MULTICAST DATA DISSEMINATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Wang Lam

September 2004

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Andreas Paepcke

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom

Approved for the University Committee on Graduate Studies.

Abstract

Distributing large amounts of information on a computer network requires a data-dissemination server that is able to accommodate many interested clients making varied requests for data. For such a server, its outgoing network connection is a bottleneck that severely limits the number of clients and the size of shared data that the server can handle. To ameliorate this bottleneck, multicast allows a server to address a single transmission of data to multiple clients at once, reducing the repeated transmission of the same data items to different clients. Consequently, the server is able to provide faster transmissions to more clients than otherwise possible over the same network connection using unicast transmissions alone.

To handle varied requests from many clients, the server must have an efficient way to decide which of the many requested items to send at any one time. This dissertation considers a variety of data schedulers to optimize this task for several performance metrics and a variety of client loads, including downloaders receiving sets of data as quickly as possible, and subscribers striving to keep changing data items current. Using modelling and simulation of such a multicast facility, we show the dramatic performance difference between various schedulers and the effects of batching clients over time.

Because clients can have differing network capacities, the server must be able to deliver its data items reliably over client connections of varying loss rates. This dissertation shows how to exploit the unique properties of this system to extend traditional reliability techniques and reduce average client wait time by over 30%. Also, the server must decide how to slice its available outgoing network capacity into data channels, how to assign its data items to those channels, and how to assign

clients to the channels given clients' varied requests and download speeds. We reveal the choices that can improve performance three-fold or more over their alternatives: Surprisingly, finding the good choices does not require advance knowledge of clients' varied download speeds, once the server chooses the slowest speed to support.

Finally, this dissertation details challenges, design, and evaluation for a prototype using IP multicast.

Acknowledgments

This dissertation exists because of the help and support of many people. First and foremost, of course, is my adviser Hector Garcia-Molina, without whose patient direction and wise guidance this dissertation would not exist.

Andreas Paepcke (in addition to kindly reading and commenting on this text) tirelessly led Stanford WebBase efforts day to day. He, Sriram Raghavan, Taher Haveliwala (Go Bears!), Gary Wesley, and Junghoo Cho made our collective work on the WebBase project an immensely enjoyable and rewarding experience for me.

Jennifer Widom, who offered many helpful comments on this text, also introduced me to some of her insights from her teaching experience, supporting my efforts to learn the challenging and rewarding art of teaching as an assistant. I am grateful also to Anand Rajaraman, Prabhakar Raghavan, and Hinrich Schuetze for offering me the happy opportunity to learn from such brilliant and accessible instructors as well as assist them in the advanced-topics courses they taught.

Finally, I owe many thanks to my friends and family, who have put up with me (much more than I deserve) being busy or buried nights and weekends. My parents' unwavering support, in particular, made my graduate study possible in the first place: I dedicate this dissertation to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Despite the growth of network capacity over time, data dissemination to many users remains a difficult and expensive proposition. Many current data dissemination systems, such as Web and FTP service, require a data server to send the same information repeatedly, once to each client that wants it. Doing so encourages network congestion for the server's limited uplink (which decreases the number of clients the server can handle), and increases the server's network costs.

In particular, when a data server becomes popular, many users often request the same data of the server, or request data that overlaps with the requests of other users. For example, a file server may find some of its collections popular and therefore frequently requested; a Web server may find that it repeatedly sends some of the same items to users (such as a Web site's front page and its cited images). High loads caused by such requests increase the server's network usage and its associated costs. Worse, such high loads can cause data service to collapse completely under network congestion, rendering the data unavailable to anyone: If a server's network connection is divided among too many requesting clients, none make sufficient progress to prevent connection timeouts, causing partial progress to be lost and prompting clients to add even more load to the system by rerequesting their data.

Fortunately, there is a more network-efficient approach available on multicast networks, which allows a server to address each data transmission to a group of clients at a time. On a multicast network, such as that available on the Internet2 backbone

Abilene and implemented in native IPv6 networks, a server can share a repository of data items over a data 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. As a result, a multicast facility can also scale to large numbers of clients and sudden increases in client load more easily.

The idea of such a multicast facility is well-established, having been envisioned and studied for Teletext [5], Datacycle [30], broadcast disks [4], and Usenet feeds [36], among others. Indeed, a multicast data-distribution facility is useful for a variety of applications:

1. A news service may have a collection of news reports that are heavily requested during the service's debut, or during an unprecedented news event.

2. It is becoming increasingly common to release new software and software updates electronically over the Internet. While Internet-based distribution makes software distribution faster and less expensive than printing and shipping the software on physical media, this form of distribution also adds stress to the servers distributing the software during its release. This stress occurs because when the software is released, many users will download various components of the software: every user will want the core components; many will want popular components; some will want more complete installations. Using a multicast data facility for network distribution and installation can make software distribution more affordable and more accessible to publishers and users.

3. In a distributed file-sharing service, many nodes participating in the file-sharing service and offering files to others may be under severe network usage constraints. For example, these nodes may include home users on low-bandwidth connections, or they may be on university networks where their aggregate network usage would panic university network administrators. Such nodes can

provide data service to more peers using a multicast facility as the file transport.

As a local case in point, a multicast facility provides a useful way to distribute data for the Stanford WebBase project, which crawls the Web to create repositories of Web pages for research.

1.1 The Stanford WebBase Project

Though typically used to build indexes for search engines such as Google, Web crawlers are also used to gather Web pages for data mining, or to produce Web caches that can be more conveniently or quickly accessed by users than the originating Web servers.

As crawlers proliferate, more and more crawlers visit each Web server on the net, repeatedly requesting the same pages. Popular servers are often visited by hundreds of crawlers in a week [35]. The extra load is especially problematic to small companies that pay their ISPs by the amount of data transmitted: A large fraction of their networking budget goes to pay for pages fetched by crawlers and not to pages fetched by paying customers.

At the other end, writing and running large-scale crawlers is also problematic. It is tricky to write crawler code that follows crawler-restriction conventions such as `robots.txt` and does not overload Web sites. (The file `robots.txt` at each Web site indicates any parts of a Web site that may not be visited.) A bug in the crawler code can cause it to visit a Web site too frequently, triggering an apparent denial-of-service attack and potential lawsuits. Even if the crawler operates correctly, it can overload the network at the crawler end, drawing the ire of network administrators. To detect and correct operational problems, an expert staff person is typically on call whenever the crawler runs. As such, the complications and attendance required in running a Web crawler may discourage users who could benefit from having a local copy of Web data. For example, an analyst may be interested in tracking the Web sites of competitors, and Web sites offering product and service reviews. Another user might be a frequent traveller who wishes to prefetch information on destinations, for offline access during trips. Also, a software developer may wish to track Web-based FAQs and documentation, perhaps to collect statistics on bug reports and software usage. These users may not have the expertise, and certainly not the resources, to craft and manage a well-behaved Web crawler.

The WebBase project presents an alternative to multiple crawlers: A single "central" crawler builds a database of Web pages, and provides a multicast service for "clients" that need a subset of this Web image. The clients in this case are all the sites that were originally running crawlers. Instead of writing and running their own crawlers, these clients subscribe to the pages they need. For example, a client may be interested in a set of Web sites, or perhaps in all `.edu` sites. This way, the Web sites supplying Web pages are only visited by one crawler representing a number of clients. Furthermore, the network is used more efficiently, since it is better to multicast a single copy of a page P, rather than having all clients get a copy of P individually.

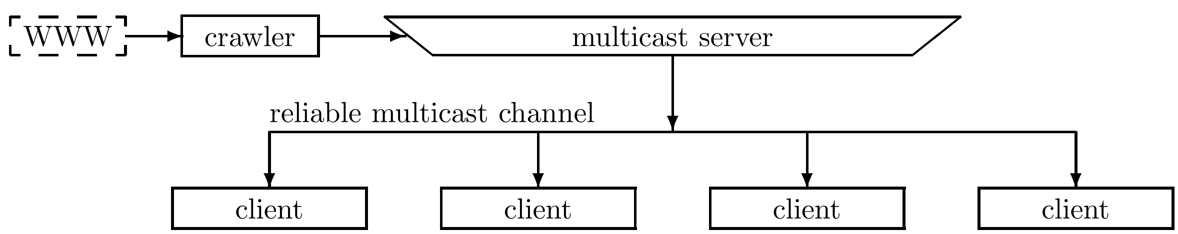

Figure 1.1: The WebBase Multicast Facility

Of course, we would expect more than one such Web crawler, repository, and multicast facility in the world. With multiple servers, clients can resubmit their unsatisfied requests to a different server should one become unavailable, and clients could choose a "nearby" server for improved service.

WebBase's crawler is currently used to build a cache that holds hundreds of millions of pages, distributed over several computers. Our multicast facility can offer the WebBase repository to other researchers so that they can request subsets of the repository in a simple and efficient way (Figure 1.1). Existing WebBase client code then allows researchers to parse the repository data for their analysis.

1.2 The Multicast Facility

All of the example applications enumerated in the beginning of this chapter, and the Stanford WebBase, share a key property: Clients may request different subsets of the available information on a multicast server, but these subsets will often overlap. In these example scenarios, researchers may request different Web sites for study, but their requests may overlap on highly cited (or frequently visited) Web sites; software users may request updates for different programs by the same vendor, but then users of a common program will request the same updates for that program; and file sharing clients are more likely to request (the same) popular new pieces of media and less likely to share requests for obscure media.

Also, in these scenarios, users may often require their requested data to arrive in full before they can use it. For example, Web indexing, mining, and ranking operations are typically (more efficiently, and more easily) done in batch rather than incrementally; software patches are often applied offline after they have been gathered and saved; and media files are retrieved in full before they are burned to disc. Therefore, a multicast facility that serves such clients should be optimized for batch use. This optimization use requires new performance metrics and new algorithms that differ from those in previous dissemination work, as described in Section 1.3.

