A Reliable Multicast Webcast Protocol for Multimedia Collaboration and Caching *

L. Kristin Wright
University of Utah
kwright@cs.utah.edu

Steven McCanne†
University of California, Berkeley
mccanne@cs.berkeley.edu

Jay Lepreau
University of Utah
lepreau@cs.utah.edu

Abstract

Large-scale, multi-point, multimedia conferencing applications designed to facilitate long-distance collaboration are enjoying growing popularity. Usually composed of real-time audio, video and shared-drawing applications, these collaborative environments help render the geographical location of collaborators irrelevant. To complement these existing collaborative applications, it would be useful to have the ability to distribute documents synchronously over the World Wide Web (WWW). One model for synchronized information dissemination within the Web is *webcasting* in which data are simultaneously distributed to multiple destinations. The WWW's traditional unicast client/server communication model suffers, however, when applied to webcasting; solutions which require many clients to simultaneously fetch data from the origin server using the client/server model will likely cause server and link overload.

A number of webcasting solutions have been proposed. Many have limited scalability because they are based on unicast while others use multicast for more scalable data delivery but require server modification or have rigid architectures. We believe that successful webcasting solutions will provide scalable, reliable delivery yet still be compatible with the existing Web infrastructure.

In this paper we describe a webcast design that improves upon previous designs by leveraging application level framing (ALF) design methodology. We build upon the Scalable Reliable Multicast (SRM) framework, which is based upon ALF, to create a custom protocol to meet webcast's scalability needs. We employ the protocol in an architecture consisting of two reusable components: a webcache component and a browser control component. We have implemented our design using the MASH multimedia application toolkit and a SRM protocol module called *libsrm*. We present the results of a simple performance evaluation and report on lessons learned while using MASH and *libsrm*.

1 Introduction

Online multimedia environments designed to facilitate long-distance collaboration are enjoying growing popularity: as modern information technology becomes ubiquitous, telecommuting has become increasingly popular; some large universities, faced with growing student populations, are exploring distance-learning as an alternative to high-cost bricks-and-mortar expansion; technical conferences are sometimes broadcast as a convenience to those who cannot attend physically. Usually composed of real-time audio, video, and shared drawing applications, collaborative environments are widely deployed [15, 10, 7, 13, 20, 9, 8, 21].

In addition to video, audio, and whiteboard applications, it would be useful to have the ability to distribute documents over the World Wide Web (WWW). For example, a speaker might display slides directly on remote listeners' desktop web browsers. Or perhaps a group of geographically remote colleagues would like to explore a common destination online and would like their individual web browsers sychronized such that the browsers display the same pages simultaneously. There are a number of slightly differing models, each requiring the synchonized distribution of web documents and their embedded objects to multiple sites. A common name for this type of collaborative session is *webcasting* [1, 4, 16, 12].

A simple webcast design might require each remote node, or session *participant*, to individually fetch the required document from the *origin server*, the web server on which the document resides. In this scenario, one participant might send a Uniform Resource Locator (URL), the unique name used to locate a document within the WWW, to the other participants. Upon receipt of the URL, each participant would fetch the corresponding page from the server. The problem with this simple solution is that the aggregate requests, or *server hits*, can lead to *implosion*—a pathology where a resource is unable to keep up with an incoming stream of messages [17]. Servers are constrained by built-in resource limitations in current operating systems such as the number of open sockets, buffer size limits, etc.. As session memberships increase to thousands, protocols which allow even a small percentage of duplicate hits could easily overload a server. Even if the server was able to keep up with a large number of incoming requests, the outgoing replies would likely cause network congestion.

1.1 Webcasting with multicast

To achieve scalability, neither the network traffic nor the number of server hits generated by a collaborative session can be allowed to significantly increase with session membership. A more efficient approach than the simple design described in the introduction would be to use IP multicast to achieve greater scalability. In [5], Deering and Cheriton describe network-layer multicast, a forwarding service that offers more efficient multidestination delivery of packets than either unicast or broadcast. Rather than send out

*This research was supported by DARPA contract N66001-96-C-8508 and by the State of California under the MICRO program.

†Now at FastForward Networks, Inc. (*mccanne@ffnet.com*).

one packet per participant, as in unicast, or replicate a single packet across every link in the network, as in broadcast, multicast protocols send packets across only those links that lead to a receiver in the multicast group. This reduces sender overhead, network traffic, and can reduce end-to-end latency.

One way to leverage multicast to improve scalability would be to modify the browser such that it multicasts data in response to requests from collaborative program participants. This is the general approach taken by [4] and [1]. While this is a more scalable approach than the naive solution suggested in the introduction, modifying the web server limits the usefulness of the resulting webcast application because server modification is only feasible when the server is within the local administrative domain. Requiring web server modifications also introduces a deployment barrier.

A second multicast approach is to build a client-side web proxy to intercept participant's browser requests, fetch the data from the origin server, and then multicast the data to session participants. This approach capitalizes on multicast's scalability without requiring server modification. [12] and [16] both use this approach. They further improve scalability by caching data locally so that any browser requests that can be satisfied using previously cached data need not be forwarded to the server. This approach capitalizes on multicast's scalability without requiring server modification but fails to prevent server implosion common in collaborative situations, e.g., when several students listening to a remote lecture simultaneously fetch a web document referenced by the lecturer.

1.2 Our approach

Our approach borrows quite freely from [12] and [16], but we improve upon existing work by leveraging the Scalable Reliable Multicast (SRM) framework [6] to build a custom protocol to address the server implosion problem mentioned above. SRM embodies a design principle called Application Level Framing (ALF) [3] that applications can leverage to build a protocol that will suit their unique needs. Although unicast protocols generally require ordered, reliable delivery, multicast applications can have widely varying delivery requirements. For example, different applications may require different flow control, rate control or error recovery mechanisms. The insight behind ALF is simply that applications know their requirements better than generic communication protocols. The protocols, then, should implement only the minimum necessary to provide the promised network service but allow applications to express their own semantics in the network protocol. Accordingly, SRM provides reliable multicast service but allows the application to determine its own semantics. We use SRM's flexibility to create a custom protocol that maximizes scalability. We present that protocol herein.

