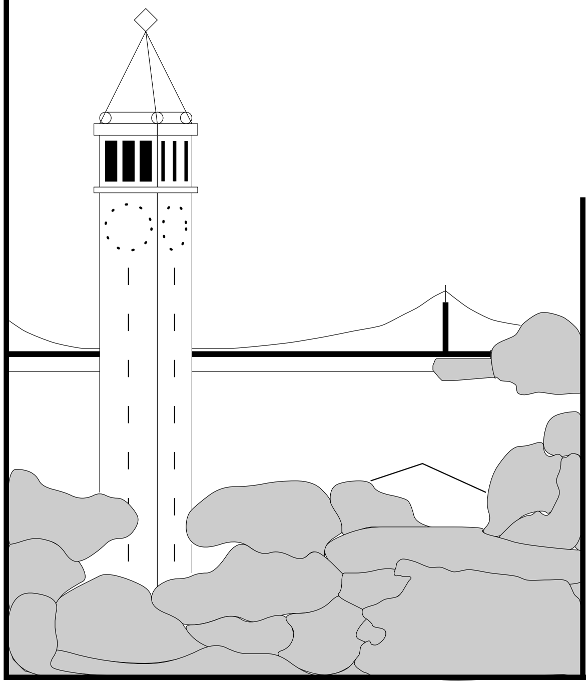
Reliable Multicast Protocol Specialization for Caching and Collaboration within the World-Wide Web

L. Kristin Wright



Report No. UCB/CSD-99-1050

June 1999

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Reliable Multicast Protocol Specialization for Caching and Collaboration within the World-Wide Web

L. Kristin Wright *

June 1999

Abstract

The World Wide Web (WWW) has become an important medium for information dissemination. One model for synchronous information dissemination is a scheme called *webcasting* where data is 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 which improves upon previous designs by leveraging the 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 webcasting's needs: reliable delivery which is scalable with respect to both origin server load and link load. We employ the protocol in a webcasting architecture consisting of two reusable components, a webcache component and a browser control component. The resulting application is a scalable webcasting application that requires no browser modifications or proxy specification. We have implemented our design in a webcast application called MASHCast. Initial measurements showed that our scalable custom protocol reduced load times from 7% to 53% depending upon the size of the page and the round-trip time to the page's origin.

*This work is supported by the Advanced Research Projects Agency (N66001-96-C-8508) and California MICRO.

1 Introduction

Online collaborative environments 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. Examples of online collaboration include group meetings held across multiple locations, distance-learning course lectures, or technical conference broadcasts. Such collaborative programs might consist of a video session of one or more sites, corresponding audio sessions, and a distributed whiteboard session.

Video [14][8] and audio [10][9] applications have been widely deployed, and a complimentary multimedia-board application has recently been developed [16]. 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 synchronized such that the browsers display the same pages simultaneously. There are a number of slightly differing models, each requiring the synchronized distribution of web documents and their embedded objects to multiple sites. A common name for this type of collaborative session is *webcasting*.

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 server *implosion* — a pathology where a resource is unable to keep up with an incoming stream of messages. 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.

Were the server able to keep up with a large number of incoming requests, the outgoing replies would likely cause network congestion. For example, a request for my home page (as it is written today) from Netscape 3.01 is 296 bytes. The objects that comprise the page total 55kB and require 12 TCP connections for transmission. If we conservatively assume a 10Mb/sec Ethernet link and ignore the fact that Ethernet performance decreases exponentially at about 50% utilization , the link will become saturated servicing a load of approximately 181 near-simultaneous requests. This estimate is conservative because it assumes a relatively high bandwidth link and ignores degrading Ethernet performance. If we assume a T1 link with 1.5Mb/sec bandwidth the link becomes overloaded servicing only 27 requests for such a page. These estimates for link saturation are merely back of the envelope calculations. They are sufficient to show, however, that traditional HTTP unicast transmission will not scale to sessions of several thousand participants.

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 as session membership increases. 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 multi-destination delivery of packets than either unicast or broadcast. Rather than send out 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 [2]. While this is a more scalable approach than the naïve 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 [15] 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.

1.2 Our Approach

Our webcasting solution is similar to [12] and [15], but we improve upon existing work by leveraging the Scalable Reliable Multicast (SRM) framework [7] to build a custom protocol to suit webcasting's scalability needs. 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 in this paper.

In addition to our custom protocol, we also 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. MASHCast requires configuration of a MASH browser plug-in, but once this plug-in is configured, no web server modification or browser proxy specification is required. In contrast to existing applications which require special setup steps such as configuring a web proxy, the MASHCast user need only start the application to begin participating in a webcast session.

In this paper we describe our custom webcast protocol and our dual-component architecture. The remainder of this document is structured as follows. Section 2 explains our architecture. Section 3 explains the basic SRM framework and how it provides reliability. Section 4 discusses the protocol we have developed. Section 5 discusses the MASHCast prototype implementation. Section 6 discusses related work in webcasting. Finally, Section 7 and section 8 present conclusions and future work.

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.

