

# Streaming Live Media over a Peer-to-peer Network

H. Deshpande, M. Bawa, H. Garcia-Molina

*Abstract*—The high bandwidth required by live streaming video greatly limits the number of clients that can be served by a source. In this work, we discuss and evaluate an architecture, called *SpreadIt*, for streaming live media over a network of clients, using the resources of the clients themselves. Using *SpreadIt*, we can distribute bandwidth requirements over the network. The key challenge is to allow an application level multicast tree to be easily maintained over a network of transient peers, while ensuring that quality of service does not degrade. We propose a basic peering infrastructure layer for streaming applications, which uses a redirect primitive to meet the challenge successfully. Through empirical and simulation studies, we show that *SpreadIt* provides a good quality of service, which degrades gracefully with increasing number of clients. Perhaps more significantly, existing applications can be made to work with *SpreadIt*, without any change to their code base.

*Keywords*—peer-to-peer networks, live streaming media, peering layer, redirect primitive, transient nodes

## 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].

Since video streams are fundamentally high bandwidth applications, even a small number of clients receiving the stream are often sufficient to saturate the bandwidth at the source. For example, a T3/DS3 connection has a capacity of 45 Mbps, while a stream with 30 fps, at 320x240 pixels can have a rate of 1 Mbps. Under such conditions, only 45 clients can be provided a maximum resolution video stream. IP Multicast, if and when deployed, will eliminate the source bandwidth bottle-neck by feeding many clients from a single stream. Until then, web sites, working under a limited bandwidth resource, have to either degrade the quality of service (QoS), or turn away a large number of clients.

*EXAMPLE 1.1:* In April 2001, Doordarshan Online (at <http://www.dd.now.com>) used Akamai’s content distribution network to web-cast India-Australia cricket matches. The company (dd.now.com) had provisioned a certain bandwidth from Akamai with its average number of clients in mind. As the matches approached a nail-biting finish,

We would like to thank Gurmeet Manku for his incisive suggestions.

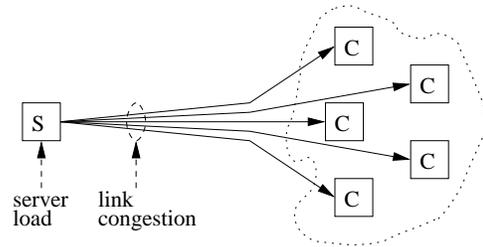


Fig. 1. Current solutions cause congestion at server and gateways

the number of clients requesting the feed increased to exceed the provisioned bandwidth. Rather than turn clients away, the web-site cut off its higher resolution feed (designed for T1 clients), and provided only a lower resolution feed (designed for DSL clients). In the final hour of the match, several clients on the West Coast were refused a feed, leading to disrepute of the site and frustration among end-users<sup>1</sup>. □

In the example above, the problem was exacerbated because a large number of clients were concentrated in a small geographic area (Stanford University). As can be seen in Figure I, the path from the server to Stanford University got congested as each data packet was replicated and sent over the link as many times as the number of clients. As a consequence, the QoS perceived by the end-user degraded.

Current trends indicate that such problems will aggravate in the near future. Increasingly, people are moving away from modems to DSL, enabling users to request multimedia feeds from various sources. It is reasonable to expect that as the edge bandwidth improves, the size of flash crowds will increase, corresponding to taller spikes in traffic. As the potential for demand increases, more events will be web-cast live by different companies to feed an increasing client base.

Designing a content distribution system which can scale with the number of clients is a challenging task. At the same time, such a system needs to be cost effective. The two requirements effectively rule out any attempt at increasing the resources of the source server. Distributed systems which use replication (e.g., Akamai) to spread out bandwidth form a viable alternative, but even such systems

<sup>1</sup>Two of the frustrated cricket fans are authors of this paper; the incident motivates the work reported here.

have a hard upper bound on the number of clients that can be supported for a given price, and a high cost associated with provisioning required bandwidth.