In addition to our custom protocol, we present our webcasting architecture. The architecture is comprised of two components, 1) a *multicast web cache* component maintained using SRM and 2) a webcasting control component. These components can be cleanly decoupled for reuse. The control component multicasts URLs to peer control components and instructs the browser as to what URL it should display. The subsequent browser request is intercepted by the caching component which either satisfies the request from within its multicast web cache or fetches the data from the origin server and multicasts the data to peer caching components. We have implemented our webcast design in an application called MASHCast using the multimedia application toolkit MASH [14] and a SRM protocol module called *libsrm* which features the SNAP scalable data naming protocol [18].

The remainder of this paper is structured as follows. We first describe our architecture, then SRM's basic reliability mechanisms and our extensions to them. We next present our implementation and initial performance characteristics. Lastly, we report on lessons learned during implementation and the future work those lessons motivate. This paper's contributions are that (i) we bridge a gap in the set of online, collaborative tools by providing a scalable webcast application, (ii) we exploit SRM's flexibility to create a new protocol that maximizes webcast scalability by minimizing origin requests and network traffic, (iii) we embody our protocol in a novel webcast architecture and implementation called MASHCast that features several reusable components, browser independence, and compatibility with existing Web infrastructure, (iv) we present initial performance characteristics, and (v) we report on lessons learned while using the MASH multimedia toolkit and *libsrm*.

2 Dual component architecture

Our webcast architecture consists of a control component called the Casting Director and a multicast caching component called the Web Cache. The Web Cache consists of two subcomponents: a Web Server, which receives requests from the browser, and the Multicache, the multicast web page cache maintained using SRM. Figure 1 shows the components that comprise our architecture.

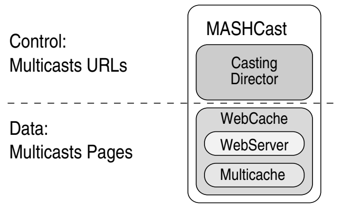

Figure 1: The MASHCast dual-component architecture features decomposable control and data modules.

The Casting Director provides control over a webcast session by synchronously multicasting URLs to peer Casting Directors and forwarding received URLs to the browsers. A sender webcasts a page by specifying the page's URL to the Casting Director via a built-in GUI.[1]

The Web Server subcomponent of the Web Cache provides an interface through which browser requests are received and satisfied. The Web Server forwards browser requests to the Multicache subcomponent of the Web Cache. If the local Multicache cannot satisfy the request from its own cache, it requests the data from peer Multicaches. If no Multicache in the session has the data, a single Multicache is selected in a distributed fashion to fetch the data from the origin server and then multicast the data to the group. To differentiate between the two types of requests, we call a request to the Multicache a *local request* and a request to the origin server an *origin request*. Selection

[1] URLs can actually be passed to the Casting Directors using any mechanism which multicasts the URL to the Casting Directors' multicast address. The GUI is a convenience for the user, not a necessity of the design.

and other aspects of our custom protocol employed by the Multicache are discussed in detail in section 4.

While we believe that our architecture's main contribution comes when used as a collaborative webcast application such as MASHCast, our architecture is novel because, in addition to being compatible with existing WWW infrastructure and addressing all sources of server implosion, its components can be cleanly decoupled for reuse. For example, though the Web Cache is not a suitable replacement for existing hierarchical web caching proxies, it can be used apart from the Casting Director as a passive multicast web cache in groups where a caching proxy server is not already in place. Members of a common research group will likely be browsing the same research related web pages. The group members could experience faster browser response times by starting a Web Server process on each machine in the group and specifying the Web Server as their web browsers' HTTP proxy. When a user requests a URL, the request would be intercepted by the local Web Server process.

The Multicache subcomponent can be used alone to cache any type of data in a distributed, scalable, reliable manner. Within MASHCast, the Multicache stores web pages and their embedded objects, but the data type is not specified in Multicache's design.

The Casting Director could be used without the Web Cache component to provide a simple webcasting service. Its use alone lacks scalability, however, because without the Web Cache's custom protocol to suppress duplicate origin requests, each session participant generates a request for each page webcasted leading to implosion.

When used together in MASHCast, the Casting Director and the Web Cache run in different processes on the same machine as the web browser. They use separate multicast addresses to communicate with peer components in the same MASHCast session across the IP Multicast Backbone (MBONE) [2]. For example, all Casting Directors in the same session might use the address 224.4.4.5 to communicate with each other and all Web Caches in the same session might use 224.4.4.10 to communicate with each other. The MASHCast application sits between the web browser and the network. MASHCast's custom protocol minimizes interactions with web servers but, when contact is necessary, data is fetched from the origin server via unicast HTTP. To participate in a MASHCast session, the user need only start the MASHCast application and a web browser on the same machine (we will call this machine the *local* machine) and fetch a special MASHCast bootstrap page into the browser. MASHCast requires configuration of a MASH browser plugin, but once this plugin is configured, no web server modification or browser proxy specification is required.

Once MASHCast is started, a sender webcasts a page by specifiying the page's URL to the Casting Director. Peer Casting Directors listen for URLs and, upon receipt, prepend the request with the URL of the local instance of MASHCast. The resulting *mangled* URL is then forwarded to the Remote Controller. Figure 2 shows how the Casting Director forwards URLs to the browser and other Casting Directors. In the figure, the URL `www.cs.berkeley.edu` is multicast among the Casting Directors. Upon receipt, the Casting Directors mangle the URL by prepending the local MASHCast Web Server address (`localhost:3123/`) onto the URL. After the URL is mangled, the Casting Director forwards the mangled URL to the Remote Controller. Because of our mangling scheme, there is no need to configure the browser's HTTP Proxy; MASHCast can coexist with any caching web proxy the user may have already configured and can also be configured to use such a proxy itself.

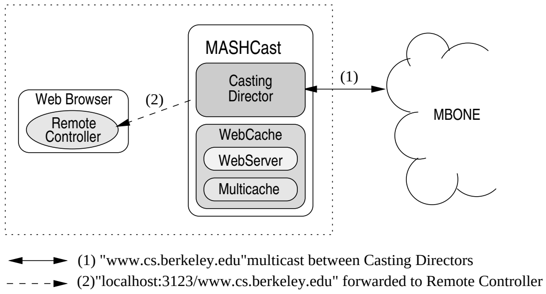

Figure 2: the URL `www.cs.berkeley.edu` is multicast among the Casting Directors. Upon receipt, the Casting Directors prepend the lcoal Web Server's address onto the URL and forward the mangled URL to the Remote Controller.

