Multimedia Systems Journal manuscript No.
(will be inserted by the editor)

Scattercast: An Adaptable Broadcast Distribution Framework

Yatin Chawathe

AT&T Labs-Research

75 Willow Road, Menlo Park, CA 94107

Abstract Internet broadcasting—the simultaneous distribution of live content streams to a large audience—has a number of interesting applications ranging from real-time broadcasts of audio/video streams for online concerts or sporting events to efficient and reliable large-scale software distribution. We identify three fundamental requirements for scalable broadcasting services: an efficient infrastructure for large-scale broadcasting, an ability to adapt the infrastructure to suit the requirements of a wide range of applications, and ease of deployment of the infrastructure. Although solutions such as the network-layer IP multicast approach and a slew of overlay distribution networks exist today, none of these technologies satisfactorily address all of the above concerns.

In this paper, we argue that an application-customizable hybrid overlay is well-suited to meet these challenges. To this end, we propose an architecture called *scattercast* that relies on a network of strategically located agents called *ScatterCast proXies* or *SCXs*. These agents collaboratively provide the broadcast service for a session. Clients locate a nearby SCX and tap into the session via that SCX. Scattercast constructs a hybrid overlay network composed of unicast links between SCXs that interconnect locally-scoped multicast regions. Rather than define a single standardized service model for transmitting data on top of the overlay, scattercast builds a customizable transport framework that provides adaptability by leveraging application-defined semantics to drive the distribution of content. We demonstrate the ability of our architecture to provide efficient distribution via a set of simulation experiments. Finally, we present our experience with the adaptability of the framework by describing two applications, a real-time Internet radio and an online slide presentations tool, both of which we have built on top of a prototype implementation of the architecture.

1 Introduction

Recently, we have seen a proliferation of proposals that use overlay networks for efficient streaming of live broadcast[1] content across the Internet. Unlike traditional solutions such as IP multicast [8] that rely on IP

[1] In this paper, we use the term “broadcast” to mean one-to-many or many-to-many transmission, instead of the traditional term “multicast”, which is often construed to mean the network-layer IP multicast protocol [8].

routers, these overlay networks use application-layer components located at IP end-points to distribute broadcast content. Examples of such overlay networks include Endsystem Multicast [19], Inktomi's Broadcast Overlay Architecture [10] and the Cisco Overcast framework [22].

The rationale behind these overlay approaches is that they address the complexity, deployability and scalability concerns associated with IP multicast. Overlay networks construct broadcast distribution trees out of unicast connections from sources toward the receivers of content. In the Endsystem Multicast approach, distribution trees are composed entirely of end-clients, while the Cisco Overcast and the Inktomi Broadcast Overlay architectures rely on special-purpose servers deployed at IP hosts within the network.

Unfortunately, although these overlay solutions alleviate some of the problems associated with IP multicast, they are often limited in their capabilities. For example, Endsystem Multicast constructs overlays entirely out of unicast connections between end-clients, and hence can scale to only small and medium sized groups. Other overlay networks that use a more infrastructure-oriented approach with dedicated broadcast servers still use unicast for the last hop to clients thereby limiting their scalability. Moreover, many of these architectures are designed with specific applications in mind. For example, the Inktomi Broadcast Overlay Architecture is tailored specifically for audio/video applications, while the Cisco Overcast architecture is designed for reliable multicast applications. This is essentially due to the fact that in the broadcast world, the conflicting needs of a widely heterogeneous set of receivers and a heterogeneous range of applications make it impossible to define a single broadcast service that can satisfactorily serve the entire spectrum of broadcast sessions. As a result, a system tailored to support real-time audio/video streaming will have substantially different requirements than those of a reliable software distribution framework.

Thus, for a broadcast distribution framework to be viable, we postulate that it must address three concerns:

- For efficient distribution, rather than completely eliminate IP multicast, we should leverage it in restricted domains where it has been proven to be efficient and usable i.e. rely on IP multicast for small locally-scoped communication where possible and use overlay unicast for wide-area dissemination.
- To support a wide range of applications and clients, the distribution framework should not provide a single standardized broadcast service model. Instead it should be designed to adapt and optimize itself based upon the needs of individual applications and operational environments.
- Finally, to ensure ease of deployability, it should be possible to roll out this broadcasting infrastructure without any major disruptions or alterations to the existing network fabric.

In this paper, we advocate that an adaptable broadcasting framework is best implemented via a twofold solution: (1) a hybrid overlay approach that combines the ease of deployment of Internet-wide overlay networks and the efficiency of locally-scoped IP multicast domains, and (2) intelligent network- and application-aware agents placed at strategic locations within the network to adapt the broadcast distribution to the specific needs of individual applications and end-clients.

1.1 Scattercast: A Customizable Overlay Architecture

Our architecture is grounded in a hybrid communication model that relies on a collection of a collection of strategically located network agents. These agents build a hybrid distribution network that is composed of a web of wide-area unicast connections and locally-scoped multicast groups. Clients attach to the broadcast network through a nearby agent via local multicast groups if possible, or else via direct unicast connections to their local agent. Agents construct distribution trees out of wide-area unicast links to each other. In this manner, whenever possible, we harness the efficiency of IP multicast using scoping rules [20, 27] to restrict the multicast flows to locally-scoped regions where it is robust and easy to manage, and rely on the simplicity of unicast for wide-area communication. We call this communication model *scattercast* and the network agents that are central to this model *ScatterCast proXies (SCXs)*.

It is impossible to support the entire range of broadcast applications through a single standardized service model. Hence, scattercast provides customization hooks that allow developers to embed intelligence within the scattercast infrastructure for optimizing individual applications. Scattercast proxies are composed of customizable service modules which interact with the underlying framework via a narrow well-defined API and contrl the behavior of the broadcast session on a per-application basis. These service modules provide richer functionality to the scattercast framework by implementing services such as congestion control, reliable multicast, or dynamic stream adaptation for adjusting to available bandwidth. Software developers can extend scattercast's functionality by designing such custom service modules.

1.2 Pros and Cons of this Approach

By migrating the broadcast service and its management to the application layer, scattercast keeps the underlying network model simple and straightforward. It isolates the use of IP multicast to well-defined locally-scoped data groups thereby eliminating or mitigating the problems associated with wide-area multicast. Moreover, by explicitly using application-level agents that can be customized via application-specific service modules, scattercast allows for a scenario where SCXs can use application semantics to adapt the progress of the broadcast session by providing assistance for efficient congestion control, reliability, and bandwidth allocation strategies.

However, these advantages come at a price. Since the scattercast network is composed of SCXs located at IP end-points rather than within IP routers, the path taken by a scattercast data stream will inevitably incur additional latency. Moreover unlike multicast routers, which execute the routing and forwarding algorithms in fast specialized hardware, SCXs are software entities that may not be able to sustain sufficient data throughput. However, by using techniques such as software clustering [5,12] to achieve scalable throughput, and by using intelligent overlay construction algorithms, we can minimize the effects of these problems.

1.3 Validation: Two Real Applications

The flexibility of our architecture is reflected in the range of applications that can be built on top of scattercast. In this paper, we describe two distinct classes of applications: a real-time audio tool, and a reliable multicast whiteboard.

- *MusiCast*, an online MP3 broadcasting service that relies on scattercast to efficiently disseminate live audio content in the form of MP3 (MPEG 1 or 2 Layer III Audio) streams. This application requires a simple real-time frame-based transport protocol without any reliability guarantees across frames.
- *An electronic whiteboard* for online presentations, whose content is composed of a number of individual presentation slides, each of which contains a collection of data objects. The whiteboard adapts the scattercast framework to provide selective reliability for these individual objects based on receiver interest. The design of this application includes a specialized SCX service module that can support a heterogeneous set of whiteboard clients ranging from high-end desktops to small hand-held PDAs.

The rest of the paper is organized as follows. Section 2 discusses scattercast's evolution from prior related work. Section 3 outlines the overlay architecture used by scattercast. We describe our protocol for constructing efficient overlay topologies in Section 4. In Section 5, we describe scattercast's service customization components. Section 6 presents a detailed evaluation of the scattercast framework. Finally, we summarize our observations and conclusions in Section 7.

