The Design and Implementation of StrandCast

Brian Begnoche
bbegnoch@cs.unc.edu

David Gotz
gotz@cs.unc.edu

Ketan Mayer-Patel
kmp@cs.unc.edu

University of North Carolina at Chapel Hill
CB #3175, Sitterson Hall
Chapel Hill, NC 27599 USA

ABSTRACT

In this paper, we present a detailed overview of *StrandCast*, an application-layer multicast protocol for stretch invariant applications. StrandCast's distribution topology is a linear list of participating receivers, which we call a *strand*. We describe three benefits that StrandCast has over traditional application-layer multicast topologies. The first is low and constant stress levels. The second is fast join and leave operations. The third benefit is a semi-reliable transmission scheme that makes long strands practical for very large groups of receivers. We also discuss various stretch invariant applications for which StrandCast is most suitable, including systems that utilize receiver-driven layered multicast for scalable congestion control.

1. INTRODUCTION

Traditional unicast delivery is designed to support one-to-one data flows where a single source sends data packets to a single recipient. In one-to-many applications, unicast requires that the source replicate every packet for each recipient. For an application with a large group of recipients, unicast is extremely inefficient because its required bandwidth scales linearly with the size of the group.

Multicast transmission is designed to remove the inefficiencies associated with unicast and one-to-many applications. First proposed as an infrastructure based network-layer solution [4], multicast is designed to replicate data packets within the network. This allows a source to transmit only one copy of its data irregardless of the number of receivers.

Unfortunately, infrastructure multicast requires the participation of ISPs due to its reliance on network-layer support within the network itself. This requirement, coupled with the slow pace of adoption among ISPs, has posed an enormous challenge for the deployment and reliability of network-layer multicast services.

In response to these challenges, several application-layer multicast protocols (i.e. [1, 2, 3, 5, 9]) have been designed that do not require support from the underlying network-layer. As a result, they rely only on end-user systems and and are relatively easy to deploy. However, application-layer multicast protocols are inherently less efficient than network-layer solutions because they require replication and forwarding to be performed by the receivers themselves.

Application-layer multicast protocols typically maintain a hierarchical overlay topology designed to distribute data from one or more sources to the group of receivers with minimal *stress*, a measure of the data replication load on overlay links, and *stretch*, a measure of the latency from source to receiver . As receivers join and leave the group, the topology must be updated to optimize these two metrics.

The minimization of stress and stretch are often in conflict for a given topology. A low-stretch distribution topology calls for few network hops between the source and any receiver. This is accomplished by creating a high-degree distribution tree. However, high-degree trees increase the stress on interior nodes. Different application-layer multicast protocols have addressed this conflict using a wide variety of solutions which attempt to balance the competing requirements for both low stretch and low stress.

However, there are a class of applications for which the stretch metric has no impact on performance. For example, applications that make use of carousel transmission, where data is repeatedly sent by the source, will be largely unaffected by changes in stretch. We refer to these applications as *Stretch Invariant*.

One such stretch invariant application is pyramid broadcasting [10]. A pyramid broadcasting system repeatedly transmits segments of a video across several channels. For receivers in this system, it is not important how long it takes for data to transit from the source. It is critical only that data begins arriving shortly after a request, even if it left the source long ago.

1.1 StrandCast

In this paper, we introduce *StrandCast*, an application-layer multicast protocol for stretch invariant applications. Because StrandCast is designed for applications immune to the effects of stretch, we attempt to minimize stress without regard to the impacts on stretch. This removes the need to balance competing optimizations and leads to a simple and efficient application-layer multicast design. StrandCast maintains an overlay distribution topology as a linear list of participating receivers, which we call *strands*.

The simplicity of the overlay topology provides several benefits over traditional application-layer multicast topologies, including low and constant stress levels, as well as very fast join and leave operations. In addition, the semi-reliable transmission portion of the StrandCast design makes long strands practical for very large groups of receivers.

1.2 Organization

The remainder of this paper is organized as follows. We begin with a review of related work in Section 2. An overview of StrandCast, together with a detailed design review, is discussed in Section

UNC Technical Report TR05-004
February, 2005

3. Section 4 defines the application programming interface for the three types of nodes defined by StrandCast. We conclude with a discussion of future work in Section 5.