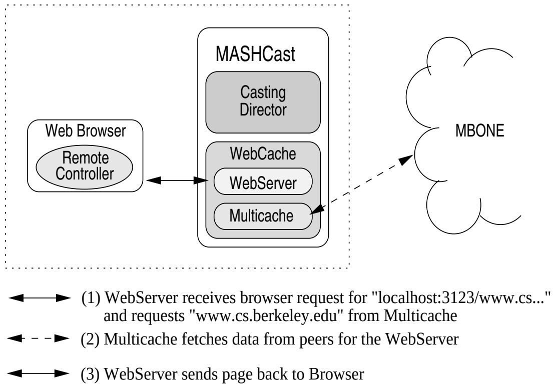

Figure 3: If the requested data are not in the local Multicache, the local Multicache requests the data from peer Multicaches.

When the Remote Controller, a Java applet, receives a URL from the Casting Director, it instructs the browser to fetch the URL. Because the local Web Server's URL has been prepended to the original URL, the request is actually directed back to the local instance of MASHCast. The Web Server subcomponent of the Web Cache receives the browser's page request and attempts to satisfy the request from the Multicache. If the local Multicache does not have the requested page, the Multicache requests the data from peers. Figure 3 shows the Web Cache satisfying the browser's request from the MultiCache. In the figure, the requested data is not in the local Multicache so the Multicache requests the data from peer Multicaches.

As an optimization, the Casting Director could pass the requested URL not only to the Remote Controller but also to the Web Cache. This would serve to prime the cache for the subsequent browser request Web Caches can also be primed before the start of a MASHCast session using scripts read by the Web Cache in *batch mode*; this would optimize response times during forthcoming lectures, for example.

3 SRM background

In this section we describe SRM's reliable service mechanisms. In the next section, we describe how we specialize

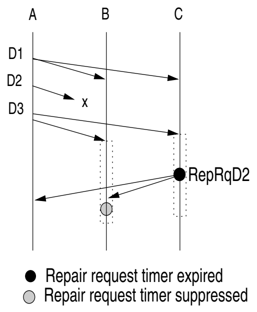

Figure 4: SRM suppresses duplicate requests by delaying requests for random periods.

the SRM framework to create a custom webcast protocol.

Web data are usually either HTML documents or images embedded within an HTML document. The browser can faithfully render these types of data only if they are received in whole. Thus, our application's semantics require reliably delivered data.

A novelty of the SRM framework is its scalable multicast retransmission scheme which relies upon *repair-requests* for loss recovery. Group members are responsible for detecting loss and requesting retransmission. Repair-requests are satisfied with a *repair* message. Repair-requests and repairs are sent to the entire multicast group.

In order to prevent the implosion of repair-request or repair packets sent from receivers to the group, SRM must suppress duplicate repair-requests and repairs. For example, when a data packet is lost near the source, many receivers experience the loss at roughly the same time. Were all of the receivers to send out repair-requests as soon as the loss was detected, the implosion of duplicate repair-requests would likely cause link overload.

Similarly, if many receivers in the group had the requested data and could, therefore, respond to the request, link overload might occur again when the replies were sent.

To avoid implosion, SRM employs a randomized and distributed damping mechanism. In this scheme, repair-requests and repairs are sent only after a randomly chosen delay. All session participants "listen" to all repair-requests and repairs and cancel any duplicates that they may have scheduled so as to prevent implosion.

Figure 4 shows how SRM's randomized suppression can suppress duplicate requests. In the figure and in similar figures to follow, the vertical lines represent increasing time on each of three nodes: A, B and C. Node A sends out a data packet named D1 which nodes B and C successfully receive. Next, node A sends out data packet D2 which is lost before reaching either node B or C. Nodes B and C detect that D2 has been lost upon the receipt of packet D3. Rather than both nodes sending out a repair-request immediately upon loss detection, they instead set request timers with delays that are chosen randomly. The timer intervals are represented in the picture with dashed brackets; timer expiration is represented with the shaded circles. In this example, node C's request timer expires first so node C sends a repair-request for D2. Node B receives this request before its own repair timer expires and suppresses it. The same randomized suppression technique is used for suppressing duplicate repairs.

All participants can cache old data and any participant with a copy of the data can send a repair message. SRM sources periodically send *session messages* reporting their current state. Participants check their state against session messages to detect tail losses.

3.1 Backoff and cancellation policy

For expository purposes, we made a simplification above that, if true, would result in SRM not being reliable under certain circumstances. One can imagine a scenario in SRM where a repair request is sent out and, according to the suppression algorithm described above, all nodes with duplicate requests scheduled cancel those requests. If, due to network congestion or some other network pathology, the request never reaches a node that can reply, the repair will never be transmitted because all other requests have been cancelled.

To ensure reliability, repair-requests are not cancelled because a single request is not always sufficient for reliability. Rather, SRM uses *exponential backoff*. That is, if a receiver sees a request with the same information as a scheduled request, then it reschedules its request with a delay, on average, double the current delay. Repair-requests are only *cancelled* when the requested data has been received.

Repairs, on the other hand, can be cancelled safely. Suppose a repair that suppressed all other repairs at potentially replying nodes never reaches the node that requested the data. We are assured that another repair-request will soon be issued by the node missing the data because repair-requests are only backed off, not cancelled. When the repair-request is resent, the repair cycle will repeat.

4 SRM extensions

The basic reliability mechanisms we have described would alone meet our reliability requirement. However, without specialization, the protocol would not yield the scalability that we desire. In this section, we describe the customizations that we made to the SRM framework to achieve our scalability goals.

