Reliably Networking a Multicast Repository

Wang Lam Hector Garcia-Molina
Stanford University
{wlam,hector}@CS.Stanford.EDU

***Abstract*— In this paper, we consider the design of a reliable multicast facility over an unreliable multicast network. Our multicast facility has several interesting properties: it has different numbers of clients interested in each data packet, allowing us to tune our strategy for each data transmission; has recurring data items, so that missed data items can be rescheduled for later transmission; and allows the server to adjust the scheduler according to loss information. We exploit the properties of our system to extend traditional reliability techniques for our case, and use performance evaluation to highlight the resulting differences. We find that our reliability techniques can reduce the average client wait time by over thirty percent.**

I. INTRODUCTION

Data dissemination to many users remains, despite the growth of network capacity over time, an expensive proposition for all but the best-funded servers. When an information server becomes popular, many users often show up requesting the same data of the server, or requesting data that overlaps with the requests of other users. For example, a Web server may find that it repeatedly sends some same items to users (such as a Web site's front page and its cited images).

Where multicast-capable networks (such as IP multicast) are available (such as Internet2), a server can instead retain a repository of data items and offer them over a multicast facility, so that users can use a corresponding multicast client to request the subsets of data items that they are interested in fetching. Such a multicast facility can send the same data once over multicast to satisfy multiple users' requests for it simultaneously, dramatically reducing the waste of network resources and lowering the corresponding network costs.

The idea of such a multicast facility is well-established, having been envisioned and studied for Teletext [1], Datacycle [2], and broadcast disks [3], among others. As a local case in point, we are interested in creating a multicast facility for our WebBase project, which crawls the Web to create a repository of Web pages for research. We would like to offer our repository to other researchers so that they can request subsets of our repository in a simple and efficient way, using a multicast facility that might look like Fig. 1. (There are other benefits to a multicast-distributed crawled Web repository, including less load on the individual Web servers that get crawled for multicast distribution, that are documented in earlier work for this multicast facility [4].)

For such a data multicast facility to be useful, however, clients must be able to receive all their requested data; the facility cannot drop bits to its clients because the resulting partial data may be unusable. Existing multicast networks, such as the IP multicast backbone (MBone), may not guarantee this; to the contrary, IP multicast is only a best-effort datagram service and prone to packet loss. One report [5], for example, found that IP multicast sites lose anywhere from less than 5% to more than 20% of the packets they request from a fixed-throughput multicast source. In short, data sent from a server over multicast may not reach all—or any—of its clients; the multicast facility must make special arrangements to ensure that its clients receive their data.

As a result, a multicast facility transmitting loss-intolerant data (such as we describe above, in contrast to loss-tolerant video or audio) will be unusable on existing multicast networks unless its data transmission can be made reliable.

In this paper, we will consider the design of a multicast facility for such lossy multicast networks. While there is existing work in general-case reliable multicast, peculiar properties of our multicast facility open up a number of new possibilities that were not available to generic reliable multicast, which we will study here.

Current approaches to reliability hinge on two basic ways to cope with loss on the network: on-demand retransmission of data to "make up" for lost data, and addition of precomputed redundant information to the data so

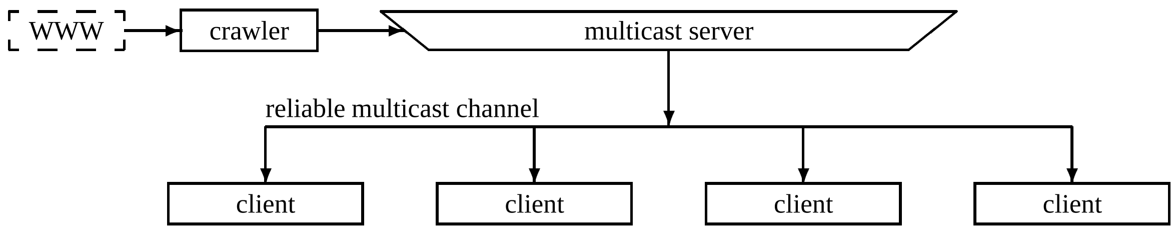

Fig. 1. The WebBase Multicast Facility

that lost data can be reconstructed using the redundant additional data (called "forward error correction," FEC). Our multicast facility adds one counterintuitive technique not available to generic multicast—ignoring the losses. Because the multicast facility schedules data for transmission as it is requested, it is possible to have clients "give up" in the face of loss and simply rerequest the data item for a new transmission later. Because the data may well be retransmitted later anyway, for other clients requesting the same data, this scheme could be more efficient. In effect, clients can get the reliability of retransmission, without the server ever committing valuable network resources to specific "make-up" or retransmission packets.

Also, because different data items are requested by different clients, and by varying numbers of clients, we can attempt to exploit these variations among data items using new, dynamically adjusted reliability schemes. Lastly, we can create hybrid approaches that combine multiple reliability schemes, new and traditional, to improve performance.

Current approaches to reliability have also typically centered on the transmission of a single data item to clients requesting it. Unlike such work, we expect clients to request (need) multiple data items at a time, and choose our metric to better match this bulk-download case. This difference in metric requires us to consider not only how long a multicast server takes to successfully transmit a data item to its clients, but also how a method for doing so will impact the transmission of other data items that clients still need.