2. BACKGROUND

In this section we discuss a selection of previous work most related to our research. We begin by reviewing IP multicast, the most widely accepted standard for network-layer multicast. We then discuss a host of application-layer multicast alternatives to IP multicast. Finally, we define stretch invariance and present two examples of stretch invariant applications.

2.1 IP Multicast

Multicast transmission is an efficient mechanism for one-to-many applications. Traditional unicast transmission requires a server to transmit a copy of each packet for every receiver. Multicast, however, allows a server to transmit a single copy of each packet regardless of the number of receivers. The multicast protocol itself manages packet replication and delivery to the group of receivers.

Multicast has been standardized around the IP multicast protocol, a design that requires core network routers to maintain group membership information and to replicate and forward packets. Group management is performed via join and leave requests that update the soft-state information maintained on intermediate routers.

Unfortunately, IP multicast's reliance on support within the core of the network requires that internet service providers adopt and support IP multicast within their networks. To date, IP multicast has not been widely deployed. In addition, the use of unreliable join and leave messages combined with the use of soft-state information can lead to long latencies in response to changes in group membership.

2.2 Application-Layer Multicast

The recognized problems with IP multicast have led researchers to search for alternatives that are more easily deployable and capable of providing improved services. Several competing protocols for Application-Layer Multicast (ALM) have been proposed [1, 2, 3, 5, 9] as alternative multicast solutions.

ALM protocols, in contrast to IP multicast, are implemented at the application layer and require no direct support from the underlying network infrastructure. This feature alone makes ALM protocols easier to deploy. However, because they rely only on participating nodes for packet replication and forwarding, the distribution tree for an ALM protocol is typically less efficient than the corresponding IP multicast distribution tree. As a result, the efficiency of ALM protocols are typically measured by two metrics [3]: *stretch* and *stess*. These metrics have been defined to compare the efficiency of an ALM protocol's distribution tree to the ideal tree as defined by IP multicast.

2.2.1 Stretch

Application-layer multicast inevitably results in longer end-to-end latencies in comparison to IP multicast. IP multicast packets follow the shortest path between source and receiver as determined by IP routing. However, ALM utilizes an overlay network in which packets are sent from the source to individual receivers via unicast links between participating peers. As a result, the typical transmission path for a packet using ALM is less efficient. The increase in server-to-receiver latency is referred to as *stretch*.

Most ALM protocols aim to minimize the impact of stretch by continuously optimizing the network overlay distribution tree. This leads to constant changes in the overlay topology and can make joining or leaving a group an expensive operation.

2.2.2 Stress

Peers within an ALM session are responsible for packet duplication and forwarding. As a result, identical packets regularly traverse individual links multiple times. This is in contrast to the more efficient IP multicast protocol which never sends a particular packet over a link more than once. The number of times a packet travels over a link is referred to as *stress*.

Stress is typically greatest towards the root of the overlay distribution topology. ALM protocols regularly employ a high-degree distribution tree designed to reduce stretch. This creates a short tree, reducing the server-top-receiver latency. However, the high degree requires nodes with many children in the overlay topology to forward the same packet multiple times. The result is a highly stressed node. Furthermore, due to the constant optimization of their overlay topology, an individual peers will be subjected to highly variable stress levels as they migrate from interior to leaf positions. The high variability in link stress can cause problems both for high bandwidth applications and for applications that manage congestion by subscribing to multiple multicast groups (i.e. Receiver-Driven Layered Multicast [8]).

2.3 Stretch Invariance

Existing proposals for application-layer multicast continuously optimize the overlay topology to satisfy the competing needs to reduce both stress and stretch. This leads to complex distribution trees and optimization algorithms. However, there are a number of applications for which stretch has no impact on performance. We call these *stretch invariant* applications.

For example, systems that employ carousel transmission are inherently stretch invariant. These applications repeatedly transmit the same unit of data over a communication channel. Individual receivers simply tune in when they are ready to receive data and wait for the information to be repeated. There are several carousel-based applications, including the datacycle architecture [7] for databases, pyramid broadcasting [10] for video-on-demand, and channel set adaptation [6] for large-scale streaming of digitized spaces work.

An application-layer multicast algorithm designed specifically for stretch invariant applications can avoid the difficult task of optimizing for the competing needs of both stretch and stress. By removing the minimization of stretch from the design criteria, we are able to design a highly efficient and scalable application-layer multicast algorithm capable of supporting stretch invariant applications.