4.1 Leveraging peers' caches

As described in previous sections, scalability is one of our key goals. Towards scalability, our protocol must minimize the number of origin requests. When a URL is webcast, ideally only a single Multicache fetches the data from the origin server and multicasts it to the group. This could be accomplished by designating the multicache where the request originated as the multicache that sends the origin request as in [12] and [16]. In other words, if participant A would like to send the document with URL `http://www.cs.berkeley.edu`, participant A would fetch the data from the server and multicast it to the session. Without additional changes in the protocol, however, this simple policy could fail to prevent server implosion in common collaborative situations. Consider a remote lecture program where the speaker is webcasting slides which contain hyperlinks to sites relevant to the lecture. It is likely that some of the students will want to immediately visit those sites causing a number of synchronized fetches, i.e., implosion at the origin server.

In an effort to avoid server implosion, we attempt to satisfy browser requests first from our local multicast web cache and, second, from a peer's multicast web cache. The origin server is contacted only if no one in the group has the requested data. Because the CastingDirector mangles the request before forwarding it to the Remote Controller so that it points to MASHCast's Web Server, MASHCast intercepts any page requests. If the data is present in the local multicast cache, the request is satisfied. If the data is not present locally, then a repair-request is multicast to peer Multicaches. Any session participants that have the data in their cache already can respond to the repair-request by scheduling a repair. Eventually, some participant's timer expires and a repair is sent. Satisfying requests from a peer's Multicache can eliminate server hits in several scenarios. Consider the example remote lecture program above. Imagine that one remote student joins the session late and requests a previously seen slide. Using previous solutions' policy where the requesting node sends an origin request and multicasts the resulting data prohibits the late student from leveraging other participant's caches. Enabling the student to benefit from others' caches leads to fewer origin requests and, if other members were within shorter round-trip times than the server, better response times. This scenario is the motivation behind using the MultiCache component alone in groups likely to browse the same pages.

4.2 Injecting data into the Web Cache

In the previous section, we explained how we can reduce server load with an SRM-based web cache. But what happens when no one has the requested data? The reliability model described in [6] assumes that all data is published into the multicast session explicitly by some source. Within the context of a webcache, however, desired data might not yet exist in the session and there must be some mechanism in place to retrieve the data from the origin server and inject it into the multicast cache. Previous solutions solve the problem by sending an origin request if the page doesn't exist locally. As explained above, this can lead to server implosion.

One solution is to have each Web Cache set a repair timer regardless of whether or not they have the data. Should a participant that does not have the data set a repair timer that expires before other repairs are received, that member sends an origin request when their timer expires rather than a repair. Although this solution works, it can lead to duplicate server hits because during the interim period when the data is being retrieved from the origin server, other participants' timers may have expired thereby triggering more origin requests.

4.2.1 Reply-pending messages

To solve this problem, we leverage SRM's flexibility by defining a new message called a *reply-pending* message which notifies other participants that an origin request has commenced and thereby suppresses duplicate origin requests. Upon receipt of a repair-request message, all nodes that do not have the requested data set a reply-pending message timer rather than a repair timer. If a node's reply-pending timer expires, that node sends a reply-pending message to the group *and* fetches the data from the origin server. All session participants listen for reply-pending messages and refrain from sending duplicate web server requests by surpressing their outstanding reply-pending timers.

Figure 5 illustrates how a reply-pending message can suppress duplicate origin requests. When node B's repair timer expires, it not only initiates an origin request but also multicasts a reply-pending message to the group. Upon receipt of the reply-pending message, Node C surpresses it's reply-pending message.

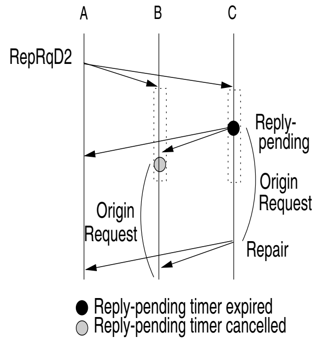

Figure 5: MASHCast reply-pending messages suppress duplicate origin requests by informing peers that an origin request in in-progress.

As was done in [6] for SRM repair-request and repair messages, we must decide between a cancellation or backoff policy for reply pending messages. Figure 6 illustrates what might happen if we implemented exponential backoff of reply-pending messages. Node A sends a repair-request and nodes B and C respond by setting reply-pending timers. Node C's timer expires so Node C multicasts a reply-pending message. Node B responds by backing off it's reply-pending timer. Recall that a timer is scaled back by first cancelling the currently scheduled timer and then setting a new timer with, on average, double the delay. Backoff is represented in the figure with a lighter-colored circle and a second, longer bracket denoting the new, longer timer delay. Despite the fact that the timer has been scaled back, due to a lengthy web server response time the timer still expires before the desired data has been multicast. Once the timer expires, an unnecessary origin request is sent. In fact, since this is more likely to happen as web server response times grow longer, unnecessary origin requests are more likely to be sent when the servers are overloaded thus exacerbating the problem—exactly the behavior we want to avoid.

For this reason, we chose a cancellation policy for reply-pending timers rather than exponential backoff. Al-

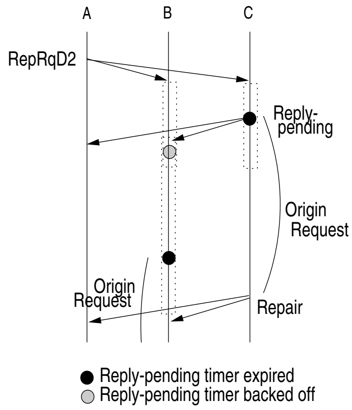

Figure 6: A Reply-pending backoff policy could cause duplicate origin requests. The better policy choice is cancellation.

though it is still possible that duplicate origin requests could be generated despite a reply-pending cancellation policy, the difference is that with a cancellation policy, the repair-request and reply-pending timer intervals can be configured such that the time between any generated duplicates is probabilistically longer than that of a backoff policy. (The next section discusses the relationships between timer suppression intervals in more detail.) We can still ensure reliability despite reply-pending cancellation for the same reason that SRM ensures reliability despite a cancellation policy for repairs: should the participant that sends out a reply-pending timer fail to successfully fetch the requested data from the origin server and multicast it to the group (due to system failure, for example), those nodes missing the data still have outstanding repair-request timers. Eventually those timers will expire, repair-requests will be sent out, and new reply-pending messages will be scheduled.