2 Related Work

This work represents a synthesis of two independent areas of research: (1) the notion of using application-layer overlay networks rather than relying simply on network-layer routers to distribute broadcast content, and (2) the use of intelligent computation within the network infrastructure to assist in distributed applications. The area of overlay content distribution networks has exploded in the recent past, particularly with the advent of the Akamai network for web-oriented content [1]. Companies such as Inktomi (formerly FastForward Networks) [10] and Cisco [22] have extended this concept to the domain of wide-area broadcast distribution. Scattercast builds upon this notion of using application-level overlay networks, and extends it to provide flexible and customizable support for a range of different applications.

Endsystem Multicast [19] was one of the first research proposals that described an actual protocol for building a multi-point distribution overlay network out of IP end-points rather than within IP routers. The ideas espoused in the Endsystem Multicast work have influenced the design of our overlay construction algorithms. However, scattercast goes beyond Endsystem Multicast and defines an entire infrastructure framework for incorporating overlay-network-based distribution into the Internet. Moreover, it defines a programming model for customizing such an infrastructure to adapt to different application requirements. Like Endsystem Multicast, other proposals such as Yoid [14] and Banana Tree Protocol [17] define protocols for constructing broadcast overlay networks. This is still a nascent area of research, and new developments in this area can be adapted into the scattercast architecture in the future.

To develop its customizable framework, scattercast combines ideas from prior works in infrastructure services for distributed applications. In the context of the web, intelligent network services such as proxies have been used to adapt web content for acceleration services [13], web anonymization [2], and to shield web clients from the effects of poor (especially wireless) networks [23]. Similar to these services, scattercast embeds application-aware entities within the network infrastructure. In the context of broadcasting, scattercast builds upon our earlier efforts with the RMX [6] architecture. In the RMX architecture, we proposed providing a reliable multicast service by using intelligent proxies distributed across the network. Scattercast generalizes the RMX approach beyond reliable multicast applications into a general purpose broadcasting framework.

3 The Application-layer Architecture for Scattercast

In this section, we elaborate on the system architecture and network service model that scattercast uses. We now look at the various components of the architecture and describe how they are put together to provide a broadcasting service.

3.1 A Cluster-based Implementation

Figure 1 illustrates the various components of the scattercast architecture. At its core, scattercast is composed of a collection of agents that form an overlay broadcast network out of unicast connections across the existing

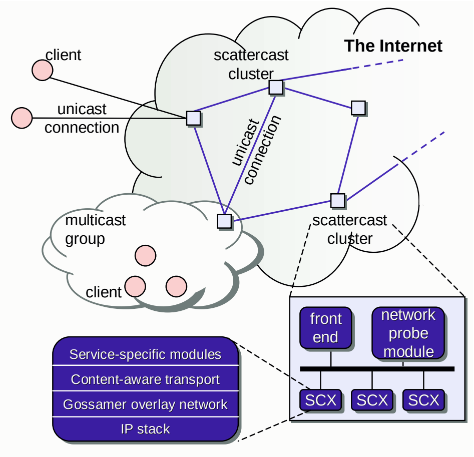

Fig. 1 The scattercast architecture.

IP infrastructure. As shown in the figure, each agent is typically implemented as a cluster of machines. Clusters provide a natural parallelism and redundancy [5,12] that scattercast exploits for scalability and availability. By replicating the scattercast functionality across a cluster of machines, we can provide incremental scaling for the broadcast service by simply adding more machines when an existing cluster can no longer sustain its offered load.

A scattercast cluster is composed of three key components: a **frontend** that interacts with scattercast clients and redirects them to appropriate nodes within the cluster, a **network probe module** that makes periodic measurements of the state of the network to assist in constructing efficient overlays, and a collection of **SCXs** that implement the actual details of the overlay broadcast network. Each SCX has a layered organization comprised of three components that reside on top of a traditional IP network stack: (1) a protocol layer called *Gossamer* that builds and maintains the broadcast overlay topology, (2) a content-aware transport layer that provides the underlying programming framework and API for customizing distribution policies, and (3) a service customization layer composed of a collection of service-specific modules that implement the actual details required to provide custom services such as reliability and congestion management.

Logically, a scattercast cluster creates a separate SCX for each session that it is serving. In practice, a cluster may implement this as a separate SCX process per session, or for better performance, as a single multi-threaded process on each cluster node or a complex non-blocking event-driven system. SCXs construct a separate overlay network for each scattercast session. This ensures that only those clusters that are actually serving clients for a given session are required to commit resources for that session. Within a cluster, however, SCXs across different sessions share information about the state of the wide-area network and their connectivity to other scattercast clusters by relying on a common network probe module.

3.2 Deployability Issues

Scattercast makes an explicit choice to keep its architecture independent of the underlying IP network. Deploying and upgrading such an application-layer system is easier than upgrading network routers. This follows from the use of commodity building blocks such as PCs rather than high-end routers to provide the broadcast service. The obvious advantage is the ability to track the best possible cost/performance ratio, and to incrementally install new software service modules to support new application types as they become available. This is in contrast to upgrading typical router configurations which require purchasing new specialized router hardware (with substantially longer time-to-market cycles) to replace old hardware.

Scattercast's per-session network model whereby it creates a separate overlay for each broadcast session allows us to incrementally deploy new transport services for supporting new types of applications. When a new software module is built, it will initially be deployed on only some of the scattercast clusters. Those clusters and the clients that they serve will immediately benefit from the use of the new module by creating their own separate overlays for the new application. As the popularity of the new application rises, more and more clusters will deploy the transport modules needed to support it and effectively improve the performance of the scattercast overlay in an incremental fashion.

3.3 Scattercast Sessions

Scattercast sessions are identified by explicit URL-like unique names. Session names are of the form `scast://creator-identity/session-name`. The creator identity is used to avoid collisions in the session namespace. For example, a broadcast of the CNN Headline News may be identified as `scast://www.cnn.com/Headline_News`. Within a single session, scattercast permits the creation of multiple independent *data channels*. Each channel represents a separate data flow within the application with its own separate transport requirements. For example, a broadcast of an online seminar may be composed of three channels, one each for the audio and video streams, and a third channel for the presentation slides. The audio and video channels require a real-time transport protocol such as RTP [33], while the slides channel needs a reliable delivery protocol.

Scattercast sessions are announced on the web using the Extended Markup Language (XML) [3]. Figure 2 shows the format of a scattercast announcement. The document enumerates the list of channels associated with the session and the corresponding transport protocols that the channels are required to use. Unicast protocol descriptors may be UDP, TCP, RTP or some other unicast transport protocol, while multicast protocol descriptors may be UDP, RTP, or a reliable multicast protocol such as SRM [11]. In addition, the announcement document

```
<SCATTERCAST name="scattercast://creator-identity/session-name">
   <DESCRIPTION> Optional textual description of the session </DESCRIPTION>

   <CHANNEL id="numeric-identifier">
      <DESCRIPTION> Optional textual description for this channel </DESCRIPTION>
      <TRANSPORT unicast="unicast-protocol-name" multicast="multicast-protocol-name"/>
   </CHANNEL>

   <CHANNEL id="numeric-identifier"> ... </CHANNEL>

   <RENDEZVOUS> rendezvous-point-location ... </RENDEZVOUS>
</SCATTERCAST>
```

Fig. 2 Format of a scattercast announcement.

lists one or more *rendezvous points* that SCXs within a session use to find each other. We discuss the details of the rendezvous mechanisms and their use for constructing the application-level overlay structure in Section 4.

A single scattercast session can contain one or more sources of data. Sources are required to explicitly join the session and announce their intent to send data. This allows the underlying scattercast protocols to use this information to build efficient source-rooted data distribution trees. When a source injects data into the scattercast session, the SCX that the source attaches to (i.e. the source SCX) inserts scattercast headers into the source's data packets. Each header contains a numeric session identifier, which is an MD5 hash [32] of the URL-based session name. The header also contains a numeric channel identifier (as published in the session announcement), the source address, and the address of the source SCX. Scattercast uses this information to efficiently route the data packets through its overlay network to all the receivers.