To be useful, a multicast facility must overcome many challenges in data scheduling (Chapters 2 and 3) and networking (Chapters 4 and 5):

Chapter 2 introduces a performance metric, *client delay*, that measures the time between a client's initial request for data and the time it first receives all the data it requested. This chapter shows how a number of different data schedulers perform under varying client loads, and how the performance of the system is affected by how much clients' requests coincide. This chapter also examines the efficiency and performance tradeoff in having clients wait to form prescheduled client batches with more shared requests. Finally, this chapter quantifies how the granularity of a data request affects the performance of a multicast system.

Chapter 3 extends Chapter 2 by considering data scheduling for data that changes over time. If the server's data changes over time, then clients may be interested in subscribing to data items (as opposed to one-time downloads)—requesting the data items if they don't have them, and remaining connected to the server to obtain new versions of the data items as they become available. For these subscriber clients, metrics based on data freshness and age become important. Chapter 3 considers how to schedule data for subscriber clients under these freshness and age metrics for a variety of client loads. This chapter also considers how a server should operate when providing data for a mix of both one-time-download clients and subscriber clients.

Chapter 4 considers a number of ways to guarantee reliable delivery, including some methods particular to our type of multicast data server. These methods, including a variable forward error correction scheme, selective retransmission of lost data, and rescheduling data items in lieu of correcting partially lost transmissions, are compared for a variety of client data-loss rates, including near-uniform and two-tiered loss-rate distributions. This chapter also considers the effect of using error-correcting data packets versus unencoded data packets for transmission; the potential benefit of a server with better knowledge of its clients' loss rates; and the effect of notifying just-connected clients when there is a relevant transmission already in progress.

Chapter 5 considers how to divide a multicast server's data and outgoing network bandwidth into multiple data channels, so that the server can accommodate clients of different download-connection speeds. Clients attempting to download data faster than their connections support will suffer transmission losses and unfairly throttle other transmissions sharing the same connection, so it is important for clients to receive no more data channels than their connection's throughput will support. This chapter considers how the number of data channels the server provides impacts system performance, and compares different ways the server can divide its outgoing bandwidth among the channels, two factors that determine the lowest-throughput client the server is willing to support. Chapter 5 also studies how to assign clients to the server's channels to consume the clients' bandwidth, evaluates how the cost of data-channel bandwidth can affect performance, and considers the performance effect of a server that knows its clients' exact connection speeds in advance and sets its data channels accordingly.

Finally, Chapter 6 describes our implementation of a multicast facility that meets the challenges detailed in Chapters 2 to 5, and measures the facility's performance when run over low-end commodity computers. This chapter describes the rationale and decision for one versus many announcement channels preannouncing data-channel transmissions; various ways for a server to verify a client's successful receipt of a data item; and how the server can break files into data items with unique fixed-length identifiers. The evaluation compares the implementation to our simulation study, determines the scaling behavior of the implementation for many hundreds of simultaneous clients, and profiles the relative computational cost of the multicast server's key activities.

This dissertation does not consider the security of multicast transmission, which requires packet authentication and/or encryption on a hostile network. For example, public-key signatures [49] can be used to manage session keys across client connections and disconnections. An analysis [56] mentions a number of alternative key-distribution proposals. A multicast server with data that must not be intercepted could adopt a multicast key management technique to ensure that only the present (and no past or future) authorized clients can decrypt the server's transmission.

1.3 Related Work

Using specialized servers to help disseminate data to large numbers of clients is not new. One technique is to have specialized intermediaries act as network "caches," reducing access latency for clients. For example, in Reference [7], the authors propose service proxies to hold popular Web data, but do not attempt to use multicast distribution to relay it to clients. As such, their design requires careful placement of proxies on the network to reduce the number of hops data must travel and therefore reduce overall network consumption. Content delivery networks such as Akamai provide and maintain such caches commercially, but their centralized control can create a single point of failure [26].

In Reference [54], the authors do consider multicast distribution (for example, using a satellite over a transatlantic link) together with caching, to improve low-latency online Web browsing. Multicast distribution becomes a secondary means of filling Web caching proxies, complementing organized hierarchical caching.

In addition to Web caching, multicast dissemination has been conceived for a number of other services, such as Teletext [5], Datacycle [30], and Usenet feeds [36]. Multicast dissemination has also been considered as part of a satellite-based Internet service [3], and as a component of publish/subscribe systems [52] such as Gryphon [46].

Perhaps most related to multicast data dissemination as described in this dissertation is the work surrounding "broadcast disks" [4], in which the intermediary disseminating data is more prominent than a cache, and is instead the primary source of data. In References [1] and [24], broadcast disks are described as a shared-distribution-channel data-dissemination technique, but driven entirely by a server without input from client requests. Later broadcast disk work [4] considers a system driven by client requests, as this dissertation assumes. Chapters 2 and 3 describe and compare work in broadcast disks against our work in more detail: Though the latter broadcast disk model resembles the multicast model used in this dissertation, different assumptions about client behavior and metrics leads to new conclusions about multicast system design.

Lastly, work on Video on Demand (VoD), which attempts to distribute videos to viewers over a broadcast network (such as television cable), appears related but only on the surface, because VoD work is able to exploit properties of video that do not apply to other data in general. For example, VoD can merge multiple requests for the same video issued at different times by slightly speeding up and slowing down video streams until they synchronize.

For additional detail, related work sections in each chapter cover additional references pertinent to that chapter, and revisit some of the above references for a more detailed comparison.

Chapter 2

Scheduling for Multiple Data Items

To design a data multicast facility optimized for the kind of data dissemination described in Chapter 1, we must address a number of interesting and challenging questions. In this chapter and the next, we focus on questions surrounding a key component of a data multicast facility, the data scheduler. The data scheduler is the part of the data server that decides how to prioritize many simultaneous and possibly conflicting client requests for data. This component is key because how well the server ranks and disseminates its requests will determine the server's effective performance to its clients.

Although our data multicast facility applies to other instances (e.g., distributing software, as described in Chapter 1), for concreteness we focus on archived Web pages, such as for the Stanford WebBase project described in Section 1.1, in these two scheduling chapters (Chapter 3 and 4).

To study the design of a good data scheduler, we consider a number of questions in this chapter:

- How does our multicast problem differ from other multicast scenarios such as broadcast disks [1]? There are a lot of similarities with broadcast disks, but also at least one key difference: in our case, a client request is not fulfilled until *all* requested data items are delivered. In traditional scenarios such as broadcast disks, on the other hand, each item would be an independent request. We will show that multicast schedules that are best for traditional scenarios may not be the best in our case. (Related work is more fully surveyed in Section 2.6.)

- What is the best way to schedule client requests? That is, how does the multicast server decide what page to multicast next? We will study various heuristics, and in particular, we present a new heuristic (which we call R/Q) that performs significantly better than other basic heuristics in our environment.

- Is it worthwhile to batch client requests? That is, is it a good idea to require clients to wait until a predetermined time or until enough other requests arrive, in order to avoid multicasting pages repeatedly? We will present experimental results quantifying potential bandwidth savings as client delays increase.

- At a practical level, what are the "units" of data that a server should offer to its clients? For example, let us consider the Stanford WebBase application: Should clients request data by individual pages? Should clients request pages by URL or by content (e.g., a query gives keywords that must appear on the page)? Or should a client request data by full sites? Or by classes of sites (e.g., `.gov`)? What are the performance implications of having larger or smaller units? Should we try to group together "popular" pages into a small number of multicast units, in order to improve performance? We will not answer these questions decisively, but we will provide some guidelines in this chapter that illustrate the impact of unit size or unit popularity on overall performance.

- What are the expected savings from our multicast facility, as compared to a scenario where all clients fetch data using individual unicast connections? For example, are the savings for Stanford WebBase significant enough to merit the expense of a central WebBase repository and multicast server? Again, we will present results that shed light on this question.

The next chapter continues our study in multicast data scheduling by considering numerous challenges that arise when the server offers changing (i.e., occasionally updated) data, so we defer questions concerning incremental updates until Chapter 3. This chapter will assume that the server's shared data is static.

This chapter makes the following contributions:

1. We define the data multicast problem for static data, and present a model and metrics for studying it (Section 2.1).

2. We define and evaluate several heuristics for scheduling transmissions (Section 2.2).

3. We address the questions outlined at the beginning of this section (Sections 2.3 to 2.5), illustrating the tradeoffs among unit size, unit popularity, client delay, number of clients, and so on.

4. We conclude with an overview of related work, contrasting the scheduling work in this chapter with other work in multicast and broadcast data scheduling.

2.1 Metrics

In a Web multicast delivery system, there is a trade-off between the network throughput the server consumes to transmit its data, and the time clients must wait for their data. Let us call the measure of the former *network cost* and the measure of the latter *client delay*, which we define formally next.

For a multicast server with a finite set of data items $\mathcal{D}$ and a finite set of clients $\mathcal{C}$, each client c is characterized by the data items it requests, $\mathcal{R}_c \subseteq \mathcal{D}, \mathcal{R}_c \neq \emptyset$, and by the time at which the client makes its request, t_c.

We define a *server schedule* S to be a sequence of elements $S = \langle s_1, s_2, s_3, \ldots \rangle$ where $\forall 1 \leq i \leq \dim(S), s_i \in \mathcal{D}$ or $s_i = \text{null}$. Intuitively, each s_i is either the data item sent in slot i, or is null, to indicate that the server sends nothing at all.

To simplify our model, in this chapter we assume that all data items $d \in \mathcal{D}$ are of equal size. Our model can be extended to variable-size items in a straightforward way, but keeping size fixed lets us focus on the more critical parameters. Furthermore, the fixed-size assumption is not unreasonable in many cases. For instance, crawlers commonly limit the number of pages that they fetch from a single Web site, to avoid overloading the site. Thus, regardless of the size of the site, roughly the same amount of data is retrieved. In such a case, if clients request whole Web sites, then items will be roughly of the same size. Because we expect Web sites to be a common and convenient unit for researchers using WebBase to request, we will think of data items as Web sites in this chapter, and use the terms interchangeably.