3. STRANDCAST

StrandCast is an application-layer multicast protocol designed specifically for stretch invariant applications. Its design avoids the overhead expenses associated with the hierarchical overlay topologies typically used by most other application-layer protocols.

In this section, we describe the distrubition topology employed by StrandCast where receivers are organized into linear lists of receivers called *strands*. Within a strand, participants assume one of three roles: source, sink, or receiver. StrandCast can scale to support large user groups, because its receivers can both join and leave strands with minimal latency and with only local effects on the overall distribution topology.

3.1 Distribution Topology

StrandCast arranges its receivers into linear lists called strands. Each node within a strand plays one of three roles: a source, a sink, or a peer. A single source node is located at the start of each strand. Similarly, a single sink node is located at the end of each strand. In between the source and the sink, there are zero or more connected

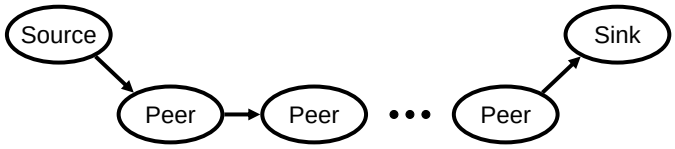

Figure 1: A single strand of nodes arranged in StrandCast's linear topology. Every strand begins with a source and ends with a sink, with a number of peers located in between.

peers. Figure 1 illustrates the linear topology of a typical strand. Data flow originates with the source and passes down from peer to peer toward the sink. Each strand is independent, so multiple strands require multiple sources, sinks, and sets of peers. However, this independence allows a group of sources and a group of sinks to be distributed across multiple machines.

3.2 Three Node Roles

The next three sections will introduce StrandCast's three node roles and explain in more detail how they participate in StrandCast. Refer to Section 4 for details on the APIs for each role.

3.2.1 The Source

The source is located at the front of the strand and is the point of origin for all data that flows down a strand. It maintains a connection with only the first peer in the strand, and this single connection has three effects. The first effect is that the source only participates in a join transaction when a peer joins the strand while it is empty. The second effect is that it only participates in a leave transaction when the first peer in the strand leaves. The third effect is that when the source sends a data packet, it only has to send it once. It is crucial that the source be this simple, because a source that uses very little resources and bandwidth makes it possible to scale StrandCast to a high number of strands.

3.2.2 The Sink

The sink serves two purposes. Its primary purpose is access control for new peers joining a strand. When a peer joins, it makes an initial connection with the sink to get the information it needs to attach itself to the end of the strand. Like the source, the sink allows StrandCast to scale to thousands of strands because it isolates access control in a separate node that can be distributed if needed. We will discuss joining a strand in greater detail in Section 3.3. The secondary purpose of the sink is to coordinate control messages when there is a break in the strand from an ungraceful leave. We will discuss ungraceful leaves in Section 3.6.

3.2.3 Peers

Peers are responsible for the receiving and forwarding data packets along the strand. Because of StrandCast's linear topology, a peer is aware of only its two neighbors. The upstream neighbor is either the source (when the peer is first in line), or the previous peer in the strand. The downstream neighbor is either the sink (when the peer is the last in line), or the following peer in the strand.

The peer is the most complex of the three node roles. It maintains a connection with two neighbors, therefore it must participate in the transactions of its neighbors joining and leaving the strand as well as when the peer itself joins and leaves. During these transactions, the peer must maintain an unbroken flow of data packets. Later, we will discuss two StrandCast properties that add complexity to the peer: semi-reliable transmission in Section 3.5 and ungraceful leaves in Section 3.6.

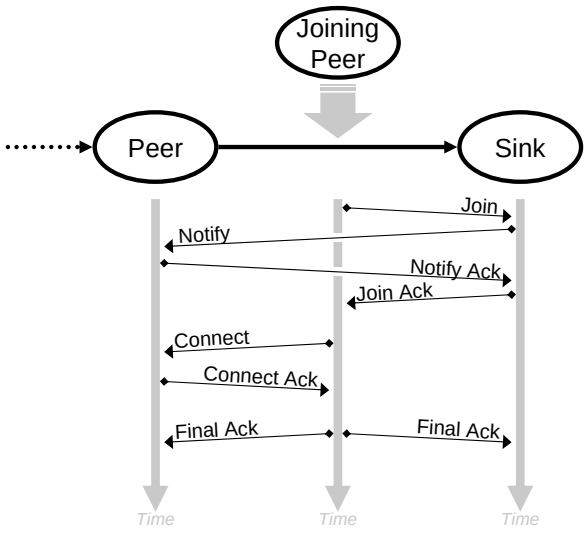

Figure 2: The message transactions of a StrandCast join operation.