The techniques presented in Sections 4.1 and 4.2 together have the novel effect of applying SRM's slotting and damping technique to origin server requests. In this manner, origin requests are minimized—something we believe is vital for scalability that previous solutions have not addressed.

4.3 Defining the suppression intervals

This section discusses how we can schedule the repair-request, repair, and reply-pending delay timers to minimize server load and network traffic.

In SRM, repair-request and repair timers are delayed for a random amount of time to avoid implosion. The *suppression interval* over which the repair-request delay is chosen is a function of the participant's estimated distance from the data source. Distances to remote nodes are estimated using timestamps sent in SRM session messages.

Equation 1 shows how SRM defines the request interval when Node A detects a loss of data sent from Node S. C_1 and C_2 are configurable parameters and $d_{S,A}$ is Node A's estimated distance to Node S.

$$[C_1 d_{S,A}, (C_1 + C_2) d_{S,A}] \quad (1)$$

Equation 2 shows how the repair interval is defined when Node B receives a request from Node A. D_1 and D_2 are configurable parameters, as in equation 1, and $d_{A,B}$ is Node B's estimated distance to Node A.

$$[D_1 d_{A,B}, (D_1 + D_2) d_{A,B}] \quad (2)$$

The addition of reply-pending messages presents a new interval definition challenge. The challenge is to define the repair and reply-pending suppression intervals such that repair-requests are satisfied by the multicast web cache before the origin server when possible. Equation 3 describes our reply-pending interval:

$$[E_1 d, (E_1 + E_2) d] \quad (3)$$

How we configure E_1 and E_2 and how we define d in relation to D_1, D_2, and $d_{A,B}$ determines whether the repair and reply-pending intervals are the same, overlapping or distinct. We might need adaptation beyond just the distances between SRM session participants. For example, in determining the value of d in equation 3, it might be useful to be able to factor estimated distance and throughputs between receivers and origin servers.

Figure 7 shows two alternative interval configurations.

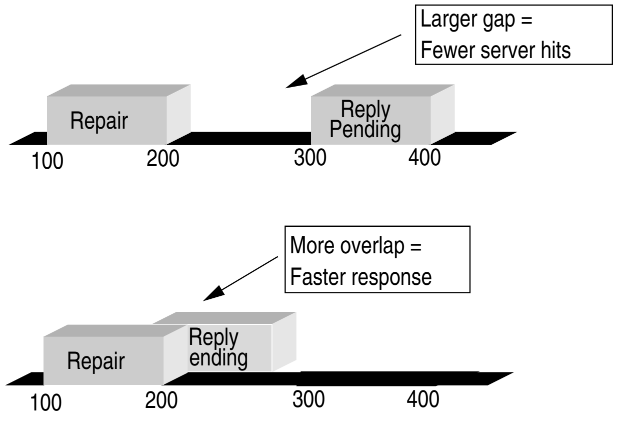

Figure 7: The addition of reply-pending messages presents a tradeoff between response time and origin server load.

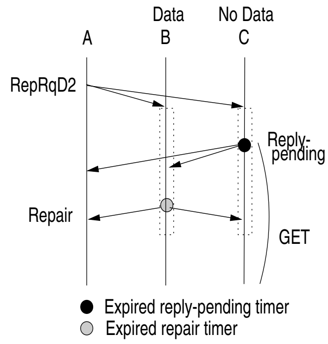

Figure 8: Unaware that node B has the requested data, Node C sends an unnecessary origin request before node B is able to send a repair.

On the upper line, the two intervals are distinct. The repair interval is between 100 and 200 milliseconds and the reply-pending interval is between 300 and 400 milliseconds. By eliminating overlap, we lessen the chance of unnecessary server hits by maximizing the chance that a participant with the requested data will send a repair before other participants' reply-pending timers expire. The tradeoff for minimizing server hits in this way is increase in the required time to inject new pages into the multicast cache when no participant has the data. In this example, 300 milliseconds must elapse before the earliest possible reply-pending timer expires. Alternatively, the lower line in figure 7 shows overlapping intervals which could improve response time but increase the likelihood of unnecessary origin requests.

Figure 8 shows what might happen if the intervals are the same. Node A sends a request. In response, Node B sets a repair timer (denoted by the R) because it has the requested data. Node C sets a reply-pending timer (denoted by the P) because it does not have the requested data. Node C happens to have randomly chosen a shorter delay than Node B and it's reply-pending timer expires before node B is able to fulfill the request. The result is that Node C unnecessarily sends an origin request. This illustrates not only that the repair and reply-pending intervals should not be the same but that, to avoid contacting the origin server when some SRM participant has the data, the repair timer interval should precede the reply-pending timer interval. With such a configuration, the delays for repair timers should precede in time the reply-pending timer delays.

To explore the parameter space, we ran experiments in which we varied the repair and reply-pending intervals and observed any changes in response times and bandwidth usage. The details of this experiment are omitted here due to length but can be found in [22]. The experiments showed that, for most documents, the protocol behaves as expected as the intervals are varied. We discuss exceptions in section 6.

5 Implementation

To verify the efficacy of our design and provide a useful tool for online collaboration, we have implemented a prototype of the MASHCast components—the Casting Director, Remote Controller, and Web Cache—using the MASH multimedia toolkit [14]. We discuss lessons learned while using MASH in section 6.

5.1 Behavior

To get an idea of the behavior of our initial implementation, we performed a simple study. We selected three web sites in geographically distinct areas. Table 1 lists the documents comprising the three web sites, their sizes, and the corresponding round-trip times from the experimental machines to the respective origin servers. Because some of the document names are lengthy, we include in the table monikers for each document which we will use for the remainder of the paper rather than the lengthy names.

Initially, we ran four experiments for each site. Each experiment measured the time taken by the Web Cache to obtain each document. To increase our control over each experiment, we instrumented the web cache to "play" requests at predetermined times from a list of requests in a file "recorded" previously using a web browser and a live user to request the documents. Using such a recording eliminates the variance introduced when using live users to trigger document requests in experiments. In this playback mode, the Web Server and Casting Director components are not needed and were not run.

The four experiments are described below:

1. **Direct.** As a baseline measurement, we instrumented MASHCast to bypass the cache and fetch the requested page immediately. By bypassing the cache, we approximate the amount of time that it would take a browser to fetch a page directly over the Internet with no help from MASHCast.