Since the data items (crawled Web sites) are the same size, it takes the same time to transmit each item. This means that the multicast for slot i ends at time $t_{\mathcal{D}} i$, where $t_{\mathcal{D}}$ is the time to transmit one site. (For simplicity, we assume that network idle periods are also a multiple of $t_{\mathcal{D}}$.) A particular client $c \in \mathcal{C}$ will have its request for Web site $d \in \mathcal{R}_c$ *satisfied* the first time after t_c that d is transmitted. That is, the request will be satisfied at time

$$t_{cd} = t_{\mathcal{D}} \min\{i \in \mathbb{Z} | t_{\mathcal{D}} i > t_c \wedge s_i = d\}.$$

We can now define two metrics for Web multicast:

- The *average client delay time* or *delay* is the (arithmetic) mean of the time clients wait for their *last* requested Web site to be satisfied. That is,

$$D = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \left(\max \{t_{cd} | d \in \mathcal{R}_c\} - t_c \right).$$

- The *amortized network cost* or *network cost* is the amount of data transmitted for the full server schedule, divided by the number of clients (whose requests are completely satisfied by the full schedule). The schedule goes from slot 1 to slot

$$\dim(S) = \frac{\max \{t_{cd} | c \in \mathcal{C} \wedge d \in \mathcal{R}_c\}}{t_{\mathcal{D}}}.$$

  We need to subtract from these $\dim(S)$ slots, the slots that were unused, and then multiply the number of resulting utilized slots by the size of each slot, $N_{\mathcal{D}}$. Thus,

$$N = \frac{1}{|\mathcal{C}|} N_{\mathcal{D}} [S - |\{t \in \mathbb{Z} | 1 \leq t \leq \dim(S) \wedge s_t = \text{null}\}|].$$

In the literature on broadcast disks and other multicast systems, clients request one data item (for us, one Web site) at a time. The assumption is that clients can start "working on" received items as soon as they arrive. Hence, the *average client* *response time* or *response time* is typically computed as:

$$R = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \sum_{d \in \mathcal{R}_c} (t_{cd} - t_c)$$

Such a metric is *not* appropriate for a Web multicast scenario, because typically clients do not start their work until *all* requested items are delivered. For example, most text indexes are built in batch mode, by specifying a directory that contains all documents to index. (Incremental indexing is possible, but not used much in practice.) In our case, we would want the images of all the Web sites we are interested in before building an index for the collection of pages. Furthermore, some indexing functions such as PageRank used by Google [12] are hard to do incrementally. So, again, it is better to wait for all data items (Web sites) to arrive, before starting to compute global statistics such as PageRank or citation counts. Thus, we believe it is more appropriate for us to use the client delay metric defined earlier, as opposed to the client response time of broadcast disks.

Although we do not use client response time, it is instructive to compare the two metrics. Interestingly, the two notions of client delay and response time are not equivalent, as the following example demonstrates.

Example. Let clients A, B, and C all request Web sites $\{a,\ b\}$. Let clients D and E both request site $\{c\}$, and let client F request site $\{d\}$. Let all six clients make their requests at the same time, and assume that $t_{\mathcal{D}} = 1$ time unit.

In this scenario, a schedule that minimizes the response time metric *will fail* to minimize the delay metric, and vice versa. Since the number of clients is fixed, we can compare the total, rather than average, delay and response time over the six clients for simplicity. The optimum schedules for response time, $\langle a, b, c, d\rangle$ and $\langle b, a, c, d\rangle$, are illustrated in the table below. Each row represents one client, and the bullets show when a client's item is serviced. (Recall that in this example the clients make their requests at the same time.) Note that both minimal response time schedules happen to be described as the same table.

	a	b	c	d	Delay for
or	b	a	c	d	each client
A	•	•			2
B	•	•			2
C	•	•			2
D			•		3
E			•		3
F				•	4
Total response time for each data item	3	6	6	4	

For example, the bullet at (D, c) indicates that, when c is transmitted third, client D's request for c is satisfied. Because clients D and E both request c, both clients accumulate three units of response time waiting for c. Therefore, clients accumulate six units of time in total waiting for c, which we record in the bottom row under c. Further, because both clients D and E have their entire requests satisfied at the end of three units of time, the client delay for both clients is 3, which we record in the rightmost column for rows D and E.

The minimum total response time is, from the bottom row of the table, 3+6+6+4 = 19, the sum of the response times for each requested data item. The total delay, from the rightmost row of the table, is 2+2+2+3+3+4 = 16, the total of the delays for each client. Although we do not show it here, no other schedules produce a better response time.

The schedules to minimize client delay are $\langle c, a, b, d\rangle$, or $\langle c, b, a, d\rangle$. Again, the two schedules correspond to the same table:

	c	a	b	d	Delay for
or	c	b	a	d	each client
A		•	•		3
B		•	•		3
C		•	•		3
D	•				1
E	•				1
F				•	4
Response time:	2	6	9	4	

The total response time, from the table: 2+6+9+4 = 21. Similarly, the minimum total delay: 3+3+3+1+1+4 = 15. No other schedules have lower delay. □

Intuitively, one can say that the difference between the response time metric and the delay metric is that the response time metric, when extended to clients making requests of more than one data item, rewards the early distribution of partial requests. By contrast, the client delay metric cares only when the last item of a request is transmitted, and is not affected by when intermediate items are sent. In the rest of this chapter, we only focus on the delay and network cost metrics.

2.2 Basic Scheduling Heuristics

The server scheduling heuristic determines what data item to broadcast to the multicast clients, given information about the currently listening clients and their requests. While one can easily make up a simple heuristic that will do the job, a more careful choice can significantly reduce average client delay.

In this section, we will describe four basic heuristics for server scheduling, two simple, one culled from existing broadcast disk work, and one we invent, tailored to our goals. We then describe the simulation in which we compare them, and note how tailoring the scheduling heuristic provides a significant win.

2.2.1 Popularity

Perhaps the first heuristic that comes to mind for multicast scheduling is to have the server, at each slot, send a data item that the largest number of clients are requesting at the time. One might reason that sending the data item that is most requested satisfies the most requests for any item. Unfortunately, this heuristic has cases in which it performs poorly, as shown in the next example.

Example. Let client A request Web site $\{a\}$; clients B and C request sites $\{b$, c, $d\}$; and all three clients appear at the same time. Again assume that $t_{\mathcal{D}} = 1$. The popularity heuristic would generate a server schedule similar to $\langle b, c, d, a \rangle$, where the more popular data items b, c, and d necessarily precede a. This schedule incurs an average client delay of $(4 + 3 + 3)/3 = 3.3$ units.

It turns out, however, that placing the three more popular items first is precisely the wrong thing to do: The server schedule $\langle a, b, c, d \rangle$ has a lower average client delay of $(1 + 4 + 4)/3 = 3$.

Clearly, we can make the penalty for doing the wrong thing arbitrarily large; adding more Web sites for clients B and C to request increases the difference in client delay between the popularity heuristic and the optimum schedule. $\square$

What is happening is clear: In a situation where a large number of clients request a lot of data, clients making very small requests are effectively forced to queue after the large requests, even though they could "jump the queue" without causing great pain for everyone else. In the schedule where the short request comes first, the client is quickly satisfied, contributing a low client delay rather than a high one to the overall average. In the schedule where the short request is forced to queue because it is not popular enough, the same client accumulates a very large delay instead, which brings up the overall average needlessly.

2.2.2 Shortest Time to Completion

The above example suggests a scheduling approach borrowed from operating systems task scheduling: have the server service first the clients that have the shortest time to completion. This heuristic is described in Figure 2.1.

Input: A set of clients and their still-pending requests
Output: The data item to transmit
  $MinRequestSize \leftarrow \infty$;
  $MinRequestClient \leftarrow$ null;
  foreach client c
    if size of remaining request of $c < MinRequestSize$
      $MinRequestSize \leftarrow$ size of remaining request of c;
      $MinRequestClient \leftarrow c$;
  Choose a data item requestedby $MinRequestClient$.

Figure 2.1: Shortest Completion Time Heuristic

In the last step, this heuristic can choose any item requested by the client with the minimum amount of data to transmit ($MinRequestClient$) and achieve its goal.

(In practice, one may choose the most popular of the items requested by MinRequest-Client, in hopes this may reduce more clients' remaining request size. This is exactly how we implement this heuristic.)

It turns out this heuristic, no matter how its last step is implemented, does not guarantee a minimal client delay either.

Example. Let client A request Web sites $\{a\}$; clients B, C, and D request sites $\{b, c\}$; all clients appear at the same time; and $S_T = 1$. The shortest-time-to-completion heuristic would generate a schedule such as $\langle a, b, c \rangle$, in which data item a comes first. This yields an average client delay of $(1 + 3 + 3 + 3)/4 = 2.5$.

Such a schedule is exactly the wrong thing to do; putting the more popular data items first (such as a schedule $\langle b, c, a \rangle$) in this case yields the smaller average client delay $(3 + 2 + 2 + 2)/4 = 2.25$. Again, we can make the penalty for doing the wrong thing arbitrarily large; this time, we simply add more clients making the request $\{b, c\}$. □

In this example, we see the inverse of the problem that plagued the popularity heuristic: There *are* times when transmitting more popular items is a better approach, because doing so satisfies a larger number of clients sooner. Even though the simple approaches are subject to pathological behavior, they will be retained for comparison in later simulated runs.

2.2.3 RxW

The *RxW* heuristic is shown in Figure 2.2. The heuristic is designed for the scenario in which every client requests a single data item (a single Web site). It seeks to favor more popular items (sites) and avoid the starvation of any clients. The name comes from the score it assigns to each item (Web site), the product of R, the number of clients requesting the item, and W, the longest amount of time any client has been listening (waiting) for the item.

