Scalable Data Naming for Application Level Framing in Reliable Multicast

Suchitra Raman and Steven McCanne
Department of EECS
University of California, Berkeley
{suchi,mccanne}@cs.berkeley.edu

Abstract

The Application Level Framing (ALF) protocol architecture [2] encourages application control over mechanisms that traditionally fall within the "transport layer", e.g., loss detection and recovery. Traditional ARQ-based reliable protocols for unicast (e.g., TCP) as well as multicast (e.g., Horus [30], RMTP [15], etc.) number data units sequentially to detect losses. Unfortunately, these transport-level sequence numbers do not permit receivers to flexibly tailor their reliability semantics. Achieving receiver-driven reliability is cumbersome in the existing "layered" architecture of the network protocol stack where the receiving application has no knowledge of how application-level objects map onto transport level sequence numbers. In this paper, we propose a new data naming scheme that exposes the structure of application data to the transport layer, thereby enhancing the expressibility of an applications' reliability and ordering semantics. We apply this data naming scheme to a reliable multicast protocol framework to achieve receiver-tailored reliability that enhances its scalability. To demonstrate the efficacy of our scheme, we have designed and implemented our scalable naming and announcement protocol (SNAP) in the multimedia application toolkit MASH [17] as a reusable protocol module.

1 Introduction

IP Multicast [6] has enabled an important new class of applications [18, 13, 12, 8, 29] that utilize large-scale multi-point communication over wide-area internetworks as it extends the traditional, best-effort unicast delivery model of the Internet architecture for efficient multi-point packet delivery. In this model, the network delivers a packet from a source to an arbitrary number of receivers by forwarding a copy of that packet along each link of a distribution tree rooted at the source subnet (or, depending on the routing protocol, at a rendezvous point [5] or core router [1]). Senders simply send their packets to a "group address" and receivers express their interest in receiving these packets by joining the corresponding multicast group through a group membership protocol. The collection of senders and receivers exchanging data over a common multicast group is often called a *multicast session* or simply a *session*.

As with unicast IP, the multicast service is not *reliable* — packets might be dropped at any point along the distribution tree. However, many multicast applications like shared whiteboards [12], webcast tools [22], and distributed simulation do not tolerate packet loss. Whiteboard state, for example, is persistent; if a piece of a drawing update is lost, the application cannot leave the drawing in an incomplete state, and instead must recover the missing packet to repair the damaged portion of the drawing. In other words, an application like a group whiteboard requires a *reliable multicast* transport protocol. Although mechanisms for reliable transmission of unicast data are comparatively well-understood and have proven extremely successful (*e.g.*, TCP), making multicast reliable at large scales remains a formidable challenge.

A fundamental challenge in the design of a reliable multicast protocols is the so-called *implosion problem* [21]. If the receivers in a multicast group all react to a packet loss simultaneously by transmitting a control message back to the source, a traffic impulse implodes upon the source, and for very large groups, this implosion effect not only overwhelms the processing capability of that source but severely congests the network.

One way to overcome this implosion problem is to simply omit feedback mechanisms altogether, an approach that has been quite successfully adopted in the multicast session directory service that is implemented by the MBone session directory tool *sdr* [11]. In this approach, the sending application disseminates a dynamic data store as a set of key/value pairs represented as a table. The sender protocol periodically multicasts each entry in the table to some agreed upon multicast group, and receivers interested in the data simply tune in to the multicast group in question and listen to the key/value bindings. Over time, each receiver builds up a copy of the data store and the table is eventually received reliably in its entirety. This style of open-loop communication is often called an *announce/listen* protocol [25] because senders periodically *announce* their data while receivers simply *listen* to the announcements to build up the data store.

Although the announce/listen framework is robust, simple to implement, and easy to understand, its performance is suboptimal both in terms of bandwidth (because data is redundantly transmitted in a continuous fashion) as well as delay (because the sender schedules data transmissions at fixed intervals ignorant of packet losses or receiver interest). Moreover, as the data store becomes large and/or receivers become interested in only small subsets of the overall available data, the approach becomes ever more inefficient. To overcome these limitations, the Scalable Reliable Multicast protocol (SRM) [7] adds a level of indirection to the announce/listen framework. In SRM, a source enlists the announce/listen framework to disseminate "meta-data" that summarizes all of the available data without actually sending it. In turn, receivers use scalable feedback mechanisms (based on "multicast damping", see [7]) triggered by the meta-data to request the delivery or retransmission of any data that is desired. (As an optimization, a source might multicast new data once upon creation to avoid the delay incurred by first announcing its existence to trigger the receiver request.) This approach is often called a "receiver reliable" protocol because the receiver rather than the source implements the reliability requirements. As a result, different receivers can implement selective reliability and tune their reliability requirements to the local user's disposition or application environment.

By allowing a receiver to selectively repair portions of the data stream, we can effectively account for an application's semantics in the design of its network protocol. This approach to protocol design, which derives from the Application Level Framing (ALF) protocol architecture [2], is a boon to protocol performance because the application is optimized for the network and vice versa. For example, in a shared whiteboard application, a receiver might issue retransmission requests for lost data on the current page and ignore missing data for pages that are not in view (or perhaps repair this data at low rate in the background). If we instead used a protocol that was ignorant to application semantics, an application might have to retrieve many megabytes worth of undisplayed data (and incur an unreasonably lengthy delay) before obtaining perhaps the small amount of data that represents the page in view.

While the premise of a receiver-reliable protocol is conceptually straightforward, a problem arises when we attempt to realize the protocol with traditional primitives. If, for instance, we name protocol data units with sequence numbers as is traditionally done in many reliable transport protocols like TCP, we would hide the structure of the underlying application data — e.g., how would a whiteboard know that the data referenced as sequence number 8792 is associated with page 12? In other words, sequence numbers map all application data onto a linear namespace and thereby discard its semantic structure. Yet it is precisely this structure that is crucial to optimizing protocol performance for the application in accordance with ALF. To this end, instead of sequence numbers, we might use a two-dimensional structure, consisting of page numbers and drawing operation identifiers within a page. The announce/listen component of SRM would disseminate the page structure as meta-data, which in turn, would allow a receiver to associate packetized data items for meaningful application structures.

In addition to requiring a rich naming structure, receiver-based reliability mechanisms must scale to very large groups and very large data sets. Not only might the data store itself become large, but the meta-data that describes that data store might also become large. Hence, we must ensure that the meta-data dissemination protocol scales gracefully with the amount of data in the session.

