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.

1 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 [2], Datacycle [4], and broadcast disks [1], among others. As a local case in point,

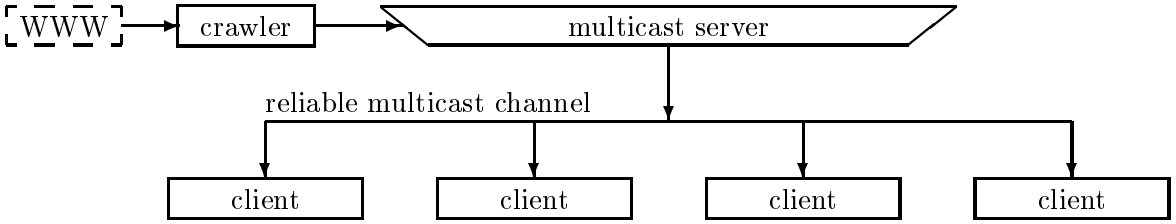

Figure 1: The WebBase Multicast Facility

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 Figure 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 [5].)

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 [12], 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 that lost data can be reconstructed using the redundant additional data (called "forward error correction," FEC, in the networking community). 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 2 a possible network traffic layout that isolates the network-consuming data transmission itself for further study and optimization. In Section 3, 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 2.1, we will propose a simple client delay metric to gauge the performance of our design decisions. In Section 4, we describe the simulation we use to evaluate our design options. In Sections 5 and 6, we will examine some results from our study to form suggestions for the design of a reliable-multicast facility over an unreliable multicast network.

2 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 multicast a requested data item to clients, the server needs to send information identifying a data item, and then the data item itself. Clients always need to receive information identifying the data item the server is about to send, to determine whether they requested the data item. On the other hand, clients do *not* need to receive all the actual data items, only the ones they request.

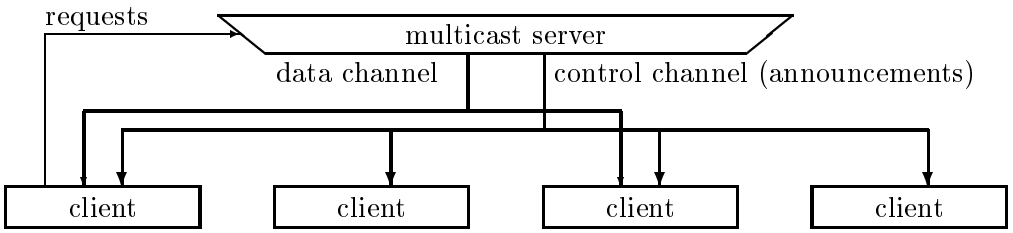

Figure 2: Information Flow in the Multicast Facility

To minimize the consumption of clients' download links, then, 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. The resulting separation of network traffic is shown in Figure 2.

All clients would always subscribe to the control channel, because they must receive all the announcements on the control channel promptly. 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.

For the control channel to work effectively, clients must receive control channel traffic reliably and promptly, so the channel must use a latency-sensitive low-bandwidth reliable multicast protocol to ensure all control channel traffic reaches its subscribers. 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 exactly this type of application, including SRM [3].

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, whose information all clients need, 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 3.

- 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.

2.1 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 [5].

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. As such, the network usage of the facility depends primarily on how much data clients request and how frequently clients appear, rather than on any particular server's design.

2.2 Multicast Operation

The server operates by gathering the clients' request subsets and using a scheduler, such as R/Q [5], 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 2.1, 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. □

3 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).

3.1 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.

Example. As a simple example, let's use a form of parity as our error-correction scheme. Further, let's suppose a data item takes up two numbered packets, each with payloads 8 bits long: 00110101 and 00001010. To ensure that each of the 8 bit positions has an even number of ones across all packets for this item, a parity packet would carry in its payload the value 00111111. (The parity packet is, equivalently, the exclusive-or (XOR) of the data packets.) (This is called even parity.)

The server could then send all three packets on its data channel (so in this instance, expanding the data by a factor of 50%), to ensure that clients can drop one packet out of the three and still receive the entire data item. For example, if a client loses the parity packet, the client can ignore the loss. If a client loses the first data packet, the client can compute it as the XOR of the two packets it does have: 00001010 XOR 00111111 = 00110101, the first data packet. Similarly, it can compute the second data packet as the XOR of the first data packet and the parity packet.

The example uses parity for demonstration, but parity is a fairly limited scheme. In practice, one would choose a different error-correcting code, which could be easily adjusted to allow the recovery of more than one data packet using as many error-correcting packets as we expect clients to recover from loss. □

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. If a system designer wishes to avoid depending on the fallback scheme, then, the forward error correction parameters must be chosen so that a client's probability of loss is negligible.