Single source multicast is a solution because it uses a single stream to feed all the clients. Multicast solutions have been attempted at various layers of the network stack. Link-layer multicast is primarily constrained by the fact that groups are statically set up by the network provider, and can be changed only by a reconfiguration of the network [2]. IP multicast is more flexible, but requires support at the routers. The complexity costs of group management, distributed multicast address allocation, and support for network management have limited the commercial deployment of IP multicast [3]. Application layer multicast has been more popular, and been deployed successfully by infrastructure companies [4], [5]. Reliable servers are installed across the Internet, to act as routers by splitting and forwarding the stream to clients in the same intranet. Here, the complexity and cost arises from deploying and maintaining the servers.

In such a context, it seems desirable to turn the problem on its head, and use the bandwidth resources of the clients themselves to multicast the stream. Peer to peer networks hold the promise of such a paradigm. However, building a stream distribution system from thin and unpredictable nodes, and yet providing a reasonable QoS, is a challenging problem.

In this paper, we propose a solution, called *SpreadIt* which builds an application level multicast tree over the set of clients. Each client node needs to be enabled with a basic *peering* layer between the application and transport layers. A node can function as a server and a client simultaneously. As a client, it receives the stream from some node in the network. As a server, it forwards the stream to other nodes in the network. The application (RealPlayer, Windows Media Player, etc.) gets the stream from the peering layer on their local machines. The peering layers at different nodes coordinate among themselves to establish and maintain a multicast tree. Since the clients are autonomous, subscribes and unsubscribes are unpredictable. The key challenge is in masking the transience of nodes while ensuring a good QoS. The peering layer is responsible for hiding changes in the multicast topology from the applications above.

As we show later, the *SpreadIt* architecture can support (to the order of) thousands of clients, and allows relatively cheap deployment. It enables a good QoS, which degrades gracefully with increasing number of clients. Importantly, existing applications do not need to be changed to use the peering layer, and are thus enabled for such a stream distribution. On the negative side, *SpreadIt* does not provide

rigid guarantees of reliability and availability as provided by high end commercial vendors providing dedicated multicast servers. Moreover, clients need to download, and execute the peering layer to enable the distribution network to be built. We have implemented *SpreadIt* which is available for download at <http://www-db.stanford.edu/peers>.

The organization of the rest of the paper is as follows. In Section II, we give a detailed model, and definitions for our peer-to-peer network. In Section III, we present our stream distribution architecture, and indicate the issues to be resolved. In Section IV, we describe the primitives and mechanisms used to enable the architecture. In Section V, we discuss the various policies evaluated for maintaining the topology of the multicast tree. In Section VI, we evaluate the various policies on the proposed architecture with respect to the QoS enabled. In Section VII, we discuss related work. In particular, we position our work in the context of application level multicast solutions. We conclude with a summary in Section VIII. In an extended Technical Report [6], we show how to use *SpreadIt* with existing implementations of the end applications.

## II. MODEL

A stream is a time ordered sequence of packets. A stream is logically composed of two channels: *data* and *control*. The stream has two end-points: a server which feeds the data to a client, and a client which receives the data from a server. The data channel comprises of the actual payload packets. The control channel is used by a client and its server to coordinate. Examples of coordination are connection setup, teardown, play, stop, etc. The stream is served over the Internet, using standard streaming protocols. We note that usually data is sent over RTP/UDP [7], while control messages are sent over RTSP/TCP [8].

Let  $s$  denote the source of a live stream. We assume that the source,  $s$ , remains up for the duration of the stream. We also assume that each live stream has a unique URI (Uniform Resource Identifier), known to all potential clients. The URI has information about  $s$  embedded in it.

A client node wishing to receive the stream sends a *subscribe* request to the source. The client can be directed by the source to a server which can support the client. When the client wants to stop receiving the stream, it sends an *unsubscribe* request to its server. At any instant, a set of nodes (clients),  $\mathfrak{N}$ , are receiving the stream. The set  $\mathfrak{N}$  changes over time, as nodes subscribe or unsubscribe.

An instance of the data and control channels, taken together, between a server and a client constitutes a *data-transfer session*. Such a session is distinguished from the *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.

A *live* stream has the important property of being *history-agnostic*: the client is only interested in the stream from the instant of its subscription onwards.