3.3 Joining a Strand

When a peer joins a strand, it performs two parallel and dependent three-way handshakes, one with the sink and the second with the last peer of the strand. They are dependent because each step of one handshake depends on steps taken in the other. In addition, there is a call and response between the sink and the last peer of the strand. Figure 2 illustrates the message transactions of a StrandCast join operation.

A new peer initiates a join by sending a join message to the sink. The sink then notifies the last peer of the strand of the join, and that peer acknowledges. From this point, the sink and the last peer of the strand no longer communicate, and the last peer waits for the new peer to establish a connection. The sink continues its handshake with the new peer by acknowledging the original join message sent by the new peer.

Once the new peer receives a join ACK from the sink, it sends a message to the last peer to initiate a new connection, and the last peer acknowledges. The new peer sends ACKs to the sink and to the last peer to finish both handshakes. At this point, both peers and the sink are in a stable state where the last peer before the join forwards strand packets to the new peer, the new peer receives packets at the tail of the strand, and the sink waits for other peers to join the strand.

3.4 Leaving a Strand

When a peer leaves the strand, the three nodes involved are the peer leaving, its upstream neighbor, and its downstream neighbor. Leaving a strand is somewhat simpler than joining because there are no interleaved handshakes. Instead, there are three call and response message transactions. Figure 3 illustrates the message transactions of a StrandCast leave operation.

A peer initiates a leave by sending a message to both of its neighboring peers that notifies them of the leave. This initial message also notifies them of the sequence number at which the leave will occur, which we call the *splice point*. By doing this, the leaving peer assumes responsibility for forwarding all packets up to that splice point, and therefore its neighboring peers know to resume data packet transfer starting with the next sequence number after that splice point. This guarantees reliable transmission since the leaving peer will forward packets before it leaves. However, but

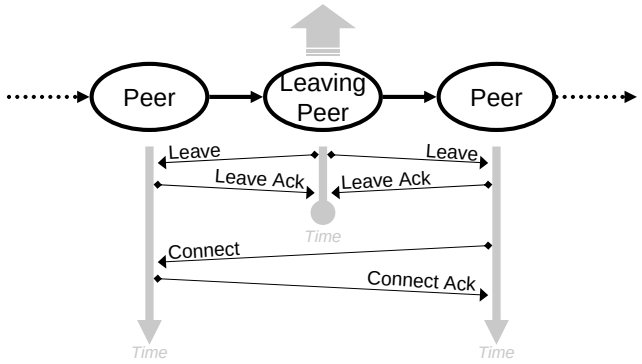

Figure 3: The message transactions of a StrandCast leave operation.

this also assumes the leaving peer will behave according to StrandCast specification. We will discuss the deviant case of an ungraceful leave in Section 3.6.

The upstream and downstream peers will acknowledge the leaving peer's leave messages, after which the downstream peer sends a connect message directly to the upstream peer. The upstream peer acknowledges the connection and begins sending data packets to the downstream peer. At this point, the upstream peer is in a stable state where it forwards data packets to its downstream peer. However, the downstream peer is in a pseudo-stable state until it receives all the data packets up to the splice point from the leaving peer. Once the leaving peer finishes draining, it severes its remaining connection and the downstream peer is in a true stable state.

3.5 Semi-Reliable Transmission

StrandCast provides semi-reliable transmission to overcome the effects of packet loss, which are magnified by the increased stretch in StrandCast's linear topology. Packets must travel through $O(n)$ links. This is larger than the $O(lg(n))$ links typical in most tree-based ALM algorithms. The increased stretch means that there is greater opportunity for a packet to be lost along its path. Furthermore, in the event of a loss, the effect is more disastrous since it affects all peers below the link where the loss occurs. In the linear case, the average impact of a loss will affect $\frac{n}{2}$ peers.