3.2 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.

3.2.1 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.

3.2.2 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 pre-computed 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.

For example, consider the forward error correction example, but now the server does not preemptively send the parity packet on the data channel. If a client reports a lost packet, the server can then send the parity packet instead of the actually-lost packet, so that the client recomputes its lost packet using the parity packet. This allows two clients to each lose a different packet, and still recover the data item using only the same parity packet transmission.

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].

3.2.3 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 [3], 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` [3]). A server could, further, instruct its clients (over a control channel) to expand its NAK-delay interval as the server's ceiling number of incoming NAKs approaches the server's incoming bandwidth.

Beyond the random delay associated with each NAK, we do not further consider the cost of the NAKs coming into the server. The cost of per-item-per-client NAKs are unlikely to determine or affect the server's network costs in practice because we expect NAK packets to be much smaller than the corresponding data item transmissions to which they respond, so the cost of the data item transmissions should dominate. Further, if a network connection or its costs are asymmetric, it is often the server's (larger) outgoing data transmissions that are expensive, not the (smaller) incoming NAK traffic. (Consider, for example, that consumer and business "broadband" connections, when asymmetric, have lower upload speed caps

than download speed limits.)

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.

3.2.4 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 $\mathrm{R}(\infty)$, 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 $\mathrm{R}(p)$.

3.3 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.

The ultimate "do nothing" scheme, in which NAKs are entirely ignored, cannot guarantee reliability because the server may never send a lost data item again if no other clients ask for it, so we ignore that scheme here.

4 Simulation

Variable	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)

Table 1: Simulation Parameters and their Base Values

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 [12]. 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. In effect, packets are dropped getting into the core backbone or getting out; they are not lost on the backbone itself.

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 1, 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 1, 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 1 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.

5 Results for Uniform Loss Rates

First, we consider how the various reliability schemes compare 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 consider how these reliability schemes are affected by the presence of clients with different loss rates.

5.1 Retransmission and Rescheduling

We first consider what happens in the simplest case, in which the multicast facility uses only retransmission (Section 3.2). In Figure 3, 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 1. 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 3.2. 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 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. As long as the packet losses are not very large, clients benefit more from the server consuming slightly more network time for each data item (which consists of the time for the original transmission plus the time for small retransmissions) than from the server delaying clients until data items are rescheduled (which incurs no retransmission delays for new data items, but forces most clients to wait for a data item to be scheduled at least twice).

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 deteriorate 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. (This is because clients can combine the distinct error-correcting packets from multiple transmissions of a data item to reconstruct missing the data.) In our case, the number of times a client needs a data item to be scheduled is very consistently two, for a large range of high-loss rates. As retransmission continues to deteriorate, and rescheduling holds steady, the penalty from clients waiting for and receiving their increasingly long retransmission requests starts to outweigh the cost of simply rescheduling the data item in its entirety for old and new clients, and letting the server transmit new data instead.

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.

5.2 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, $R(\infty)$.

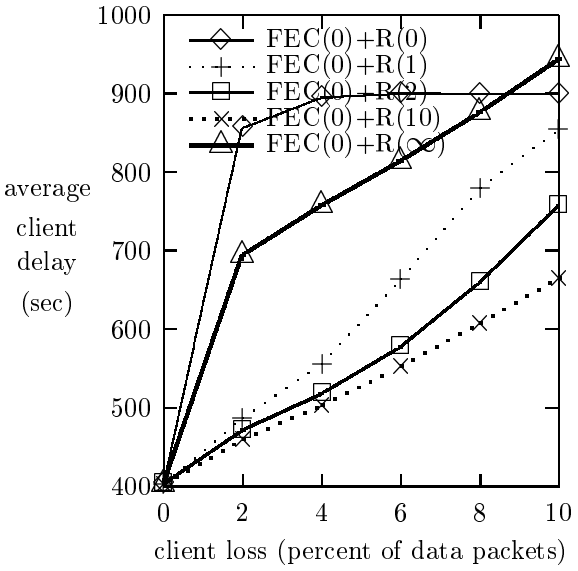

Figure 3: Retransmission Schemes

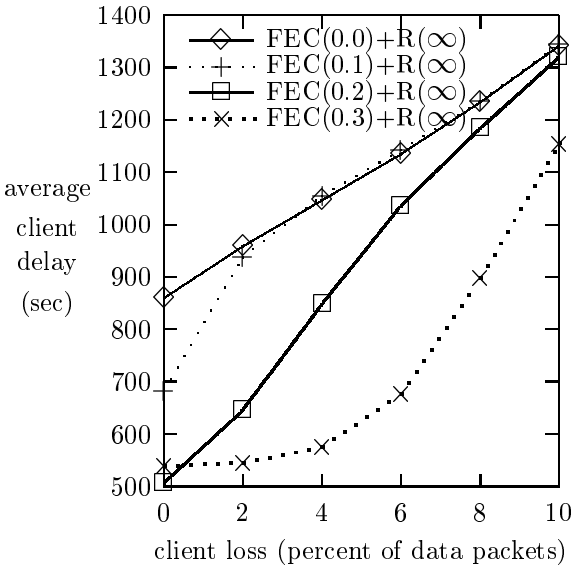

Figure 4: Constant Forward Error Correction Schemes

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 Figure 4. 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. 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.