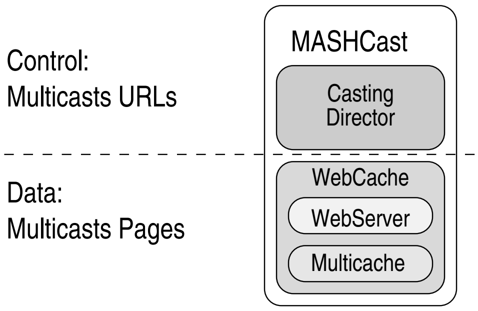


Figure 1: **MASHCast Architecture.**

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]

[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.

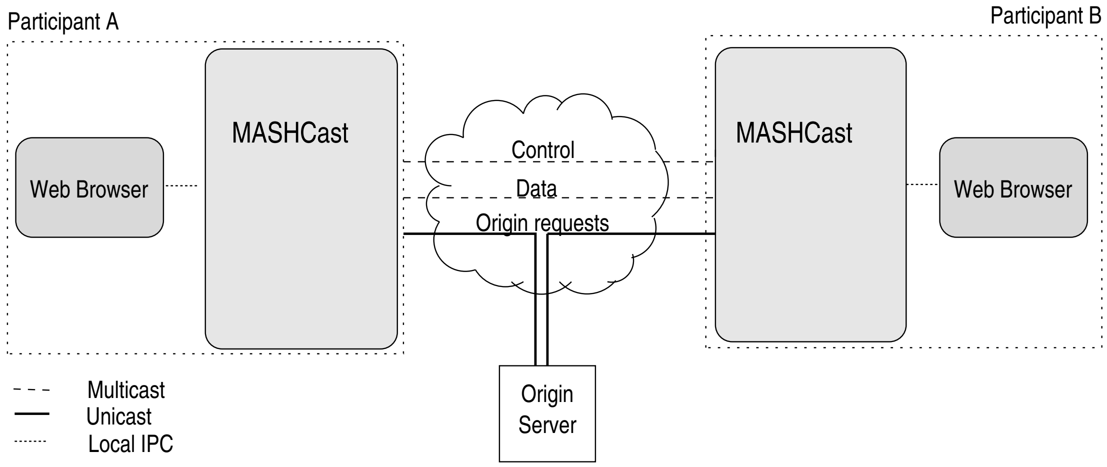


Figure 2: **MASHCast Communication Paths.**

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

In addition to being compatible with existing WWW infrastructure, our architecture is novel because its components can be cleanly decoupled for reuse. The Web Cache component can be used apart from the Casting Director as a passive multicast web cache. For example, 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). 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 the Hypertext Transport Protocol (HTTP), a unicast, TCP-based transport protocol used for the WWW. 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 for the remainder of the paper) and fetch a special MASHCast bootstrap page into the browser. Figure 2 shows a MASHCast session with two participants.

Once MASHCast is started, a sender webcasts a page by specifying 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 3 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 onto the URL. In this example, the local address is `localhost:3123/`. After the URL is mangled, the Casting Director forwards the mangled URL to the Remote Controller.

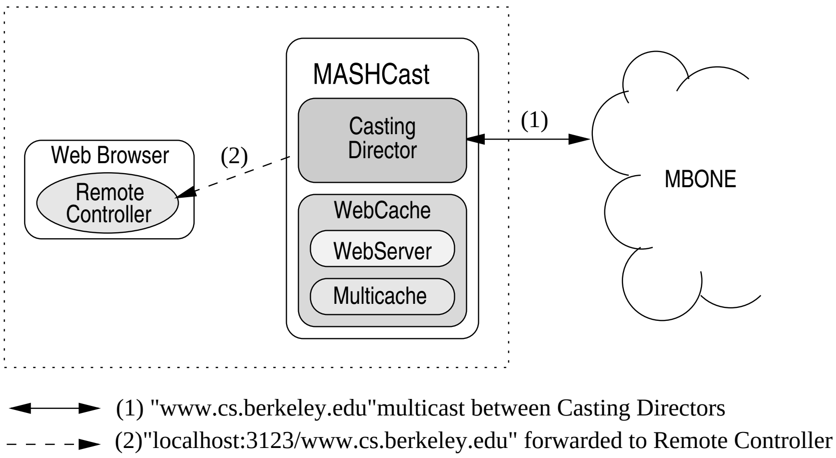


Figure 3: **Casting Director.**

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 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 4 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 its peer Multicaches.

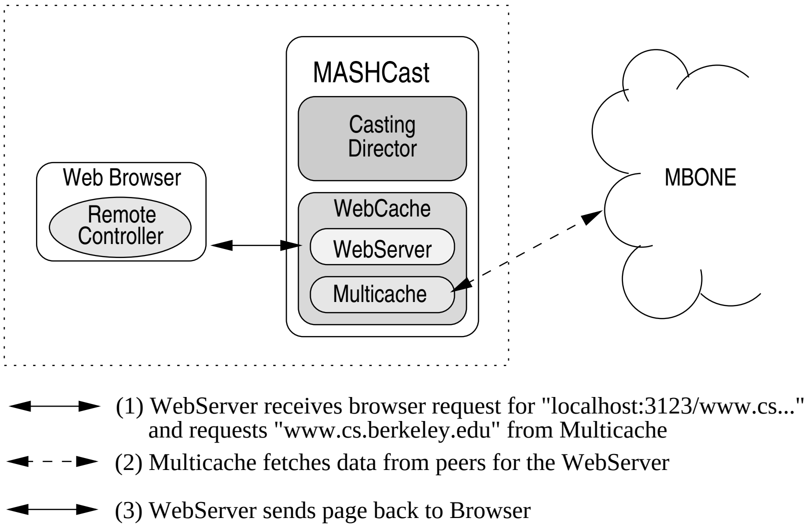


Figure 4: **WebCache.**

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 request from the browser.