In this paper, we will present new extensions to existing reliability techniques, and define a metric by which to measure their performance. Finally, we will use simulations to evaluate these techniques, and to address the following questions:

- When clients lose data, should a multicast server send retransmission packets or simply schedule the data item for later transmission? Would it be advantageous for a server to send retransmission packets selectively?
- Is forward error correction effective in our multicast facility, even though the data is latency insensitive, and clients can wait for retransmission? Does a dynamically-adjusted forward error correction scheme improve performance?
- If forward error correction fails to recover all lost packets, should the multicast facility revert to retransmitting lost packets or rescheduling the data item for later?
- When a new client connects to a multicast server, requesting a particular data item already being sent at that instant, should the server have the client pick up partial data and request retransmission, even if it requires more implementation complexity and causes more retransmission traffic? Or, should the server not inform the client of the transmission and make the client wait for the data item's next transmission?
- Can the multicast scheduler exploit client data-loss information to improve performance? For example, would performance improve for a scheduler that uses client loss rates to more accurately estimate the time a client needs to receive a data item over its lossy link? If so, how much improvement would such a modification gain?

We will outline in Section II a possible network traffic layout that isolates the network-consuming data transmission itself for further study and optimization. In Section III, we describe traditional approaches such as retransmission and forward error correction, and introduce new possibilities that are possible for our multicast facility as suggested above. In Section II-A, we will propose a simple client delay metric to gauge the performance of our design decisions. In Section IV, we describe the simulation we use to evaluate our design options. In Sections V and VI, we will examine some results from our study to form suggestions for the design of a reliable-multicast facility over an unreliable multicast network.

II. The Multicast Facility

In our multicast facility, the multicast server has a number of data items (intuitively, static files) ready for dissemination, from which each client will request some (not necessarily proper) subset. The server breaks each data item up into a number of same-size packets.

A new client requests a subset of the server's data items using a reliable unicast connection to the server. Because reliable unicast connections are widespread (e.g., TCP), we will assume the connection exists and always delivers its data.

To minimize the consumption of clients' download links, we should separate the server's traffic into a low-bandwidth *control channel* announcing data items, and a *data channel*, consuming the server's remaining outgoing bandwidth, to carry the data items themselves.

All clients would always subscribe to the control channel, to receive announcements of data items on the data channel. Clients would use the announcements to determine when to subscribe to the data channel, so that they receive the data items they request and skip the ones they didn't request.

Clients must receive control channel traffic reliably and promptly, so the channel must use a latency-sensitive low-bandwidth reliable multicast protocol to ensure its delivery. The simplest solution for this channel is to use an existing reliable multicast protocol to ensure the probability of loss is negligible; protocols have been proposed for this type of application, including SRM [6].

On the data channel, on the other hand, the traffic is relatively high-bandwidth and latency-insensitive: that is, performance may suffer if packets take longer to arrive, but there is no deadline before which packets must arrive to be useful. Even more notably, data channel traffic has two properties, enumerated below, that are particular to this multicast facility and a related system called "broadcast disks," in which a server sends small single-item requests on a local broadcast network to satisfy local clients' requests. (Broadcast disks are briefly described in Related Work.) As we shall see later, we can attempt to exploit these properties for more efficient reliable transmission:

- The information on the data channel is "interesting" to only a (varying) subset of the server's clients at any one time. Unlike the control channel, the data item being sent on a data channel is probably needed by some but not all of the server's clients. As a result, we may want our reliability scheme for the data channel to vary in some way with the clients that actually need the data on the channel at the time. We consider some of these possibilities in Section III.
- Any information on the data channel can, and is likely to be, retransmitted at a later time as a result of new requests from new clients. This is a stronger claim than saying that the data channel is latency-insensitive and so we can always resend lost packets; this claim actually says that if a client does not receive a data item successfully from the data channel, the multicast server could decide to not retransmit the client's losses at all, instead compelling the client to wait the next time the entire data item is transmitted again.

When a client has received all the data items it requested, it disconnects from the control channel and its request is considered complete. The sooner a client receives its data, the sooner a user is happy and able to use the data, so we will use this as our performance metric.

A. Metric

In a number of scenarios—such as researchers analyzing subsets of a Web repository, software updates being saved and applied, and media downloaders downloading material to burn to disc or export to a portable device—the measure of performance users care about is that of client delay, the time it takes a client to receive all the data it requested. This simply reflects how users often request multiple data items because they need them all, and because even where it is not strictly necessary to do so, it is often more convenient to batch-process data than incrementally process it.

Because of this, we will focus on client delay as our performance metric. We define this notion below. It is intuitively similar to the notion of client delay defined in [4].

For a multicast server with n data items $D = \{d_1, d_2, d_3, \ldots, d_n\}$, and k clients $C = \{c_1, c_2, \ldots, c_k\}$, each client c_i is characterized by the data items it requests, $R_i \subseteq D, R_i \neq \emptyset$, and by the time at which the client makes its request, t_i.

We define the *client delay* of a client c_i using the earliest time $T_i > t_i$ when the client has all the data items it requests R_i. We simply say that for this client, its delay is $d_i = T_i - t_i$.

For k clients C, we define the *average client delay* over all clients as $d = \frac{1}{k} \sum_{i=1}^{k} d_i$.

We might also measure the network usage of the multicast facility, and use it as a metric; unfortunately, this is not particularly enlightening because a multicast facility striving to serve its clients will always be using the network whenever there is new, requested data to distribute.

B. Multicast Operation

The server operates by gathering the clients' request subsets and using a scheduler, such as R/Q [4], to decide which requested data item to send on the data channel.

The R/Q heuristic is designed to help minimize average client delay, defined in Section II-A, and operate quickly even when scheduling large numbers of data items. In this heuristic, for each data item i, the server determines R_i, how many clients are requesting the data item i, and Q_i, the size of the smallest outstanding request for a client requesting that data item i. The heuristic chooses to send a data item with the highest R_i/Q_i score.

Example. Suppose three clients, A, B, and C have pending requests on a multicast server. Client A needs three data items, numbered 1, 2, and 3; B needs items 2 and 4, and C needs items 2 and 3.