5.3 Clients Requesting Data in Transmission

In all our simulations so far, we have been assuming that a new client begins receiving data from the server not when the client first connects to the multicast server, but at the earliest time the server sends a new data item after the client joins the system. This is because a new client would not have received the announcement (or headers) for a data item in progress at the time the new client joins, and so the client would not know of the data item unless the server explicitly notifies the client of the transmission in progress.

Intuitively, it seems that the server should in fact notify the new client of a transmission in progress when it joins, if the transmission is relevant to the client's request. Then, the client could behave as if it had simply lost early packets of the transmission due to an unreliable network, and send a negative acknowledgment for them as usual.

Intuitively, it is comparable to a bus stopping and waiting for late passengers: All passengers already on board take a delay, but provide a dramatic benefit to the new but late passenger, who gets to board an earlier bus. Like a late passenger whose alternative is to wait the entire interval between buses, a client that misses a transmission in progress would otherwise be compelled to wait a long time for the next transmission. Is it worth making passengers on board wait for a latecomer?

In our case of the multicast system, however, there are additional factors that can affect the result of our modification. Because other clients may also need retransmissions anyway to cope with network losses, the cost of retransmitting for (catching up) a new client could be slightly lower, by the cost of retransmission for other clients. Also, a server might not choose to retransmit at all; if the server does not retransmit in response to the new client's NAK, then the client gets a slight benefit from getting a few packets early, but neither gets the benefit of receiving an entire data item early nor negatively impacts the rest of the system.

We consider the effect of new-client notification here, which we will highlight in our results with a "+notify" notation. For example, since we denote a default rescheduling scheme "FEC(0)+R(0)," we would denote a rescheduling scheme with the new-client notification "FEC(0)+R(0)+notify." (Clearly, new-client notification is a design decision independent of the forward error correction and retransmission schemes we have considered.)

In Figures 5 and 6, we show the effect of new-client notification on our multicast facility. We plot the relative performance of several multicast schemes that do not do this notification to variants that do. Along the horizontal axis, we vary the percentage of packets each individual client loses. On the vertical axis, we see the average client delay of the multicast system. In Figure 5, the server link loses no packets, a low-loss scenario (as we had used when we considered retransmission earlier). In Figure 6, the server link loses 10% of its packets, a high-loss scenario (as we had used when we considered forward error correction earlier).

For example, we see that if the only packet loss comes from clients losing 6% of their incoming data packets, a retransmit-always multicast server would yield an average client delay of about 815 seconds (13.5 minutes). If we modified it to notify new clients of relevant transmissions in progress, then the average client delay rises just over two minutes to 944 seconds. Similarly, in Figure 6, if the server link loses 10% of its data channel packets, and clients each additionally lose 6% of the data channel packets that remain, a multicast server that sends data items with 20% error-correction padding and retransmits on demand would yield an average client delay of just over 17 minutes, versus just over 22 minutes if we added new-client notification.