StrandCast uses three customizable transmission rate parameters based on a target rate to provide semi-reliable transmission. We say that StrandCast's transmission scheme is *elastic* because it allows a node to temporarily deviate from its constant data transmission rate to recover from loss. StrandCast assumes that all peers can sustain this constant transmission rate without suffering congestion. If a peer cannot meet that requirement, then it cannot participate in the strand. We define StrandCast's three transmission parameters as the total transmission rate, the data transmission rate, and the retransmission rate.

R_T: The total transmission rate defines a maximum of how fast a node can send data packets, both to forward new packets and to retransmit rerequested ones. R_T is the rate that peers must be able to sustain in order to participate in a strand.

R_D: The data transmission rate is the rate at which a node can forward incoming data packets, and it cannot exceed the total transmission rate. Equation 1 is the first requirement for semi-reliable transmission.

$$R_D <= R_T \tag{1}$$

R_R: The retransmission rate is the rate at which a node can retransmit lost data packets. It also cannot exceed the total transmission rate, so Equation 2 is the second requirement for semi-reliable transmission.

$$R_R <= R_T \tag{2}$$

There is no requirement that $R_D + R_R = R_T$. In fact, the three values can be anything as long as Equations 1 and 2 are satisfied. If $R_D + R_R > R_T$, then there is some overlap in the allotment of data transmission and data retransmission. In this case R_R has priority over R_D. The priority of R_R means that StrandCast attempts to handle any retransmission requests first up to the rate R_R, and it attempts to send data second and at a rate as close to R_D as possible without exceeding R_T.

If a node sending data packets experiences no loss, it will forward data packets at the same rate at which it receives them, since it does not spend any bandwidth on retransmitting packets. The amount of total bandwidth not alloted to data transmission, $R_T - R_D$, is not used in this case. However, if a node does experience loss, one of two things may happen. The first possibility is that the node is able to both forward packets and satisfy retransmission requests without exceeding R_T. Otherwise, if enough loss occurs such that R_T is not sufficient to forward new data packets and retransmit others, then some portion of R_D must be sacrificed and data packets will accumulate in a buffer. When the sum of the number of incoming packets and the number of retransmit requests drops back below R_T, then the node will use the remaining portion of R_R to send buffered packets. StrandCast's transmission scheme is elastic because it uses unused retransmission bandwidth to "catch up" and send data packets that were buffered during periods of congestion. This elasticity is what allows StrandCast to provide the same quality of service to all peers.

StrandCast's reliability depends on the size of R_R relative to R_T. If R_R is low, then a strand will use a small portion of R_T to handle loss. Therefore the overhead of alloting retransmission bandwidth is minimal, but the likelihood of a permanent loss occurring because retransmission requests exceed R_R is greater. On the other hand, if R_R is high, then a strand is more reliable because it uses a large portion of its total bandwidth to handle loss. The drawback is that a smaller amount of the total bandwidth is used to forward data. We can express StrandCast's reliability with the ratio $\frac{R_R}{R_T}$. If $R_R = R_T$, then the strand is totally reliable, because in the case of total loss, a peer will use all its available bandwidth to rerequest lost packets before it will forward packets it has received. However, it will reliably accomodate total loss at the expense of possibly having to buffer a large number of packets it cannot send until congestion decreases and the peer then can use some of R_T to send them.

3.6 Ungraceful Leaves

An ungraceful leave happens when a peer disappears from a strand without performing a leave operation. This could happen, for example, if a client's host loses its network connection or crashes. The upstream neighbor will detect such a break by a timeout resulting from data packets not being acknowledged. When a peer detects a break, it sends a break message to the sink, which will respond by sending a repair message up the strand that contains the address of the peer above the break. Peers will pass this repair message up the strand until the peer immediately below the break will attempt to send a repair message to the missing peer, that message will timeout, and the peer will detect that the break is above it. Since the repair message passed up the strand contains the address of the peer above the break, the peer below the break can initiate a a transaction between the two peers to splice the strand back together.

3.7 StrandCast as a Receiver-Driven Multicast

StrandCast's constant and minimal stress on a link due to its linear topology makes it an ideal candidate for receiver-driven multicast (RLM) [8] for stretch invariant applications. RLM subdivides a high single high bit-rate stream into a handful of lower bit-rate streams, each of which is transmitted to its own multicast group. The low bit-rate streams are called layers. Because each layer is relatively thin, receivers can typically subscribe to multiple layers at the same time. Individual receivers can control the bit-rate of their incoming data stream by changing the number of layers to which they subscribe. In this way, RLM enables course-grained congestion control that scales to very large groups of users.