Each node,  $n$ , has a unique identifier (IP address) which remains fixed and unique over its application session. However, each node is autonomous, and can subscribe or unsubscribe from a stream at will, even without notification. Each node is connected to the Internet with a certain bandwidth capability. A node which is unable to serve an incoming client  $c$  with an acceptable QoS is said to be *saturated* with respect to  $c$ , otherwise it is *unsaturated*. A node which expects the stream, but is not receiving it is said to be in a *transient* state. The time taken by the node to recover from a transient state is called *transience* time.

The goal is to design an efficient architecture for feeding a set of nodes,  $\aleph$ , with a particular live stream sourced from  $s$  while providing a high quality of service. Typically, QoS is characterized by packet loss, packet delay, time to first packet (time elapsed between a subscribe request send and the start of stream), and jitter. Jitter is effectively eliminated by a huge client side buffer [9]. Hence, we shall not worry about jitter in the rest of the paper.

### III. THE SPREADIT ARCHITECTURE

The architecture we propose performs multicast at the application-layer. A multicast tree (see Figure 2) rooted at  $s$ , is built over the set of nodes,  $\aleph$ . Nodes are organized into different levels. For each node,  $n$ , at level  $l + 1, l = 0, 1, 2, \dots$ , there is a node, called its *parent*,  $p$ , at level  $l$ ;  $n$  is called a *child* of  $p$ . All nodes in the sub-tree rooted at  $p$  are called its *descendants*.

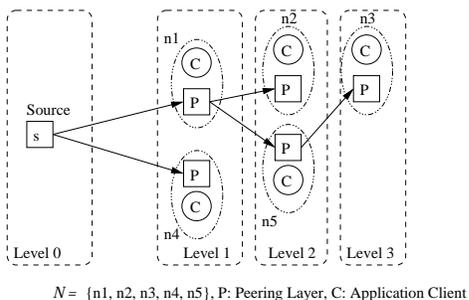


Fig. 2. An application level multicast tree built on the peers

A data-transfer session is established between a node and each of its children. Each node,  $n$ , at level  $l$  forwards the stream to all its children (if any) at level  $l + 1$ .

An application level multicast helps to balance out the

server bandwidth load. To illustrate, assume each node can support  $d$  clients, and that the nodes have arranged themselves to form a  $l$  level complete tree. Thus,  $|\aleph| = O(d^l)$ . If the server were feeding all clients by itself, it would need to send  $|\aleph|$  packets per time unit. Under the SpreadIt architecture, it has to feed only  $d$  nodes. As a result, congestion is removed from the network near the server, and the bandwidth requirements are spread over the network of clients.

However, for the above benefits to be realized, the following issues need to be resolved. We shall discuss the solutions in Sections IV and V.

1. *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. There needs to be a mechanism to enable the new nodes joining the multicast to identify an existing node in the tree, which can act as its parent.

2. *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 disconnections 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.

3. *Managing Load*: The multicast tree is built over nodes whose primary intent is to act as individual clients. The resources of the nodes should be judiciously used while constructing the multicast tree, so as not to overload any node. Most of the nodes today have significant processing power and memory resources. The constraining resource at a node is predominantly bandwidth capacity. Each node is aware of its own capacity, and can, for a given stream bit-rate, fix for itself a maximum limit on the number of children it will support.

Note that, using an application-layer multicast solution results in increased packet loss and delay at a node. In particular, the data channel of the stream typically uses UDP, which does not provide guarantees on reliable delivery of packets. A level  $i$  node receives data from the source over a path of  $i$  hops. So the packet losses and delay accumulate across each hop. Our experiments (Section VI) suggest that the effects of packet loss and delay are limited, and a good QoS can be enabled using SpreadIt.

### IV. PEERING LAYER: PRIMITIVES AND MECHANISMS

We introduce a basic peer-to-peer infrastructure layer between the application and the transport layers for streaming. All communication between the application and the transport layers passes through the peering layer, as shown in Figure 3.