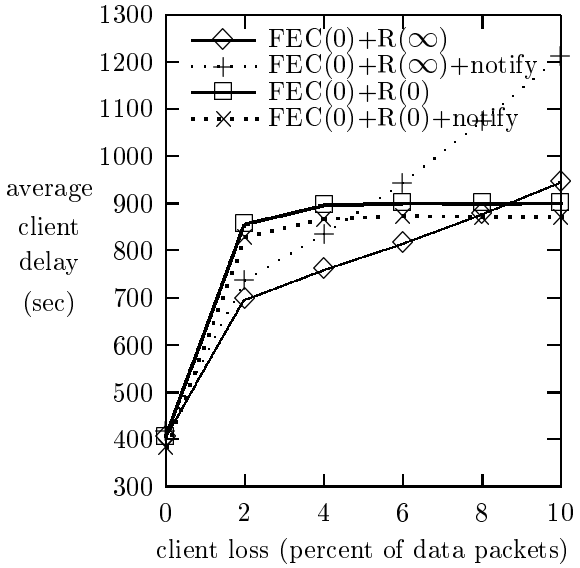

Figure 5: Retransmission Schemes

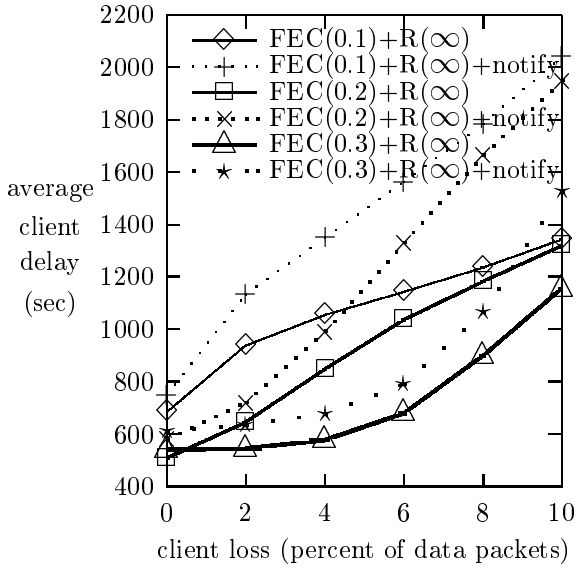

Figure 6: Constant Forward Error Correction Schemes

We see, then, that for retransmission-backed schemes, including those using forward error

correction (labelled R(∞) in Figures 5 and 6), new-client notification actually hurts average client delay. This suggests that, especially given our high simulated load, the large number of clients that need other data are significantly hurt by a new client triggering extra NAKs under new-client notification, and the penalty to waiting clients outweighs the benefit to the new one. In fact, we can also observe in Figure 6 that, when client losses are poorly covered by forward error correction, new-client notification hurts client delay much more than if client losses are mostly covered by FEC. This suggests that, as clients lose more packets than forward error correction can cover, more clients (waiting for other retransmissions and new data) are forced to suffer additional delay caused by a new client. Further, this additional delay apparently outweighs the ameliorating effect of a new client "sharing" its cost of retransmission with other clients.

By contrast, if we consider a multicast system that never sends retransmissions on demand (plots labelled R(0) in Figure 5), we get a slight benefit from new-client notification, as we might expect. A client in this system can catch the tail of a transmission already in progress, but that in itself is not enough for a client to reconstruct the data item. Because its NAKs would be ignored, it is unable to fill in the data it is still missing, and must wait for the next transmission of the data item to fill in the rest. In effect, a late client must still wait for the next transmission anyway; its benefit is that it may not need the entirety of the next transmission, since the client already knows how the transmission will end. This may allow clients to complete their requests before the end of a transmission of a data item, and reduce their delay accordingly. Alternatively, this reduces the chance that a client needs yet another transmission of a data item to reconstruct lost packets, delaying itself and other clients. Lastly, unlike the retransmit-always server, a rescheduling server does not change its dissemination to accommodate new clients, so there is no client delay penalty. The net result, as we see in the figure, is a consistent, small client delay benefit for new-client notification, of about half a minute (3%-5%).

6 Results for Non-Uniform Loss Rates

Under less-controlled environments, different clients may be connecting to a multicast server over links of varying quality, leading to different loss rates to different clients. As a result, the reliability schemes we study can behave differently while trying to accommodate (or not) clients that lose more packets than others. Also, we can try to improve the performance of the system by adjusting our reliability schemes, or our multicast facility scheduler, to fit the demands of the requesting clients.

In this section, we will first review how retransmission and rescheduling fare in this new environment, then consider the addition of forward error correction. In particular, we will see the effect of using a variable expansion factor for forward error correction. We will also separately tune forward error correction and the multicast scheduler itself using client loss information, and evaluate their impact on system performance.

To study how our schemes are affected by the presence of poorly connected clients, we run simulations in which clients are grouped into two classes. One class of clients, having "good" links, lose only 5% of the data channel packets sent to them. The other class of clients, having "bad" links, lose 50% of the data channel packets sent to them. Clearly, poorly linked clients' loss rate is extreme; this is chosen to amplify the effects they would have on the system, so that we can see them more visibly in our results. To observe the behavior of our multicast facility as poorly linked clients enter the system, we vary the relative fraction of well-linked and poorly linked clients in the system.

With clients having different loss rates, we can consider another metric that biases client delay by the loss rate of the client suffering it. Unlike the average client delay metric we have been using so far, we could consider a *loss-weighted* average client delay that averages, for each client, the product of the client's delay and one minus its average loss rate. (Intuitively, this makes the delay of low-loss clients more important the delay of high-loss clients.) Though we study this metric in addition to average client delay, we report only on average client delay here.