In this paper, we propose a novel naming scheme that simultaneously solves the meta-data scaling problem and provides a rich naming structure that is amenable to ALF. Our naming scheme exploits hierarchy to effectively add a level-of-indirection to the meta-data dissemination protocol. In our approach, hierarchy provides a summarization structure that we utilize to reduce the amount of overhead in the announce/listen component of the meta-data dissemination protocol. Here, senders announce meta-data summaries to the multicast session, which in turn, trigger namespace recovery mechanisms at each receiver in a receiver-directed fashion. In short, senders announce "meta-meta-data" that describes the namespace and receivers use the SRM recovery mechanism to reliably retrieve an arbitrarily large namespace. An elegant consequence of our hierarchical representation is that we can control which pieces of the namespace are disseminated at what rates using announce/listen and which pieces are repaired using receiver-directed recovery. This provides a tunable tradeoff between the consistency among data stores at each receiver and the bandwidth consumed by the protocol's control traffic. In addition to this tunable tradeoff, the namespace hierarchy allows an application to easily impose its own data structure over that hierarchy and thereby infer application-level meaning from the transport-level name, which is not generally possible with a flat sequence number space.

The remainder of this paper describes the manifestation of our hierarchical naming concept in a concrete protocol that we call the Scalable Naming and Announcement Protocol (SNAP). SNAP provides an application-aware data naming scheme for receiver-tailored reliability and generalizes the announce/listen protocol concept in order to handle large namespaces. In the next section, we describe the core hierarchical data naming scheme. Like SRM, SNAP is independent of network-layer addresses and Section 3 describes our approach for global and unique identification of sources. The actual meta-data dissemination protocol is described in detail in Section 4, and we report on its resulting performance in Section 5. In Section 6, we describe our current prototype API to SNAP. Finally, in Sections 7 and 8, we summarize related work, the contributions of the work presented herein, and directions for future research.

2 SNAP: Hierarchical Data Naming

The SNAP naming system uses a hierarchical structure to represent and name application data. Hierarchical naming allows data transmission from different objects in the application data store to proceed independently. The mapping from application-level structures to a SNAP namespace is flexible and under application control and provides a sufficiently rich structure for applications such as shared whiteboards, webcast applications, etc. The new naming scheme also provides a *common syntax* between the the application and transport that enables receiver reliability.

We start with an overview of the main components of the SNAP naming hierarchy — application data units, fragments, containers, namespaces, and name maps, illustrated in Figure 1.

An *Application Data Unit (ADU)* is the smallest unit of data that can be processed independently by an application. Applications hand ADUs to the transport protocol "atomically" to be delivered to the session. Examples of an ADU include a single scan of a JPEG image, or a whiteboard drawing operation [12].

While ADUs can be of arbitrary sizes, a *packet*, which is the unit of network transmission, is limited by the characteristics of the path between sender and receiver(s). The maximum transmission unit, or MTU, of a network determines the amount of data that the transport protocol can transmit at a time without having to fragment it en route to the receiver. While the path MTU in a unicast session is clearly defined as the smallest MTU of any link in the path from a datagram's source to its destination [14], the same is not true in a multicast session because it involves multiple receivers and hence multiple data paths from the sender to receivers. If a link has a smaller MTU than the sender's packet size, the packet must be fragmented and reassembled at the edges of the low MTU segment. Relying on IP to perform link-level fragmentation and reassembly is

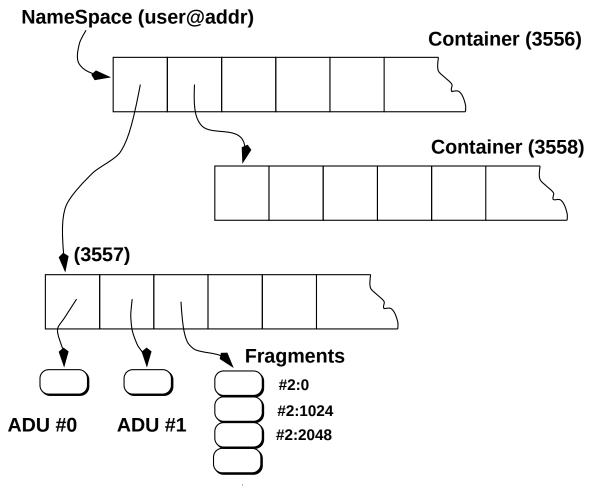

Figure 1: Example illustrating namespace, containers, ADUs, and fragments.

undesirable because fragment loss can result in a (multicast) retransmission of the entire packet. To limit the inefficiency resulting from packet retransmissions, it is important for the transport protocol not to choose a large packet size. Since there is no clear method to determine the optimal transmission size in a multicast session, we use 1024 B, a common transmission size for most link technologies[1].

If an ADU handed to the SRM framework exceeds the packet size, it is fragmented into multiple pieces each of which fits into a packet. A fragment loss does not trigger the retransmission of the whole ADU, but just the lost fragment, and that too only if the application cares for it. Since an application is incapable of processing a fragment without receiving the entire ADU, ADU fragments are reassembled before propagating to the receiving application. While the pure ALF model recommends that the application be "network aware" and only transmit ADUs that don't exceed the MTU, handling fragmentation within the transport protocol shields the details of the network for indifferent applications, and applications that genuinely have large ADUs. Fragments are identified using a starting byte offset and length in bytes. Byte offsets are required to account for the fact that different members may fragment an ADU differently depending on their interface MTU.

A *namespace* is a hierarchical structure onto which an application maps its data. We presently allow each data source to create a distinct namespace in the session. Allowing multiple data sources to modify a shared namespace introduces problems of inconsistency when concurrent and conflicting operations are performed on the same namespace. For this reason, we defer the design of consistency algorithms for our framework as future work, and instead, only allow source-specific namespaces.

Nodes in the namespace hierarchy are called *containers*. Containers refer to ADUs or other containers or both. A containers is also the unit of selective reliability, i.e., by selecting specific containers, a receiver can choose to repair data belonging to the corresponding nodes in the namespace. This, as we have argued, is essential for customizing the receiver's data set. Each container is identified by a *container ID* (CID), that is assigned to it by the source sequentially in the order of creation. The key advantage of hierarchical namespaces is that the application can generate data at any time into a container. In other words, there is no notion of "closing" a container once it has been opened and written into.