Input: A set of clients and their still-pending requests; current time t
Output: The data item to transmit

```
ChunkToSend ← null;
MaxRW ← −∞;
foreach data item i
    R_i ← 0;
    W_i ← 0;
    foreach client c
        if c requests i
            R_i ← R_i + 1;
            if t - (time c began requesting i) > W_i
                W_i ← t− (time c began requesting i);
    RW_i ← R_i × W_i;
    if RW_i > MaxRW
        ChunkToSend ← i;
        MaxRW ← RW_i;
return ChunkToSend.
```

Figure 2.2: RxW Heuristic

2.2.4 R/Q

It appears from observing the popularity and shortest-time-to-completion heuristics that balancing the two heuristics may yield better results than either of the two alone; while either heuristic can make very poor suggestions, a combination of the two seems less likely to do the same, since the two heuristics make poor suggestions under different circumstances.

We invent one possible way to combine the wisdom of the two heuristics, which we call the R/Q heuristic, shown in Figure 2.3. In this heuristic R counts how many clients request a data item, and Q holds the $MinRequestSize$ for that item (from the shortest-time-to-completion heuristic).

```
Input: A set of clients and their still-pending requests; current time t
Output: The data item to transmit
  ChunkToSend ← null;
  MaxRQ ← −∞;
  foreach data item i
    R_i ← 0;
    Q_i ← 0;
    foreach client c
      if c requests i
        R_i ← R_i + 1;
        Q ← number of items c still has pending;
        if Q < Q_i
          Q_i ← Q;
    RQ_i ← R_i/Q_i;
    if RQ_i > MaxRQ // Save item with largest R_i/Q_i
      ChunkToSend ← i;
      MaxRQ ← RQ_i;
  return ChunkToSend.
```

Figure 2.3: R/Q Heuristic

2.3 Simulating the Heuristics

To compare the heuristics in our Web multicast model, we use a simulation to determine the client delay and network cost each heuristic would incur under a variety of conditions. We first describe the parameters of the simulation, then examine how the heuristics fare relative to each other and to unicast distribution.

2.3.1 Description

In our simulation, a server is presumed to have all its $|\mathcal{D}|$ uniform-size data items (representing crawls of Web sites), ready to distribute. Clients appear at exponentially-distributed random intervals, so that on average a new client appears every $t_{\mathcal{C}}$ units of time.[1] Each client, when it appears, listens to a (unique) multicast tree with root

[1]Given our simplifying assumption of fixed data item size, we model time as fixed-size slots. Thus, clients may appear only at the beginning of a slot. This simplification does not have significant effect

Variable	Description	Base value
$\lvert\mathcal{D}\rvert$	Total number of data items (Web sites)	10000
$h_{\mathcal{D}}$	Fraction of data (Web sites) deemed hot	0.20
$h_{\mathcal{R}}$	Fraction of requests that are of hot sites (approximate)	0.80
$t_{\mathcal{C}}$	Average time between clients	30 minutes
r	Average size of client requests (Web sites)	20
$N_{\mathcal{D}}$	Size of each item (Web site) (kilobytes)	7800
$t_{\mathcal{D}}$	Time to transmit one item (Web site)	1 minute

Table 2.1: Simulation Parameters and their Base Values

at the server, and remains listening until it has received every data item it requested. After its requests are satisfied, the client leaves the multicast tree and requires no further service. The server is presumed to have a client's site requests at the time the client appears; that is, a client takes zero time to issue its requests to the server. The costs of setting up the multicast tree, and of issuing data requests, are dismissed because they are small relative to the data transfer costs.

To refine the model further, each data item has the property of being either "hot" or "cold," indicating how much it is requested relative to other items. A fraction $0 \leq h_{\mathcal{D}} \leq 1$ of the items are designated *hot*, the remainder *cold*. A fraction $h_{\mathcal{R}} \geq 0.5$ of all requests, made by any client on any item, are of hot items. The remainder of all requests $1 - h_{\mathcal{R}}$ are made on cold items. The clients make requests of an exponentially-distributed random number of items, with r sites as the arithmetic mean.

Table 2.1 summarizes the simulation parameters, and shows the base values we use initially. The base values in the first five lines of the table were chosen simply to have a reasonable scenario that differentiated the performance of the heuristics. (If the system is either too loaded or too unloaded, all the heuristics will end up having similar performance results.) For example, we invoke the commonly observed 80-20 rule in differentiating hot and cold Web sites. To estimate the average size of client requests, we consider some of the example scenarios for WebBase introduced in Section 1.1. We guess that analysts may track on the order of twenty competitor

on our results.

and review sites, and similarly, that a developer may collect information from about twenty sites, so we chose twenty as the initial value for r. For $|\mathcal{D}|$, the total number of Web sites crawled, we chose 10000. This is the largest number our simulation could handle without making run times huge. While one can envision crawlers that visit more than 10000 sites, a crawler that only visits 10000 "important" sites is also feasible. Such a more focused crawler would be able to revisit its sites more frequently, maintaining its cached images more up-to-date, and hence provide a higher-quality service to its clients.

The values in the last two lines of the table are chosen to approximate expected transmission costs. As mentioned earlier, many crawlers limit the number of pages retrieved from a single site to a few thousand. In particular, our own WebBase crawler fetches 3000 pages, so we use this number. The average size of a Web page in our WebBase is 2.6 kilobytes compressed (gzip), so a Web site would be about 7800 kilobytes. We assume that clients request full Web sites, so we set the data item size $N_{\mathcal{D}}$ to this value. The unit of time ($t_{\mathcal{D}}$), one minute, comes from an assumption that we have at least 1 megabit/sec at our disposal (a reasonable value if 10BaseT Ethernet is the limiting factor for multicast throughput). If so, then 7800 kilobytes takes about 61 seconds to transmit.

There are of course many other "reasonable" settings, and we do study changes to the base values later on. In particular, in Section 2.3.2 we study the impact of varying $|\mathcal{D}|$, and show that the *relative* performance of the heuristics is unaffected by this scale.

2.3.2 Initial Results

First, let us compare the scheduling heuristics for a multicast facility. In our plots for this chapter, "MinTime" refers to the shortest-time-to-completion heuristic; "Pop" refers to the popularity heuristic; "RxW" and "R/Q" refer to the heuristics of the same name as described earlier. "Unicast" refers to the corresponding network cost for unicast distribution, and is represented as a constant cost of one client getting all its data; in client delay, that is $rt_{\mathcal{D}}$. All simulations are run at least until the 95%

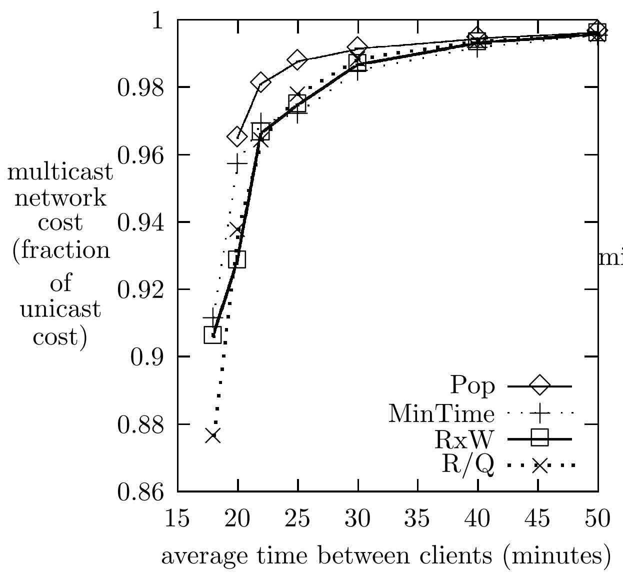

Figure 2.4: Benefit in amortized network cost

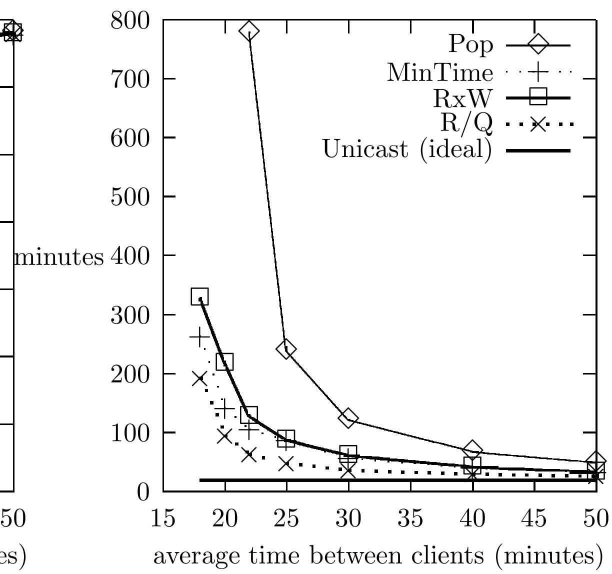

Figure 2.5: Client delay

confidence interval for average client delay is smaller than 10% of the value itself.

A typical network cost plot appears in Figure 2.4. In this plot, we consider the network cost of multicast distribution as a function of the average client interarrival time ($t_{\mathcal{C}}$). The network cost of multicast distribution is plotted on the vertical axis, as a fraction of the network cost unicast distribution would incur providing the service. For example, at $t_{\mathcal{C}} = 20$ (clients arriving on average twenty minutes apart), we see that multicast using RxW and R/Q incurs just under 94% of the network cost unicast would have incurred in sending each client request separately. Similarly, Pop and MinTime incur about 96% of the unicast cost at the same load. Because multicast sends only requested data, it cannot send requested data more than once per client; therefore, it is impossible for multicast network cost to exceed unicast network cost. (Note that in Figure 2.4 the results for Pop end at lighter load. This is because the Pop results did not stabilize even in long runs.)