A multicast server implementing R/Q would determine each of the item's R/Q scores, which begin as (in increasing order of item number) $\frac{1}{3}$, $\frac{3}{2}$, 1, and $\frac{1}{2}$. In absence of other clients, then, the server would send item 2 first, then item 3, then the remaining two items in arbitrary order (after items 2 and 3 are sent, both items 1 and 4 have R/Q score 1).

As we can see, the heuristic helps minimize client delay because it sends the items that are most requested, and most quickly help to satisfy a client request, first.

To keep the computation simple, let us say each item takes one unit of time to send on the data channel, and arrives without loss or delay. Let us suppose the R/Q implementation chose to send item 1 before item 4. Then the client delay for the three clients would be $d_A = 3$, $d_B = 4$, and $d_C = 2$, for an average client delay of $d = 3$. (If the implementation chose the reverse, the average client delay is the same.) In this simple example, this is an optimal average client delay, preferrable over the other possible values of $d = \frac{10}{3}$, $\frac{11}{3}$, or 4. □

III. Making the Data Stream Reliable

To protect an arbitrary data item from loss over an unreliable data channel, the server can apply several complementary approaches. First, the server can try to prevent the data item from being lost, by sending redundant packets of (error-correcting) data with the data item so that clients can use redundant packets to recompute the data in some lost packets (forward error correction). Next, the server can react to lost packets by retransmitting data for the data item as needed (retransmission). Finally, clients that still do not have the data item can have their request for the data item added back to the multicast server's request list, so that the scheduler can try to send the data item again in the future (rescheduling).

A. Forward Error Correction

One well-known approach to reliable multicast is to add a predetermined amount of error-correcting (FEC) data to the data being sent, so that if some of the data is lost during the transmission, receivers (clients) can use the error-correcting data they receive over the data channel to mathematically reconstruct the lost data. Typically, the redundant packets are constructed using error-correcting codes (or more specifically, erasure codes) that allow any subset of the packets to be used in reconstructing data. Packets made using such codes require more precomputation than would simply cloning particular data packets, but the error-correcting packets are more widely usable than the simple copies.

We can apply forward error correction to a multicast server in a number of ways.

- We can choose a fixed *expansion factor* $f > 0$ for the server. For d data packets, the server would then send an additional $\lceil fd \rceil$ error-correcting packets.
- We can have the server increases its expansion ratio with the number of clients interested in the data item, such as *expansion factor* $= fR$, for R clients and a parameter $f > 0$. For d data packets, the server would then send a total of $\lceil (1 + f)d \rceil$ packets of data and redundancy. Intuitively, we are backing more popular data items with more redundancy so that its many (more) clients are less likely to lose the data item.
- We can make the expansion factor inversely proportional to the number of clients interested in the data item, such as *expansion factor* $= f(R)^{-1}$, for a parameter $f > 0$. Then, the server would send a total

of $\lceil (1 + \frac{f}{R})d \rceil$ packets of data and redundancy. Intuitively, we are backing less popular data items with more redundancy, because their repeated transmission would be more costly to the client delay of other clients. Also, more popular data items are likely to be scheduled for transmission again shortly, as new clients request them, so there is less delay in losing, and waiting for, a popular data item.

- We can attempt to match the expansion factor to a factor of the maximum estimated loss rate any client requesting the data item is suffering. (The server can determine a client's average loss rate simply by determining what percentage of packets sent to the client trigger NAK responses.) That is, if of all clients requesting data item i, the client losing the most packets has a loss rate of l, then we can choose an expansion factor of fl, for a parameter f presumably near one.

We will denote the forward error correction scheme, with expansion factor f, as FEC(f). For example, to denote a varying FEC scheme in which the server increases the expansion factor from zero, by 1% per client requesting a data item, we will use FEC($0.01R$).

Should forward error correction fail, because a client does not receive enough redundant packets to reconstruct an original data item, the multicast facility must fall back on some other reliability scheme, such as the two schemes described next, so that the client's request for the data item is eventually satisfied.

B. Retransmit

Another well-known approach to reliable multicast is for the server to receive NAKs from clients indicating their lost data, then retransmit lost data so that clients have another chance to receive it.

1) Retransmissions Are Multicast: We notice that the server must consume the same network resources to send these additional packets unicast or multicast: It must consume the same number of bytes on its network link to do so, a resource that could not be spent sending other data. Because multicast can benefit multiple clients at the same time, though, it is therefore to the server's advantage to always send its additional packets of data over the data channel, where it originally sent its data items. As a special case, if only one client needs the additional packets, sending those packets unicast is equivalent to sending them multicast, except that over a reliable unicast stream (TCP), we are locked into TCP's retransmit-as-needed reliability scheme, rather than being able to take advantage of the design choices we make for the data channel. Therefore, we assume below that any additional packets of data the server must send will be sent over the data channel.

2) Retransmissions Use Error-Correcting Codes: The simplest implementation of retransmission could determine which packets clients need from the NAK, and retransmit exactly those packets again. As an enhancement, instead of actually resending the lost data, a server using a retransmission scheme should send precomputed error-correcting packets for the data instead, because a same-size error-correcting packet can "make up" for the loss of (allow the reconstruction of) one data packet, even if different clients lost different data packets.

Using error-correcting packets allows the server to send no more error-correcting packets than the largest number of data packets lost by one client, even though the union of all data packets lost by all clients may have larger cardinality. This scheme is mentioned, for example, in MFTP [7].