3.4 Client Attachment

When a client wishes to join a scattercast session, it must first locate an appropriate scattercast cluster. For optimal performance, the client should be directed to a lightly loaded cluster in close proximity to the client. Moreover, the cluster must support all the service-specific modules required by that session. To automate this process, clients should rely on a sophisticated content-aware redirection scheme that takes into account not only cluster distance and load, but also the software capabilities of the scattercast clusters. However, this problem is essentially orthogonal to the design of the scattercast infrastructure itself. Hence, in this paper, our prototype implementation relies on a simple static configuration scheme to locate scattercast clusters. Such static configuration may be disseminated to the client via DHCP or using a well-known DNS name for the cluster frontend, e.g. `scattercast.mydomain.com`. That said, an intelligent redirection system that can automatically locate an appropriate scattercast cluster is a subject of future work, and we expect that a commercially deployed scattercast infrastructure would rely on such a redirection system.

Upon locating an appropriate cluster, the client communicates directly with the cluster frontend using either HTTP or RTSP [34]. In its request to the frontend, the client includes the name of the session it wishes to join and either the contents of the corresponding session announcement document or simply a pointer to its location on the web. Upon receiving a request, the frontend locates the SCX for that session, starting a new one if necessary, and redirects the client to that SCX. The frontend responds to the client with a list of the channels that form the session, along with the addresses, port numbers, and protocol descriptions used by the session's SCX for each of the data channels. The client uses this information to retrieve the session data directly from its SCX. As shown in Figure 1, SCXs can transmit the session data to their clients using either locally-scoped multicast groups or direct unicast. Hence, clients should first attempt to communicate with the SCX using multicast, and revert to unicast communication if that fails. This allows us to leverage the efficiency of IP multicast in the local domain when it is available.

4 Gossamer: Overlay Network Construction

A key challenge for the scattercast architecture is to construct an efficient overlay network. For this, SCXs rely on a protocol that we call *Gossamer*. Our approach is based on the End System Multicast protocol [19]. Like End System Multicast, Gossamer is a randomized three-step protocol: (1) it discovers the set of SCXs that are participating in a given session, (2) builds a strongly-connected graph overlay structure or *mesh* across those SCXs, and (3) runs a routing protocol on top of the mesh to build source-rooted distribution trees. However, unlike End System Multicast which is designed for small and sparse sessions and hence does not worry about the overhead of its control traffic, Gossamer's design explicitly attempts to limit the control traffic overhead especially as the number of SCXs participating in the overlay grows. We now look at each of the three stages—membership discovery, mesh construction, and tree building—in more detail.

4.1 Membership Discovery

Each SCX uses a gossip-style [9, 30, 36] resource discovery protocol that is based on a variant of the Name Dropper scheme[16] to find the existence of other SCXs in its session. When an SCX (say X_i) joins a session, it bootstraps itself via a well-known list of rendezvous SCXs. Let $\Gamma(X_i)$ represent the set of other SCXs that X_i knows of at any time. Periodically, X_i performs a discovery round. During this round, it picks a random SCX $X_j \in \Gamma(X_i)$ and sends a DISCOVERY message to it. This message includes a *bounded random list*

$\gamma(X_i) \subseteq \Gamma(X_i)$. When X_j receives the message, it merges this list into its own set $\Gamma(X_j)$ of known SCXs. In addition, it returns a DISCOVERY_RESPONSE message that includes its own list $\gamma(X_j) \subseteq \Gamma(X_j)$. X_i in turn merges this list into its own set $\Gamma(X_i)$, and thus gradually learns of all or most of the other SCXs in the session. By bounding the size of the DISCOVERY messages, Gossamer prevents the discovery traffic from overwhelming the network even as the session membership grows.

To bootstrap the discovery process, an SCX needs to first locate some initial group of other SCXs. We rely on explicit rendezvous points as the bootstrapping mechanism. Each scattercast session has associated with it one or more rendezvous SCXs that are advertised in the session announcement (see Figure 2). The redundancy provided by the existence of multiple rendezvous points within the same session ensures that new SCXs can join the mesh even if one of the rendezvous points fails. We note, however, that even if all rendezvous points in the session fail, existing mesh members can continue to operate.

4.2 Mesh Construction

When a new SCX joins the session, it gradually discovers other SCXs and selects some of these SCXs to be its neighbors in the mesh. Depending upon its access bandwidth, each SCX has a limited number of neighbors that it can connect to in the mesh. To avoid requiring explicit coordination across SCXs, we split this neighbor degree constraint into two: a maximum number of edges (k_1) that the SCX is allowed to insert from it to other nodes, and a maximum number of edges (k_2) that it is willing to accept from other nodes. As long as we ensure that $k_2 > k_1$, any new SCX joining the mesh will eventually find some k_1 nodes that have room to accept connections from it. We use the notation $<k_1,k_2>$ to represent these degree constraints, effectively resulting in a total degree constraint of $k_1 + k_2$. Under these constraints, if an SCX X_i tries to insert a connection to another SCX X_j, we require that X_j accept the connection as long as it has not reached its limit k_2 of the maximum number of edges that it is willing to accept.

When an SCX first joins the overlay topology, its neighbors will most likely be sub-optimal, since initially the SCX does not have complete membership information and may not know of nodes that would provide better distribution paths. Hence, Gossamer periodically adapts the mesh structure to incrementally improve its topology. This is especially important since as new SCXs join the session and existing ones leave over time, the mesh will have to be reconstructed to adapt to the new reality.

4.2.1 Incrementally Optimizing the Overlay Gossamer's optimization algorithms rely on routing information generated by an application-layer routing protocol that is run on top of the mesh. This protocol, which will be

```
compute_cost_function($X$) {
    Let C.F.[$X$] = 0.0
    Foreach source $S$ in $X_i$'s routing table {
        Let C.F.[$X$] $+=$ normalized cost$^*$ of routing to $S$ via $X$
    }

    If ($X_i$'s routing table is empty) then {
        Let C.F.[$X$] $=$ normalized cost$^*$ of the edge between $X_i$ and $X$
    }
    Return C.F.[$X$]
}
```

***Note:** Normalized routing cost is defined as the cost of the route to S via X divided by the maximum of the corresponding such costs for all $X' \in \Lambda$ (see Figure 4 for definition of Λ). Similarly the normalized edge cost is defined as the ratio of the cost of the edge between X_i and X to the maximum of the corresponding edge costs for all $X' \in \Lambda$. We note that the normalized cost is always a value between 0.0 and 1.0.

Fig. 3 Algorithm used by node X_i to compute the cost function for node X.

```
optimize($X_j$) {
    Let $\Lambda =$ (set of neighbors of $X_i$) $\cup$ $X_j$
    For each $X \in \Lambda$ { Let C.F.[$X$] $=$ compute_cost_function($X$) }
    Let $Y (\in \Lambda) =$ node with maximum C.F.
    Let $H =$ hysteresis value
    If ($Y == X_j$) then reject $X_j$
    Else if (C.F.[$Y$] $-$ C.F.[$X_j$] $> H$) then { accept $X_j$ ;   reject $Y$ }
    Else reject $X_j$
}
```

Fig. 4 Algorithm used by node X_i to determine whether to accept node X_j as a neighbor.

described in Section 4.3, executes a limited form of distance vector routing [18] resulting in a small number of entries in the routing table: one for each source SCX. Gossamer uses the routing table to determine the utility of edges in the mesh structure. It attempts to add edges that result in better routes towards sources and remove edges that are not as useful.

SCXs periodically probe other mesh nodes to evaluate the usefulness of adding new edges. A node X_i probes another node X_j using a REQUEST_STATUS message. The STATUS response from X_j contains a copy of its current routing table and a CAN_ACCEPT flag that indicates whether X_j has reached its limit k_2 of connections it is willing to accept from other nodes.

If X_i has not yet filled its limit k_1 of edges it is allowed to add, then it will automatically accept X_j as a neighbor. But, if X_i already has k_1 neighbors, then to accept X_j, it will have to remove one of its existing neighbors. To do this, X_i runs an optimization algorithm that evaluates the "cost" of each of its existing neighbors and the new node X_j. This per-node cost is a function of the routing delays that would result if X_i were to use that node as its next hop to each of the individual sources. The cost function is computed using the algorithm shown in Figure 3, and is used as input to the optimization algorithm which itself is described in Figure 4. X_i will accept X_j as a neighbor only if X_j's cost is less than that of one of its existing neighbors by at least H, where H is a hysteresis value that allows us to trade off the stability of the mesh versus the level of optimization. A higher value of H will result in fewer changes to the mesh structure, but may result in a less efficient mesh. Our preliminary experiments showed that a hysteresis value of 0.15 times the number of known source SCXs provides reasonable stability without greatly affecting the efficiency of the resulting mesh.