As we move left in the figure, we see that the multicast scheme is a win in network cost over unicast distribution, especially as multicast gains more clients listening at the same time from higher load. For the scenarios plotted, multicast distribution saves up to some 10% of the network cost of unicast. Since the scenario assumes an average of 20 Web sites per client (yielding 15600 kilobytes of data), this is a savings of over one megabyte per client. As load increases, this network savings increases as shown by the drop in relative multicast network cost at the left of the graph. Of course, as the reader recalls, these graph omits the cost of the crawl that creates the server's copies of its Web sites, but that is a fixed cost that is divided over the number of clients served by the data (or in this simulation, how long the server runs). The multicast curves, then, would translate up the graph slightly, but as the server gets more use, this translation quickly becomes negligible. Meanwhile, the Web servers whose data is available through the multicast facility accumulate network cost savings from not being repeatedly crawled. With our base parameter values, for example, in which clients appear every half hour at our multicast facility instead of crawling the Web servers directly, these Web servers would save 607345 kilobytes of network transmission costs every week.

The graph also assumes a paying-by-the-bit perspective, in which multicasting one data item incurs the same cost as unicasting it. On the other hand, one might consider a more "global" view of network consumption in which a multicast data item consumes more network resources in the world than a unicast one, because a multicast item traverses more network links to more clients; in this case, one would have to adjust the curves. Even in this case, though, multicast is a win over unicast distribution, if the multicast facility multicasts data only to the clients that request it. This idea can be implemented by adding a low-traffic "control" multicast tree through which the multicast facility indicates its upcoming data transmission. By monitoring the control tree, clients can subscribe to the main "data" dissemination multicast tree only when the data items they wish to receive are being sent. As a result, the data tree always has exactly the clients that want the data being distributed, and no more. With such a design, multicasting one data item would only incur the negligible additional network cost of the "control" tree, and whenever multiple clients are listening to the "data" tree, multicast is saving the network cost of duplicating the data for each client.

From Figure 2.4 we also see that the network cost of the multicast schemes does not vary much by scheduling heuristic; the gap between good and poor heuristics is under 4% of the unicast cost. So, it is the client delay that will differentiate the schedulers.

Figure 2.5 is the average client delay graph that corresponds to the same simulations plotted in Figure 2.4; again, the results in this plot are typical. The average client delay is plotted as a function of varying average client interarrival time ($t_{\mathcal{C}}$). For example, if clients arrive on average every thirty minutes, they suffer an average client delay of 36 minutes on a multicast scheme using R/Q, around 58 minutes on a multicast scheme using RxW or the shortest-time-to-completion (MinTime) heuristic, and nearly 122 minutes on a multicast scheme using the popularity (Pop) heuristic. (In Figure 2.5, Pop's delay (4075 minutes) at 20 minutes inter-arrival time is not shown because it would compress the vertical axis.) As we can see in the figure, the average client delay grows rapidly as the average time between client arrivals falls; this is natural, as falling interarrival time indicates more clients (and their different data item requests) competing for the server's constant transmission capacity.

We see most notably in this figure RxW and MinTime's significant gains over Pop, and then an even lower average client delay from our tailored-to-the-task heuristic R/Q. This suggests that there is a pure benefit to careful choice or design of the multicast scheduler. This benefit arises because of the difference between our multicast scenario and the typical broadcast disk or broadcast delivery scheme, in the way clients request data and the way we measure how long clients wait for their data. For the values shown in the figure, this benefit from R/Q is an average client delay of less than 80%, and as low as 54%, of the average client delay incurred by the next-lowest-delay heuristic. Compared to the poor-performing Pop, we see that R/Q incurs up to an order of magnitude smaller client delay.

Lastly, we point out that these results apply to scenarios of widely varying scale in the same way. For instance, in Figure 2.6, we plot the average client delay of our heuristics as a function of the number of data items the server maintains ($|\mathcal{D}|$). The request skew—the number of hot data items—is scaled with $|\mathcal{D}|$, meaning that $h_{\mathcal{D}}$ is fixed and hot and cold items remain a fixed fraction of the total number of server items. Other parameters are set to their base values. Notice that the delay begins

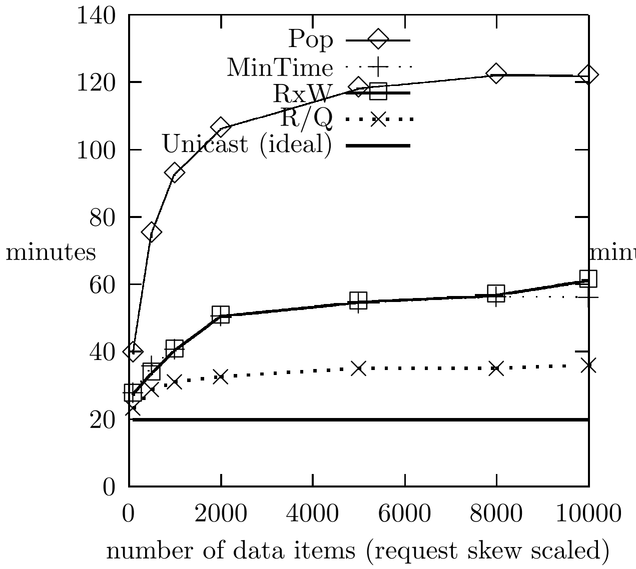

Figure 2.6: Heuristics compare similarly with more data

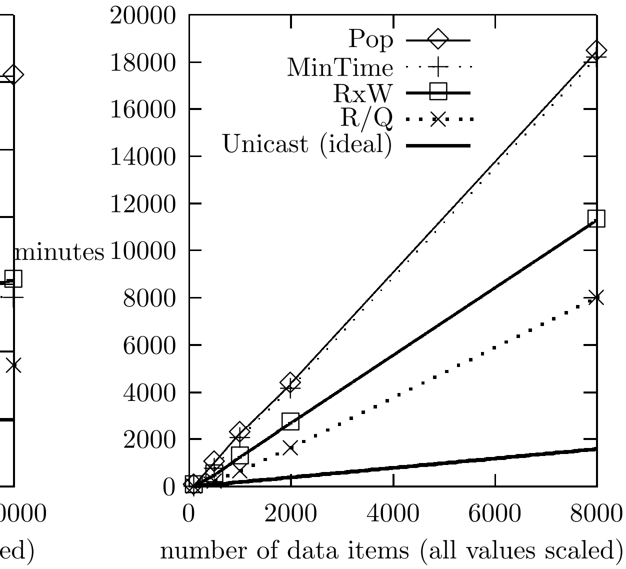

Figure 2.7: Heuristics compare similarly at larger scale

to taper off for the range we have plotted. The average client delay tapers off as we move to the right of the figure because the falling probability of overlapping requests (induced by a rising $|\mathcal{D}|$) marches towards multicast's worst case: client requests having no overlap at all. As we approach the worst case asymptotically, client delay approaches its worst-case value. Notice, also, that the system is stable, no matter what the scale. That is, every client that appears will be satisfied, within a period of time no longer than the time it takes the multicast server to multicast all its data.

We see in this figure that the heuristics perform the same way relative to each other even with widely varying $|\mathcal{D}|$. For example, for $|\mathcal{D}| = 1000$ data items, a multicast scheme with R/Q still incurs less than 80% (about 77%) of the client delay of the next-lowest-delay heuristic, and incurs about a third of the client delay of a multicast scheme using Pop, numbers not far from what we saw for the next order of magnitude ($|\mathcal{D}| = 10000$, our base value).

In Figure 2.7, we see that the same is true if we took a small case and then increased the scale of all the factors relating the number of data items (total number of items $|\mathcal{D}|$, number of hot items $h_{\mathcal{D}}|\mathcal{D}|$, and average client request size r) concurrently.

Again, we plot average client delay for our heuristics as we vary $|\mathcal{D}|$, the total number of data items, over a large range. For this latter case, unlike Figure 2.6, we had clients request a significant fraction ($r = 0.2|\mathcal{D}|$) of the entire repository, so that clients would request an average of at least 20 Web sites even in the smallest case shown. Otherwise, the parameters in each simulation remain proportional to our base parameter values for the $|\mathcal{D}| = 10000$ case. We notice that in Figure 2.7, unlike Figure 2.6, delay continues to increase. This is because the number of Web sites requested by each client continues to grow as we move to the right in the figure.

Here in Figure 2.7, too, we see that the proportion between client delays between differently-performing heuristics stays fairly constant for a wide range of values. For example, we can see in Figure 2.7 that Pop accumulates about twice as much client delay as RxW and MinTime, which in turn accumulate about one and a half times as much delay as R/Q, for the range of values shown.

To confirm the relative client delay performance of our heuristics that we are seeing in these figures, we also ran and examined a large number of smaller, faster running simulations ($|\mathcal{D}| = 100$ items (Web sites)). In these experiments, we found again that R/Q typically incurs between 60% and 85% of the client delay that RxW, the next lowest-delay heuristic, incurs over the same parameters. There are a few notable exceptions, in which R/Q incurs nearly as much delay as RxW (nearly enough for the difference in delay incurred by the two heuristics to be statistically insignificant), but these exceptions are fairly easy to describe. For example, R/Q incurs nearly as much delay as RxW when

- the average client request size is very small (r is near 1). In this case, the times to completion for almost all clients are very small (and comparable). Because all clients will finish in about the same time, the shortest-time-to-completion heuristic contributes less information.

- the average time between new clients is very large ($t_{\mathcal{C}} \geq 50$, meaning in this case that $t_{\mathcal{C}} \geq \mathcal{D}/2$). This indicates a very lightly loaded case, in which clients are so infrequent that very few clients are listening to the multicast stream at any one time. In this case, the server simply provides service to the few clients that are present, and less room remains for improving the average client delay. (As $t_{\mathcal{C}}$ grows, multicast and unicast converge into the same form of data delivery.)