The weakness of RLM is that it does not perform well with typical tree-based ALM protocols because links suffer from highly variable stress that can reach high levels. RLM was initially intended for IP multicast where a node could forward one packet to multiple recipients and the packets would replicate within the network. In the case of tree-based ALM protocols, a node must replicate each packet and forward it once for every recipient. For example, a node with ten children will have to send out ten packets for every one it receives, and the resulting stress on its link due to the outgoing packets will limit the node's subscription to an RLM group.

StrandCast, however, minimizes stress because each node only has one downstream peer and therefore only has to forward each packet once. As a result, an application can extend StrandCast to use the ideas of RLM by using a group of many thin strands instead of one to accomodate receivers with different amounts of bandwidth.

4. NODE ROLE APIS

The APIs of the StrandCast nodes are designed to be simple and customizable through the use of event handlers. They use object-oriented programming techniques, and each of the three node types is a class: Source, Sink, and Peer. StrandCast also contains a virtual class called Handler, which a user can use to create a Handler subclass that implements the event handler functions he choses to behave according to a particular application using StrandCast. Each of the three node classes has three basic function type pairs that are (1) allocating and deallocating a node, (2) joining and leaving a strand, and (3) registering and unregistering event handlers.

4.1 Source API

Source allocation methods:

- **Source(char* src_name, unsigned short src_port, double total_rate, int max_burst_length, double rexmit_rate, int max_rexmit_burst_length, int max_packet_size)**: Source constructor. *src_name* and *src_port* specify a local address and port. *total_rate* and *max_burst_length* specify a token bucket that defines the strand's total transmission rate, R_T, *rexmit_rate* and *max_rexmit_burst_length* specify a token bucket that defines the strand's retransmission rate, R_R, and *max_packet_size* specifies the maximum allowable payload size of a data packet in the strand. The data transmission rate of the strand, R_D, is not specified explicitly, but rather a source sets it implicitly by sending data packets at that rate with the *send* method.

- **~Source()**: Source deconstructor

Source methods to open and close a strand:

- **void open(char* sink_name, unsigned short sink_port)**: Opens a new strand by connecting to the sink specified by its address *sink_name* and its port *sink_port*.

- **void close()**: Closes the strand. After *close* is called, *open* cannot open a new strand. Instead, a new source object must be used.

Source data transmission methods:

- **void send(char* data, int size)**: Sends a data packet through the strand. The size of the data block should not exceed the maximum packet size specified in the constructor. The source implicitly sets the strand's data transmission rate, R_D, by sending packets at that rate.

- **int canSend()**: Returns 1 if the source is in a state in which it can send.

Source event handler registration functions: The only option for a source event handler is the control handler, which is mostly used for debugging by reporting incoming and outgoing StrandCast control messages and state changes. Refer to Section 4.4 for more information on StrandCast event handlers.

- **void registerControlHandler(Handler *ch)**: Registers a control event handler.

- **void unregisterControlHandler(Handler *ch)**: Unregisters a control event handler.

4.2 Sink API

Sink allocation methods:

- **Sink(char* sink_name, unsigned short sink_port)**: Sink constructor. *sink_name* and *sink_port* specify a local address and port.

- **~Sink()**: Sink deconstructor

Sink methods to open and close a strand:

- **void open(char* src_name, unsigned short src_port)**: Joins with a source specified by its address *sink_name* and *sink_port* to create a new strand.

- **void close()**: Closes the strand. After *close* is called, *open* cannot open a new strand. Instread, a new sink object must be used.

Sink event handler registration functions: The only option for a sink event handler is the control handler, which is mostly used for debugging by reporting incoming and outgoing StrandCast control messages and state changes. Refer to Section 4.4 for more information on StrandCast event handlers.

- **void registerControlHandler(Handler *ch)**: Registers a control event handler.

- **void unregisterControlHandler(Handler *ch)**: Unregisters a control event handler.

4.3 Peer API

Peer allocation methods:

- **Peer(char* peer_name)**: Peer constructor. *peer_name* specifies a local address.