Certain reliable multicast applications such as shared whiteboards use a persistent data model, where data lingers in the session for the entire lifetime of the session. In general, this is a convenient model for applications that need to support late joiners[2]. However, data persistence causes difficulties in the CID assignment scheme. Consider a scenario where a source, after generating some data in containers, say 1 through k, crashes and re-joins a session. Assigning container descriptors sequentially starting from 0 causes collisions between the first k new containers generated by the source and data already generated during its earlier incarnation. By randomizing container descriptors, we can greatly reduce the likelihood of such collisions. The following analysis of collision probabilities shows that picking a random initial container descriptor (ICD) and subsequent sequential assignment results in a smaller probability of collision than picking each container descriptor randomly. Figure 2 illustrates these two cases. n is the aver-

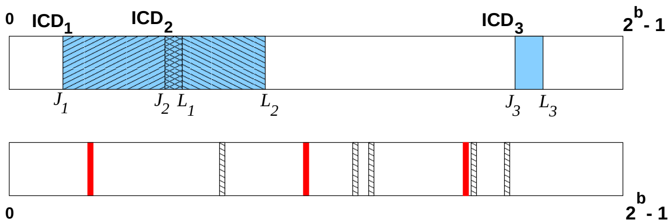

Figure 2: Two randomized descriptor allocation schemes. J_i (Join) and L_i (Leave) represent a source's ICD and final descriptor in incarnation i. In the first case, where container IDs are allocated sequentially, collision occurs when two ranges have even one ID in common. In the second allocation scheme, a collision occurs when an ID is picked that coincides with any one of the n previously allocated container IDs. In this case, we assume that collisions within the same incarnation can be detected and avoided.

age number of containers generated during a single incarnation. b is the number of bits used for container descriptors[3]. Then, $N = 2^b$ is the total number of available containers. Assume that b is sufficiently large so that $n << N$. We evaluate the probability of collisions among container descriptors in different incarnations. The probability that no collisions occur with the randomized ICD is given by $(1 - \frac{2n-1}{N})$. With the scheme that picks a random descriptor for each container, the probability that there are no collisions is $\prod_{i=0}^{n}(1 - \frac{n+i}{N}) \leq (1 - \frac{n}{N})^n \leq (1 - \frac{2n-1}{N})$. Hence, the randomized ICD scheme with subsequent contiguous allocation is better. Another advantage is that selecting contiguous container descriptors allows receivers to detect lost containers in a data-driven manner from gaps in the CID

[1] For links that have smaller transmission sizes than 1024B, the protocol can limit the bandwidth consumed by retransmissions using scoped or localized repair to avoid retransmission floods affecting all group members.

[2] This does not require that all the data be stored in main memory. The application may spool old data to secondary storage.

[3] In our implementation, we use $b = 32$.

sequence.

As an optimization to efficiently handle applications such as *wb*, that generate a large number of (small) ADUs within each page or container, we allow applications to use sequence numbers at the lowest level. ADUs are numbered sequentially within a container. The advantage of this optimization is that data-driven, loss detection is possible in a number of cases. Another advantage is that the extra *ADU sequence number* reduces the state of the index data structure that maps application-level ADU names to containers. An ADU is therefore identified within the transport protocol by the tuple $(CID, ADUseqno)$.

The binding between application names and container IDs is called the *name map.* This is an optional component of the namespace system that is used by the application maps its data onto the SRM namespace by assigning a name to each container. For example, names could be URL strings in the webcast application. However, it is conceivable that a sophisticated application may use other types of names, perhaps with a different profile specification just as different application-specific RTP [26] profiles are possible. Our framework allows a different namemap module to be used with the core naming and announcement protocol.

3 Source Identifiers

A namespace is source-specific and reflects the structure of data generated by that source. In typical reliable multicast applications such as whiteboard, each receiver maintains a copy of every active source's namespace. We need a source identification mechanism that meets the following requirements.

- **Uniqueness.** Two distinct sources must have distinct identifiers within a session to enable receivers to determine the owners of data and maintain source-specific state. A source ID collision occurs when two distinct sources use the same identifier while transmitting data; this is an event that must ideally be avoided, or at the very least, its occurrence must be minimized.

  RTP [26] uses randomly generated 32-bit source identifiers for multicast audio/video transmission and detects collisions and resolves them, but does not perform any repair when one is detected. The only ill-effects of a collision in this case are degraded quality for a brief duration of time, until the collision is resolved. For reliable multicast applications, collisions could potentially cause irrecoverable damage to application data.

- **Time invariance.** Unlike RTP applications like audio and video where application state is ephemeral, reliable multicast applications require that a source identifier for a given source has to be the same at all points in time within the session. In particular, when a source leaves and rejoins the session at a later point in time, it must be possible to re-use the same source ID as in its previous incarnation. The same applies to source crashes and subsequent recovery. Existing schemes like RTP's randomized identifiers violate this requirement.

- **Terminal independence.** In several reliable multicast applications, a source must have the choice of retaining the same identity independent of the terminal or host used to participate in the session. A source (user) must also be able to retain ownership of data created by it in an earlier incarnation in the same session. Furthermore, a single reliable multicast session should be amenable to multiple sources from the same end host or user. These requirements preclude the use of user- and machine-specific identifiers based on a combination of host IP address and user name.

To support time invariance, our source identification scheme allows sources to use arbitrary application-level strings. For example, a user in a whiteboard application could identify herself using a user name and host address, as shown in Figure 1. However, since string names have variable length and incur a high per-packet overhead especially for small ADUs, we hash the name into a fixed-length 64-bit integer derived from the MD-5 function applied on the string. Because the original identifier is an arbitrary user-supplied string that can be retained when moving from one terminal to another, terminal independence is easily achieved as well. Our solution minimizes the probability of collisions and resolves any collisions that might occur using the technique described in the RTP specification [26]. In addition, we ensure that the two important requirements of time invariance and terminal independence are met. The probability of source ID collision in this scheme is the same as in RTP, but that we have achieved the additional goals of time invariance and terminal independence.