With the growth in client delay, it is likely that the limits on a multicast facility's scale depend on the time clients are able to wait for their data. For Figure 2.6, for example, we see that when clients request about 20 Web sites from a (R/Q) multicast facility's repository of over 5000 such data items, the client can expect to wait on average 35 minutes (a little over half an hour) for its 156 megabytes of data. Half an hour is probably tolerable to the analyst going to lunch, or the traveller packing suitcases, during the download. In fact, one would be hard-pressed to do better with a well-behaved personal crawler. If we say, for example, that we have a bulk-processing applications with a delay limit of one day, then from Figure 2.7, we see the multicast facility has an effective limit of around 6000 Web sites even with R/Q, if clients request on average around 20%, or 1200, of them at a time. This suggests that in very large scale applications, it may be helpful to relegate very cold Web sites to unicast distribution, where they will not incur client delay to most clients, and use the multicast facility to distribute the most popular sites (the most popular 6000 of them) for significant network cost savings.

Still, it is worth pointing out that the limits of a multicast server's scale also depends on the popularity skew of the data. In Figures 2.8 and 2.9, we see how the skew in access requests affects the multicast system's performance. Along the horizontal axis, we vary the number of hot data items, so that very few items get most of the requests at the left side of each figure, and every item has the same chance of access at the right edge of each figure. We plot the network cost per satisfied client and the average client delay of a multicast facility using R/Q on the vertical axes. For example, we see that if h falls from our base value of 0.20 to 0.05 (500 hot items), average client delay falls 5% and network cost per client falls 2%. The gains are more significant as $h_{\mathcal{D}}$ falls further—that is, as clients make more similar requests (fewer items account for more of the requests). It appears, therefore, that having multiple multicast servers not only benefits clients by allowing them to reach a closer server, but more significantly, it allows servers and clients to self-select into pools of similar interests for improved multicast performance—reduced network

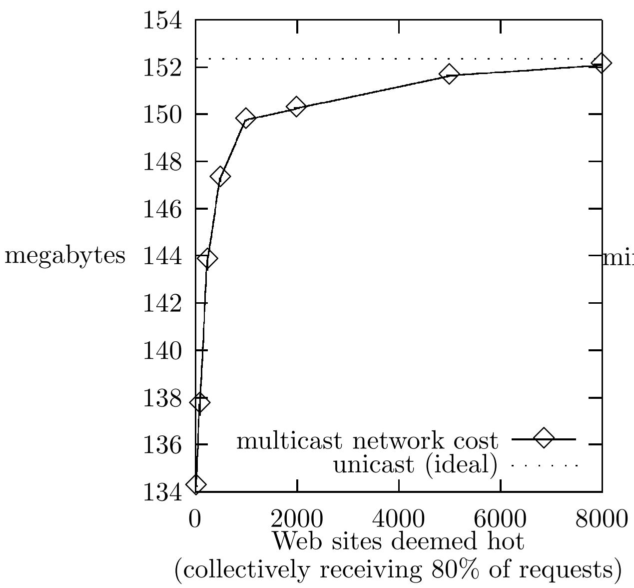

Figure 2.8: Network cost varies by number of hot items

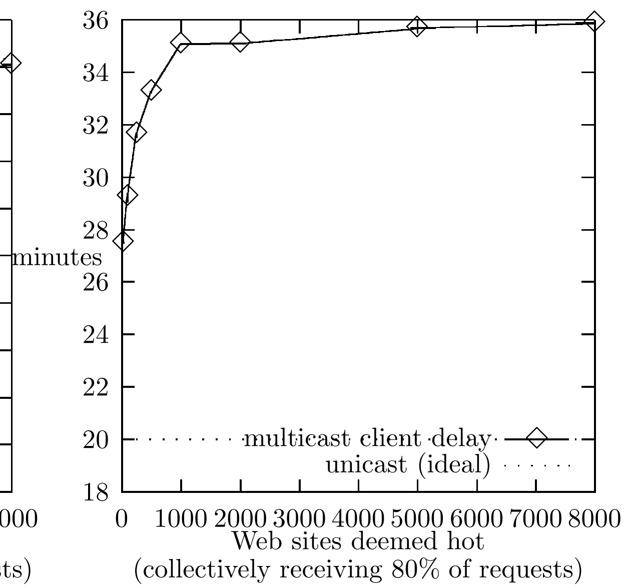

Figure 2.9: Delay varies by number of hot items

consumption and lower client delay.

2.4 Batching Clients

Another decision to consider in the design of a multicast facility is whether to delay clients until the next start of a periodic interval, thereby batching them to enhance the overlap in their requests. If we batch clients into fixed intervals this way, then we may be able to broadcast a server's Web site data less often, saving network cost by incurring client delay. In the meanwhile, the client may be released to do other things before the next interval begins. To consider the effects of batching clients, the simulation runs in this section assume that a given number of clients start at the same time; to get total client delay, we then add the expected time a client will wait for the next interval to the client delay incurred in the simulation.

First, let us note that the decision of grouping clients into batches is orthogonal to the choice of server scheduler. That is, the relative ordering of the heuristics is roughly the same, regardless of whether clients arrive at once, or arrive as scheduling

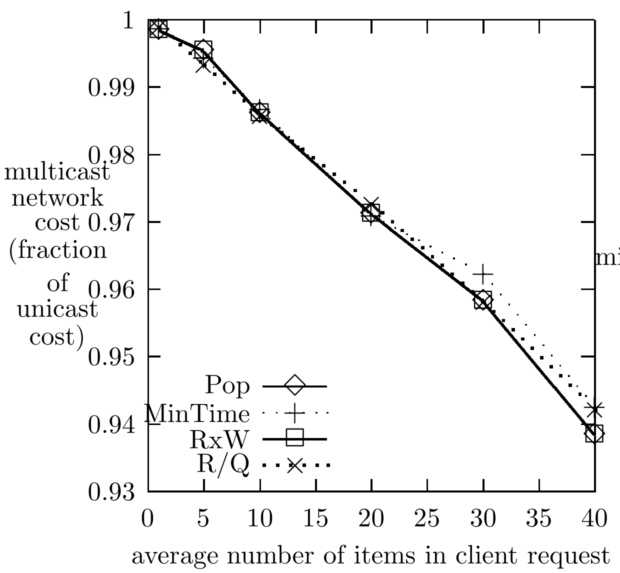

Figure 2.10: Benefit in network cost for batched clients

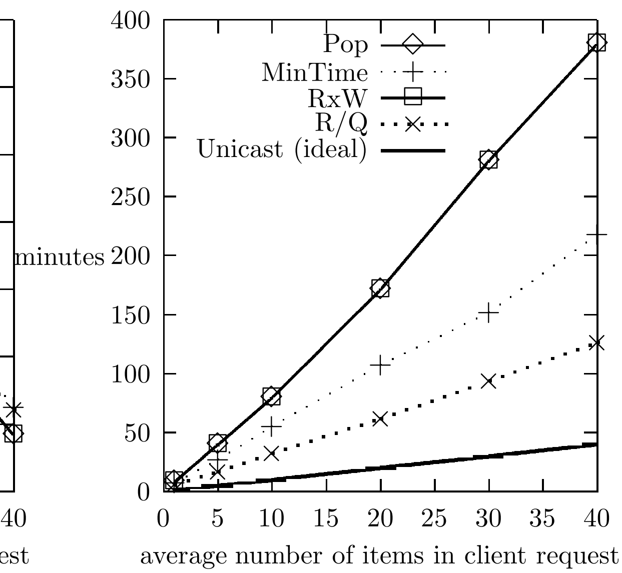

Figure 2.11: Client delay for batched clients

decisions are made. Figures 2.10 and 2.11 compare the heuristics when clients arrive at once. The scenario graphed starts with our base parameters, except the client average interarrival time, which does not apply. Instead, we have ten clients start at once, requesting a random number of Web sites, whose average we varied along the horizontal axis. (The number of clients starting at once was chosen small to emphasize the graph's shape; curves grow higher and details become harder to see as the parameters grow large.)

Here, with clients batched, the "W" in RxW (which represented the longest time any client has waited for a particular data item) is identical for every data item at every point in time. As a result, the W does not provide any information to the heuristic, and the RxW scheduler becomes equivalent to Pop. Also, because all the clients begin listening at the same time, we find that as they listen to more data, multicast becomes more efficient relative to unicast distribution. For example, when clients request forty items (Web sites), the multicast schemes send only 93% of the data that a unicast scheme would. In the limit, clients request all the multicast server's data, in which case client delay is exactly the same for multicast or unicast, but the data is transmitted only once in the multicast case. Last, we observe that as before, R/Q shows that a tailored heuristic can substantially improve the performance of the multicast facility. For instance, with forty items requested, R/Q cuts delay by a factor of three from Pop and RxW, and nearly a factor of two from MinTime. As a result, from here we will now simply assume that the multicast system uses R/Q as its scheduler.