- **~Peer()**: Peer destructor

Peer methods to join and leave a strand:

- **void join(char* sink_name, unsigned short sink_port)**: Joins a strand by connecting to the sink that performs as access control to the strand. *sink_name* and *sink_port* specify the address and port of the sink.
- **void leave()**: Leaves a strand. After *leave* is called, *join* cannot join another strand. Instead, a new peer must be used.
- **int isOpen()**: Returns 1 if the peer is subscribed to a strand.

Peer accessor methods to get strand transmission parameters:

- **double getRate()**: Returns the total transmission rate, R_T, of the strand.
- **int getRateDepth()**: Returns the token bucket depth of the total transmission rate, R_T, of the strand.
- **double getRexmitRate()**: Returns the retransmission rate, R_R, of the strand.
- **int getRexmitDepth()**: Returns the token bucket depth of the retransmission rate, R_R, of the strand.
- **int getMaxPacketSize()**: Returns the maximum packet size allowed on the strand.

Peer event handler registration functions: The peer API allows the registration of event handlers for when a peer receives a data packet, rerequests a data packet, retransmits a packet, experiences a local loss, recognizes a global loss, successfully joins a strand, and successfully leaves a strand. It also allows the registration of a control handler, which is mostly used for debugging by reporting incoming and outgoing StrandCast control messages and state changes. Refer to Section 4.4 for more information on StrandCast event handlers.

- **void registerRecvHandler(Handler *rh)**: Registers a receive event handler.
- **void unregisterRecvHandler(Handler *rh)**: Unregisters a receive event handler.
- **void registerRereqHandler(Handler *rrh)**: Registers a rerequest event handler.
- **void unregisterRereqHandler(Handler *rrh)**: Unregisters a rerequest event handler.
- **void registerRexmitHandler(Handler *rxh)**: Registers a retransmit event handler.
- **void unregisterRexmitHandler(Handler *rxh)**: Unregisters a retransmit event handler.
- **void registerLocalLossHandler(Handler *llh)**: Registers a local loss event handler.
- **void unregisterLocalLossHandler(Handler *llh)**: Unregisters a local loss event handler.
- **void registerGlobalLossHandler(Handler *glh)**: Registers a global loss event handler.
- **void unregisterGlobalLossHandler(Handler *glh)**: Unregisters a global loss event handler.
- **void registerJoinHandler(Handler *oh)**: Registers a join event handler.
- **void unregisterJoinHandler(Handler *oh)**: Unregisters a join event handler.
- **void registerLeaveHandler(Handler *lh)**: Registers a leave event handler.
- **void unregisterLeaveHandler(Handler *lh)**: Unregisters a leave event handler.
- **void registerControlHandler(Handler *ch)**: Registers a control event handler.
- **void unregisterControlHandler(Handler *ch)**: Unregisters a control event handler.

4.4 Handler API

Handler is a virtual class that a user can use to create an event handler or group of event handlers that respond to StrandCast events. The user can select which callback methods of Handler to implement and can implement all of them in one Handler subclass or multiple Handler subclasses. If the callback methods of a Handler subclass are implemented for more than one kind of event, then that handler must be registered for each kind of event. All five control events are registered by the same method, *registerControlHandler*. The twelve StrandCast events are described in the following paragraphs, followed by a specification of the callback methods of Handler.

Receive Event: A receive event occurs when the peer receives a data packet from its upstream neighbor.

Rerequest Event: A rerequest event occurs when the peer must rerequest the retransmission of a lost packet from its upstream neighbor.

Retransmit Event: A retransmit event occurs when the peer must retransmit a packet rerequested by its downstream neighbor.

Local Loss Event: A local loss occurs when the peer is unable to rerequest a lost packet from its upstream neighbor because all of the available retransmission rate, R_R, is used. A local loss is a permanent loss in the strand.

Global Loss Event: A global loss occurs when a peer does not receive a packet because it was permanently lost somewhere in the strand above it. A local loss at some peer implies a global loss at all peers below.

Join Event: A join event occurs when a peer successfly joins a strand.

Leave Event: A leave event occurs when a peer successfly leaves a strand.

Control Packet In Event: A control packet in event occurs when a node receives a StrandCast control message.

Control Packet Out Event: A control packet out event occurs when a node sends a StrandCast control message.

Control State Change Event: A control state change event occurs when a node changes state.

Control Address Event: A control address event occurs when a node establishes its local address and port.

Control Transmission Parameters Event: A control transmission parameters event occurs when a sink or peer receives the transmission parameters of a strand.