4 Discovering the SRM Namespace

SNAP receivers need to discover each source's namespace so that they can issue retransmission requests. To disseminate the namespace structure reliably, we use the basic SRM mechanism to disseminate the namespace tree and session messages to trigger namespace repairs. A special index container `/map` is allocated to contain the namemap bindings. The /map container always appears as the left-most child of the root node in a namespace. Figure 3 illustrates this. Receivers maintain a snapshot of the namespace generated by each source in the session. When a source creates a new namespace node, a message is transmitted from the `/map` container. Transmitting new data from this container indicates to the receivers that a new node in the namespace has

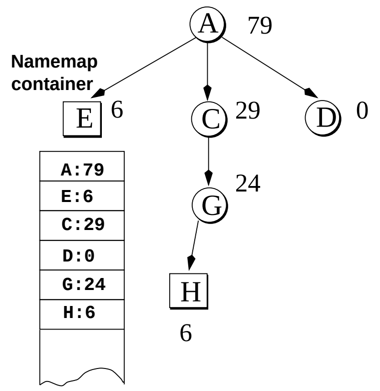

Figure 3: Namemap bindings are treated as regular data by the SRM recovery algorithms, permitting re-use of protocol machinery. It also allows receivers to detect lost containers in a data-driven manner. For simplicity, we have ignored fragmentation in this example.

been created. In response, each receiver updates its snapshot of the namespace tree for the corresponding source.

4.1 Data-driven Loss Detection

We describe the data-driven and control-driven loss recovery mechanisms that enable receivers to learn about undiscovered pieces of the namespace tree. Since data is identified by CID and ADU seqno, losses within the same container can be identified by gaps in the ADU sequence number field. A gap in the CID sequence number indicates a missing container. Since ADUs are allowed to span multiple packets, we also need a mechanism to detect missing fragments. Fragments carry byte boundaries which can be used to detect missing fragments. Each fragment also carries a "more" bit that is used to signal to the receiver(s) that more fragments are to be expected within an ADU. The last fragment of an ADU has the "more" bit set to zero. On receipt of a fragment with "more" set to 0, the receiver checks to see if an entire ADU has been received before propagating it up to the receiving application. When a receiver receives a new ADU before completely receiving the previous ADU, it schedules a repair request for the trailing fragment(s) of the ADU. For example, assume that a receiver has seen the first n bytes of ADU a when ADU $a+1$ starts to arrive. A request is scheduled for $a : n+1, EOA$, where EOA is a special value that indicates the end of an ADU. In response, the rest of the ADU is retransmitted either by the original source, or by any peer member in the session that successfully received the data. The transport protocol does not act independently of the application in repairing losses. Lost data is repaired only if the application so desires. After performing the necessary name translation from container ID to name, SRM queries the application with the container name to determine whether a repair request needs to be scheduled[4]. If so, a request is scheduled according to a randomized timer algorithm similar to the basic SRM loss recovery scheme [7]; if not, the loss is ignored. In response to a repair request, any peer member in the session is allowed to respond with data. Data is buffered at the application at both the source and receivers, and the transport protocol only maintains the structure of the namespace. Upon receipt of a repair request, the transport protocol queries the application for the requested data.

4.2 Announcement-driven Loss Detection

While data-driven loss detection mechanisms handle several cases of losses, they do not detect "tail losses" within a container, or tail losses of missing containers. Late joiners to the session also need a special mechanism to discover interesting portions of the namespace hierarchy. In *wb*, a new receiver queries the source using for a list of all its pages by transmitting a "page vector request". The source in response replies with a list of all its pages. This scheme, where the source transmits a vector of available pages does not scale well to long-lived sessions with a long history. Imposing an application-defined hierarchy on the namespace permits a receiver to navigate the namespace and selectively fetch only branches of the tree it is interested in. In order to transmit a concise representation of the namespace hierarchy, we summarize the state of its namespace using the signature function.

[4]Alternatively, an application declares its preferences in a profile at startup time.

Source-based announcements fall in the general category of announce-listen protocols where sources periodically announce their data and receivers listen to these announcements to reconstruct the data. Announce-listen protocols are conceptually simple and do not suffer from feedback implosion problems when used over IP multicast. The session announcement protocol (SAP) [10] is an "open loop" reliable protocol that multicasts data periodically. A receiver simply waits for source announcements to receive all data. In the basic SRM algorithm with linear sequence numbers, the source periodically transmits "meta-data" indicating the last ADU sequence number transmitted so far. SRM receivers use this information to NACK data. In SNAP, each source announces its signature which is "meta-meta-data". Receivers NACK meta-data, to repair pieces of the namespace, and data, just as in SRM. We use the SRM slotting and damping algorithms [7] to reduce the amount of redundant SNAP traffic.

4.2.1 Signatures

The exact structure of the namespace tree can be conveyed completely and without any loss of information by providing an in-order and pre-order traversal of the hierarchy. For long-lived sessions with large and persistent namespaces, this is too large and expensive to disseminate to the entire multicast group. In order to limit the bandwidth consumed by state announcement messages, we need a more compact representation, or "fingerprint", of the namespace hierarchy. We call this compact representation of the namespace hierarchy its signature and define it as follows. The signature serves to inform all receivers about the current structure of a container and all its descendants.

The signature, s, is a function from a container into the set of 64-bit integers. The signature of a node is defined recursively as the last ADU sequence number contained in it. If a container C is a leaf-level container that contains only ADUs, its signature is simply the right edge of the container. This is represented as the highest sequence number in its ADU sequence, and the highest byte of the last ADU. The highest ADU sequence number helps receivers detect trailing losses within a container. The highest byte of the last ADU helps receivers determine missing fragments.

For an internal container that points to other containers, the signature is computed as a hash function over its own right edge and the signatures of its children containers $C_1, C_2, \ldots, C_n$. An example of such a hash function h is MD-5 [24]. MD-5 has the special property that no known computationally feasible technique exists to produce two distinct messages having the same hash value. Such a hash function gives us signatures that are unique with high probability. The signature of a source's namespace can be viewed as a unique (with high probability) fixed length representation of all its data. However, we note that the signature is a one-way function and cannot be reversed to reconstruct the tree that generated it.

$$s(C) = \begin{cases} \texttt{right\_edge}, & \text{if } D \text{ is a leaf-level container;} \\ h(\texttt{right\_edge}, s(C_1), \ldots, s(C_k)), & \text{otherwise} \end{cases}$$

where `right_edge` of a container C is defined as the tuple (max ADU seqno, last byte offset in last ADU) within C.