2. **Miss.** For this experiment, we started a single MASHCast application. We requested the documents and measured the time to fetch the page. This measures the overhead to inject a page into the Multicache. Because of the necessity of an origin request in addition to the overhead of the per-request reply-pending delay, we expected this experiment to yield the longest load time.

3. **Peer Hit.** For this experiment, we started a MASHCast application on one machine and primed its cache with the appropriate pages. We then started a second MASHCast participant on a second machine .25ms and requested the page a second time. Because the first participant had the page in its multicache, we expected it to respond to the second MASHCast's request for data. This experiment measured the time saved by taking advantage of data in a peer's multicache. We expected this to be the second fastest load time after a Hit (described below).

4. **Hit.** For this experiment, we measured the time to load a page from the participant's local multicache. We expected this time to be the fastest since no network communication is necessary.

Each experiment was repeated thirty times and the results were averaged. Figure 9 shows the load times by site for each experiment. Note the logarithmic y-axis rendered necessary by the orders of magnitude difference between Peer Hit and Hit times and Direct and Miss times.

As we expected, the load times for both Hit and Peer Hit were faster than either the Direct or Miss. We were surprised, however, that the Peer Hit and Hit times were so similar. The Peer Hit time should have shown document transmission time that the Hit case didn't require. Upon investigation, we found that, based on SRM session messages multicast by the first node, the SRM recovery mechanism at the newly started node was detecting the missing documents and recovering them before the web cache layer requested them. Our experiment was set up such that the first document was requested 9.8 seconds after the application was started. Within that 9.8 seconds, the local node was easily able to detect missing previously sent data, request the data from the session, and recover the data. The effect was that each request resulted in a Hit rather than a Peer Hit.

Rather than try to beat *libsrm*'s efficient recovery mechanisms, we introduced a new recovery policy and a fifth test case. SRM detects and informs the application of data losses but only recovers those losses if the *application* explicitly requests SRM to do so. Our default *aggressive-recovery* policy specifies that lost data is always recovered. We used SRM's *selective reliability* capabilities to implement a second recovery policies for MASHCast: our *lazy-recovery* policy specifies that the SRM layer only recover data that the browser has explicitly requested. We expected that lazy-recovery would yield slower response times than those of aggressive-recovery due to the overhead of retrieving data from a peer's cache rather than from one's own cache at the time of request. However, lazy-recovery could save cache space and reduce network traffic to achieve greater scalability in many circumstances—for example, when late joiners in a collaborative session aren't interested in seeing previously webcast data or when the Web Cache is being used as a passive cache.

Our additional test case is described below:

5. **Peer Hit w/Lazy Recovery.** This experiment is identical to the Peer Hit experiment except that the lazy-recovery policy is enabled so that the application will only request recovery of a missing document if the user has requested that document.

Discussion. The advantage gained by using another participant's cache is represented by the difference between the Peer Hit bar and the Direct bar or, if lazy-recovery is desired and enabled, the advantage gained is represented by the difference between the Peer Hit w/Lazy Recovery bar and the Direct bar. Similarly, the advantage gained by using the local cache is represented by the difference between the Hit bar and the Direct bar. These experiments

No.	Document	Moniker	RTT in ms	kB
1	www.cs.berkeley.edu/	berk	38.00	9.2
2	www.lcs.mit.edu/	mit	61.00	5.2
3	www.lcs.mit.edu/lcs_style.css	mit-style	61.00	.6
4	www.lcs.mit.edu/images/nav.gif	mit-nav	61.00	12.8
5	www.lcs.mit.edu/images/banner.gif	mit-banner	61.00	12.4
6	www.lcs.mit.edu/images/divider.gif	mit-divider	61.00	.3
7	www.lcs.mit.edu/images/mld.jpg	mit-mld	61.00	8.5
8	www.lcs.mit.edu/images/w3c.gif	mit-w3c	61.00	1.4
9	www.lcs.mit.edu/images/jupiter.gif	mit-jup	61.00	7.0
10	www.lcs.mit.edu/images/cilklogo.gif	mit-cilklogo	61.00	4.2
11	www.lcs.mit.edu/images/bluepix1.jpg	mit-bluepix	61.00	1.9
12	www.cs.utah.edu/	utah	0.85	4038
13	www.cs.utah.edu/mountains3.jpg	utah-mts	0.85	77.7
14	www.cs.utah.edu/banner_title.gif	utah-banner	0.85	1.6

Table 1: Documents requested in experiments.

do not show the scalability gained by fewer net server hits and fewer data transmissions resulting from suppressed duplicate origin requests. These gains are measured on a small scale in [22] and are also the subject of future work discussed in section 6.

We would expect that the difference between the Miss and Direct response times would include the following items:

I On average, half of the repair-request interval plus the minimum delay (150ms).
II On average, half of the reply-pending interval plus the minimum reply-pending delay (175ms).
III Overhead of our cache software.

Given the above, we would expect to see an average of at least 325ms (the sum of first two items above) difference between the Miss and Direct response times. Accordingly, if a horizontal line is drawn across the graph at 325ms, we see that that is indeed a lower bound in the graph.

From individual document response times, we can approximate the minimum and maximum overhead experienced. `mit-mld` shows the smallest difference between Miss and Direct response times, 359ms of overhead, and `mit-nav` shows the largest overhead, 620ms. By subtracting the 325ms lower bound found above, we estimate a minimum overhead of 35ms and a maximum overhead of 295ms for our dataset.

Peer Hit w/Lazy Recovery response time is comprised of

I On average, half the repair request delay interval (150ms).
II On average, half the repair delay interval (100ms).
III The transmit overhead to and from the peer.
IV At least the minimum amount of overhead (35ms).

This yields a lower bound of only 285ms. In fact, we experienced much higher load times than expected. Upon investigation, we found that a combination of implementation choices accounted for the poor Peer Hit w/Lazy Recovery performance. Our reply-pending parameter configuration did not allow enough time for the peer cache to satisfy our request so that, in the meantime, a spurious origin request was sent and satisfied. Coupling this with the fact that the MASHCast components are optimized not for performance but for ease of implementation and policy change (for example, the Web Cache is all interpreted Tcl), and a mismatch in our data naming scheme accounts for the poor performance. Section 6 covers the mismatch in our data naming scheme in more detail.