3 SRM Background

In this section we describe SRM's reliable service protocol.In the next section, we describe how we specialize the SRM framework to create a custom webcast protocol.

Web data are usually either Hypertext Markup Language (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.

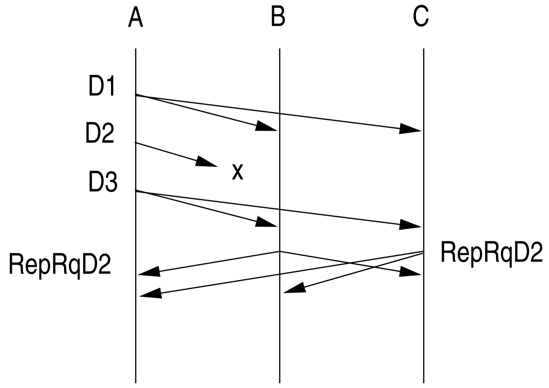


Figure 5: Several nodes experiencing simultaneous loss can cause duplicate repair-requests.

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, or sent to a subset of the group if local recovery is in effect.[2]

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.

Figure 5 illustrates this scenario. In the figure (and in similar figure 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 a second data packet, D2, which is lost before reaching either node B or C. Node A, unaware that packet D2 was lost, next sends packet D3 which is successfully delivered. Nodes B and C, upon receiving packet D3, detect that they have lost packet D2 and both send out a repair-request for the packet D2 to the multicast group. If we consider a much larger group — say thousands of nodes instead of three — then it is not difficult to imagine how unchecked duplicate requests could cause link overload. Similarly, if many receivers in the group had the requested data and could, therefore, respond to the request, link overload might again occur once all of the replies were sent.

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

[2]Local recovery is the subject of ongoing research and is not yet implemented.

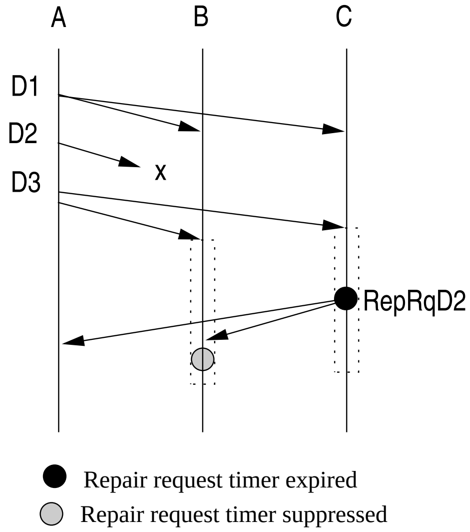


Figure 6: SRM suppresses duplicate requests by delaying requests for random periods.

Figure 6 shows how SRM's randomized suppression can suppress duplicate requests. As in Figure 5, data packet D2 is dropped before reaching nodes B and 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 boxes; the time at which timers expire is represented with solid circles; the time at which suppressed timers would have expired is represented with shaded circles. In this example, node C's request timer goes off first and it 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 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 that request. 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 canceled.

To ensure reliability, repair-requests are not canceled 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 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 request will soon be issued by the node missing the data because repair-requests are only backed off, not cancelled, when a repair has been received. When the request is resent, and the repair cycle will repeat.

4 SRM Customizations

The basic reliability mechanism we have described alone would 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 Data Naming

In virtually all transport protocols, data is named based on a [source, sequence number] scheme where the source refers to the node sending the data and the sequence number is the number of bytes that have been sent previously. However for many applications, including webcasting, this common naming convention does not allow the application to name its data with a convenient naming scheme. For example, in the HTTP protocol, web data is named using unique strings called Uniform Resource Locators (URL) [6]. Their use in HTTP makes URLs the natural choice for data naming in a webcasting application. To use a protocol with the more common [source, sequence number] naming scheme, it is necessary to map the URLs to sequence numbers, perhaps in the order that the URLs are seen. Tables would be necessary to map URLs to sequence numbers and back. Furthermore, once a series of numbers are selected to represent a certain URL, they are associated with a source. This can falsely differentiate the same document. For example, node A might request the URL `http://www.cs.berkeley.edu` and name it `[A, 500]`. Simultaneously, node B requests the same URL but names it `[B, 700]`. The result is that an unnecessary copy of the document is transported among the multicast group.

Using SRM, applications can determine their own naming scheme. In our protocol, data are named using the document URL and the offset within the document: [URL, offset]. The total length of the page is transmitted in the packet headers along with the name. Since the application knows the document length and what fragments it has received, losses within a document are detected from gaps in the offsets. For example, if a participant receives bytes 256-512 but has not received 0-255, it assumes a loss and schedules a repair-request. Losses at the end of a document are discovered using SRM session announcements.