Note that it is necessary to transmit the highest byte offset in addition to highest ADU for the following reason. A source announcement may be transmitted after a source has begun transmission of an ADU, but has not completely transmitted it. This may occur because fragments of the

ADU are streamed through a token bucket that has a maximum burst size. If we allow source announcements to be sent only on complete ADU boundaries, we can use the highest ADU sequence number alone to represent the right edge.

4.2.2 Source Announcements

As explained earlier, each source announces "meta-meta-data" and "meta-data" to the session. Periodically, each source multicasts a session announcement message containing the signature of the root and the signatures of at most k containers. The period between announcements and the amount of data to send in each announcement are scaled to constrain control traffic and the exact rate is determined by external policy. For example, if we constrain sources to allocate no more than 5% of their session data bandwidth on announcements we restrict the size and/or rate of announcement messages. From the announcement rate and the network MTU, we can compute a maximum announcement packet size, which in turn determines k. A higher frequency of session announcements results in lower loss detection latency at the receivers. We now outline a heuristic that picks the k "best" containers whose meta-data is transmitted along with the root signature. Since we cannot transmit meta information about all the containers in a source's namespace, we give priority to containers that have not been announced for the longest period of time. We use a randomized scheme similar to lottery scheduling [31]. The analysis of this algorithm is akin to the coupon collectors problem [9]. The expected time of the algorithm is proportional to $(n \log n + n \log^2 n + |C|)$, where n is the target number of containers to be selected. The total number of containers is C. The details of the algorithm and its analysis are described in [23].

4.2.3 Receiver Processing

We now proceed to discuss the receivers' processing upon receiving an announcement packet from source S. Each receiver uses the signature to determine if it has a correct snapshot of S's namespace. If the local copy of S's signature does not match the one carried in the announcement, the receiver transmits a request to repair the root node of the namespace. All requests are multicast to the session and use a randomized backoff scheme to reduce the number of duplicates when losses are correlated.

If the receiver is interested in the entire namespace, it performs a systematic *recursive descent* procedure to explore the namespace, detect mismatched nodes, and repair them. However, the recursive descent algorithm is the last resort and is not invoked in the common case, where applying the meta-data updates transmitted with each signature is sufficient to bring the receiver up-to-date.

Below, we describe the receiver's processing for the different loss cases.

1. **No Loss.** On receipt of an announcement message from source S, a receiver R first compares the root level signature sent by S with the corresponding local version. If the two signatures match, R assumes that it has a current snapshot of S's namespace. Note that there is a vanishingly small probability that two different trees will generate the same MD-5 root signature. However, the inconsistency is only transient as a new signature is periodically transmitted by the source as more data is transmitted and the namespace tree develops.

2. **Missing ADUs.** If the local signature computed on S's namespace does not match the one carried in the announcement, the receiver invokes loss recovery for the namespace meta-data by issuing a a *namespace repair request*. The purpose of the namespace repair request is to query the source (or any other eligible member in the SRM session), to repair the state of a mismatched node. In response, the source, or any member with an up-to-date copy of the source's tree, multicasts a namespace repair packet for a requested node with a list of all the children and their signatures. The receiver descends the namespace tree recursively in this manner to locate missing branches. When a leaf-level container with only ADUs is reached, the container's signature gives the last ADU generated, and the receiver is now in a position to schedule a repair request for the missing ADUs. This is illustrated in Figure 4.

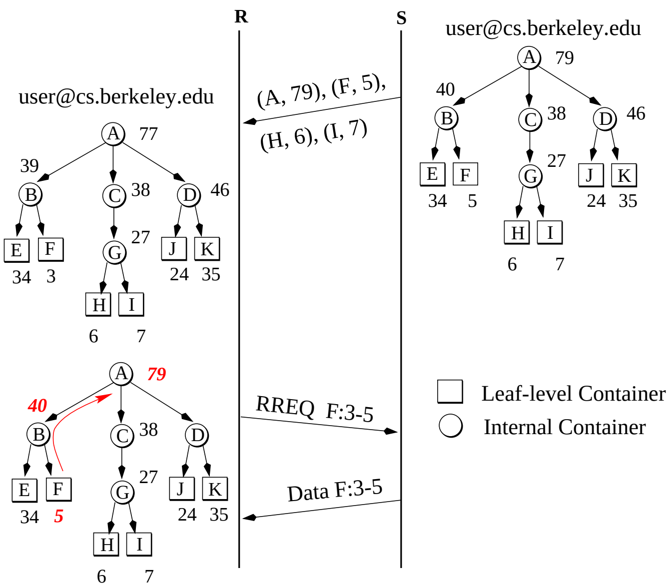

Figure 4: Recovering from tail losses using SNAP announcements. A-K are CIDs and the number beside each container is its signature. For leaf-level containers, the signature is also the right edge of the container, and provides sufficient information to schedule repair requests. This figure shows a receiver recovering from a tail loss in container F.

3. **Missing namespace nodes.** A receiver may lose entire namespace nodes and the corresponding meta-data information from the /map container, perhaps due to persistent congestion. Missing namespace nodes can be detected by gaps in the sequence of map entries and recovered from. Tail losses from the /map container are detected when a receiver receives a session announcement containing the map container's signature. This is illustrated in Figure 5. If a receiver receives data in a container whose name mapping it does not yet know, it generates a repair request for the missing mapping. Rather than discard ADUs whose names are unknown at the receiver, we buffer the data at the SRM layer until the corresponding name of the ADU

is available to propagate up to the application[5]. Mapping information appears as regular data to the SRM protocol and we leverage the SRM request/repair machinery to recover lost portions of the namespace.

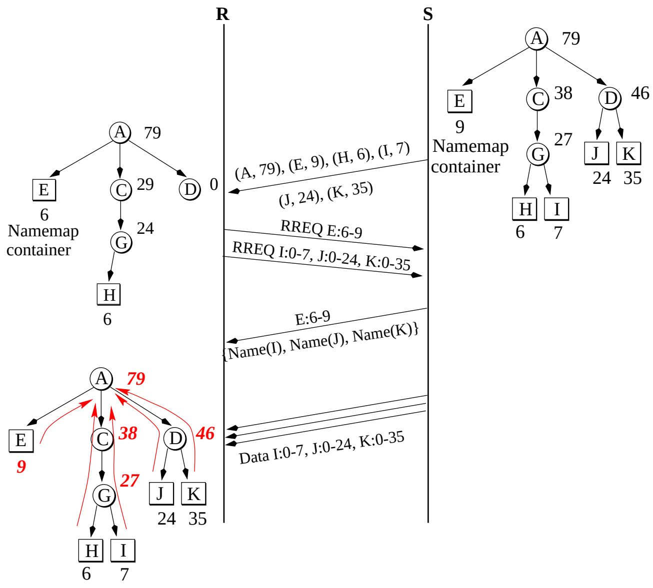

Figure 5: Receiver R recovers from missing containers that went undetected because of tail losses in the namemap container `/map`. Once again, the notation is the same as in Figure 4. First, R recovers missing bindings from the map. Missing data from the containers I, J, and K is then recovered.