6.1 Retransmission versus Rescheduling

In this environment, we find that a rescheduling server consistently outperforms a retransmitting server. Though this observation appears to invert the results in the last section, it is actually consistent with our earlier conclusions there: Rescheduling is more effective than retransmission when clients lose a large fraction of their packets, and in this environment, some of the clients lose a large fraction indeed. Whether in the minority or majority, the high-loss clients compel a retransmit-always server to incur all the expense of retransmission for high-loss clients. Hence, reschedule-only performs better than retransmission in this environment. Data from this two-tiered case is available as part of Figure 10, where it is compared to an equivalent scenario without the use of error-correcting packets.

6.2 Forward Error Correction

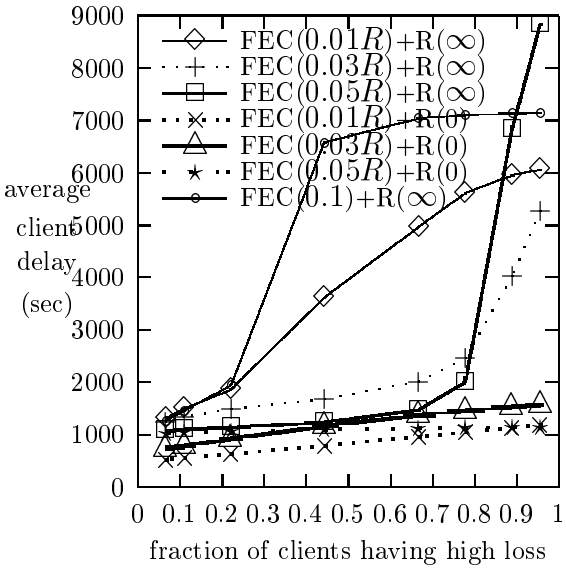

Figure 7: Variable Forward Error Correction

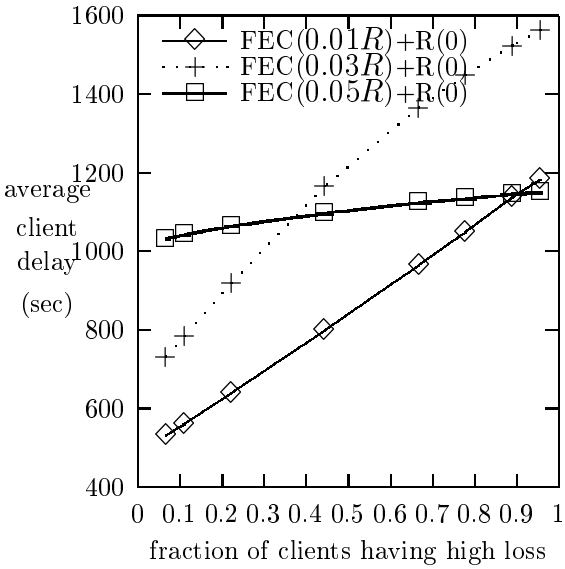

Figure 8: Variable Forward Error Correction

In Figures 7 and 8, a zoomed-in view of Figure 7, we see how various forward error correction schemes perform for clients of different data-channel loss rates. Here, we use a server whose link to the multicast backbone does not lose packets, and vary the proportion of low-loss clients versus high-loss clients. For example, at the left edge of the plots, all clients have low-loss rates (5%), and at the right edge, all clients have high-loss rates (50%). Along

the vertical axis, we plot the resulting average client delay for various forms of forward error correction. Plots labelled "nR" indicate a forward error correction scheme whose expansion factor (as defined in Section 3.1) is a linear function of the number of clients wanting the data item being padded. The plot labelled "FEC(0.1)" is a fixed error-correction factor, included for comparison for Figure 7. (Other fixed expansion factors are omitted from this graph to reduce clutter; their curves are similar to the curve shown.)

For example, in a scenario with about 10% of clients having very lossy links, a multicast system using a fixed expansion factor of 10% with retransmission would have an average client delay of about 1450 seconds (24 minutes). For multicast systems with linearly varied FEC and retransmission, the delays are generally better; for example, a server using an expansion factor of 0.05R, the client delay is 1100 seconds (18 minutes).

The more sustained improvements come, however, when we consider linearly varied FEC without retransmission. The minimum delays plotted are for the non-retransmitting servers: Using an expansion factor of 0.01R, 0.03R, and 0.05R, they net an average client delay of 560 seconds (9 minutes), 780 seconds (13 minutes), and 1040 seconds (17 minutes). Intuitively, these results are an extension of rescheduling's better performance than retransmission in this environment, suggesting that to optimize the use of forward error correction, we must use it without the penalizing interference of retransmit-always as backup.