4.2 Avoiding Web Server Implosion

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 webcasted, 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. 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 the cache 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 the simple 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.

4.3 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 [7] 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.

One solution is to have each participant 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. Figure 7 shows this pathological case where the delay caused by the HTTP origin request allows a second node's timer to expire and a duplicate origin request is sent. In the figure, node A multicasts a request. Node B and C respond by setting

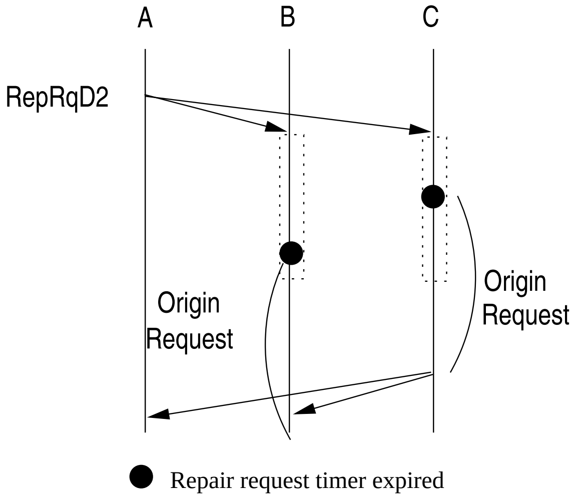


Figure 7: Lengthy HTTP response times can cause duplicate origin requests.

repair timers. Neither node has the requested data. Node B's timer expires before node C's and, having no data, it sends an origin request to the server. Because of a lengthy HTTP response time (denoted by the lower bracket) node C's repair timer expires before node B receives the data and multicasts it to the group. Consequently, Node C does an unnecessary origin request.

4.3.1 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 a 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 time may also be sending origin requests may also be sending origin requests may also be sending origin requests rather than a repair timer. If a node's reply-pending timer goes off, 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 cancelling their outstanding reply-pending timers.

Figure 8 illustrates how a reply-pending message can suppress duplicate origin requests. When node B's repair timer expires, it not only initiates a origin request but also multicasts a reply-pending message to the group. Upon receipt of the reply-pending message, Node C cancels it's reply-pending message.

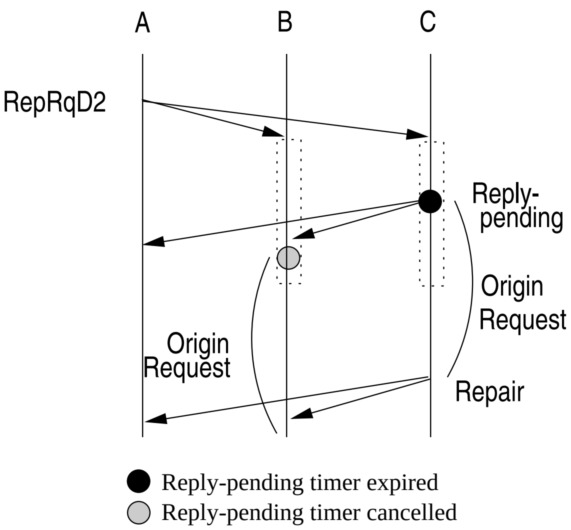


Figure 8: MASHCast Reply-pending message suppresses duplicate origin requests.

As was done for SRM repair-request and repair messages, we must decide between a cancellation or backoff policy for reply pending messages. Figure 9 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 first 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 double the delay. In the figure, a shaded circle represents the cancelled timer and a second, long dashed box denotes 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 an exponential backoff policy. Although 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

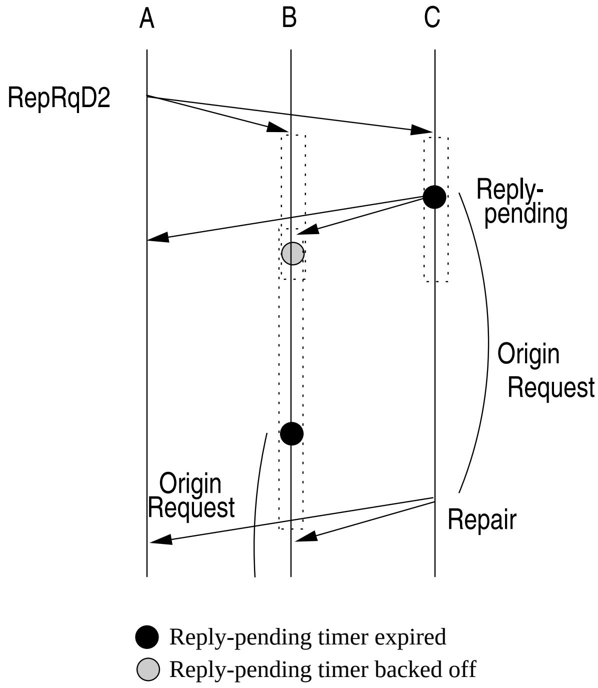


Figure 9: Reply-pending backoff can cause duplicate origin requests.