4. **Selective reliability.** Because SNAP uses hierarchical naming, receivers can perform selective reliability by deciding when and when not to generate repair requests for nodes in the namespace tree. This is especially beneficial when a new user joins a long-lived *wb* session to review a specific page, say the agenda page from the a weekly meeting in January 1997. By organizing its data hierarchically, the source allows receivers to tailor its requests to receive only the data it requires. Figure 6 shows an example where R does not care for data below container C. Nodes whose contents are uninteresting to a receiver are termed *don't care* nodes. In order to prevent spurious signature mismatches at higher levels of the namespace tree, the receiver maintains the most up-to-date signature for the root of the *don't care* subtree.

5 SNAP: Performance Evaluation

To evaluate the performance of SNAP within the SRM framework, we conducted a simulation study using the network simulator *ns* [16]. We used a multicast group with one source and up to 55 receivers[6]. Background traffic in the simulations was generated using TCP connections, which induced packet losses. The topology used was a tree of degree 4,

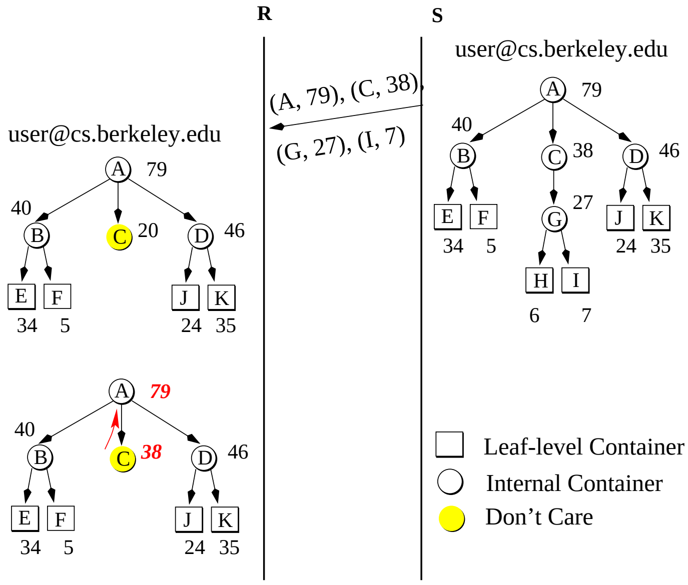

Figure 6: Selective recovery using SNAP. If the receiving application at R does not care for data belonging to a certain subtree, it simply updates the signature of the container at the root of such a subtree, and refrains from scheduling a repair request for portions of the namespace below those branches. The correct signature at the root of the *don't care* subtree allow the R to compute the correct signature at the next higher levels and eventually the signature at the namespace root. This prevents the receiver from generating spurious namespace repair requests.

with the source at the root. A constant bit-rate data source was used with a randomly generated namespace. We looked at two metrics: (i) convergence time, or the latency to recover from a tail loss, and (ii) session size scalability which is measured by the bandwidth used by SNAP control traffic as the session size is increased. In order to measure the worst case convergence time for a late joiner, we used the recursive descent scheme, without any heuristics or selective reliability.

We define the convergence time as the elapsed time from the instant at which a packet is dropped in the network (as a result of overflow at a router queue) to the instant that a receiver receives the data packet. We measure the worst case convergence time in our simulations, i.e., time for the last receiver to recover from the loss. The convergence time has three components, the average waiting time for the first update from the source, the time taken to discover the location of the loss using SNAP, and the time to recover from a loss. Figure 7 shows the convergence behavior of the state update protocol with varying periodicity of updates. As expected, the convergence time improves with more frequent SNAP announcements. However, there is a tradeoff between the amount of bandwidth consumed by SNAP control messages, and the recovery latency of the protocol in discovering and reacting to losses. This convergence time is significantly reduced when applications exercise selective reliability to selectively repair portions of the namespace. Our simulations also used containers with exactly one ADU to stress the control-driven repair mechanisms. Therefore, the observed latencies represent the worst-case scenario. When used in applications that continuously generate data in each con-

[5] Alternatively, unnamed data may be passed up to the application and the application could be notified when the name is learned.

[6] Because of the prolific memory requirements of *ns* multicast simulations, we were unable to experiment with larger groups.

tainer, data-driven recovery is likely to repair most losses with lower latency.

The chief concern with multicasting control messages such as repair requests, state announcements and namespace repair requests/replies is the amount of bandwidth consumed in very large session sizes. In order to evaluate the effectiveness of suppression resulting from the "slotting and damping" mechanism in SNAP, we measured the number of copies of each control message multicast to the group. Figure 8 shows this behavior as the group size is scaled up to 50 nodes. For SNAP updates, the average number of copies per message was about 3, and remained approximately constant with increasing session sizes. On the average, about 2 copies of a request message are transmitted to the session. This remains roughly constant with large group sizes. Schemes such as local recovery that are generally applicable to SRM can also be applied to this case to improve scaling behavior and eliminate the extra duplicates.