If we look more closely at the reschedule-backed schemes (as shown in Figure 8, which plots a different range on the vertical axis), we also see that the optimally tuned expansion factor varies by average loss rate, suggesting the importance of tuning the expansion factor appropriately. In this case, we see that a server using a large expansion factor ($0.05R$) can provide a benefit to even the high-loss clients; its performance is relatively stable, even as high-loss clients dominate the system. On the other hand, a server using a smaller expansion factor can only help low-loss clients from requiring repeated transmission of their data item, and is not nearly as effective on high-loss clients. As a result, its performance deteriorates as more high-loss clients enter the system. We see, then, that is advantageous to provide enough forward error correction to cover the low-loss clients ($0.01R$), but not much more than that ($0.03R$), as the excess does not contribute to the remaining clients (who need another scheduled transmission anyway). If high-loss clients dominate, however, then it may

be helpful to adjust the expansion factor to follow, as we see at the far right of our plot.

The behavior of retransmit-backed forward error correction schemes is slightly different. Here, because losses are always filled by retransmission anyway, the negative effect of high-loss clients is more pronounced, and the partial benefit of a larger expansion factor directly helps to reduce the time devoted to retransmission. (Consequently, a server using expansion factor $0.03R$ does better than a server using expansion factor $0.01R$ in this case.)

Also, excess padding ($0.05R$) causes a particularly painful delay penalty when its padding is not quite enough for a high-loss client; in that circumstance, not only does the expansion factor approximately double the size of the data item (and therefore, the time to transmit it), it approximately doubles the later rescheduled transmission needed for the high-loss client to fill in its last missing packets. We see the effect of this penalty as we move right in the plot; the more high-loss clients we have in the system, the more likely we are to have clients lose many more packets than expected, simply because of chance.

In summary, we conclude that 0.01R-expansion-factor FEC with rescheduling provides a good performance tradeoff for this scenario, and one that is useful when facing widely-varying loss rates in general. The FEC expansion factor is sufficiently large to ensure low-loss clients receive their data without waiting for repeated rescheduling, while high-loss clients wait for their data to be rescheduled so as not to raise low-loss client delay needlessly. Excess transmission than that necessary to accommodate low-loss clients appears to hurt client delay overall, hurting low-loss clients more than it helps lossy clients.

6.3 Exploiting Client Knowledge

Suppose a multicast server tracked the performance of its clients, and could therefore estimate their loss rates. Could the multicast server use this information to improve performance? With this information, we could modify the reliability scheme so that it sends just enough data for all of the clients requesting each data item. Alternatively, we could modify the multicast scheduler itself to take client loss rates into account. We consider each of these possibilities in turn in this section. To see what the potential gains are for each option, we assume in this section that the multicast server has "perfect knowledge" of the loss rates of its clients. That is, when we assign each client its average data-channel loss rate, we provide

the value to the server as well. This allows us to see the ideal behavior of each option, by assuming that the server's estimates of client losses are perfect.

6.3.1 FEC with Client Knowledge

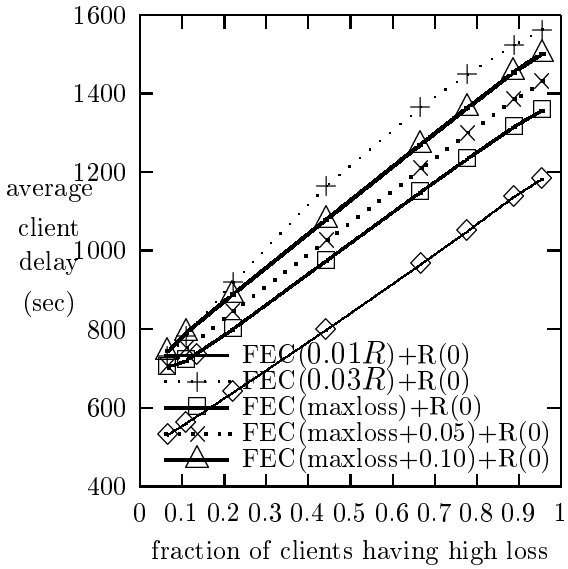

Figure 9: Perfectly Loss-Aware Forward Error Correction

In Figure 9, we consider how a forward error correction scheme without retransmission would perform if the server knew the largest loss rate of any client requesting a data item, and used that loss rate to determine its FEC expansion factor. That is, the server determines the minimum FEC expansion factor that should allow every client to reconstruct the transmitted data item immediately, assuming that each client loses as many packets as expected for its loss rate. In our two-tiered-client simulation, the effect is that the expansion factor matches the loss rate of a high-loss client if a high-loss client is requesting the data item being transmitted. The expansion factor matches the loss rate of a low-loss client otherwise.

We denote this minimum expansion factor "maxloss" in the plot. So, "FEC(maxloss+0.05)" represents forward error correction whose expansion factor is always 5% higher than the highest loss rate of a client requesting the item being sent. As in the prior section, we use a server whose link to the multicast backbone does not lose backets, so the server link does not affect the server's estimates of client data-channel loss.