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.

4.3.2 Selective Reliability

In a collaborative webcasting session, participants will likely desire that their cache reliably receive all webcasted pages so that their local browser can faithfully render the most recently webcasted page. However, if the Web Cache component of MASHCast is used apart from the Casting Director as a passive multicast web cache, participants may not desire each page to be reliably delivered. Rather, they might desire only those pages that they themselves have requested be reliably delivered. In this way, the local participant can preserve bandwidth and storage space for the data they are most interested in. This is only a passive policy, however, in that data not locally requested that does happen to reach the cache is not rejected.

SRM allows the application to implement such a selective reliability policy. Before repair-requests are scheduled, the SRM layer determines the application's interest in recovering the missing data via an application-level callback. We take advantage of this by specifying that only those pages that the browser has requested be repaired when the Web Cache component is used as a passive cache. If the application indicates that reliability is not desired for certain data, neither a reply-pending message nor a repair-request message is scheduled to repair that data.

4.4 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. The suppression interval over which the repair delay is chosen is a function of the participant's estimated distance from the requester. 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.

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

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.

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

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:

$$[E_1 d, (E_1 + E_2) d] \tag{3}$$

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. Although we have not yet studied the behavior of these terms, 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. What data would be ideal and how to gather are currently open questions. We discuss this issue more in section 7.

Figure 10 shows two alternative interval configurations. 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

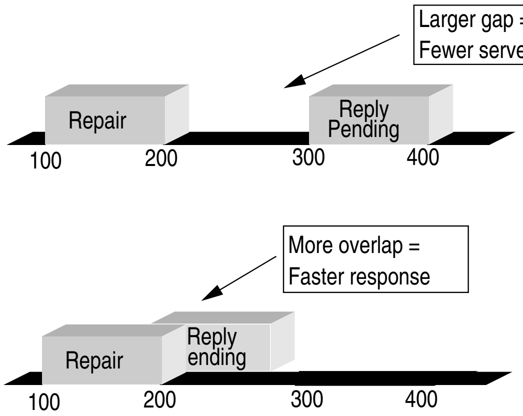


Figure 10: Timing tradeoff

hits by maximizing the chance that a participant with the requested data will send a repair, if not finish, before other participants' reply-pending timers goes off. 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 10 shows overlapping intervals which could improve response time but increase the likelihood of unnecessary origin requests.

Figure 11 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 an interval configuration, repair timer delays will be shorter than reply-pending timer delays.

We have not yet studied the effect of different parameter settings on MASHCast's performance. Some alternatives for d are explored in section 7. Table 1 shows our current definition for C, D and E. We are using the default values in the MASH implementation of SRM for the request and repair parameters (C_1, C_2, D_1, and D_2).

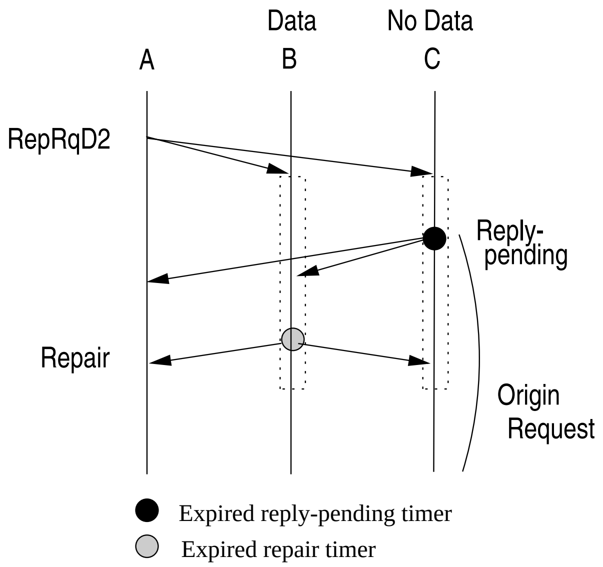


Figure 11: Two nodes responding to a request. Node C unnecessarily fetches before node B is able to fulfill the request.

Request	Repair	Reply-pending
$C_1 = 1$	$D_1 = 1$	$E_1 = 2$
$C_2 = 2$	$D_2 = 2$	$E_2 = 3$

Table 1: **Current values for MASHCast interval parameters.**

5 MASHCast 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 — within the MASH [13] shell. 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) [13]. Further, C++ code can invoke OTcl code and vice-versa. We leveraged the existing MASH components and the Tcl libraries to lessen the implementation effort. Table 2 shows the borrowed components and their uses. The Multicache's SRM extensions described in section 4 are implemented using C++ with the exception of our loss-detection data structure which, for ease of implementation, we wrote in OTcl. We created the reply-pending message object by subclassing the SRM reply message object and introducing `E1` and `E2` as new object data. In order to use URLs as our data naming convention, we needed to map URLs to the cached web page data. We store the

Module	Use
MASH SRM Object	Used as the base for our custom SRM protocol base.
MASH TCP Server Object	Intercept URL requests from the browser.
	Used for communication among Casting Directors.
Tcl 8.0 HTTP Library	Send and receive GET requests to and from the HTTP server after winning a reply-pending election

Table 2: **Modules and their communication uses.**

data in a hash table and use the hashed URLs as keys. The Casting Director object is implemented in Tcl as a simple subclass of the MASH TCP Server object.