Though the Peer Hit requests were satisfied via the disk, they uniformly took just slightly longer than Hit requests, which were also via the local disk. At this time, we don't know why this is but believe that analysis of the two code paths will reveal the difference.

In general, the MASHCast components are optimized not for performance but for ease of implementation and policy change. For example, the Web Cache is all interpreted Tcl. *libsrm* implemented in C++ and, at the time of experimentation, was also largely unoptimized.

6 Lessons learned and future work

The MASH multimedia toolkit provides a low barrier-to-entry, multimedia research platform. We believe that our implementation experience with MASH is relevant to the broader multimedia research community looking for such a platform. In this section, we report on lessons learned during implementation and the future work and our observations defined.

The MASH software model is an object-oriented architecture in which objects' methods can be implemented in either C++ or the MIT Object Tcl system (OTcl) [14]. This split object model was especially conducive to our research because it enabled backend mechanisms to be implemented using efficient C++ code while policy could be implemented using TCL for ease of policy modification. We found that this ease of policy change facilitated the experiments discussed in [22] and section 5. In addition to allowing policy modification ease, implementing within the MASH environment allowed us to leverage existing, well-tested MASH components to lessen the implementation effort. Specifically, we used MASH's Announce Listen Manager, HTTP Server and Adaptive Timers. MASH offers many more multimedia components including standard audio and video transmission protocols.

libsrm is an instantiation of the framework presented in [6]. In addition to the SRM reliability mechanisms, *libsrm* incorporates the SNAP hierarchical, scalable naming protocol [18]. *libsrm* is available as a standalone library and as a MASH component. In general, we found *libsrm* simple to use. All of the SRM extensions described in section 4 were implemented in approximately 1200 lines

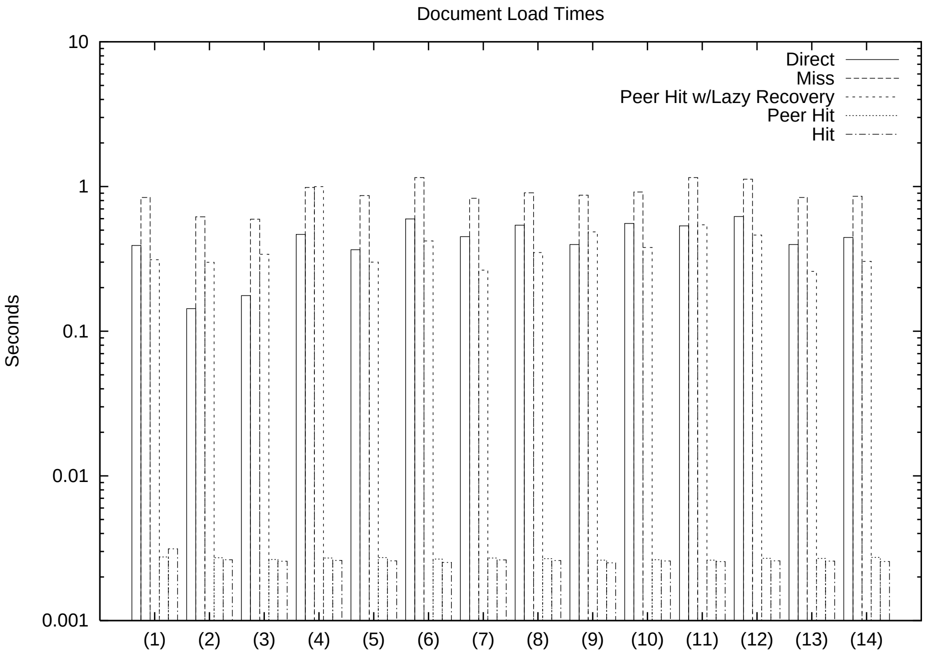

Figure 9: Page load times for each of the five experiments by site. The numbers across the x-axis reference the document numbers in Table 1.

of Tcl code. Lazy recovery was implemented simply by adding a one-line check to see if the URL had been explicitly requested in the application's implementation of the *libsrm* upcall invoked when a loss is detected.

We found that, while we gained critical namespace scalability from SNAP our application didn't conform naturally to SNAP's naming convention. One assumption central to both SNAP and SRM is that data are source-specific. Accordingly, a source is associated with each data name. While this association is desirable for the vast majority of multimedia applications, it is not optimal for a webcast application. Associating an SRM source with Web documents can falsely differentiate the same document. For example, node A might obtain and source `www.cs.berkeley.edu`, naming it `[A,www.cs.berkeley.edu]`. Simultaneously, node B gets the same URL but names it `[B,www.cs.berkeley.edu]`. Fortunately, duplicates can be easily detected at the application level where, in accordance with ALF principles, caching and recovery are managed so that duplicate cache entries are avoided. However, duplicate transmissions do sometimes occur (see below). A non-source-specific naming option would be desirable for transmitting web data. Currently, SNAP maintains namespace trees per source. By adding a new *universal source* with a well-known source-id, we enable SRM to detect non-source specific, duplicate data. This alone does not solve the problem, however, because all applications would have to agree on the same container ID and application data unit (ADU), the two components in a SNAP data name, for like data. We intend to investigate solutions to make SNAP more amenable to applications which do not have source-specific data.

Secondly, we learned that the natural choice for a Web ADU is not optimal. At the outset, the single document or image corresponding to a URL seemed natural. However, this proved to be too coarse-grained for large documents. Since the SRM layer doesn't notify the application layer that an ADU has arrived until after the entire ADU is received, we occasionally observed the application, unaware that a document was already partially received and would shortly be fully received, spuriously request the document a second time. Compounded with the naming mismatch described above, this led to a transient state in which different SRM names existed for duplicate URLs and duplicate data were multicast. Depending on the document's size, the transmission of duplicates could account for significant amounts of unnecessary data. In hindsight, we would break documents into large fragments so that the application would be notified in a more timely manner of incoming documents. We implemented an earlier version of MASHCast designed for a non-fragmenting SRM library that *did* perform fragmentation and reordering. We should be able to merge this code into the current version with little change other than to expand the fragmentation granularity from 1K to 10K. This size avoids duplicating SRM's underlying 1K fragmentation efforts and avoids fragmenting in the common case of small documents.

