node deletions, or short lifetimes of nodes is not analyzed.

The middle-right in Figure 1 are taken by solutions that build a mesh-based overlay topology on the group-members, with each node maintaining a neighborhood table. Meshes are thus less susceptible to partitioning than tree-based methods because each node has many potential parents. A routing protocol is devised to use an optimal spanning tree embedded in the mesh for stream forwarding. The routing protocol is simplified in Bayeux [9], CAN-based [8], and Delaunay-triangulation based [10] proposals. A level of indirection is created by assigning logical addresses to nodes. Geometric properties of these logical address are used to resolve routing paths. The efficiency of such logical-address architectures depends on physically close nodes getting addresses that are close in the logical space as well. The respective architectures then rely on an underlying scalable addressing and routing capability to scale to thousands of nodes.

However, mesh-based protocols inherently incur several control messages to keep the neighborhood tables of nodes updated. Several messages need to be passed when a node joins or leaves, and it takes some time for the mesh to stabilize. Hence the effects in terms of our end-system metrics of packet and packet-block losses due to peer transience, are difficult to predict. To the best of our knowledge, none of the mesh-based protocols have been analyzed for arbitrary node failures, short node lifetimes, and sudden bursts of node additions. It is not clear how long the respective mesh-based protocols take to stabilize for flash crowds, or their ability to deliver good end-system performance over an unstable mesh, or avoid partitions for several short-lifetime members. The last is especially important because partition repairing mechanisms in meshes are much more expensive than in PeerCast.

We believe there is a need to compare tree and mesh-based protocols using a peer-behavior model, apart from studying their efficiency vis-a-vis unicast. The trade-offs between the greater robustness promised by a mesh, against its greater message passing (and hence greater time to stabilize), and their effects on the end-to-end application-specific metrics need to be explored.

We remark that our work complements the issues explored in the projects discussed above. The heuristics suggested by the proposals above to maximize efficiency can be used to improve the efficiency of topologies constructed by PeerCast. In turn, the above projects can gain from our evaluations, and tune their policies to match the performance on end-system metrics observed in this work.

IX. Conclusions

In this paper, we proposed an architecture, called PeerCast, for streaming live media over a P2P network. PeerCast distributes the media stream to clients using a self-organized, source-specific, end-host multicast tree built over the clients themselves. A peering layer at each subscribed client is used to maintain the stream feed under arbitrary joins, leaves, and failures of peers in the network. The peering layer at a node uses a redirect mechanism to provide hints to a requesting client and guide it to an unsaturated server in the network. The applications at a client need not be aware of a change in the server providing the feed. Different policies for topology maintenance were suggested, with indications on the use of previously discovered heuristics to improve the efficiency of the overlay.

We argued for the necessity to evaluate proposals on end-system performance metrics. A peer behavior model was proposed that enabled us to characterize application-router behavior in a P2P network. Experiments show that the basic primitives are efficient, and extensive simulations indicate that the suggested policies perform well on end-system metrics. We can summarize the information gained from simulations as follows.

1. The PeerCast policies SP/GFA and SP/RTA construct a near-optimal overlay tree with respect to end-system metrics. The policies SP/GF and SP/RT construct trees that are within $1.2\times$ the optimal cost for upto a 1000 clients.
2. The PeerCast architecture, with SP/GF and SP/RT policies, shows the ability to scale gracefully with number of clients for upto hundreds of peers, and with expected lifetime of clients upto 5 minutes, on the end-system metrics.
3. The number of packet losses in the active state are $2.5\times$ more for PeerCast than for Star for a 1000 clients. However, Star assumes a server with enough bandwidth along a path to each client to support replicated streams, which might not be practical.
4. The number and duration of *consecutive* lost packets is small for PeerCast with SP/RT and SP/GF policies. We believe such rare and small disruptions will be acceptable to the end-user, and may be overcome by increased redundancy in the encoding of the multimedia data.
5. The response times returned are upto $4\times$ better for PeerCast policies than Star. The difference increases upto $10\times$ for flash crowds of size 1000 with the expected time to join of a node of 1 minute. The reason is that PeerCast, with its lighter redirect, avoids the convoy effect in handling subscription requests by a single source.
6. Among the various policies suggested, we recommend a join policy of SP, and a leave policy of GF or RT for a P2P domain.

References

[1] Shannon Dorey, "Streaming media - will it overtake television?," *http://www.the-surfs-up.com/news/news4p1.html*, 2001.
[2] S. Deering, "Multicast routing in a datagram internetwork," *PhD Thesis, Stanford University, California*, 1991.
[3] P. Francis, "Yoid: Extending the internet multicast architecture," 2000.
[4] Y. Chu, S. Rao, and H. Zhang, "A case for end system multicast," in *Measurement and Modeling of Computer Systems*, 2000, pp. 1–12.
[5] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang, "Enabling conferencing applications on the internet using an overlay multicast architecture," in *Proc. of the ACM SIGCOMM*, 2001.
[6] Y. Chawathe, "Scattercast: An architecture for internet broadcast distribution as an infrastructure service," *PhD Thesis, University of California, Berkeley*, 2000.
[7] J. Jannotti, D. Gifford, K. Johnson, M. Kaashoek, and J. O'Toole, "Overcast: Reliable multicasting with an overlay network," in *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000, pp. 197–212.
[8] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Networked Group Communication*, 2001, pp. 14–29.
[9] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, 2001.
[10] J. Liebeherr, M. Nahas, and W. Si, "Application-level multicast with delaunay triangulations," Tech. Rep. CS-2001-26, University of Virginia, CS Dept., 2001.
[11] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, "ALMI: An application level multicast infrastructure," in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001, pp. 49–60.
[12] "Reference deleted for anonymous review," .
[13] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications," *RFC 1889*, Jan, 1996.
[14] H. Schulzrinne, A. Rao, and R. Lanphier, "Real-time streaming protocol (rtsp)," *RFC 2326*, Apr, 1998.
[15] R. Cohen and G. Kaempfer, "A unicast-based approach for streaming multicast," in *Proceedings of the IEEE INFOCOM*, 2000.
[16] "Reference deleted for anonymous review," .
[17] P. Francis, S. Jamin, C. Jin, Y. Jin, V. Paxson, D. Raz, Y. Shavitt, and L. Zhang, "Idmaps: A global internet host distance estimation service," in *IEEE/ACM Transactions on Networking*, Oct, 2001.
[18] V. Roca and A. El-Sayed, "A host-based multicast (hbm) solution for group communications," in *1st IEEE International Conference on Networking*, 2001.