5.1 Performance

To get an idea of the behavior of our initial implementation, we performed a simple study. We selected eight web sites from geographically distant areas. We determined round-trip times from our host machine `vine.cs.berkeley.edu` to the remote sites with the `ping` program, which computes round-trip times by sending echo packets to network hosts. The web sites, their home page's size in kB, and their corresponding round-trip times are in Table 3.

WWW Site	RTT in ms	kB
www.cs.berkeley.edu	2	17
www.cs.ucsd.edu	33	60
www.cs.utah.edu	39	81
www.cs.iastate.edu	70	74
www.cs.duke.edu	71	34
www.cs.brown.edu	83	20
www.cs.ucl.ac.uk	210	11
www.cs.princeton.edu	285	32

Table 3: **Sites used for measurements.**

We ran four experiments for each site. Each experiment measured the time to individually fetch each site's home page. The experiments and 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 one MASHCast application and a browser on `vine`. We requested a site's page and measured the time to fetch the page. This measured the overhead to load a page into MASHCast's 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.** We started a MASHCast application on `vine` and primed its cache with the appropriate site's page. We then started a second MASHCast participant on `berryman` (in the same subnet and .24 milliseconds away) and sent the page to the session a second time. Because the first participant had the page in its multicache, it should have responded 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 ten times and the results averaged. Figure 12 shows the load times by site for each experiment. As we expected, the load times for both Hit and Peer Hit are faster than either the Direct or Miss.

The advantage gained by using another participant's cache is represented by the difference between the Peer Hit bar and the Direct bar. Similarly, the advantage gained by the local own cache is represented by the difference between the Hit bar and the Direct bar. This information is graphed in Figure 13. The upper line graphs the seconds saved by hitting in the local cache. The lower line shows the seconds saved by hitting in your peer's cache. Figure 13 highlights how MASHCast can enhance the performance of collaborative webcasting application and improve the user's response time. These response time improvements are also applicable when using the WebCache component as a passive collaborative cache without the Casting Director.

Figure 12 shows another trend that is worth a closer examination. As the round-trip times increase, we expect the Direct and Miss times to increase. However, the Peer Hit times should not increase as a function of round-trip time since they should not be contacting the origin server. Rather, they should be a function of the page size. Moreover, it is reasonable to expect that they are a multiple of the Hit time. Table 4 shows that the Peer Hit load times do not appear to be a function of the Hit times. (The table also repeats the round-trip times and page size for comparative purposes.) It also appears that they are not strictly tied to round-trip time or size. We suspect that this is a side-effect of inefficiencies in our choice of reply-pending interval parameters. The cause of this effect remains open.

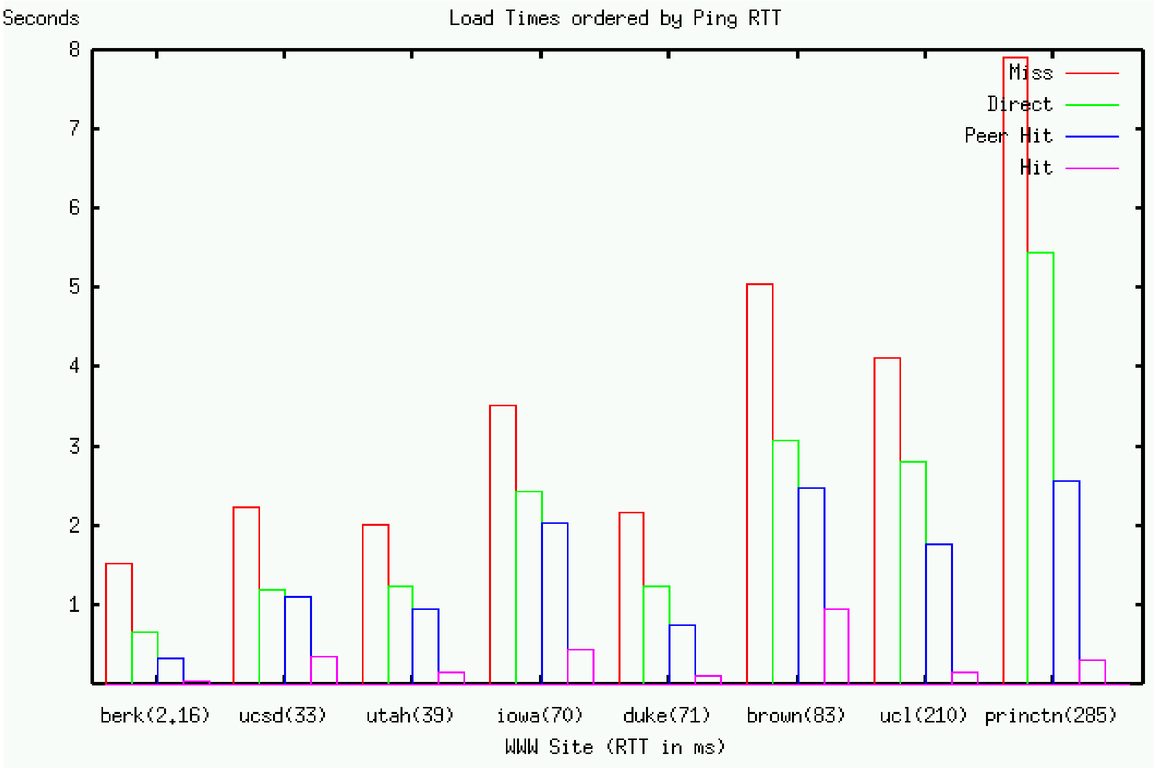


Figure 12: Page load times for each of 4 experiments by site. The round-trip times are in parenthesis beside each site. As we expect, load times generally increase with RTT.