We see that it is a net loss to increase the expansion factor beyond the value that matches the highest client loss rate for a data item; across the plot, padding the expansion factor "just in case" a client loses more packets than expected is an overall loss. Also, we see from this plot confirmation for what we had observed before—that it is helpful to focus on the low-loss clients to the relative neglect of high-loss clients. In this perfect-knowledge system, the server chooses an expansion factor that matches that of a high-loss client if a high-loss client is requesting the data item being transmitted. Doing so incurs a worse client delay than a linearly varied FEC system, despite the server knowing in advance the minimum number of packets to send to such clients, so it is the attempt to accommodate the high-loss clients that causes client delay to deterioriate in a perfect-knowledge system. As a result, this use of client-loss information does not improve the performance of the multicast facility.

6.3.2 Tuning the Scheduler

We can attempt to improve our results by breaking the abstraction barrier between the scheduler, which takes information about clients' requests for data and determines which data item to transmit, and the reliability scheme, which ensures that clients receive the data item being transmitted or ensures that their requests are returned to the scheduler for later transmission. In particular, we could change the scheduler itself so that it takes into account the clients' loss rates. We consider how we could make this change, and evaluate its potential effect by simulating the resulting server with perfect knowledge of clients' data-channel loss rates.

A Loss-Adjusted Scheduler For example, for our R/Q scheduler, we are intuitively computing a score for each data item that rises with the number of clients needing the data item, and rises with the smallest request-size of any client wanting the data item. The latter is useful because we would like to complete almost-satisfied and very-short requests, so that clients do not wait for transmission needlessly. To adjust the scheduler, therefore, we observe that the latter factor is really a "shortest time to a satisfied client" factor, and that the time to completion for a client depends not only on how many data items it still needs, but also how long it will take to receive them over a lossy network.

We observe that clients suffering high loss will take longer to receive a same-size data item than clients suffering little loss; therefore, we should raise the score of a data item not by the smallest request-size of any client needing the data item, but rather by the shortest expected time to completion of any client needing the data item. So, we compute the request size divided by the fill rate (the fraction of packets the client receives on its link), and use that as the client's expected time to completion. In the revised R/Q scheduler, we find the client with the smallest expected time to completion, divide the number of clients wanting the data item by it, and choose a data item with the highest such score.

A Minimal Effect The resulting performance is shown in Figure 11. In this plot, we see the average client delay as we vary the proportion of high-loss to low-loss clients, for several multicast systems with the normal and revised R/Q schedulers. The server is assumed to lose no packets on its link to the multicast backbone. The plots labelled "R/(Q/fill)" use the revised R/Q scheduler as described above.

As we can see, the relative performance of the original R/Q scheduler and the revised scheduler are fairly similar. For example, if nearly half (about 45%) of clients are high-loss clients, then both the basic R/Q scheduler and our revised R/Q scheduler incur an average client delay of about 110 minutes.

This suggests that, in general, adding complexity to the scheduler to take client loss rates into account nets no appreciable benefit. To see whether our intuition for the scheduler is incorrect, we run simulations with other variants of R/Q, in which we bias the scheduler towards clients with higher loss rates rather than lower ones as above (arguing that clients with high loss rates need more service to receive their data). The results there suggest that our original intuition for adjusting R/Q is correct, and that the standard R/Q matches or outperforms the resulting scheduler.

6.4 The Advantage of Error-Correcting Packets

As we have asserted earlier, when a multicast server needs to compensate for losses on the network, erasure-coded packets that a client can use to compute the value of any one other missing packet is at least as useful as an exact transmission of a particular data packet. These