Data transfer sessions are established between the peer-

ing layers at the two nodes. The peering layer allows

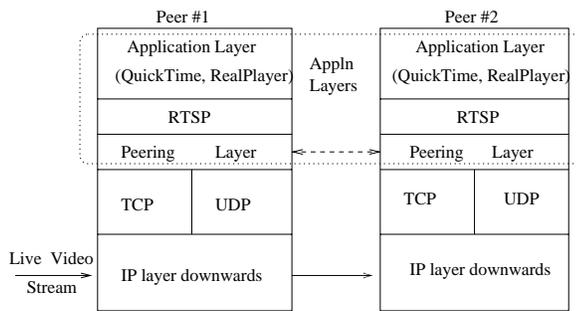


Fig. 3. A layered architecture of a peer

the application layer above to specify the stream to obtain, through a “get-stream” interface. Given a stream URI, the peering layer locates a server which can provide the stream, negotiates with the server for opening a data-transfer session, and establishes a transport session to the server. It interacts with the peering layer at the server to detect, or report a data-transfer session termination. In the event of a data-transfer session termination, it cleans up the transport layer session, 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 survives these changes, and is *unaware* and, indeed, *unconcerned*, of shifts in the underlying multicast topology, as it should be. Furthermore, the code of existing applications does not need to be changed to enable the use of our peering layer (see [6]). We also discuss the design philosophy for the peering layer, and discuss its end-point addressing and guarantees in the extended version of the paper.

### A. The Redirect Primitive

The peering layer supports a redirect primitive used to effect changes in the topology. The primitive is simple and light-weight. It hides a lot of implementation details, and is a good building block for tree-maintaining algorithms. The peering layer uses redirects as hints, to enable discovery of an unsaturated node in the network.

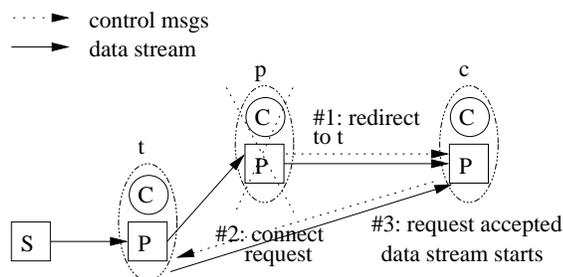


Fig. 4. An example of the redirect primitive in use

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 URI. Note that such redirect messages are produced and used at the peering layer, and the application above is unaware of such messages. Thus, it is the redirect primitive which allows the application session to persist despite changes in the data-transfer sessions.

### B. Discovering an Unsaturated Server

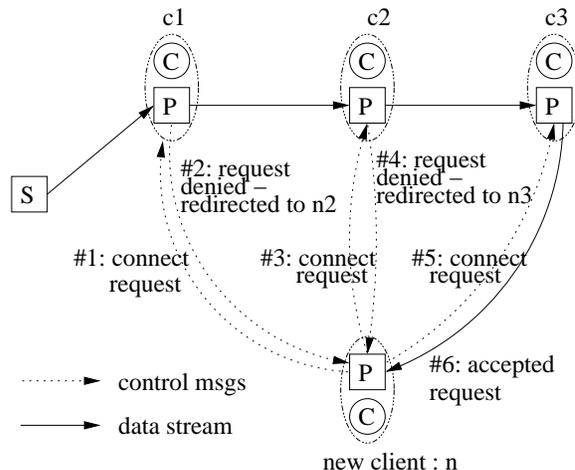


Fig. 5. Adding a new node to a multicast tree

A new node,  $n$ , seeking the live stream needs to be able to discover an unsaturated node in the peer-to-peer network, which can act as its parent. Such a resource discovery mechanism is built using the redirect primitive.

The process of addition of a new node is shown in Figure 5. To start the process,  $n$  contacts the source,  $s$ , of the stream at the universally known URI. When  $s$  receives a data-transfer session setup request, it checks to see if it is itself unsaturated. If  $s$  is unsaturated, it accepts  $n$  as its child and starts feeding it the stream by establishing the data-transfer session. If  $s$  is saturated, it responds to  $n$  with a redirect message with one of its immediate children  $c$  as the target peer. Upon receiving the redirect,  $n$  attempts to setup a data-transfer session with  $c$ . The process continues iteratively, and  $n$  gets bumped down the multicast tree until it is 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 Deletion of Nodes

When one of the nodes,  $d$ , in a multicast tree unsubscribes from the stream, the descendants of  $d$  become tran-

sient. To recover from such an event, the descendants have to be grafted back into the source-rooted tree after  $d$  has been removed.