4.5 Handler Callback Methods

- **virtual void handle_data(DataPacket *p)**: Handles a receive event. The user can access the new data packet through *p*.

- **virtual void handle_rereq(u_int16_t seqnum)**: Handles a rerequest event. *seqnum* is the global sequence number of the rerequested packet.
- **virtual void handle_rexmit(u_int16_t seqnum)**: Handles a retransmit event. *seqnum* is the global sequence number of the retransmitted packet.
- **virtual void handle_local_loss(u_int16_t seqnum)**: Handles a local loss event. *seqnum* is the global sequence number of the lost packet.
- **virtual void handle_global_loss(u_int16_t seqnum)**: Handles a global loss event. *seqnum* is the global sequence number of the lost packet.
- **virtual void handle_join()**: Handles a join event.
- **virtual void handle_leave()**: Handles a leave event.
- **virtual void handle_packet_in(CntrlPacket *p)**: Handles a control packet in event. The user can access the control packet through *p*.
- **virtual void handle_packet_out(CntrlPacket *p)**: Handles a control packet out event. The user can access the control packet through *p*.
- **virtual void handle_state_change(conn_state_t state)**: Handles a control state change event. *state* is the new state.
- **virtual void handle_address(char* addr, u_int16_t port)**: Handles a control address event. *addr* and *port* are the local address and port.
- **virtual void handle_transmit_info(double total_rate, int max_burst_length, double rxmit_rate, int max_rxmit_burst_length, int max_packet_size)**: Handles a control transmission parameters event. *total_rate* and *max_burst_length* describe the token bucket of the total transmission rate, R_T, of the strand, *rxmit_rate* and *max_rxmit_burst_length* describe the token bucket of the retransmission rate, R_R, of the strand, and *max_packet_size* is the maximum allowed payload size of a data packet in the strand.

5. CONCLUSION AND FUTURE WORK

We have presented the design for StrandCast, a novel application-layer multicast algorithm designed for stretch invariant applications. StrandCast employs a linear overlay topology that reduces complexity and alleviates most of the overhead in topology management associated with more traditional tree-based application-layer multicast algorithms.

StrandCast maintains a linear strand of connected users for each multicast group. Within a strand, participants assume one of three roles: source, sink, or peer. Each strand begins with a source that is responsible for transmitting the stream of data packets to the group. Each strand ends with a sink that is responsible for access control and maintenence of the strand. Between the source and sink are zero or more peers which participate as data receivers and forwarders.

The simplicity of StrandCast's distribution topology provides a number of desireable properties. Both join and leave operations can be completed with very low latencies. In addition, the stress placed on each participating receiver is minimal and held constant throughout the session. This makes StrandCast an ideal solution for stretch-invariant applications that rely on receiver-driven layered multicast techniques for congestion control.

There are several possible improvements that can be made to the current StrandCast implementation. For example, improving recovery from ungraceful leaves can be made more efficient by increasing peer awareness between nodes. Our current solution can often require long delays before repairs are made as the repair messages propogate through the strand. Knowledge of neighboring peers can provide protection against ungraceful leaves and reduce the repair to a faster local operation.

Another future addition to StrandCast could be the ability to manage the peers' positions within the strand to overcome congestion points. This could be accomplished via peer swapping, where neighboring nodes within the strand could mutually decide to trade positions in an effort to improve the overall performance of the strand.

6. REFERENCES

[1] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of ACM Sigcomm*, 2002.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A largescale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 2002.

[3] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. of ACM SIGMETRICS*, 2000 2000.

[4] S. E. Deering and D. R. Cheriton. Multicast routing in a datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[5] P. Francis. Yoid: Extending the multicast internet architecture, 1999.

[6] D. Gotz and K. Mayer-Patel. A framework for scalable delivery of digitized spaces. *International Journal on Digital Libraries*, To Appear. Special Issue on Digital Museums.

[7] G. Herman, K. C. Lee, and A. Weinrib. The datacycle architecture for very high throughput database systems. In *Proc. of ACM SIGMOD*, 1987.

[8] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proc. of ACM SIGCOMM*, 1996.

[9] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast uisng content-addressable networks. In *Proc. of 3rd International Workshop on Networked Group Communication*, 2001.

[10] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *Multimedia Systems*, 4:197–208, 1996.