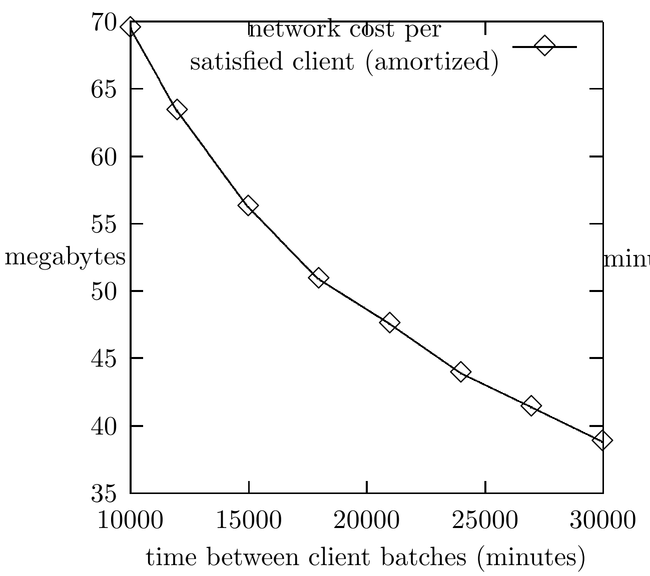

Figure 2.12: Amortized network cost for batched clients

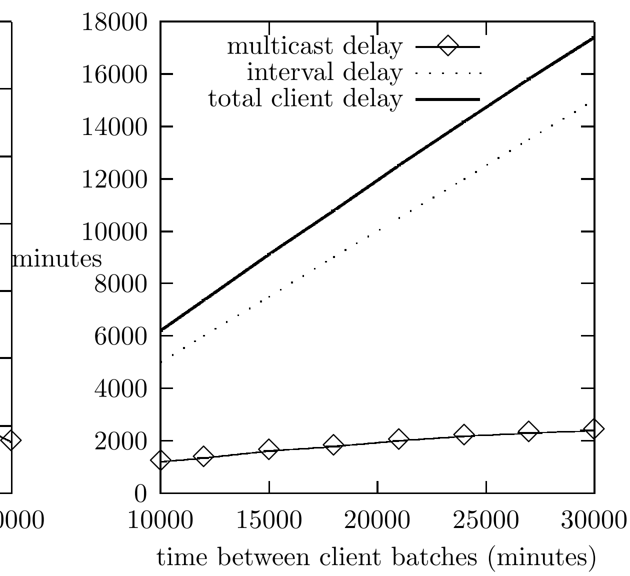

Figure 2.13: Client delay for batched clients

Now, we turn to consider how we might be able to batch clients into intervals. In Figures 2.12 and 2.13 are the network cost and client delay of a batched multicast facility. The simulation scenario again uses our base parameter values. On the bottom axis is the time between intervals at which clients begin receiving service. A client that misses the start of an interval waits until the beginning of the next interval for service.

Notice that the minimal interval spacing on the horizontal axis is 10000 minutes; this is because clients at most request all the server's data items (Web sites). Therefore, to ensure all clients can be serviced within their interval, the interval spacing t_I must be chosen so that $t_I \geq |\mathcal{D}| t_{\mathcal{D}}$. Notice that this argument applies even if we bound the number of data items a client can request; this is because clients individually need not request all a server's data, but it is easy to see how several clients with little overlap in their requests can collectively request all of a server's data. Further, because a client waits for the start of the next interval, it will, on average, be delayed by extra time $t_I/2$ waiting for the next interval. This $t_I/2$ delay is plotted in Figure 2.13 as "interval delay" (the dotted line).

Also, notice the dramatic fall in network cost as the interval spacing t_I grows. This is because with clients appearing on average every $t_\mathcal{C}$ time, we expect the server to accumulate $t_I/t_\mathcal{C}$ clients at the start of each interval. Even if the server must multicast all its data during an interval, this means the server is incurring only $|\mathcal{D}|N_\mathcal{D}/(t_I/t_\mathcal{C}) = |\mathcal{D}|N_\mathcal{D}t_\mathcal{C}/t_I$ units of network cost per client, an upper bound that falls rapidly as t_I grows. In contrast, if we did not batch clients in our multicast service, the upper-bound network cost per client is a larger $N_\mathcal{D}t_\mathcal{C} \geq |\mathcal{D}|N_\mathcal{D}t_\mathcal{C}/t_I$. (Intuitively, this value comes from assigning the network cost incurred by the multicast server to a client during the time that client has appeared onto the multicast tree, but the next client to appear has not.)

As we can see in the figures, the substantial component of client delay in a batched environment comes not from the additional delay during the multicast dissemination, in which the server copes with a higher load; instead, it comes from the expected delay for clients waiting for the next interval to begin. On the other hand, we see that this delay brings a substantial reduction in network cost even for the minimal interval spacing. (Seventy megabytes of network cost is less than half the unicast network cost for distributing twenty Web site images, which is over 152 megabytes.)

For situations in which clients are relatively impatient, as might be the case for end users requesting Web sites or software components for offline use, the over-five-fold increase in client delay might become too painful to bear. In scenarios where clients are not as sensitive to delay, however, such as periodic multicast to specialized search engines building new Web indexes, this tradeoff is easier to make. We can find the longest acceptable delay, then use it to estimate the longest acceptable time between client batches and keep the network cost savings. For instance, for WebBase we expect that a week-long average delay is acceptable to clients, so batching clients every week and a half is a good choice. (Using Figure 2.13, the average total client delay is 10080 minutes, or a week, is achieved when the time between client batches is a little over a week and a half; recall that on average, clients will not wait the entire interval length.) Alternatively, we may have a budget for network costs from which we derive a minimum time between client batches, and use it estimate the average client delay our clients would suffer.

As an alternative method of batching clients, a server might decide to allow smaller interval spacings than $|\mathcal{D}|t_{\mathcal{D}}$. To do so, however, the server must track the size of the union of all its clients' requested data, to ensure that the size can still be transmitted within an interval. This means that the server must be willing to start a new interval prematurely once the server can no longer accommodate any new items in its union of requested data, or that the server can arbitrarily delay clients beyond the next scheduled interval if their requests would grow the union of requested data too much. Also, should a client request more data than an interval allows, the server must still adjust its interval period at least once to accommodate it. In any case, this means that the server will have difficulty reliably predicting a client's start time, the time when the client must be present to receive the multicast data. In contrast, the batching scheme described in this section allows a client to make a request, learn when the next fixed-size interval begins, and go do other things until then. Whether a server should consider variable intervals, then, may depend on whether its clients value a known, prescheduled start time, and how much this information ameliorates the longer interval delay.

2.5 How Big a Data Item?

One last decision in the design of a multicast dissemination system that we will consider is the unit of transfer of data. For example, one might propose that one server data item should be one Web resource—one Web page. This provides fairly high granularity, but creates its accompanying problems of scale. On the other hand, one might propose that a data item should be one Web site, as we have assumed so far, or even a set of sites.

Let us assume that clients continue to request data items (e.g., Web sites), but that the server has grouped the items (sites) into larger *chunks* of the Web. In particular, let us define a parameter $u \geq 1$ that represents the number of items in a server chunk. With this parameter, a server now manages fewer ($|\mathcal{D}|/u$) large chunks, rather than more ($|\mathcal{D}|$) items.

As chunks get larger (u grows), it becomes increasingly likely that a client receives excessive amounts of data to get the items it seeks. That is, a client wants a particular item, but must request a large server chunk (of lots of items) in order to get it. This leads to increased client delay and possibly, higher network cost, as the multicast server sends more data than its clients strictly need.

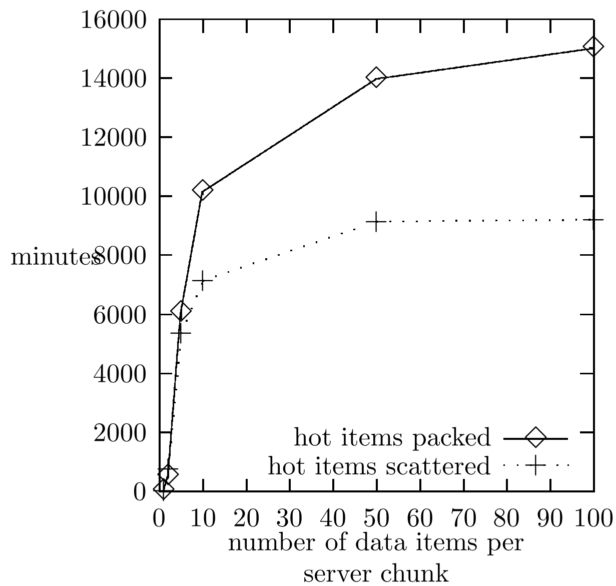

Figure 2.14: Client delay grows as server chunks grow larger

In Figure 2.14, we see how client delay is affected by varying u on the horizontal axis. We begin with a server having our base parameter values (10000 items, clients appearing randomly, thirty minutes apart on average). With this scenario as the base case, we increase u so that the server has fewer ($|\mathcal{D}|/u$) chunks to manage, but clients still require data in terms of items. Therefore, they request as many server chunks as they need to acquire all the items they seek. Because we cannot take for granted that the server will group hot items together into chunks (the server may not know how, if it must define its chunks in advance of request statistics), we consider the possibilities in which hot items are grouped into chunks and in which hot items are uniformly scattered across chunks. They are called "hot items packed" and "hot items scattered," respectively, in the figure.

We can see the resulting impact on client delay in our figure. As the chunk size grows, and encompass more items, we see that client delay grows rapidly. In this resulting plot, the client delay becomes approximately inversely proportional to the number of chunks remaining.

With chunks consisting of more data items, not only are clients forced to request more data than they need, increasing their delay because of the extra time they consume receiving the (more) data they requested, but also because other clients are doing the same, the system suffers the effect of higher load until their requests overlap more and the rising load tapers off. In the extreme case, only one chunk remains, consisting of all data items, and so the server simply cycles through all its data and client delay approaches $|\mathcal{D}|t_{\mathcal{D}}$.

In summary, our results show that it is critical to match the clients information needs (items) to what the server provides (chunks). If chunks are too large, performance degrades quickly, and if chunks are smaller than items, the management overhead is high. For WebBase, we expect clients to be interested in full sites (to index them or mine them), so we clearly need to use a site as the server chunk.

2.6 Related Work

As we shall see in this section, work related to our multicast facility ranges from caching in the network (as an alternative to multicast for efficient data dissemination) to "broadcast disks" (which our scheduling work extends). Our data scheduling work in particular may evoke references to scheduling in other contexts, but our multicast data scheduling differs in a key way: the shared benefit of scheduled multicast dissemination.