3) Clients Unicast One NAK Per Data-Item Transmission: In our multicast system, clients will send a NAK only at the end of a transmission of a data item, when it has already attempted to use whatever error correction packets it has already received, and when it has determined the packets it still needs to complete its copy of a data item. We require NAKs to be sent this way for two reasons: The first is that having clients send a separate NAK for each packet lost can collectively fill the network links to a multicast server more quickly than if they sent their NAKs in larger aggregate. The second is that retransmission may be used as a reliability scheme only after variable forward error correction fails to make up for client losses; if so, only after the client has tried to receive all the sent packets can it determine whether it actually needs any more packets at all, and if so, how many the client actually needs to reconstruct a data item.

The NAK is sent unicast instead of multicast because only the server needs to receive the NAKs; unlike some other published reliable multicast schemes such as SRM [6], in this system one client sending a NAK does not prevent any other client from sending its NAK. It is perhaps worth noting that unicasting NAKs rules out SRM-style schemes where clients can transmit data to fill NAKs in place of the server, but by restricting retransmissions to the server, we also easily avoid accidental or malicious data corruption from clients sending mangled retransmits to each other.

If all clients send their NAK at the same time following the end of a data item's transmission, then the instantaneous spike in network traffic to the server may flood the server's incoming link and cause NAKs to be lost or delayed. To prevent this "NAK implosion" on the server at the end of each data item, each client delays its NAKs by a random time, chosen uniformly from a small interval (e.g., as in `wb` [6]).

Because error-correcting packets are themselves sent on the data channel, they too can be lost. Therefore, when an announced retransmission concludes, clients may again send NAKs for more packets they still need, and the server may accommodate. The server could complete as many rounds of retransmission as necessary for its clients to reconstruct the data item, to guarantee reliability.

4) Selective Retransmission: In our multicast facility, we can apply our retransmission scheme as we have describe it so far, but we have other options. In particular, the server can choose to ignore NAKs and reschedule the affected clients, in effect using rescheduling (described below) as a backup for a weak-retransmission scheme. With this flexibility, we could try to choose when NAKs should be best handled by retransmission, and when NAKs should be best handled by rescheduling the client.

In particular, we can choose to honor only retransmissions of small size, on the intuition that they have little time cost compared to the cost of making the affected clients wait for the next scheduled transmission of the data item. The server could begin retransmission, for example, only when it receives a NAK of no more than p packets, and reschedule otherwise.

When the server is retransmitting, since it is already delaying a new data item to retransmit packets for an old one, the server could also accept and coalesce subsequent NAKs so that those NAKs are also satisfied. Outside of this retransmission period (i.e., if the server does not notice any clients nearly completing a data item, or if the server has already finished helping a client complete a data item), the server ignores NAKs from clients. In effect, a server is "convinced" to honor retransmission requests only if doing so completes a very-nearly-completed data item.

We will denote the retransmission scheme with R(∞), if the server retransmits for any NAK. If the server sends retransmissions for a data item only if it receives a NAK of no more than p packets as described above, we will call it R(p).

TABLE I
SIMULATION PARAMETERS AND THEIR BASE VALUES

Description	Base value
Number of data items available	100
Size of a data item (in packets)	60
Number of data items requested per client (mean)	9
Time between new clients	4 000 ms (4 sec)
Server link loss rate	0%
Client link loss rate	5%
Time to send one packet	80 ms
NAK delay time (range)	[30ms - 210ms)
NAK send time (minimum)	80 ms
NAK send time (mean)	100 ms
Simulated time (length)	86 400 000 ms (1 day)

C. Reschedule

The server can decide not to change its data transmission to accommodate losses. Instead, it would receive NAKs from clients as indications that the client needs the data item again, and add back the client's request for the NAK'd data item. The scheduler would decide when to reschedule the data item for retransmission. This scheme is easy to implement and requires no additional network resources, but at possible cost in higher client delay.

Rescheduling is a reliability option that is particular to this multicast facility: even if we choose this nearly "do nothing" scheme, we can still assert that clients will eventually receive their entire data request, because the scheduler holds the client's data request.

We will denote this reliability option as R(0). Intuitively, the notation considers a reschedule-only scheme to be equivalent to a retransmission scheme that sends additional packets only if the server receives a NAK of zero packets—a NAK that would never be sent.

IV. SIMULATION

To assess the performance of these different possibilities, we turn to simulation to predict their effect on a multicast system under a variety of loads. We describe the simulation in this section, then present the results of the simulation in the next.

We assume here a simple star-shaped topology for the multicast network as an approximation of the real multicast backbone's (MBone's) loss behavior, as suggested in [5]. In this topology, the server and each client have separate lossy links to the multicast backbone. Because the multicast "backbone" was observed to lose relatively few packets that correlate among multiple but not all clients, functionally we can approximate the core "backbone" as reliable and push the dropped packets to server and client links. Therefore, the backbone is represented as a single node that simply connects all the lossy links.

We assume also that data transmissions are nonpreemptible; that is, once a server decides to send a data item, or send retransmissions for a data item, it will send all of that chosen transmission before choosing to send something else. This reduces the number of announcements that the server must make to its clients, and reduces the unneeded (irrelevant) traffic that clients receive from the data channel.

Because a multicast facility is uninteresting when it is not loaded enough for client requests to overlap, we will consider a hypothetical high-load scenario that would be impractical to service using unicast (TCP) delivery alone. One can easily imagine a variety of such scenarios, such as

- a newswire over a low-throughput wide-area wireless network;
- an ISP reserving a small portion of its network capacity to deliver its own popular content to its subscribers;
- a software vendor disseminating widely needed patches for its products; or,
- a government sending reports and instructions to its diplomatic missions through expensive and full satellite links.

In Table I, we enumerate the parameters for our simulation's *base case*. In this base case, we try to portray a Web server for a small business that faces a sudden increase in requests over its limited network connection. In our experiments, we vary some of these base values.