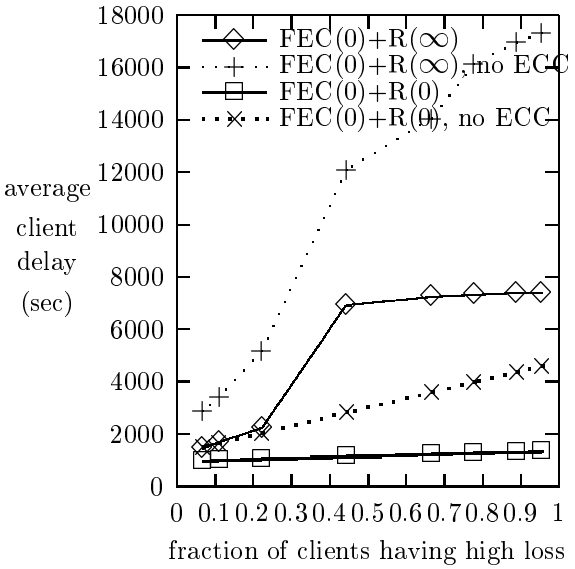

Figure 10: Exact Versus Error-Correcting Packets

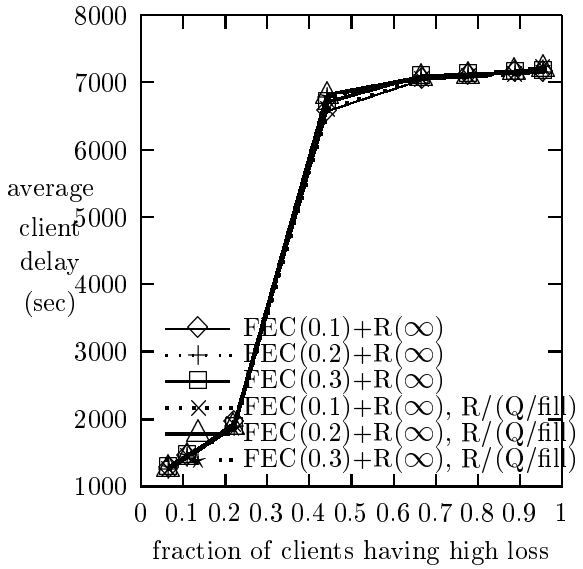

Figure 11: Applying a Loss-Sensitive Scheduler

erasure-coded packets, however, may be expensive to compute. The simpler alternative to computation is for the multicast server to send only the original data packets, and to resend those exact packets if they are lost.

To determine the effect of error-correcting packets on the client delay of the facility, we run simulations in which the multicast server retransmits the specific data packets a client lost, versus ones in which the server sends packets from a large pool of error-correcting packets.

The results of the simulations are shown in Figure 10. In this plot, we chart the average client delay versus the fraction of clients of high loss, for multicast systems that use error-correcting packets as before, and systems that must transmit exact data packets. This plot shows the delay that results in scenarios where the server link loses 3% of data channel packets; plots for other server-link loss rates are similar, so we omit them for brevity. The systems that do not use error correcting codes for their data packets are marked "no ECC," to contrast the systems that do use them by default.

For example, if about 10% of clients have very lossy links, then retransmit-always and reschedule-only schemes using error-correcting packets have average client delay of about 28 and 16 minutes, respectively. By contrast, if these two schemes used only exact data packets, then they would incur a much higher 57 and 28 minutes of client delay, respectively.

Clearly, we see that there is a dramatic difference in performance between a basic exact-

packet transmission scheme and an error-correcting packet scheme. Whether the server sends retransmissions on demand, or ignores them for later scheduling, average client delay from the multicast facility is much lower when error-correcting packets are used.

In an exact-packet scheme, clients' NAKs for data must coincide for any retransmitted data to help more than one client at a time, negating much of the advantage of multicasting retransmissions of data. In the error-correcting-packet scheme, every packet helps every client that receives it. This difference makes the latter scheme's retransmissions much more effective, as shown in the results.

For a retransmit-never scheme, the issue is the same. When a large pool of error-correcting packets are available, the server can use a different set of packets for a data item when it transmits the item another time. This allows clients missing some packets to benefit from every received packet of a later transmission of the item. By contrast, a multicast system using only exact packets runs the risk that a client can simply miss the same data packets again in a rescheduled transmission, by coincidence, making the retransmission useless for those packets.

Of course, when a multicast facility runs out of new error-correcting packets to transmit, the risk of coincidental, repeated packet loss returns for any very high-loss clients that notice the repetition. The results here suggest, therefore, that it is helpful to pregenerate and use as much error-correcting data as system resources allow.

7 Related Work

General-purpose reliable multicast protocols have been widely studied, with numerous variations proposed. A networking text [10] details the design of IP multicast; a multicast book [6] can survey a number of such protocols, such as [3] 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" [1] as well as our own multicast facility [5]. 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 [8]) and Web caching (e.g., [9]), but there, client latency is expected to remain very small, limiting the flexibility of the reliability mechanism.

8 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] Demet Aksoy and Michael Franklin. RxW: A scheduling approach for large-scale on-demand data broadcast. *ACM/IEEE Transactions on Networking*, 7(6):846–860, December 1999.

[2] M. H. Ammar and J. W. Wong. The design of Teletext broadcast cycles. *Performance Evaluation*, 5(4):235–242, December 1985.

[3] 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*, 5(6):784–803, December 1997.

[4] 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*, pages 97–103, May 1987.

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

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

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

[9] 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.

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

[11] Nitin H. Vaidya and Sohail Hameed. Scheduling data broadcast in asymmetric communication environments. *Wireless Networks*, 5(3):171–182, 1999.

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