The simplest case is when  $d$  was acting just as a client, and was not serving any other node in the network. Then,  $d$  sends an unsubscribe request to its parent. The parent frees up resources dedicated to  $d$  at its side. No further changes in the tree are needed.

If  $d$  was the direct parent of a set of nodes  $C$ , it also sends a redirect message to all the nodes in  $C$  specifying a target  $t$ . The value of  $t$  might be the parent of  $d$ , or the source  $s$ . 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 able to inform their parent, or send redirect messages to their children. The 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. If a peer detects that the parent has skipped a specified number of heart-beats, it deems the parent as dead. It then recovers from its transience state by sending a redirect message to itself specifying the source,  $s$ , as the target. The process of discovering an unsaturated server in the network, as mentioned in Section IV-B, is then started.

### V. POLICIES FOR TOPOLOGY MAINTENANCE

Observe that the tree topology is defined by specifying the policy for choosing a target peer in a redirect message. Such a policy will have implications on keeping the tree balanced. The time taken to discover an unsaturated server depends on the policy, which in turn affects the time to first packet of a new node, and the transience time of nodes. The policy also determines the information about the network that needs to be maintained at each peer.

A topology maintenance policy must specify the target peer to contact under addition, and deletion (or failure) of nodes. Such a policy can then be implemented using the mechanisms we discussed in Section IV. The tree can be optimized with respect to a variety of cost functions (available resource, bandwidth, delay, packet loss). For a cost function, a host of policies are feasible, ranging from light-weight to a completely centralized heavy-state policy.

In this section, we shall discuss some of the policies, based on simple cost functions, that are appealing because of their light-weight and distributed nature. In particular, we assume that each node knows only the source  $s$ , and its

local topology: its parent, and the set of children it supports.

#### A. Addition of a peer (or Add Policies)

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

1. *Random* :  $Y$  chooses one of its children at random as the target  $t$ , and redirects  $X$  to  $t$ . Such a policy requires minimal state at  $Y$ . On an average, the tree is expected to be balanced.
2. *Round-Robin (RR)* :  $Y$  maintains a list of its children.  $Y$  forwards  $X$  to the child,  $t$ , at the head of the list. The child,  $t$ , is then moved from the head 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)* :  $X$  sends traceroute information along-with its request.  $Y$  maintains the network locations of its children.  $Y$  redirects  $X$  to a child that is closest to  $X$  [10]. Such a policy helps in creating trees taking network proximity into account. Packet losses and delays are expected to be minimized, thereby improving the QoS metric.
4. *Knock-Downs* :  $X$  sends its traceroute information along-with its request.  $Y$  maintains the network locations of its children. If  $X$  is closer to  $Y$  than any of  $Y$ 's children,  $Y$  redirects its child which is the farthest, and accepts  $X$  as a child. Otherwise, it redirects  $X$  to a child that is closest to  $X$ . Such a policy helps in creating trees taking network proximity into account. Packet losses and delays are minimized.
5. *Smart-Bandwidth* :  $X$  sends its bandwidth capacity along-with its request to  $Y$ . If the bandwidth capacity of  $Y$  is greater than or equal to that of  $X$ , act according to one of the above policies. Otherwise,  $Y$  closes its data-transfer session with its parent,  $t$ . It then redirects  $X$  to  $t$ , and redirects itself and all its immediate children to  $X$ .

#### B. Deletion of a peer (or Delete Policies)

When a node  $X$  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  $X$  is unsubscribed, all of its descendants become transient. Each transient descendant can try to recover by contacting  $t$ . Alternately, only the children  $C$  of  $X$  attempt

to recover by contacting  $t$ . The rest of the descendants of  $X$ , are also the descendants of  $C$ . Thus, these nodes automatically recover when the feed to  $C$  is restored. Hence, we have the following policies.