In this scenario, a Web server linked to the world over a relatively small business "broadband" connection with IP multicast, is suddenly being deluged with requests because of its newfound popularity. Normally, using only unicast traffic, such a Web server would be crippled by a large spike in requests, because its outgoing link would be divided by so many requesting Web clients that none of them get enough throughput to make progress. The Web server would appear down, and be unable to provide any service at all for as long as it remains popular—the time when its service is most needed.

Fortunately, in our scenario, while the server's main Web page might be delivered to clients over unicast using HTTP, most of the requested data by volume—the images for the Web page, the animated vector graphics, and the Web client scripts—are sent using a multicast facility, and Web clients have plug-ins that support this method of download.

The multicast server could be busy sending about a hundred such very popular items, from images to scripts, to clients making new requests every few seconds. (In Table I, we arbitrarily choose four seconds, a number small enough to induce numerous simultaneous clients.) Clients would request about nine such items on average to fill a typical Web page request.

For these kinds of items (images, vector animation, and scripts), we estimate an average size of about 30 kilobytes each, which would be broken up into packets of 512 bytes each. The packets size is necessarily so small because Internet hosts send UDP packets over IP multicast, and IP(v4) requires UDP packets of only up to 576 bytes to be supported for delivery. (Larger packets may be fragmented or dropped over individual network links.) Consequently, some popular UDP-based services such as DNS restrict their packets to a maximum 512 bytes of payload, to ensure their operation over IP. Our hypothetical multicast server would be wise to do the same, so the server's 30-kilobyte data items are divided into sixty 512-byte packets each.

The multicast channel is assigned a relatively small sliver of outgoing network throughput so that even Web clients running behind modem connections can keep up. At 50 kilobits per second (just under the 53 kilobits per second at which the fastest POTS-line modems can currently download in the United States), the multicast data channel could send 512 bytes (one packet of data) about 50 * 1024 / 8 / 512 = 12.5 times per second. This means one packet is sent every 80 milliseconds.

The NAK delay-time range in Table I is chosen to match the NAK delay-time range described for `wb`, an already-available whiteboard application for IP multicast; a multicast client chooses a delay time for each NAK uniformly randomly from the given range of delay times. Lastly, we estimate the mean NAK send time, the time it takes a NAK to arrive at the multicast server after a multicast client decides to send it, at 100 milliseconds, as a loose estimate of the time it takes a small packet to travel half the world's circumference over currently available IP networks.

We simulate a multicast system over a day of load to determine its performance for clients over that time.

V. Results for Uniform Loss Rates

To evaluate the reliability schemes over various conditions, we have performed numerous experiments, changing assumptions, base parameter values, and algorithms. Due to space limitations, however, we report on only a subset of our results here, and refer to the extended version of this paper for more results [8].

In this section, we consider how various reliability schemes compare when when all clients have the same data-channel packet-loss rate. This is plausible, for example, when we have relatively controlled conditions (such as large-area internal networks or experimental high-capacity networks) in which a multicast server's clients can connect to the server over comparable network links. This allows us to study the performance of our various reliability schemes when clients are of fairly similar capability. In the next section, we will briefly summarize findings for other loss models and algorithm enhancements.

A. Retransmission and Rescheduling

We first consider what happens in the simplest case, in which the multicast facility uses only retransmission (Section III-B). In Fig. 2, we compare several forms of retransmission, including no retransmission. In the figure, we graph the average client delay as a function of client loss rates. Other parameters of the simulation are as specified in Table I. The client delay is the time it takes clients to request their items, measured in seconds. The client loss rates are specified in percentage of data-channel packets lost, fixed for all clients. In this relatively low-loss scenario, the server's link to the multicast backbone is assumed to lose no packets. The plot labelled "R(∞)" represents a retransmit-always scheme, as described in Section III-B. The plot labelled "R(0)," by contrast, represents a reschedule-only server, which never sends packets in response to NAKs. The other plots represent a spectrum of selective retransmission schemes that lie between sending packets for all NAKs and no NAKs.

The plot shows, for example, that in a multicast scenario where the server loses no packets and clients independently lose 2% of packets in the data channel, a retransmit-always multicast system will have an average client delay of about 695 seconds, or about 11.5 minutes. If the system starts retransmitting only on NAKs for few

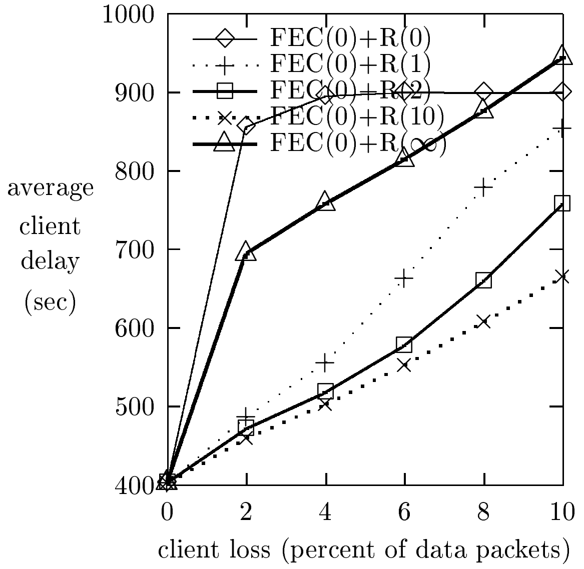

Fig. 2. Retransmission Schemes

packets, on the other hand, clients get substantially lower delay. For NAKs of up to 10 packets, for example, clients have 460 seconds delay, or about 7.5 minutes (just over half the delay they would suffer under the reschedule-only system, which does worst in this group at over 14 minutes). Increasing the size of accepted NAKs further does not dramatically improve performance, so we plot only the lines shown to reduce clutter. As we can see, there is advantage to selective retransmission, a system that retransmits principally for small losses where retransmission is short, and that forces larger losses to be rescheduled.