Finally, though our small-scale experimentation is adequate to demonstrate MASHCast's performance gains, they are not sufficient to demonstrate proven scalability. Because we have built upon two protocols whose scalability has already been proven in the literature, SRM and SNAP we are reasonably confident that our design is also scalable. Nonetheless, we plan additional experimentation within the year aimed at stressing MASHCast's tolerance to scale within an emulated WAN environment using a large-scale testbed we are building in a local but orthogonal project.

7 Related work

Webcast [1] uses NCSA Common Client Interface to communicate with the Mosaic web browser via TCP/IP to fetch URLS, request Mosaic to report URLs selected by the user, and pull in data from web to local program space. They utilize RMP (Reliable Multicast Protocol) which provides ordered, reliable multicast delivery. RMP uses a token-ring based algorithm to ensure delivery which requires an explicit list of receivers. A per-receiver list precludes them from targeting large groups as we do. A second difference between MASHCast and Webcast is that Webcast is specific to the Mosaic browser.

mMosaic [4] uses a modified Mosaic browser to send documents out via multicast. Reliability is achieved via an announce/listen protocol [19]: documents are sent every four seconds and receivers listen to and store the multicast data. Once a document has been fully constructed it is displayed. When losses occur, they may not be repaired for four seconds. The potential four-second delay to repair losses coupled with the time to download the data can add up poor response time. Lastly, mMosaic is specific to the Mosaic web browser.

mWeb's [16] solution is similar to ours in that they use SRM for reliable distribution. However, they do not take advantage of SRM's flexibility as we do to to further improve scalability by minimizing origin requests.

WebCanal [12] also has a design that is similar to MASHCast's. WebCanal uses a modified version of SRM called Light-weight Reliable Multicast Protocol (LRMP) to multicast documents reliably. Like mMosaic, there is no solution offered for server implosion that can result from many nearly-simultaneous origin requests. Also, LRMP uses a constant for suppression rather than the random, adaptive suppression algorithms found in SRM. We have found that a non-adaptive algorithm can cause pathologies in certain circumstances [11].

8 Conclusions

Large-scale multimedia conferencing applications are enjoying growing popularity. To complement existing audio, video and drawing applications, we have designed a novel webcast architecture which features (i) reusable components and (ii) a custom protocol built upon the SRM framework that maximizes scalability. Our proposal improves upon existing work by leveraging SRM to tailor a custom protocol to satisfy webcasting's unique needs. We have implemented our design in a useful collaborative application that is freely available for download with the MASH multimedia toolkit. Through simple experimentation, we have verified the efficacy of our design, demonstrated the power of ALF to enable applications to tailor protocols to meet their unique needs, and reported on the lessons learned using the MASH multimedia toolkit and *libsrm*.

Acknowledgements

We thank Suchitra Raman, Tina Wong and Yatin Chawathe for the implementation of *libsrm* and significantly improving MASHCast. Yatin also conceived the URL mangling scheme which frees the MASHCast user from having to configure a browser's HTTP proxy.

References

[1] E. Burns et al. Webcast – Collaborative Document Sharing via the MBone. Information online at http://www.ncsa.uiuc.edu/.

[2] S. Casner. Frequently asked questions (FAQ) on the multicast backbone (MBone), Aug. 1994.

[3] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM '90*, September 1990.

[4] G. Dauphin. mMosaic: Yet Another Tool Bringing Multicast to the Web. In *Workshop on Real Time Multimedia and the WWW*, October 1996.

[5] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[6] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proceedings of the 1995 ACM SIGCOMM Conference*, August 1995.

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

[8] T. Frivold, R. Lang, and M. Fong. Extending WWW for Synchronous Collaboration. In *Proceedings of the 2nd International WWW Conference*, Chicago, Oct. 1994.

[9] M. Handley and J. Crowcroft. Network Text Editor (NTE): A scalable shared text editor for the MBone. In *Proceedings of SIGCOMM '97*, Cannes, France, Sept. 1998.

[10] V. Jacobson and S. McCanne. Visual audio tool. Software online at ftp://ftp.ee.lbl.gov/conferencing/vat.

[11] M. Kornacker and K. Wright. Experimental evaluation of a reliable multicast application. U.C. Berkeley CS268 Computer Networks term project and paper. Unpublished report., May 1997.

[12] T. Liao. Webcanal: a Multicast Web Application. In *6th International WWW Conference*, Santa Clara, CA, April 1997.

[13] S. McCanne. A distributed whiteboard for network conferencing. U.C. Berkeley CS268 Computer Networks term project and paper, May 1992.

[14] S. McCanne et al. Toward a Common Infrastructure for Multimedia-Networking Middleware. In *Workshop on Network and OS Support for Audio and Video '97*, 1997.

[15] S. McCanne and V. Jacobson. vic: a Flexible Framework for Packet Video. In *ACM Multimedia '95*, San Diego, CA, November 1995.

[16] P. Parnes et al. mWeb: a framework for distributed presentations using the WWW and the MBone. In *W3C Workshop: Real Time Multimedia and the Web*, Sophia Antipolis, France, October 1996.

[17] S. Pingali, D. Towsley, and J. F. Kurose. A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols. In *Proceedings of ACM Sigmetrics '94*, Santa Clara, CA, 1994.

[18] S. Raman and S. McCanne. Scalable data naming for application level framing in reliable multicast. In *ACM Multimedia '98*, Bristol, UK, September 1998.

[19] E. M. Schooler. A Multicast User Directory Service for Synchronous Rendezvous. Computer Science Department, California Institute of Technology, Sept 1996.

[20] G. Sidler, A. Scott, and H. Wolf. Collaborative Browsing in the World Wide Web. In *Proceedings of the 8th Joint European Networking Conference*, Edinburgh, May 1997.

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

[22] K. Wright. Using *libsrm* for Caching and Collaboration. Technical Report UUCS-99-04, University of Utah, Mar. 1998. http://www.cs.utah.edu/.