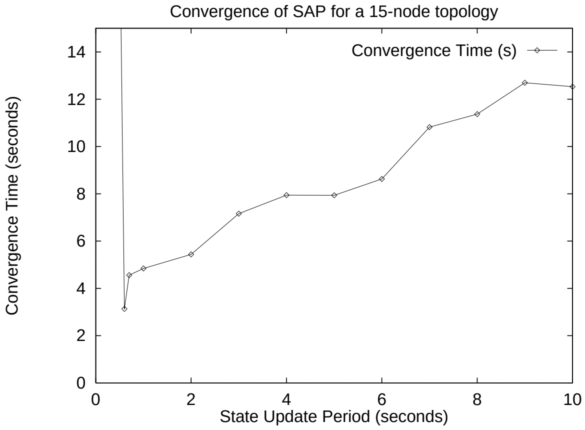

Figure 7: Convergence of SAP with decreasing frequency of updates

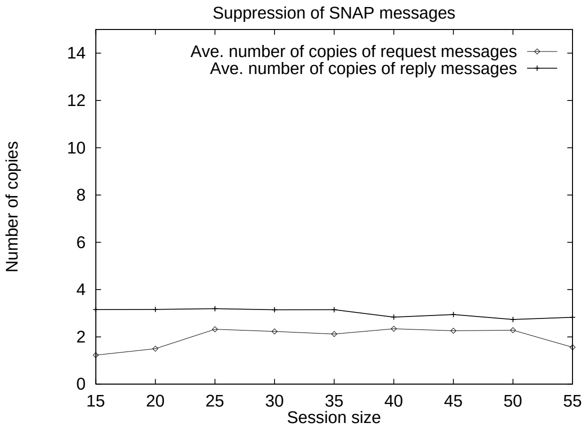

Figure 8: Effectiveness of SRM-style suppression in SNAP. On the average, about 2 identical copies of a request and 3 identical copies of a response are multicast.

6 Implementation

We have implemented a prototype of the naming scheme and SNAP in the MASH [17] toolkit. The MASH platform is a scripting-based programming environment for networked multimedia applications. It provides composable basic building blocks such as network objects, codecs, widgets, and an event-driven programming model. We implemented SNAP as a library of classes in the toolkit. SNAP runs at user-level in the same address space as the application. Since the application and protocol share an address space, upcalls [2] by the transport protocol to query the application are implemented as function calls. The namemap that implements the mapping from names to transport level identifiers is a separate module as shown in Figure 9. The software

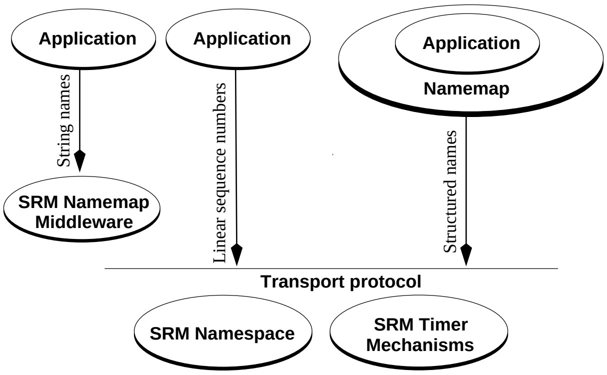

Figure 9: Namemap is a middleware component that implements the mapping from structured names used by the application to transport level identifiers. SRM timer mechanisms and SNAP are core transport layer functions. The transport layer exposes a low-level API to the application or middleware component that enables selective reliability.

architecture of SNAP is shown in Figure 9. The protocol framework has two interfaces: one with an event system that allows the library to register events and invokes the specified handlers when these occur. In our implementation, the Tcl [19] event loop provides these functions, but it is possible to install an alternate event handling system provided it supports an equivalent API.

The other important interface to SNAP is the application-transport interface. Our basic framework has the following API. The direction (upcall/downcall) of each function call in this API is also indicated.

- ↓ `srm_alloc_container parent`
  `srm_alloc_container` allocates a container as a child contained within `parent`, which is the ID of the parent container. Each container ID is 32 bits wide, allowing at most 4 billion containers per source within each application. This function returns the container ID of the new allocated container to the caller.

- ↓ `srm_start`
  `srm_start` signals to the SNAP layer that there is data ready for transmission. Since the transmission of data is clocked by the underlying rate regulating mechanism, the application merely registers its interest in transmitting data. The application is immediately called back with a request for data. (See `srm_get_data` below.) SNAP sets a timer in its rate control leaky bucket to accumulate enough tokens for the ADU just transmitted. Subsequent ADUs that arrive during this period are maintained in an application level ADU queue.

↑ `srm_recv_data cid seqno data`
`srm_recv_data` is a notification delivered to the application by the transport protocol when a complete ADU has been received either resulting from the original transmission or from a retransmission. Retransmissions are indistinguishable from original data and are subjected to the same rate regulations. If an ADU has to be fragmented at the sender, it is reassembled before delivery up to the application.

↑ `srm_get_data cid seqno`
On receiving a repair request, the transport protocol requests the application for the ADU corresponding to container `cid`, sequence number `seqno`. This function is invoked when the token bucket internal to the SNAP library has accumulated enough tokens to transmit an ADU that was registered using a prior `srm_request_to_send`. This is also used by the SNAP library when responding to a repair request generated by a session member. This notification to the application is necessary since buffering is the responsibility of the application.

↑ `srm_recover cid seqno`
Receiver-tailored retransmission is achieved via this function call. `srm_schedule_request` explicitly queries the application to check if a repair request for the lost container needs to be scheduled by SNAP on behalf of the application.

7 Related Work

There has been much recent and ongoing work in reliable multicast protocols and architectures. Most of the existing work focuses on algorithms that ignore application interests. While the independence of transport-level reliability and congestion control algorithms from the application have obvious value, augmenting these with a naming scheme that enables receiver-reliability enhances the scaling properties of the protocol.

A recent Internet draft [3] on RMFP proposes a *two-layered* data stream with embedded objects and sequence offsets within objects. This work is orthogonal and complementary to our scalable session announcement protocol. Their work assumes properties of the data stream such as bounded liveness of data, i.e., the application performs a *close* signaling the end of the object. This is a restriction for applications such as *wb*, where the user may revisit an old page, and add to its existing contents. In addition their approach uses a special `BIND` message that binds application level names to object IDs. This introduces extra message types and restricts applications to use ASCII string names. Separating the namemap module from the core SRM namespace mechanisms provides the application with the flexibility to use any naming convention of choice.

Other work on a reliable multicast framework (RMF) [4] proposes *universal receivers*, a concept that allows several protocols to be defined using a set of common packet formats. While RMF does not disallow specialized naming schemes built above the "transport-level" framework, the authors do not address issues of structuring and naming application data, and discovering the structure of namespaces at the receivers.

8 Summary and Future Work

In this paper, we have presented a generalized framework for reliable multicast that supports diverse reliability semantics. The two key components of the framework are (i) a hierarchical naming system, and (ii) a scalable session announcement protocol. The result is a receiver-driven reliability protocol where receivers tailor their reliability requirements. Based on simulation studies of convergence time and message complexity of the protocol, we demonstrate that the protocol's bandwidth consumption scales well to large group sizes. We have designed and implemented this Scalable Naming and Announcement Protocol (SNAP) as part of the MASH [17] toolkit. We discuss some avenues for future work in this area.