As we expect, the average client delay rises as clients lose more of their data; this is because clients need more attempted packet transmissions to get the same number of packets needed to form a data item.

Also, we see that for relatively low client loss rates, retransmission (FEC(0)+R(∞)) is a better choice than rescheduling (FEC(0)+R(0)). This is because clients will typically lose a few packets from each data item they request, and having just a few extra packets retransmitted is much less time-consuming than waiting for the data items to be rescheduled.

On the other hand, rescheduling outperforms retransmission for higher loss rates (starting at about 8%). Most strikingly, the performance of rescheduling only does not degrade substantially from 6% up to 30% client data-loss rates (not plotted), holding at about 15 minutes average client delay.

In retrospect, we can see why this is so: Clearly, retransmit-always servers must deterioriate over time; as clients lose more data, the server must spend more time retransmitting. Worse, as clients lose more data, they become increasingly likely to drop packets from a retransmission, compelling them to send another round of NAKs and wait for another round of retransmission. On the other hand, for reschedule-only servers, the loss rate affects only the number of times a client needs to have a data item scheduled for transmission, before the client receives enough packets to reconstruct the data item.

We also observe that limiting server retransmissions to benefit small-NAK clients can provide an improvement over both schemes discussed so far. In particular, the improvement is most noticeable when selective retransmission consistently covers the clients losing fairly few packets. For example, the 1-packet NAK scheme nets lower client delay than a retransmit-always scheme, but falters compared to more lenient selective-retransmission schemes as client losses rise above one packet per data item (which is about 2% of data-channel packets). This is because clients having a higher loss rate have a correspondingly smaller chance of losing no more than one packet for a data item. As the chance of some client losing sufficiently few packets to trigger retransmission falls, more and more data items must be rescheduled for clients to receive them in full, incurring higher delay.

Another benefit of selective retransmission is more subtle. It turns out that in practice, it is possible for new multicast clients to send large NAKs for data items they missed, causing additional delay in a retransmit-always system; this delay, selective transmission can avoid.

The reason a new client can send large NAKs is that a new multicast client connecting to the data channel will not only receive new data items, it will also receive retransmissions for data items the client missed while joining the system. This means that the client can send NAKs after receiving the partial retransmission, requesting the remainder of the data item that the client doesn't yet have. Relatively few of these clients requesting relatively large retransmissions in this way would cause delays on the data channel, increasing clients' average delay; therefore, the selective retransmission scheme, which is able to ignore these large NAKs, can reduce the impact of these requests. A selective retransmission scheme can better choose to retransmit when retransmission's cost is small, and avoid retransmission when its benefits are small.

We see, therefore, that the choice of retransmission scheme a designer should apply to the multicast facility depends on the expected loss rates clients will suffer. Given the two simplest-to-implement choices, retransmission is preferable for low loss rates, and rescheduling is preferable for high loss ones; the two break even near 10%, a loss rate that effectively divides relatively clear networks from congested or faulty ones. If more complex server implementations are possible, though, a selective-retransmission scheme can significantly improve performance even more. By triggering retransmissions only for small NAKs—NAKs of many more packets than necessary for the expected loss rate, but much fewer packets than the size of a data item—we can help the server appropriately isolate small losses for retransmission and large losses for rescheduling.

More generally, we would like to combine to the most effective retransmission scheme with the most effective forward error correction scheme. To evaluate forward error correction schemes, as we do next, we must choose a baseline retranmission scheme as a backup for forward error correction. For simplicity, we will choose retransmit-always below.

B. *Forward Error Correction*

If we are willing to consider applying forward error correction in addition to retransmission, we are able to do better than retransmission alone. Because in this section, we are considering a scenario in which client losses are similar, we could apply a fixed expansion factor of error-correcting data to each data item. If forward error correction fails, we must have some form of retransmission or rescheduling as backup; for simplicity, we will choose retransmit-always here, R(∞).

To evaluate these schemes, we consider a higher-loss scenario, in which the server's link drops about 10% of data-channel packets from the source. Besides allowing us to illustrate another representative section of the parameter space that we have studied, this scenario allows us to highlight variations between the forward error correction schemes.

The resulting performance for various expansion factors are shown in Fig. 3. Along the horizontal axis, we vary the percentage of packets independently lost by each client. On the vertical axis, we plot the average client delay of forward error correction schemes applied with various expansion factors. For our base case, recall that FEC(0.0) represents no forward error correction; the plot so labelled is simply a retransmit-always server.

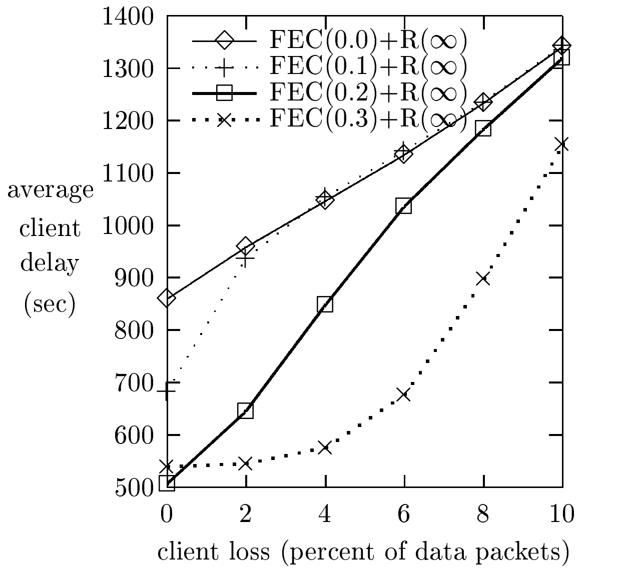

Fig. 3. Constant Forward Error Correction Schemes

If clients lose 4% of data-channel packets, in addition to the server losing 10% of its outgoing data-channel packets, for example, we see that a server acting without forward error correction and a server with 10% expansion show similar performance (at 17.5 minutes average client delay). Higher expansion factors improved performance here; 20% expansion cut client delay over 15%, and 30% expansion almost by half.