4.2.2 Mesh Updates: SCX Leaves and Failures When an SCX leaves the session, it floods a time-stamped notification to the rest of the mesh. This notifies the SCX's neighbors of its impending departure and allows them to reconfigure their edges to other existing SCXs. The departing SCX sends the notification to its immediate mesh neighbors which in turn propagate it to the rest of the session. Since the mesh is not loop-free, SCXs use the time-stamp in the notification to detect duplicate copies of the notification and stop forwarding the copies. In the event that an SCX fails without any warning, its neighbors detect the failure and notify the rest of the session. Upon receiving an SCX leave/failure notification, other SCXs mark that SCX as dead in their membership list, and trigger updates in the routing protocol.

4.2.3 Mesh Partition It is possible that the death of an SCX causes the mesh to be partitioned. Although such an occurrence is rare, it must be dealt with and the mesh repaired. We rely on a periodic HEARTBEAT that is generated by one of the rendezvous points and propagated over the mesh. The rendezvous points run a simple distributed election algorithm and pick one of themselves as the heartbeat generator. As long as every SCX in the session continues to receive this heartbeat, the entire mesh is connected. When an SCX stops receiving heartbeats, it attempts to heal the partition by re-contacting the heartbeat generator. To avoid an implosion of all partitioned SCXs contacting the heartbeat generator at the same time, we use a randomized damping interval before the SCX attempts to heal the partition. In the event that the heartbeat generator itself dies, the remaining rendezvous points elect a new heartbeat generator and the healing process continues.

4.3 Tree Building: Application Layer Routing

The mesh construction stage builds the basic overlay structure. To create data distribution trees on top of this overlay, Gossamer relies on a variant of distance vector routing [18] that is run over this dynamically created mesh topology. The routing protocol uses unicast latencies between SCXs as its distance metric. This ensures that paths in the overlay topology reflect the underlying IP topology by favoring nodes that are closer in the IP topology over more distant nodes.

Standard distance vector routing algorithms maintain routing tables that contain one entry for every node in the network. In Gossamer, to reduce network overhead, only data sources actively advertise route information. A source notifies its SCX about its intent to send data, and the SCX in turn advertises a zero-length route for itself to the rest of the session. All other SCXs passively collect this route information. Thus each SCX builds a routing table that contains one entry per source SCX. The cost associated with a routing path is defined as the sum of the unicast distances between neighboring nodes along the path. To detect loops within routes and to avoid the counting-to-infinity problem [7], each routing table entry contains not only the routing cost to the source SCX, but also the complete path associated with that cost. This approach is similar to that used by the Border Gateway Protocol (BGP) [25] for preventing routing loops across IP domains.

4.3.1 Route Measurements: Network Probe Module To measure unicast latencies between nodes in the overlay topology, scattercast uses shared learning within individual scattercast clusters. Each cluster has a single Network Probe Module (see Figure 1) that all SCXs within the cluster share. Since machines within a cluster are connected via a low latency high speed network, we assume that a single probe module per cluster is sufficient to measure unicast distances to and from the entire cluster. The probe module performs periodic *ping* experiments with the corresponding modules on other clusters, and maintains a soft state cache of the results of previous experiments. Each ping experiment consists of a small sequence of time-stamped packets that the probe module sends to its counterpart in a different cluster. When the counterpart receives the ping packets, it simply reflects them back to the sender. The sender uses the average time difference between sending the packets and receiving the responses to compute the round-trip times and thus the approximate one-way latencies.

4.3.2 Data Distribution Based upon the routing tables generated by the routing protocol, Gossamer constructs source-rooted reverse shortest path data distribution trees. This form of distribution trees is similar to those created by the Distance Vector Multicast Routing Protocol (DVMRP) [37] used in IP multicast networks. Data trees are built out of an independent set of transport connections that are separate from the control connections used by the mesh construction and routing protocols. Since a session may consist of many independent data channels, Gossamer constructs a separate logical tree for each channel. The transport protocols used for the channels depend upon the information specified in the session announcement. As we shall see in the following section, the details of how each SCX forwards data on top of the distribution trees for each channel can be customized depending upon the requirements on individual sessions.

Transient changes in the distribution tree due to routing updates or mesh changes may result in temporary disruption of data flow. To minimize data loss during a route change, data continues to be forwarded along the old route for a short while until the downstream SCX establishes the new tree connections and starts receiving data along the new route.

5 Content-aware Transport

While Gossamer provides the underpinnings for building a broadcast overlay network, we have yet to address the problem of how scattercast allows this network to be customized for different applications. Rather than impose a single service model on all applications, scattercast allows different applications to use different forwarding and transport policies. It does so by providing software developers with a general-purpose programming framework to adapt the infrastructure depending upon the content that it is expected to carry. Scattercast encapsulates this customization within well-defined software modules that can be installed into individual SCXs. As shown in Figure 1, these *service-specific modules* interact with the underlying scattercast infrastructure via a narrow API that is defined as part of a content-aware transport framework. Developers can use this API to build service modules for different kinds of transport services. For example, an RTP service module can define an RTP-based transport that applications such as audio/video broadcasting can use to forward data within the scattercast network. On the other hand, a software distribution application may use an HTTP-based service module to reliably distribute software packages from a vendor to all of his clients, thereby inheriting the congestion-friendly streaming behavior of HTTP.

Scattercast's content-aware transport framework defines the basic set of mechanisms required to glue individual service modules into the scattercast infrastructure. What service modules are used for a given session depends upon the transport requirements of the individual channels that make up the session. The session announcement document (see Figure 2) defines these requirements and allows the transport framework to pick the appropriate service modules for the session. As we described in Section 3.2, not all scattercast clusters need to support all kinds of service modules. If a cluster does not have an appropriate module for a given transport service, then that cluster cannot participate in sessions that require that service. Yet, clients near the cluster can still join the session by finding a different cluster that does support the transport services required by the session.

The transport framework allows service modules to customize the behavior of the infrastructure in two distinct ways, and we now look at both of them in detail:

– Delegating control of packet forwarding policies to the service modules, thus allowing individual services to determine how data flows through the infrastructure for their applications.
– Allowing service modules to incorporate application-specific transport functions such as reliability, congestion control, and bandwidth management, thus allowing the infrastructure to help in optimizing those transport functions.

5.1 Customizing the Data Forwarding Model

Rather than define a single standardized packet forwarding model for all scattercast sessions, the transport layer allows individual service modules to control the forwarding function by exposing to them a localized view of the underlying overlay topology. At each SCX, the transport framework keeps the service modules informed about the set of *links* associated with each channel. A link is defined as the interface to either a locally scoped multicast group that the SCX uses for communication with local end-clients, or a unicast connection between the SCX and another SCX or an end-client. Whenever a change occurs in the underlying distribution topology, the transport framework performs an upcall to the service module to inform it of the change. In addition, service modules can directly query the transport framework at any time to determine the upstream link and the set of downstream links along the distribution tree for a source.

This exposure of the local topology at each SCX to the higher service layers allows us to define richer forwarding mechanisms beyond the simple best-effort transmission of data from a source to its receivers. For example, the RTP service module forwards RTP data packets that arrive along the upstream link from a source towards all the downstream links. On the other hand, RTCP receiver reports that arrive from receivers or from downstream SCXs are handled differently: the service module aggregates individual receiver reports and periodically forwards the aggregate reports along the upstream link towards the source.

As another example of how a service module can customize the forwarding behavior within the scattercast network, consider a reliable multicast service module that implements a NACK[2]-based end-to-end reliability scheme similar to PGM (Pragmatic General Multicast) [35] or BCFS (Bread-Crumb Forwarding Service) [38]. In this case too, the service module forwards data packets that arrive from the upstream link along all downstream links. When it receives a NACK indicating packet loss, it forwards the NACK upstream towards the source. At the same time, it maintains a soft-state record of the link along which the NACK was originally received. If another NACK for the same data arrives on a different link, the service module suppresses the new

[2] Negative ACKnowledgment

NACK thus avoiding implosion at the source. In addition, it augments its previously saved state for that data to include the new link. When retransmitted data arrives, the service module forwards it only along those links that had sent a NACK for that data. The per-link state is "soft" and is eventually timed out to ensure correct operation in the event of loss of the retransmitted data.