5.2 Future Implementation Work

Although session messages are being multicast from session participants, we have not yet added information regarding what data MASHCast participant have sent out. Rather, when data is missing before the end of the page, a tail loss is assumed. This might lead to premature repair-requests in our current implementation because any data fragment that comes in before the last fragment in a page will cause a repair-request to be scheduled. If the repair-request timer expires before the tail has been received, a spurious repair-request is multicast.

As with any cache, MASHCast's web cache can fill to capacity so a cache victim policy is necessary to determine what data to throw out of the cache. MASHCast currently has no intelligent cache management policy. It would be interesting to study the effectiveness of both a random cache policy, thereby increasing the likelihood of at least someone in the group having a desired page in their cache, or a more traditional Least Recently Used policy. The effectiveness of the cache policy

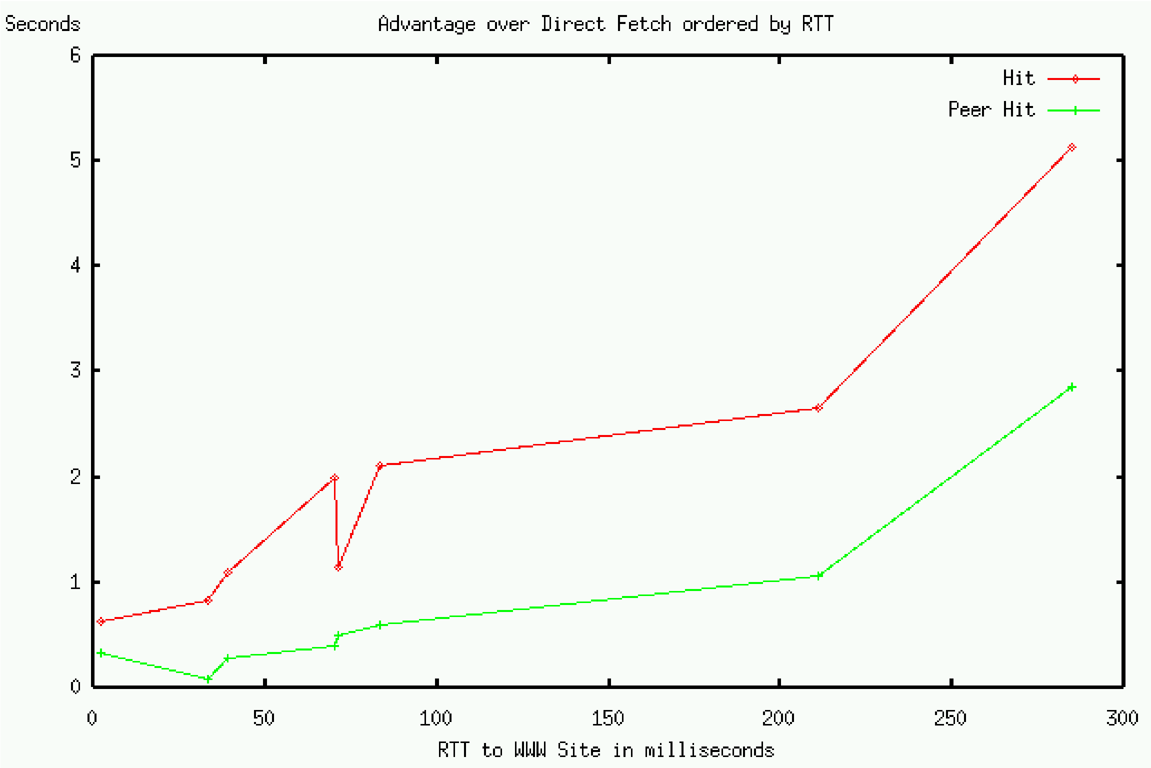


Figure 13: Seconds Saved Ordered by Round-trip Time.

could be related to the mode in which the MASHCast components were being used. For example, a distance-learning lecture program might perform better with a LRU policy, but a passive web cache program might benefit from a random policy.

Currently, the GUI with which a sender can webcast pages is not integrated into the Casting Director. The user must start the GUI separately.

Selective reliability is not yet implemented.

6 Related Work

Work is ongoing in both academia and industry. In this section we discuss related projects and how we differ from those projects.