From this data, we notice that as client losses consume the forward error correction, the average client delay rises until FEC's effect fades and the system approximates the performance of retransmission alone. Worse, because FEC might add unnecessary but fixed error-correcting data to some data items while neglecting other losses, inadequate FEC with retransmission could, as losses rise, approach a level of performance slightly worse than that of retransmission alone. (At 10% server loss and 4% individual client loss as before, for example, a server with 10% expansion factor actually had a client delay about 1% worse than a retransmit-always server.)

We also see that the performance of forward error correction is, like retransmission schemes, sensitive to the actual loss rate of the clients. Excess error correction simply pads client delay by about the amount of the excess: at the far left of the plot, for example, a 10% increase in the expansion factor between 20% expansion and 30% expansion increased client delay by just over 6%, from 507 seconds to 539 seconds. Though not plotted here, the gap can be larger: As packet losses decline, client delay under 20% expansion will decline until stabilizing near the expected client delay when the server needs no retransmission.

We note also, however, that the optimal expansion factor is not the same as the clients' net loss rate; the expansion factor must be set higher, because clients can lose more or less data than their average rate during any one data item. For example, FEC with 30% expansion outperforms FEC with 20% expansion in performance dramatically even when clients lose but 14.5% of their data channel (10% loss due to server, 5% loss due to clients).

If carefully chosen, though, an appropriate expansion factor can improve the performance of a multicast system, beyond that of retransmission alone. This plot, for example, suggests that a designer can choose an expansion factor about twice the expected packet loss rate of the system's clients to exploit added forward error correction without making it excessive, and yield better delay than a retransmit-only scheme.

Ideally, a multicast implementation could combine the best of both forward error correction and retransmission. After setting the forward error correction expansion factor, the system designer could then have a multicast server use selective retransmission to decide between retransmission and rescheduling when forward error correction fails. The use of forward error correction reduces the delay of clients waiting for retransmission, and makes up for most of the clients' losses so that selective retransmission could be configured to retransmit only for small NAKs as suggested in the previous section.

VI. Additional Results

There are a number of results that we are unable to fully describe here, but are detailed in the extended version of this paper [8]. We mention some of this work in this section.

A. Clients Requesting Data in Transmission

When a new client first connects to a server, the server is likely in the midst of transmitting a data item. If the client requested the data item already being sent, the server could decide that the client missed the tranmission in progress (having certainly missed the announcement or headers for the data item), and simply let the client wait for the item's next scheduled transmission. Alternatively, the server could specially identify the data item to the new client, so that it can receive partial data as if it simply lost earlier packets.

Our results so far have assumed the former design, but we also consider the effect of having the server explicitly notify new clients of a transmission in progress. We find that it provides a slight benefit (for our scenario, about 3%-5%, or about half a minute) to reschedule-backed schemes, because this notification does not impact the server's data transmissions, but provides clients more data. On the other hand, this notification actually hurts retransmission-backed schemes, even those with forward error correction, because this notification encourages new clients to issue large NAKs for data whose tail they received when they joined.

B. *Results for Non-Uniform Loss Rates*

We also consider how reliability schemes behave for clients of different data-link loss rates, by seeing how they are affected by a new presence of very poorly connected clients in varying proportions. To see how our reliability schemes accommodate these poorly connected clients, we assign a multicast server's clients into two groups, low-loss clients (losing 5% of their data-channel packets) and high-loss clients (losing an extreme 50% of their data-channel packets), and vary the relative fractions of these two groups in the multicast system. This model allows us to amplify high-loss clients' effects on the system, so that we can see it more visibly in our results.

We find that a rescheduling server consistently outperforms a retransmitting server (sometimes by over a factor of 3), because retransmission requests are dominated by high-loss clients, and as we found in the last section, retransmitting servers falter relative to rescheduling servers when losses are high and retransmissions, many.

When we consider forward error correction, we find that FEC with rescheduling performs substantially better than FEC with retransmission, as the above would suggest. Further, a linearly-varied forward error correction scheme, with an expansion factor that increases linearly with the number of clients requesting the data (e.g., 1% times the number of interested clients, denoted FEC($0.01R$)), could be tuned to provide enough forward error correction to satisfy low-loss clients, while forcing high-loss clients to wait for rescheduled transmissions. A server using such a scheme outperforms a fixed-expansion-factor FEC system in the same environment; similar linearly-varied schemes using much higher expansion factors (whose performance is less affected by the proportion of high-loss clients in the system); and similar linearly-varied schemes using much lower expansion factors (which may be insufficient for the losses of clients that dominate the system).

C. *Exploiting Client Knowledge*

We can also attempt to estimate and exploit information about individual clients' loss rates in the multicast server. To evaluate this possibility, we consider a hypothetical "omniscient" server, which knows the exact data-channel packet-loss rate of each client in the multicast system, and modify the server to exploit this information, in its reliability scheme and in the scheduler. We find that the modifications create little benefit even in the omniscient server.