5.2 Incorporating Application-aware Transport Services

In addition to controlling the forwarding policies within an SCX, service modules can incorporate application-aware transport mechanisms to assist in implementing selective reliability, congestion control, and bandwidth management. For example, by using a UDP-based channel for data communication between SCXs, an RTP service module can ensure timely delivery of real-time traffic. On the other hand, a software distribution application may use TCP-based streaming connections to achieve hop-by-hop reliability and on top of that use an end-to-end reliability scheme such as the PGM-based protocol described above to ensure that all the bits of the software package reliably reach their destinations.

Similarly, service modules can control the congestion behavior of the overlay by using a combination of any of the following mechanisms. First, if a service module uses TCP to communicate across SCXs, it can automatically inherit TCP's congestion-friendly behavior. As it forwards packets from a faster connection to a slower one, the service module may simply buffer the incoming data until it can be handled by the slower connection. With finite buffer space, the service module may eventually drop data as its buffers fill up and rely on end-clients to recover the dropped data. Alternatively, it may initiate a pushback by sending an explicit congestion notification to inform its upstream SCX to slow its sending rate. This feedback may eventually percolate all the way back to the source and bring the source transmission down to a rate that is suitable for the SCXs to handle.

Finally, at each SCX, the service modules can control the policies that affect how data packets are scheduled for transmission and how the available transmission bandwidth is distributed across the session. The transport framework exports a hierarchical transmission scheduler that uses the Start-time Fair Queuing (SFQ) [15] scheduling scheme. Service modules can prioritize certain data packets over others by assigning them to different "traffic classes" within the hierarchical scheduler. For example, the RTP module can, if its applications support it, use layered transmission [26] to adjust to varying bandwidths across the session. The source transmits its data stream encoded in the form of multiple layers. At each SCX, the RTP service module forwards as many layers along a downstream link as the available bandwidth along that link permits. Moreover, it prioritizes the base layers over higher layers to ensure that the data from the base layers reaches the receivers before data from the enhancement layers.

To fully exercise the various flavors of customization afforded by the scattercast transport framework, we have built a reliable broadcasting service module that provides applications with a flexible ADU[3]-based framework for selectively controlling the extent of reliability within the application's data stream, the ordering constraints across ADUs, and the transmission of individual ADUs across the session. We used this service module to implement support for an online presentation application that we will describe in Section 6.6. For lack of space, we refer the interested reader to [4] which describes the implementation details of this service module.

6 Evaluation

In this section, we evaluate the ability of our architecture to provide efficient and adaptable broadcast communication. To measure the performance of the scattercast network with respect to the efficiency of the overlay, we rely on a collection of simulation experiments. To demonstrate the adaptability of scattercast to support different kinds of applications, we present our experience building two applications on top of a prototype scattercast implementation.

6.1 Simulation Setup

For most of our experiments, we used a protocol simulator that implements the various components of the Gossamer protocol: membership discovery, the mesh construction and optimization algorithms, and a distance vector routing protocol for building distribution trees. The input to the simulator is an Internet topology generated using the Georgia Tech Topology Generator [39]. We used the Transit-Stub model to construct 50 different topologies, each consisting of 1000 nodes and approximately 4200 edges. The simulator assumes shortest path Internet routing and accordingly computes unicast distances between nodes in the topology. Some of these nodes are selected at random as SCX nodes and the Gossamer protocol is run across these nodes. The simulator does not take into account the effects of any cross traffic and queuing delays on the behavior of the protocol. In each of our experiments, the simulator assumes a well known rendezvous SCX. All remaining SCXs join the session at a random instant within the first five seconds of the experiment. Unless mentioned otherwise, each session consists of a hundred SCXs and one randomly chosen source SCX, and each SCX has a node degree constraint of <3,4>[4]. SCXs execute the Gossamer algorithms every 5 seconds. The routing layer performs routing updates

[3] An *Application Data Unit* or *ADU* is the smallest unit of data that is meaningful to the application, such as a block of a video frame or a scan of a progressive JPEG image.

[4] The degree constraint is composed of two components, k_1 and k_2 as described in Section 4.2.

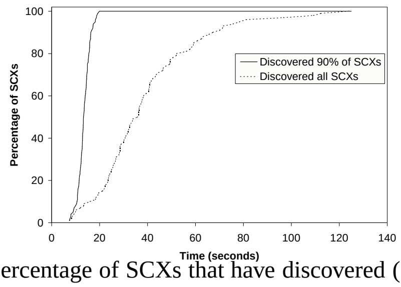

Fig. 5 Percentage of SCXs that have discovered (a) at least 90% of the other SCXs, and (b) all of the other SCXs vs time.

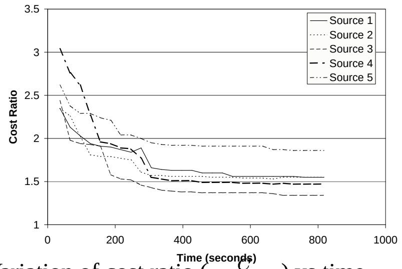

Fig. 6 Variation of cost ratio ($\frac{C}{C_{mcast}}$) vs time.

every 30 seconds. The experiment ends when there are no changes to the mesh structure for at least 100 seconds.

Our experiments evaluate the performance of Gossamer with respect to its ability to discover other SCXs participating in the session and to form stable efficient overlay meshes. We compare the efficiency of the overlay topology to an optimal topology that uses router-based distribution such as IP multicast. The main metric that we use for evaluating the efficiency of the scattercast network is the average cost C of routing from a source over scattercast. C is defined as the average latency between a source SCX and all the other SCXs over a scattercast distribution tree. We compare this cost to the cost C_{mcast}, which is the average latency in the corresponding optimal source-rooted IP multicast tree. The *cost ratio* $\frac{C}{C_{mcast}}$ represents how close the latencies using scattercast distribution are to the native multicast routing latencies. For example, a cost ratio of 1.7 means that the average latency using Gossamer is 1.7 times worse than multicast. Our evaluations will demonstrate that typical latencies with Gossamer are within a factor of 2 of multicast latencies. We note that the simulator only captures the latencies introduced within the network, and does not account for the additional latency at each SCX that is required to process a packet and forward it along to the next SCX. Later, we present experimental results that quantify this processing latency.

6.2 Membership Discovery Performance

Figure 5 depicts the performance of the gossip-based membership discovery algorithm during an experiment consisting of 100 SCXs. In our experiment, we restricted the size of the membership set $\gamma(X)$ exchanged during each DISCOVERY round to 30. We see that all SCXs discover at least 90% of the rest of the nodes within the first 20 seconds of the experiment. Over 85% of the SCXs discover everyone else within the first minute of the experiment. The rest of the nodes discover everyone else within the first 2 minutes. This indicates that most session participants quickly discover each other and can start forming a mesh structure.

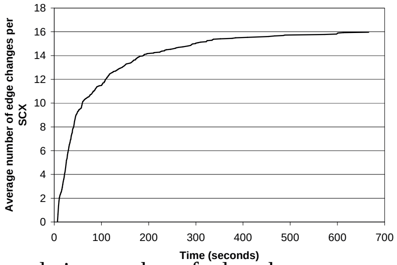

Fig. 7 Cumulative number of edge changes per node in the mesh structure over time. Each edge that is added or removed is counted as two changes, one for each node in the edge.

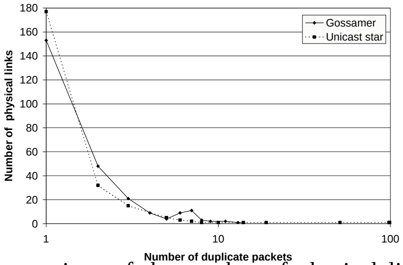

Fig. 8 Comparison of the number of physical links with packet duplication for the Gossamer topology vs direct unicast.

6.3 Mesh Construction

This section presents results to prove the effectiveness of Gossamer as a smart overlay construction protocol. The results highlight various properties of Gossamer such as the variation of cost ratio during the lifetime of an experiment, the stability of the mesh structure, and the level of packet duplication experienced using Gossamer.