Webcast [2] uses NCSA Common Client Interface to communicate with the Mosaic web browser via TCP/IP to fetch URLS, ask 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,

WWW Site	Peer Hit:Hit	RTT in ms	kB
www.cs.brown.edu	2.59	83	20
www.cs.ucsd.edu	3.05	33	60
www.cs.utah.edu	6.30	39	81
www.cs.iastate.edu	4.62	70	74
www.cs.duke.edu	6.78	71	34
www.cs.berkeley.edu	8.55	2	17
www.cs.princeton.edu	8.4	285	32
www.cs.ucl.ac.uk	11.01	210	11

Table 4: **Peer Hit Ratio.** The second column lists the ratio of Peer Hit load time to Hit load time. Although we expect to see some relationship between the two, there to be no relationship.

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. The RMP project has also spun off a commercial company called GlobalCast [1].

mMosaic [4] uses a modified Mosaic browser to send documents out via multicast. Reliability is achieved via an announce/listen protocol [18]: 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 [15] solution 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] 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].

7 Future Performance Analysis

Our implementation shows that the design is feasible and has promise, but more work is needed to fully evaluate the resulting performance and refine the protocol machinery. While the simple experiments described in this paper show that leveraging peers' multicaches can be beneficial, further study is needed to determine if our protocol meets the scalability goals we have described: scalability with respect to both network traffic and the number of generated server hits.

In anticipation of this work, we included mechanisms to continue our study: tracing facilities within MASHCast and an extension to the packet tracing facility `tcpdump` that understands MASHCast's packet format. The next step is to vary delay timer parameter choices (E_1 and E_2) and evaluate the effects in response time and traffic.

In addition to the parameter choices above, we would also like to study the choices for determining the bias for the reply-pending interval (d). The choices include:

- **The estimated distance to the requester.** Using the estimated distance to the requester would optimize the response time for the requester (this is the logic used for configuring the repair timer interval based on the estimated distance to the requester). A potential problem with this choice is that distance to the origin server may dwarf the distance to the requester.

- **The estimated distance to the origin server.** This choice would minimize the time to begin multicasting but might not be scalable if requests are being made to a number of web servers. However, it seems feasible that many types of webcasting sessions enjoy a large amount of server locality. If this is the case, then saving data about just the last web server contacted and using that might yield great results. Statistics would need to have a lifetime such that they reflect the origin server's current load.

- **Shared-learning for origin server distances.** Because webcast responses are generated by a random of the group, a single node is unlikely to fetch successively requested documents. This makes the likelihood of the last-contacted origin server being the current server decrease as sessions grow large. Since all data originates from some participant who did contact the origin server, estimates of the time taken to contact that server could be included in the packet header. server. This option could become complicated, however, if the receiving node needed to adjust the distance to the server depending upon its distance to the participant that sent the repair.

- **Use SPAND or a similar service.** SPAND [17] enables applications to report and query observed server performance to remote network nodes. If several participants in a close area share a SPAND server, they can benefit from the other's knowledge without increasing the amount of multicast traffic. If other applications besides MASHCast are also registering their knowledge of the network with a SPAND server, MASHCast can benefit from their knowledge, also.

8 Conclusions

We have described a webcasting architecture which features a decomposable architecture and a custom protocol built upon the SRM framework. Our protocol is novel because we leverage the SRM framework to tailor the protocol to the specific needs of webcasting. The result is a protocol which is scalable with respect to server load and link traffic. We have implemented our design in an application called MASHCast. Initial experimentation shows that the MASHCast can leverage data from peers' multicast caches effectively, resulting in reduced server traffic and less wide-area traffic.

9 Acknowledgments

Yatin Chawathe conceived the URL mangling scheme which frees the user from having to specify an HTTP proxy when using MASHCast. In addition to Yatin, I'd like to thank three fellows who have influenced this work greatly: my son David, who was considerate enough to be a wonderful little boy so that I could accomplish at least some work — but not much — while watching him; my advisor Steven McCanne who provided not only guidance but an academic example to which I can only hope to aspire; my husband, Alan Domonoske, without whose unfaltering love and constant support, neither this nor most other things in my life would be possible.

References

[1] GlobalCast. Information online at http://www.gcast.com/.

[2] Ed Burns et al. Webcast. Information online at http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/webcast.html.

[3] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM '87*, September 1990.

[4] Gilles 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] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol - HTTP/1.1. Request for Comment 2068, IETF Network Working Group, 1997.

[7] Sally Floyd, Van Jacobson, Steven McCanne, Ching-Guang Liu, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *Proceedings of the 1995 ACM SIGCOMM Conference*, August 1995.

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

[9] V. Hardman, M. A. Sasse, M. Handley, and A. Watson. Reliable Audio for Use Over the Internet. In *INET '95*, Ohahu, Hawaii 1995.

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

[13] 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.

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

[15] P. Parnes et al. The mweb presentation framework, Sophia Antipolis, France, October 1996.

[16] S. Raman and T.-L. Tung. Mediaboard using the Scalable, Reliable Multicast Toolkit. U.C. Berkeley CS262 Operating Systems term project and paper, December 1996.

[17] M. Stemm S. Seshan and R. H. Katz. SPAND: Shared Passive Network Performance Discovery. In *Usenix Symplosium on Internet Technologies and Systems*, December 1997.

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