Global recovery in SRM results in request and reply floods transmitted to the entire group, even for losses that are localized. Several solutions have been proposed for local recovery in the literature [15, 7, 20, 27]. Integrating SNAP with a local recovery scheme could provide greater scalability. Finally, the deployment of IP multicast in the Internet has been impeded to some extent by the absence of a multicast congestion control algorithm. This problem has recently received significant attention in the research community. We hope to use our framework as a vehicle to design, test, and deploy different congestion control schemes on the Internet.

A key ingredient to the success of the general framework approach to reliable multicast is experience drawn from design and implementation of a variety of applications. We plan to port a number of existing applications including the MASH mediaboard [28], and mashcast [32] to effectively demonstrate the power of application-level framing.

9 Acknowledgments

We would like to thank Hari Balakrishnan, Teck-Lee Tung and the anonymous reviewers for their useful feedback. This research was supported by DARPA contract N66001-96-C-8508, by the State of California under the MICRO program, and by NSF Contract CDA 94-01156.

References

[1] Ballardie, T., Francis, P., and Crowcroft, J. Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing. In *Proceedings of SIGCOMM '93* (San Francisco, CA, Sept. 1993), ACM, pp. 85–95.

[2] Clark, D. D., and Tennenhouse, D. L. Architectural Considerations for a New Generation of Protocols. In *Proceedings of SIGCOMM '90* (Philadelphia, PA, Sept. 1990), ACM.

[3] Crowcroft, J., Wang, Z., Ghosh, A., and Diot, C. RMFP: A Reliable Multicast Framing Protocol, Mar. 1997. Internet Draft (RFC pending).

[4] DeCleene, B., Bhattacharaya, S., Friedman, T., Keaton, M., Kurose, J., Rubenstein, D., and Towsley, D. Reliable Multicast Framework(RMF): A White Paper, Mar. 1997.

[5] Deering, S., Estrin, D., Farinacci, D., and Jacobson, V. An Architecture for Wide-Area Multicast Routing. In *Proceedings of SIGCOMM '94* (University College London, London, U.K., Sept. 1994), ACM.

[6] Deering, S. E. *Multicast Routing in a Datagram Internetwork.* PhD thesis, Stanford University, Dec. 1991.

[7] Floyd, S., Jacobson, V., McCanne, S., Liu, C.-G., and Zhang, L. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proceedings of SIGCOMM '95* (Boston, MA, Sept. 1995), ACM.

[8] Frederick, R. *Network Video (nv).* Xerox Palo Alto Research Center. ftp://ftp.parc.xerox.com/net-research.

[9] Graham, R. L., Knuth, D. E., and Patashnik, O. *Concrete Mathematics*, second ed. Addison-Wesley, 1994.

[10] Handley, M. *SAP: Session Announcement Protocol.* Internet Draft, Nov 19, 1996.

[11] Jacobson, V. *Session Directory.* Lawrence Berkeley Laboratory. ftp://ftp.ee.lbl.gov/conferencing/sd.

[12] Jacobson, V., and McCanne, S. *LBL Whiteboard.* Lawrence Berkeley Laboratory. ftp://ftp.ee.lbl.gov/conferencing/wb.

[13] Jacobson, V., and McCanne, S. *Visual Audio Tool.* Lawrence Berkeley Laboratory. ftp://ftp.ee.lbl.gov/conferencing/vat.

[14] Kent, J. C., and Mogul, J. C. Fragmentation considered harmful. In *Proc. ACM SIGCOMM* (October 1987).

[15] Lin, J. C., and Paul, S. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings IEEE Infocom '96* (San Francisco, CA, Mar. 1996), pp. 1414–1424.

[16] McCanne, S., et al. UCB/LBNL/VINT Network Simulator - ns (version 2). http://www-mash.cs.berkeley.edu/ns/.

[17] McCanne, S., et al. Towards a Common Infrastructure for Multimedia-Networking Middleware. In *Proceedings of the Seventh International Workshop on Network and OS Support for Digital Audio and Video* (St. Louis, CA, May 1997), ACM.

[18] McCanne, S., and Jacobson, V. *vic: video conference.* Lawrence Berkeley Laboratory and University of California, Berkeley. ftp://ftp.ee.lbl.gov/conferencing/vic.

[19] Ousterhout, J. K. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[20] Papadopoulos, C., Parulkar, G., and Varghese, G. An Error Control Scheme for Large-Scale Multicast Applications," IEEE. In *Proceedings IEEE Infocom '98* (San Francisco, CA, 1998).

[21] Pingali, S., Towsley, D., and Kurose, J. F. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *Proceedings of SIGMETRICS '94* (Santa Clara, CA, May 1994).

[22] PointCast Inc. *PointCast Home Page.* http://www.pointcast.com.

[23] Raman, S. Design and Analysis of a Framework for Reliable Multicast. UCB CS Masters Thesis, May 1998.

[24] Rivest, R. *The MD5 Message-Digest Algorithm.* MIT Laboratory for Computer Science and RSA Data Security, Inc., 1992. RFC-1321.

[25] Schooler, E. M. A multicast user directory service for synchronous rendezvous. Computer science department, California Institute of Technology, Sept. 1996.

[26] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications.* Internet Engineering Task Force, Audio-Video Transport Working Group, Nov. 1991. Internet Draft expires 3/1/96.

[27] Speakman, T., Farinacci, D., Lin, S., and Tweedly, A. Pretty Good Multicast (PGM) Transport Protocol Specification, Jan. 1998. Internet Draft (RFC pending).

[28] Tung, T.-L. MediaBoard: A Distributed, Shared Collaboration Application. UCB CS Masters Thesis, Feb. 1998.

[29] Turletti, T. *INRIA Video Conferencing System (ivs).* Institut National de Recherche en Informatique et an Automatique. http://www.inria.fr/rodeo/ivs.html.

[30] van Renesse, R., Birman, K. P., and Maffeis, S. Horus: A Flexible Group Communication System. *Communications of the ACM* (1996).

[31] Waldspurger, C. A., and Weihl, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *First Symposium on Operating Systems Design and Implementation (OSDI)* (1995), USENIX Association, pp. 1–11.

[32] Wright, K. MASHCast: Applying SRM Middleware to Webcast. Presentation at the Winter '98 MASH Retreat.