1. *Grandfather-All (GFA)*:  $X$  chooses its parent (which is the grandfather of  $C$ ) as  $t$ .  $X$  sends a redirect message to all its children to contact  $t$ . Each child of  $X$  in turn sends a redirect message to its children with  $t$  as target, and so on. Thus, all the descendants of  $X$  try to recover by contacting  $t$ . The advantage of such a policy is that the height of the affected portion of the tree can be reduced.
2. *Root-All (RTA)*:  $X$  chooses the source as  $t$ . As in GFA, all the descendants of  $X$  attempt to recover by contacting  $t$ . The advantage is that the affected nodes move closer to the source, reducing the height of the tree.
3. *Grandfather (GF)*:  $X$  chooses its parent (which is the grandfather of  $C$ ) 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 effects of failures are localized. Other nodes in the tree are not affected. Moreover, it avoids the explosion of requests to  $t$  as in GFA.
4. *Root (RT)*:  $X$  chooses the source as  $t$ . As in GF, only nodes in  $C$  attempt to recover by contacting  $t$ . The advantage of such a policy is that the sub-trees rooted in  $C$  could be accommodated near the source, causing fewer packet losses, and delays for the affected nodes. Moreover, it also avoids the explosion of requests to  $t$ .

Note that the policy to recover from failure is similar to delete, 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.

## VI. PERFORMANCE EVALUATION

In this section, we study the performance of SpreadIt. We implemented SpreadIt, and obtained values for basic parameters through empirical studies. The values were then used in simulations to evaluate the various policies for the QoS metrics.

### A. Experimental 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 addition, deletion, and failures. The add policy was set to Random, and the delete policy to Root (RT). Failures of nodes were

detected by the absence of heartbeats. The implementation details are discussed in [6].

We performed experiments 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 with a 100Mbps ethernet. The video clip was 37.12s long, and 834,352B in size. The video resolution was 240x180, and the sound sampling rate was 11 KHz mono. The experiments were designed to measure the following parameters at a peer.

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. Note that packet delay is caused by network latencies as well as processing time in a node’s network stack and peering layer.
2. *Time to discover parent*: defined as the interval between the times a new node sends a subscribe request to the source, and to an unsaturated node. Thus, the time to discover parent measures the time spent in tree-transience management by a node.
3. *RTP/UDP packet drops*.

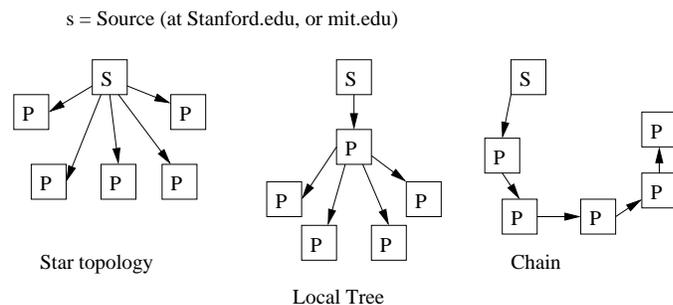


Fig. 6. Topologies used in the experiments

In one set of experiments, the source was placed in the same intranet (in Stanford) as all the clients (peers). In the other set, the source was placed across the internet (at MIT) while all the clients were in the same intranet (in Stanford). Measurements were also made for different peer topologies as shown in Figure 6. A *Star* topology models the conventional architecture, in which all clients connect directly to a single source. We assume that in a *Star* architecture, the source has a sufficiently high bandwidth to support *all* the clients. Intuitively, *Star* models the best solution that does not use the aggregation provided by multicast. The *Local Tree* allowed us to study the effects of network proximity, while the *Chain* showed us the effects of increasing number of hops of a peer from the source.

Data was collected over five runs for each setting, and then averaged. Upto six clients were used in different experiments. Note that experiments 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.

### B. Results from Empirical Tests

1. *Packet Delays* increase linearly with the number of hops between the source and client. The average packet delay observed for a Chain topology over a 1 minute run, when the source and the clients were all within Stanford is shown in Figure 7. Note that the order of magnitude of the time delays, even at five hops, is about  $0.1s$ , while RealPlayer 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% of buffering delays. Thus, the number of clients supported can be in thousands (a tree has average height of 10, each node has 3 children, number of clients can be  $(3^{11} - 1)/2 = 88572$ ) before delays become a dominating factor.