6.3.1 Cost Ratio Variation We executed the Gossamer protocol across a number of experiments involving 100 SCXs. We discovered that once a stable overlay was constructed, the resulting cost ratio ($\frac{C}{C_{mcast}}$) for most experiments varied between 1.65 and 1.75. For a small number of experiments, we observed cost ratios as low as 1.35 and as high as 2.35. Figure 6 shows how the cost ratio changes during the lifetime of a single experiment before the overlay reaches a stable structure. This experiment involved 5 source nodes, and the figure plots the variation in cost ratio over time for each of the five corresponding distribution trees. We notice that over time, the cost ratio progressively decreases for each of the five distribution trees. Within about 300 seconds all of the cost ratios mostly stabilize to their final value. This follows from the manner in which SCXs gradually locate other nodes, determine their utility as neighbors, and over time, discover their optimal neighbors thus causing the mesh to stabilize into its final overlay structure.

6.3.2 Mesh Stabilization In Figure 7, we study the stabilization properties of Gossamer with respect to the edge changes that occur in the mesh over the lifetime of the experiment. The figure shows the cumulative distribution of the average number of changes made to the mesh per node. Each edge that is added to or removed from the mesh is counted as two changes, one for each node in the edge. As in the previous experiment, we notice that most of the changes to the mesh topology occur in the initial stages of the experiment. Within about 300 seconds, the mesh stabilizes to almost its final structure. The above two experiments demonstrate that the Gossamer algorithms are effective in rapidly converging to a stable mesh structure.

6.3.3 Packet Duplication Overhead We now demonstrate the effectiveness of the Gossamer distribution trees in limiting the number of duplicate copies of data that any Internet link needs to carry. We ran a single experiment and computed the number of physical links in the underlying Internet topology that carried duplicate data copies. We note that, by definition, source-rooted multicast distribution will result in no packet duplication: all links carry exactly one copy of the data packets. Since Gossamer does not rely on router support to provide optimal tree branching points, it will necessarily result in a few Internet links having to carry multiple copies of the data packets. We compare the packet duplication overhead for Gossamer with that for the limiting case of direct unicast from the source SCX to all destination SCXs.

The plot in Figure 8 depicts the distribution of packet duplication across the physical Internet links, where the x-axis represents the number of data copies that any link may see and the y-axis represents the number of links that carry a given number of copies. We notice that most links carry only one copy of the data: in the Gossamer distribution tree, 153 of the physical links carry only one copy, and in the unicast star topology 177 links carry a single copy. However, the tail of the plot shows that with unicast at least two links carry over 95 copies of each data packet and many other links still have a substantially high level of packet duplication. The worst duplication occurs at the links near the source. In comparison, in the Gossamer mesh structure, no physical link carries more than 14 copies of the data. Thus, we see that Gossamer is quite effective in limiting the packet duplication overhead in comparison to naïve unicast. Even though the packet duplication overhead is high when compared to native multicast, it is evident that a smart distribution protocol such as Gossamer can provide substantial improvement over naïve unicast. We note, however, that the amount of packet duplication in the Gossamer overlay network is a function of the node degree ($<k_1, k_2>$) associated with the various SCXs in the network. The node degree is not a fixed constant for the entire network. SCXs that are connected to the Internet via “fat” pipes can tolerate higher packet duplication and can hence be assigned a higher node degree. This allows Gossamer to improve latencies within the overlay structure by reducing the number of hops that a packet has to encounter before reaching its destination.

6.4 Scaling Behavior

The above experiments depict the behavior of Gossamer for a fixed number of SCXs and a fixed node degree. We now look at how the protocol operates as we vary these parameters. For all of the following experiments, we computed each data point by running 25 independent simulations and calculating the average and the 95% confidence interval.

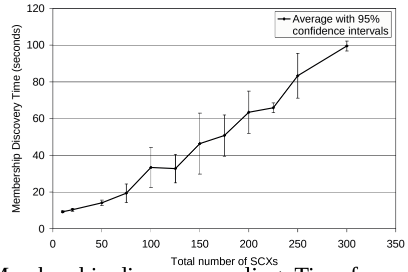

Fig. 9 Membership discovery scaling: Time for everyone to discover at least 90% of the other SCXs vs session size.

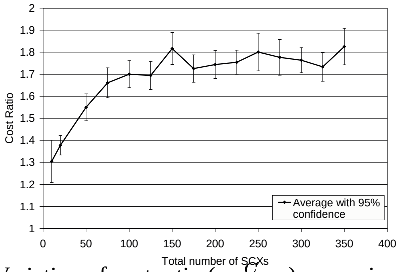

Fig. 10 Variation of cost ratio ($\frac{C}{C_{mcast}}$) vs session size.

6.4.1 Membership Discovery Scaling Figure 9 shows the scaling behavior of the membership discovery algorithm. The x-axis plots the session size in terms of the number of SCXs and the y-axis plots the time when all of the SCXs in the session have discovered at least 90% of the other SCXs. As expected, the time to completion of the algorithm increases with increasing session size. We note that the performance scales essentially linearly as opposed to the $O(log^2 n)$ performance for the original Name Dropper algorithm [16] that our discovery protocol is based on. This is due to the fact that we use a bounded list size during each DISCOVERY round unlike the original algorithm which exchanges the entire membership set $\Gamma(X)$ in each round. This penalty is incurred to limit the communication overhead in each round. A higher (or infinite) bound on the size of the membership list that is exchanged during each DISCOVERY round would reduce the amount of time that SCXs require to discover everyone else. However this reduction in time would come at the cost of increased control traffic as a result of larger DISCOVERY and DISCOVERY_RESPONSE messages. We hypothesize that a reasonable solution would be for an SCX to start with a high value for the bound on the size of its DISCOVERY messages when it initially joins the session and gradually reduce the bound as it discovers more and more SCXs. Although we have not experimented with this form of adaptive bounds, intuition suggests that it will allow SCXs to discover most of the other session members quickly while still keeping the communication overhead low.

6.4.2 Scaling of Cost Ratio with Session Size and Node Degree Figure 10 shows the variation in cost ratio across a range of session sizes. With a small number of SCXs, most of the SCXs are directly connected to the source SCX, and hence the cost ratio is low. As the number of SCXs increases, most SCXs receive data through other transit SCXs, thereby resulting in a higher cost ratio. We measured the cost ratio for sessions with up to 350 SCXs. For most sessions, the cost ratio remains within 1.6 and 1.9. This implies that typically the average latency associated with a Gossamer distribution tree is within a factor of 2 of the corresponding latencies for source-rooted native multicast.

Figure 11 shows how the cost ratio varies with node degree for sessions with 100 SCXs. As expected, the cost ratio decreases with increasing node degrees. As the node degree increases, the depth of the distribution

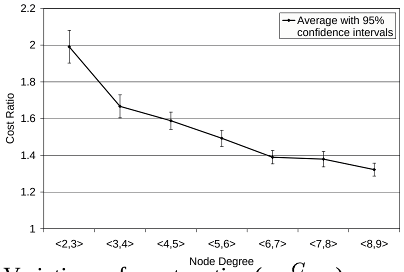

Fig. 11 Variation of cost ratio ($\frac{C}{C_{mcast}}$) vs node degree ($<k_1,k_2>$ as defined in Section 4.2).

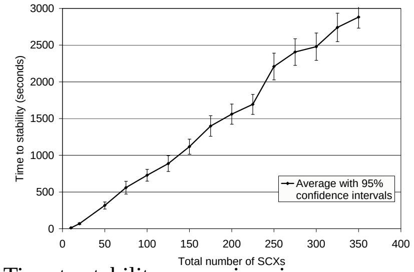

Fig. 12 Time to stability vs session size.

trees decreases, thereby decreasing the cost of routing over the tree. In the limit, Gossamer degenerates to n-way unicast and the cost ratio falls to 1.0.

6.4.3 Scaling of Stabilization Time Finally, Figure 12 shows the variation in the running time of the Gossamer protocol with respect to the total number of SCXs. We notice that the time it takes for the mesh to stabilize increases with increasing number of SCXs. This is expected since a larger session size implies more SCXs to discover and more SCXs to attempt to optimize for. We note, however, that although the mesh may not stabilize for a long time, as shown in Figure 7, most mesh changes occur early on and subside fairly quickly; only a small number of nodes continue to optimize their connections for a while. Moreover, SCXs and clients do not have to wait until the mesh has completely stabilized to start receiving data from the sources. Clients start receiving data as soon as any path is established between the source and the destination. This initial path may be sub-optimal, and the SCXs will eventually reconfigure themselves into a more optimal overlay structure.