We adjust the scheduler so that its "smallest outstanding request" factor Q is replaced with a more meaningful factor $Q/fill$, the shortest expected time to completion of an outstanding request. Though the latter factor takes each client's loss rate into account, the resulting scheduler creates no appreciable benefit. We also try a forward error correction scheme that sets its expansion factor "just high enough" for every interested client to reconstruct a data item in one transmission, and find that a system using such a scheme performs better than FEC($0.03R$) but worse than FEC($0.01R$). By setting the expansion factor high enough for high-loss clients to reconstruct data, the client-loss-aware FEC scheme performs worse than a scheme that neglects high-loss clients.

D. *The Advantage of Error-Correcting Packets*

Lastly, we find that not using error correcting packets in transmission and retransmission causes a dramatic penalty. Retransmitting (R(∞)) and rescheduling (R(0)) servers that do not use error-correcting packets often have double or higher client delay than corresponding servers that do. This suggests that it is helpful to generate and use as much error-correcting data as system resources allow.

VII. Related Work

General-purpose reliable multicast protocols have been widely studied, with numerous variations proposed. A networking text [9] details the design of IP multicast; a multicast book [10] can survey a number of such protocols, such as [6] and [7]. In this paper, we extend general-purpose techniques to optimize our multicast application, which not only has varying numbers of interested clients from data packet to data packet, but also allows longer client delays to improve network efficiency, and later rescheduling of requested data items.

The work in this paper is designed to complement reliable multicast data dissemination, such as in "broadcast disks" [3] as well as our own multicast facility [4]. Work in this area often assumes a reliable network, but multicast networks such as IP multicast can often lose packets.

Closest to our work in broadcast disks, [11] considers scheduling given a known fixed loss rate and no specific client information; here, by contrast, we consider clients of different loss rates and specific client requests.

Of course, reliable multicast dissemination is also assumed and considered for use in other contexts, such as publish/subscribe (e.g., Gryphon [12]) and Web caching (e.g., [13]), but there, client latency is expected to remain very small, limiting the flexibility of the reliability mechanism.

VIII. Conclusion

In this paper we studied how to use a multicast facility to reliably disseminate data to interested clients over an unreliable network. Since data is repeatedly transmitted from a repository, reliability can be achieved by either rescheduling requests, adding redundancy to transmissions, or adding retransmissions.

From numerous simulations, we find that retransmission is most effective when applied selectively, so that clients suffering large losses wait for the entire data item to be rescheduled rather than having their losses retransmitted. This can complement forward error correction, which in our scenario performs best when set to expand each data item by a fraction about twice that of the expected packet-loss rate of the interested clients—a fraction that accommodates clients that may lose more data than expected for any one item. If a server faces clients of widely different loss rates, it should have its forward error correction tuned to satisfy only lower-loss clients, so that high-loss clients must wait for rescheduled transmissions.

Under a server that only reschedules data items for losses, it is helpful to direct newly joining clients to the data item being sent at the time; for other servers, however, such a direction can actually hurt performance.

Lastly, we find that error-correcting packets are essential to good performance, but that having the server estimate client loss rates for its reliability scheme or scheduler can provide little benefit.

We find that the proper choice and tuning of reliability schemes can improve performance by over 30%, suggesting the importance of careful design choices. The results provide insights that are guiding the design of our own multicast facility for Web data.

References

[1] M. H. Ammar and J. W. Wong, "The design of Teletext broadcast cycles," *Performance Evaluation*, vol. 5, no. 4, pp. 235–242, December 1985.

[2] Gary Herman, Gita Gopal, K. C. Lee, and Abel Weinrib, "The Datacycle architecture for very high throughput database systems," in *Proceedings of ACM SIGMOD 1987 Annual Conference*, May 1987, pp. 97–103.

[3] Demet Aksoy and Michael Franklin, "RxW: A scheduling approach for large-scale on-demand data broadcast," *ACM/IEEE Transactions on Networking*, vol. 7, no. 6, pp. 846–860, December 1999.

[4] Wang Lam and Hector Garcia-Molina, "Multicasting a Web repository," in *Fourth International Workshop on the Web and Databases (WebDB)*, 2001, pp. 25–30, Available at http://dbpubs.stanford.edu/pub/2001-28.

[5] Maya Yajnik, Jim Kurose, and Don Towsley, "Packet loss correlation in the MBone multicast network," in *Proceedings of IEEE Global Internet*, November 1996.

[6] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 784–803, December 1997.

[7] Kenneth Miller, Kary Robertson, Alex Tweedly, and Marc White, "Starburst multicast file transfer protocol (mftp) specification," draft-miller-mftp-spec-03.txt, April 1998, Internet Draft.

[8] Wang Lam and Hector Garcia-Molina, "Reliably networking a multicast repository (extended version)," Tech. Rep., Stanford University, 2002, Available at http://dbpubs.stanford.edu/pub/2002-37.

[9] Andrew S. Tanenbaum, *Computer Networks*, Prentice Hall, Upper Saddle River, New Jersey, third edition, 1996.

[10] C. Kenneth Miller, *Multicast Networks and Applications*, Addison-Wesley, Reading, Massachusetts, 1999.

[11] Nitin H. Vaidya and Sohail Hameed, "Scheduling data broadcast in asymmetric communication environments," *Wireless Networks*, vol. 5, no. 3, pp. 171–182, 1999.

[12] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman, "Exploiting IP multicast in content-based publish-subscribe systems," in *Middleware*, Joseph S. Sventek and Geoff Coulson, Eds. 2000, Lecture Notes in Computer Science, pp. 185–207, Springer-Verlag.

[13] Pablo Rodriguez, Ernst W. Biersack, and Keith W. Ross, "Improving the latency in the web: Caching or multicast?," in *3rd International WWW Caching Workshop*, Manchester, UK, 1998.