6.5 Overhead of Application Layer Processing

The above simulations do not capture the overhead introduced by processing broadcast packets entirely at the application layer instead of within fast specialized routing and forwarding hardware. We now quantify these overheads in terms of delays and throughputs and describe how, by intelligent use of replication within a scattercast cluster, each cluster can sustain its offered load. Our experimental setup was composed of one Sun Ultra-60 running SunOS 5.8 and five Sun Ultra-10 machines running SunOS 5.6 interconnected via a switched gigabit network. We ran an experimental SCX on the Ultra-60 and used the Ultra-10 machines to act as the upstream and downstream links for the SCX. Since the goal of these experiments is to measure the performance of a single SCX, we ignore the wide-area characteristics of the overlay network.

For all of the following experiments, the test SCX was connected to one or more upstream nodes which transmitted data packets along a TCP connection to the SCX. We used TCP rather than UDP for these measurements so as to allow us to capture the additional overhead that can result from using reliable transport

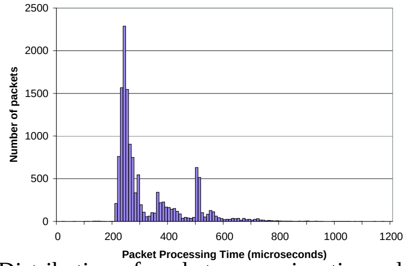

Fig. 13 Distribution of packet processing times during the lifetime of an experiment composed of one upstream link and four downstream links.

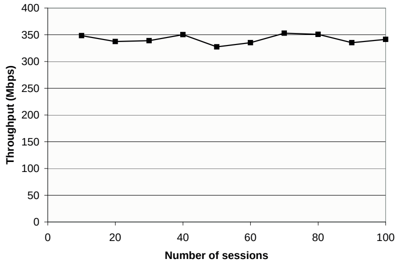

Fig. 14 Total throughput handled by the SCX machine as a function of the number of sessions being processed by that machine.

protocols between SCXs. When the SCX received a packet, a simple forwarding service module in the SCX determined the set of downstream nodes to forward the packet to, and retransmitted the packet along separate TCP connection to the appropriate downstream nodes.

6.5.1 Packet Processing Delay To determine the packet processing delay, we executed an experiment containing a single upstream source and four downstream links attached to the SCX. We measured the time that each packet took between arriving on the SCX's network interface and the last copy of the same packet departing the SCX for one of the downstream nodes. Figure 13 is a histogram depicting the distribution of the processing delays that we observed during a single run of this experiment. The y-axis represents the number of packets that incurred a processing delay of between $\pm 5\mu s$ of the corresponding x-axis value. As seen from the figure, most of the data packets encounter a processing delay of between 240 and $250\mu s$. A smaller number of packets experienced a higher delay of $500\mu s$, while a handful of packets experienced delays as high as 2–$3ms$ (not shown in the figure). Thus, for most packets, the processing overhead is substantially lower than typical wide area latencies, which can range anywhere from a small number of milliseconds to hundreds of milliseconds. We note that we achieve these performance results by running our SCX on a stock SunOS kernel. We expect that the SCX performance can be significantly improved by using more finely tuned operating system kernels such as the IO-Lite system [29], which eliminates kernel overhead due to redundant copying and multiple buffering of I/O data within the kernel and the application.

6.5.2 SCX Throughput To determine the throughput that a single machine within a scattercast cluster can sustain, we ran an experiment that measured the total number of incoming and outgoing bytes processed by the machine per second, as we varied the number of sessions being processed on that machine. We assumed that each session had two independent sources of data and two outgoing downstream links.[5] Figure 14 shows the

[5] Other experiments revealed that the aggregate throughput was not affected by the upstream/downstream configuration of each session.

aggregate throughput within a machine as a function of the number of sessions being handled by that machine. We notice that even as the number of sessions increases, the aggregate throughput remains the same (although, it does result in progressively reducing the amount of total bandwidth that is available to a single session). The figure shows that a single Ultra-60 can handle approximately 350Mbps worth of data. By replicating SCXs on multiple machines within the cluster, and spreading the sessions being handled by the cluster across all of those machines, we can expect a cluster of three machines to handle the entire throughput of 1Gbps that can be sustained by our experimental hardware.

6.6 Scattercast Adaptability: Real Applications

The above experiments demonstrate the efficiency, scaling behavior, and processing overheads of the scattercast architecture. To evaluate scattercast's ability to adapt its behavior to different kinds of applications, we present our experience building two applications on top of a prototype scattercast implementation.

6.6.1 Internet MP3 Radio Our first application is an Internet audio broadcasting system called MusiCast. Unlike traditional audio broadcasting systems, which rely either on a simple non-scalable client-server model (e.g. Internet Radio web-sites) or on ad hoc hand-configured distribution networks such as the Real Audio splitter network, MusiCast relies on the flexibility of the scattercast framework to provide an efficient and dynamically configurable audio distribution environment.

MusiCast servers inject audio content into the scattercast network using MP3 (MPEG 1 or 2 Layer III Audio) as the underlying audio format. To allow standard clients such as WinAmp [28] and RealAudio [31] to "tune" into the broadcast, SCXs export an HTTP interface to the clients. Clients contact their local scattercast cluster frontend which in turn redirects them to the appropriate SCX for their broadcast. If the client is capable of using multicast to receive MP3 broadcasts (e.g. WinAmp with a multicast plugin [24]), it may directly communicate with its SCX using a locally scoped multicast group. The MusiCast SCXs use an RTP-based service module to communicate with each other. If the SCX notices packet loss based on RTCP receiver reports that provide reception quality feedback from downstream nodes, it is possible for the service module to then dynamically transcode the audio stream using lower bit-rates or notify the source to encode at a lower rate. Our prototype MusiCast implementation however does not include this feature.

6.6.2 Electronic Whiteboard for Online Presentations Our second application—an online presentation system—extensively uses the adaptability of the scattercast framework to not only adjust its level of reliability and bandwidth usage depending upon available capacity, but also adapts the source's data stream depending upon the

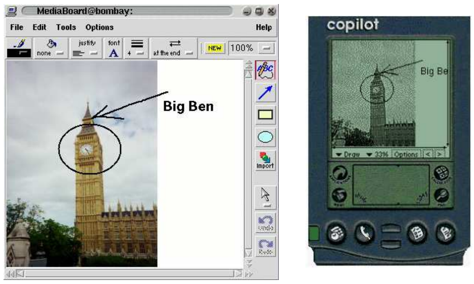

Fig. 15 The desktop whiteboard application interacting seamlessly with a PalmPilot whiteboard client. The PalmPilot screenshot was taken from the xcopilot, a software emulator for the PalmPilot.

characteristics of the individual receivers. Sources and receivers in this application use a shared electronic whiteboard based on the *wb* application [21] developed at the Lawrence Berkeley Laboratories. A source's data stream is composed of a number of individual slides, each slide containing different kinds of objects such as text, scribbles, and high-resolution images. The service module for the whiteboard's transport protocol adapts the data stream depending upon receiver interest and network characteristics. Receivers periodically send reports to their SCX indicating their level of interest. We use a binary interest indicator for each slide: the slide that a receiver is currently looking at has a high interest level, while all other slides have low interest. The SCX's service module collects aggregate interest information from all of its clients and forwards it to its upstream SCX. Each SCX uses this information to control how bandwidth allocation and loss recovery take place within the session. For example, packet losses on slides with high interest level are recovered with a higher priority than losses on slides with lower interest levels. Similarly, ADUs on slides with greater interest levels are prioritized for transmission over ADUs on other slides.

The interface that the whiteboard service module exports to its receivers depends upon the capabilities of the receivers' devices. For example, if a receiver tunes into a presentation using a PalmPilot PDA, then the receiver's SCX accordingly adapts the application's ADUs to allow faster delivery and rendering at this impoverished device. It transcodes all image ADUs into simple 2 bits-per-pixel bitmaps by throwing away excess resolution and color information, thereby speeding up data transmission across the typically restricted wireless networks that PalmPilots use as well as eliminating the need for complex rendering algorithms within the PalmPilot application. Figure 15 shows a snapshot of two receivers tuned into an online presentation via scattercast; one receiver is a typical desktop application while the other is a simpler PalmPilot program. This application demonstrates that not only can scattercast provide an adaptable framework for customizing transport behavior, but also can provide an integrated environment for interfacing a wide variety of end clients and networks into a unified broadcasting system.

7 Conclusion

This paper describes scattercast, a unified architecture for Internet broadcasting that addresses three important concerns: (1) the need for an efficient infrastructure that is optimized for Internet broadcasting, (2) an ability to customize the infrastructure to provide a range of broadcasting services rather than a single standardized service model, and (3) ease of deployment of the infrastructure. The scattercast architecture exploits application-layer intelligence to build a broadcast overlay network instead of the traditional network-layer approach. Scattercast uses a distributed protocol, Gossamer, to build and to maintain a dynamically configurable overlay topology on top of which it distributes broadcast content. Finally, by creating a flexible programmable transport framework that integrates application awareness into service-specific modules, we provide an avenue to adapt the broadcasting framework for existing and future application environments, and to incrementally deploy new broadcast services.

Our detailed evaluations demonstrate the ability of scattercast to provide an efficient broadcasting substrate. The network delays that we observed during our simulations are within a factor of two of the corresponding delays using a network-layer approach such as IP multicast. Yet scattercast provides far greater flexibility that router-based approaches by allowing service developers to customize the behavior of the infrastructure depending upon the needs of their applications. The MusiCast and whiteboard applications that we have built on top of a prototype scattercast implementation attest to this flexibility. We believe that such a value-added service infrastructure that combines a flexible programming framework with an intelligent application-layer network will become an important Internet paradigm of the future.

References

1. AKAMAI TECHNOLOGIES INC. FreeFlow: High Performance Internet Content Delivery, 1999. `http://www.akamai.com/`.
2. ANONYMIZER.COM. Privacy is your right. `http://www.anonymizer.com/`.
3. BRAY, T., PAOLI, J., AND SPERBERG-MCQUEEN, C. M. *XML: Extensible Markup Language*, Dec. 1997. W3C Proposed Recommendation. `http://www.w3.org/TR/PR-xml-971208`.
4. CHAWATHE, Y. *Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service*. PhD thesis, University of California, Berkeley, Dec. 2000.
5. CHAWATHE, Y., AND BREWER, E. System Support for Scalable and Fault Tolerant Internet Services. In *Proceedings of Middleware '98* (Lake District, U.K., Sept. 1998).
6. CHAWATHE, Y., MCCANNE, S., AND BREWER, E. A. RMX: Reliable Multicast for Heterogeneous Networks. In *Proceedings of INFOCOM 2000* (Tel Aviv, Israel, Mar. 2000).
7. CHENG, C., RILEY, R., KUMAR, S. P. R., AND GARCIA-LUNA-ACEVES, J. J. A Loop-Free Extended Bellman-Ford Routing Protocol Without Bouncing Effect. In *Proceedings of SIGCOMM '89* (1989).
8. DEERING, S., AND CHERITON, D. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems 8*, 2 (May 1990), 85–110.
9. DEMERS, A. J., GREENE, D. H., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H. E., SWINEHART, D. C., AND TERRY, D. B. Epidemic Algorithms for Replicated Database Maintenance. *ACM Operating Systems Review 22* (Jan. 1988), 8–32.

10. FASTFORWARD NETWORKS/INKTOMI MEDIA DIVISION. Broadcast Overlay Architecture, 2000. `http://www.ffnet.com/`.
11. FLOYD, S., JACOBSON, V., LIU, C., MCCANNE, S., AND ZHANG, L. A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing. In *Proceedings of ACM SIGCOMM '95* (Boston, MA, Aug. 1995), pp. 342–356. A more complete version of this paper appeared in IEEE/ACM Transactions on Networking, December 1997, 5(6) pp. 784–803.
12. FOX, A., GRIBBLE, S., CHAWATHE, Y., BREWER, E., AND GAUTHIER, P. Cluster-based Scalable Network Services. In *Proceedings of SOSP '97* (St. Malo, France, Oct. 1997), pp. 78–91.
13. FOX, A., GRIBBLE, S. D., CHAWATHE, Y., AND BREWER, E. A. TranSend Web Accelerator Proxy, 1997. Free service deployed at UC Berkeley.
14. FRANCIS, P. Yoid: Extending the Internet Multicast Architecture. `http://www.aciri.org/yoid/`.
15. GOYAL, P., VIN, H. M., AND CHENG, H. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM '96* (Stanford, CA, Aug. 1996).
16. HARCHOL-BALTER, M., LEIGHTON, T., AND LEWIN, D. Resource Discovery in Distributed Networks. In *Proceedings of the 18th Annual ACM-SIGACT/SIGOPS Symposium on Principles of Distributed Computing* (Atlanta, GA, May 1999).
17. HELDER, D. A., AND JAMIN, S. Banana Tree Protocol, an End-host Multicast Protocol. Unpublished Report, July 2000.
18. HENDRICK, C. *Routing Information Protocol (RIP)*, June 1988. RFC-1058.
19. HUA CHU, Y., RAO, S., AND ZHANG, H. A Case For End System Multicast. In *Proceedings of ACM Sigmetrics '00* (Santa Clara, CA, June 2000).
20. JACOBSON, V., AND DEERING, S. Administratively Scoped IP Multicast. In *Proceedings of the 30th Internet Engineering Task Force* (Toronto, Canada, July 1994).
21. JACOBSON, V., AND MCCANNE, S. *LBL Whiteboard*. Lawrence Berkeley Laboratory. Software available at `ftp://ftp.ee.lbl.gov/conferencing/wb`.
22. JANNOTTI, J., GIFFORD, D. K., AND JOHNSON, K. L. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI)* (San Diego, CA, Oct. 2000), USENIX.
23. LILJEBERG, M., HELIN, H., KOJO, M., AND RAATIKAINEN, K. Enhanced Services for World-Wide Web in Mobile WAN Environment. Tech. Rep. C-1996-28, University of Helsinki, Computer Science, Apr. 1996.
24. LIVE.COM. A Multicast Plugin for WinAmp. `http://www.live.com/multikit/winamp-plugin.html`.
25. LOUGHEED, K., AND REKHTER, Y. *A Border Gateway Protocol (BGP)*, June 1989. RFC-1247.
26. MCCANNE, S., JACOBSON, V., AND VETTERLI, M. Receiver-driven Layered Multicast. In *Proceedings of ACM SIGCOMM '96* (Stanford, CA, Aug. 1996), pp. 117–130.
27. MEYER, D. *Administratively Scoped IP Multicast*, July 1998. RFC-2365.
28. NULLSOFT INC. The WinAmp MP3 Player. `http://www.winamp.com/`.
29. PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI)* (New Orleans, LA, Feb. 1999), USENIX.
30. PELC, A. Fault-tolerant Broadcasting and Gossiping in Communication Networks. *Networks 28*, 5 (Oct. 1996).
31. REAL.COM. RealPlayer. `http://www.real.com/`.
32. RIVEST, R. *The MD5 Message-Digest Algorithm*, Apr. 1992. RFC-1321.
33. SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. *RTP: A Transport Protocol for Real-Time Applications*, Jan. 1996. RFC-1889.
34. SCHULZRINNE, H., RAO, A., AND LANPHIER, R. *Real Time Streaming Protocol (RTSP)*, Apr. 1998. RFC-2326.
35. SPEAKMAN, T., FARINACCI, D., LIN, S., AND TWEEDLY, A. *Pragmatic General Multicast (PGM) Reliable Transport Protocol*. CISCO Systems, 1998. Internet Draft.
36. VAN RENESSE, R., MINSKY, Y., AND HAYDEN, M. A Gossip-style Failure Detection Service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)* (Lake District, U.K., Sept. 1998), pp. 55–70.
37. WAITZMAN, D., PARTRIDGE, C., AND DEERING, S. *Distance Vector Multicast Routing Protocol (DVMRP)*, Nov. 1988. RFC-1075.
38. YANO, K., AND MCCANNE, S. The Breadcrumb Forwarding Service: A Synthesis of PGM and EXPRESS to Improve and Simplify Global IP Multicast. *ACM Computer Communication Review 30*, 2 (Apr. 2000).
39. ZEGURA, E. W., CALVERT, K. L., AND BHATTACHARJEE, S. How to Model an Internetwork. In *Proceedings of IEEE Infocom '96* (San Francisco, CA, Mar. 1996).