As noted in Section 1.3, one way to use intermediaries to help disseminate data (besides multicast) is to have them strategically placed to act as network caches. Work to this end appears in Reference [7], where the authors argue that Web page access follows a Zipfian distribution; by contrast, in this chapter, we approximate the distribution using a simpler model of hot and cold data items for simplicity.

Reference [54], which considers multicast distribution in concert with caching, focuses on low-latency online Web browsing, unlike our work, which allows for noticeable client delay to reduce network consumption.

In contrast to caching work is the work related to "broadcast disks," where the server is the primary source of data for its clients. When conceived as a server distributing data without input from client requests [1, 24], the broadcast disk server must know or guess in advance the access patterns of its clients, so that it can correctly schedule more popular data for broadcast more frequently. To this end, Reference [60] determines scheduling algorithms for broadcast disks that minimize average "access time" (average response time) for a given data-access probability distribution.

Reference [4] develops a scheduling heuristic (RxW, described briefly in Section 2.1) to minimize a similar "average wait" measure by using specific client-request information, as opposed to overall access distributions. (Notice it is difficult to use RxW, and our R/Q as well, without specific request information. We, too, assume that specific client-request information is available to our multicast facility.) The paper further describes how RxW performs nearly as well as a longest-wait-first heuristic (omitted from this text, represented instead by RxW), and how RxW can have its results and performance approximated when run-time for the scheduler is limited.

Still, in the related scheduling work cited in this section, a lingering assumption remains: clients request a single piece of data at a time. This assumption is true in much scheduling work for broadcast delivery in general (for example, Reference [57]). For example, this assumption remains even when the authors consider less conventional measures of client waiting (e.g., Reference [2], which defines "stretch" as client response time divided by the size of the data item the client is requesting; and Reference [64], which supposes client requests to have deadlines that may or may not be met). We remove this single-data-item-request assumption, and as a result our scheduling work extends existing scheduling work in broadcast delivery.

Outside the context of data dissemination, our work on scheduling data for multicast may remind readers of process scheduling in operating systems [58, 55], and of job scheduling in operations research [25, 32]. Neither process nor job scheduling, however, have the key property of our multicast facility: data being scheduled for multicast can benefit multiple clients simultaneously. A CPU assigned to a process, for example, benefits only that process, and the general job-scheduling problem [25] does not allow one machine doing one operation to benefit multiple jobs requiring it.

2.7 Conclusion

In this chapter we have modeled a multicast facility, which unlike existing schemes, allows clients to request multiple items from the repository at a time, and does not deem them satisfied until all their requests are next multicast. The new data-scheduling heuristic we introduced, R/Q, is able to substantially outperform existing heuristics for this metric. We have also studied various multicast issues, such as granule size and batching delay. Next, in Chapter 3, we shall extend our study to include clients that subscribe to changing data in the server's repository, along with the one-time download clients considered in this chapter.

Chapter 3

Subscribers and Downloaders

As we recall from Chapter 2, the data scheduler is a key component of a multicast data facility; its behavior determines the performance of the system to its clients. In this second chapter on data scheduling, we extend our scheduling work to tackle the challenges of incremental updates, wherein the server's items may mutate over time, and clients may wish to track such changing items. Considering the Stanford WebBase for example, clients may wish to keep their requested Web images up-to-date (matching the live World Wide Web) by maintaining a subscription to a WebBase multicast facility that periodically or continuously crawls the Web.

This extension to our multicast facility leads to several questions we need to address in the design of our multicast system. As before, we focus on the design of such a facility for the Stanford WebBase project, as proxy for other multicast data-dissemination applications such as those listed in Chapter 1. In particular, we consider the following questions in this chapter:

- How should we measure the performance of the multicast facility under a subscription model? We have a number of possible metrics: We might measure the "lag" between when a change happens on the live Web and when clients are next synchronized with the live Web. Or, we could measure the lag between a change and when the change is next broadcast. We can also use metrics described in [16], freshness and age, which measure how up-to-date a client's image is at each point in time. We will see how the choice of metric affects the design decisions we make for the facility.

- How should the multicast facility disseminate the changes that it receives? We could try to send clients all the changes as the server receives them. This is applicable, however, only under very constrained conditions; if clients have limited resources with which to receive or process updates, for example, they would be unable to keep up with server updates. Further, the updates from a fast Web crawler would have poor locality and organization, because a fast crawler would have to interleave Web pages from the many Web sites it is crawling simultaneously to maintain high throughput.

  It would be preferable for the multicast server to buffer arriving changes, then disseminate them at the clients' speed as consolidated data items. In the simplest case, we have the server maintain a ring of data items that are updated with incoming changes, and cyclically disseminate the updated data items in ring order. This is similar in concept to disseminating changes as they arrive, but is more efficient and can scale with client speed. Of course, the server can order the dissemination of updated data items differently, by using a different scheduler. This leads to the natural question:

- How should the multicast server schedule requests? A multicast facility will typically receive requests from different clients for more data than it can send at one time. How should the multicast facility choose from its requests what data to send first, so that its performance is optimized?

While there is existing work in multicast systems, such as that of broadcast disks and publish/subscribe systems, our multicast scenario in this chapter diverges from prior work significantly. Unlike broadcast disks, and unlike Chapter 2, our data is updated over time, so our metrics—and our design—must take the freshness of clients' data into account. Unlike publish/subscribe systems, we cannot assume that every relevant update on the Web can be disseminated to clients in real-time, or even at all, because of the relatively large size of Web data, and its rapid and widespread changes online. Instead, our multicast system must choose between many competing updates wisely. Also, unlike both of these systems, our scenario can accommodate a mix of not only subscribers keeping up with changing data, but also downloaders seeking a single copy of their requests in minimum time, introducing new complexity to our system. We survey related work more fully in Section 3.6.

To answer the design questions above, we need to explore different performance metrics and multicast server schedulers. We will then model our multicast facility and the Web data it will disseminate, and use a simulation of our model to study the tradeoffs in our many server design decisions.

3.1 Model and Metrics

The multicast facility's performance to an individual client can be measured over a number of metrics, computed over the data the client receives and strives to keep up-to-date. Let us define our model for the multicast facility, and define metrics formally for an individual client, before finally defining them for multiple clients.

3.1.1 Design Model

In this chapter, we assume a client c requests to track, or subscribes to, up to $|\mathcal{D}|$ data items from the server's set of available data items $\mathcal{D}$. Let us call the set of items the client tracks $\mathcal{R}_c$. (Intuitively, these data items are Web pages for our WebBase facility.) This means the client is connected to the multicast facility, and strives to keep up-to-date copies of its $|\mathcal{R}_c|$ data items. If a client does not already have a copy of a particular data item, the client effectively requests a copy of the data item for the first time. To the multicast facility, a client not having a data item is similar to the client having an out-of-date data item; in both cases the up-to-date item needs to be sent to the client.

The client, by staying subscribed, indicates its interest in receiving any updated versions of its requested data items as they become available. When the client is no longer interested in receiving updates, it notifies the multicast facility of its withdrawal and disconnects. Alternatively, the multicast facility can, through occasional pings of its clients, discover that a client has disconnected or crashed, and remove that client's request from consideration.

We call W_{dt} the version number for the data item $d \in \mathcal{D}$ on the live Web at time t. The version numbers start at 1 and increment monotonically with each change of the data item. As a consequence, for example, we can say $W_{dt} > W_{dt'}$ only if $t > t'$. It is not meaningful to compare version numbers of different data items.

An individual client, in turn, may have its own copy of the data items it is tracking (by virtue of tracking them). We denote with C_{dt} the version number for the data item d stored at the client at time t. If a client has at time t the same version of data item d as appeared on the Web at time t', then $C_{dt} = W_{dt'}$. If a client does not have any version of data item d at time t then we define $C_{dt} = 0$.

When the multicast facility disseminates a data item d, it disseminates the highest-numbered version of item d in the repository at the time. Clients then have the same version of the item the repository does. Now, in practice the repository may not have the same version of a data item as appears on the live Web, leading to a dilemma. Clients, in principle, care about the live Web and not any particular repository, and so we can measure the performance of the multicast facility by seeing how fresh client data is compared to the live Web. On the other hand, how the repository gets data from the live Web is not controlled by the multicast facility, so it does not make sense to include that component in our evaluation. Thus, to isolate our study to the multicast facility, we define the repository to have the "latest" version of each data item at all times, and call these data items the *source data*.

For simplicity, let us say that time is discrete, having units of equal size representing the average time the multicast server takes to transmit one data item. Of course, the actual time to transmit a data item may vary from item to item, but approximating the actual time to send each data item by a single average value will allow us to model the performance of the multicast facility on average, over time.

Similarly, we assume that the multicast network is reliable, so that the delivery of data items is guaranteed. For example, in a controlled network where losses are small, dropped data can be hidden using error-correcting or erasure codes uniformly during transmission. In a next chapter, we consider in detail how to best provide reliable delivery for a multicast facility, over an unreliable (best-effort) datagram multicast network such as the IP multicast backbone.

3.1.2 Metrics for One Client

A client can measure the performance of the multicast facility (to it) in a number of ways, such as the ones we enumerate now. We will extend these metrics in the next section to measure the performance of the multicast facility for all of its clients.

For the following metrics, each of which is defined for a single client, let us define an auxiliary function $\delta(i)$ as $\delta(0) = 1$ and $\delta(i) = 0$ for $i \neq 0$. Then, we can define for this client, over discrete time from 1 to $t_{\max}$:

time-averaged freshness the fraction of the client's items that are up-to-date, averaged over time:

$$F = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} \frac{1}{|\mathcal{R}_c|} \sum_{d \in \mathcal{R}_c} \delta(W_{dt} - C_{dt})$$

Intuitively, we simply give a client data item one "point" if it is up-to-date at each unit of time and no points if it is not, then sum up all the points and average them over time to determine freshness.

For example, if a client is tracking four data items over time 1 to $t_{\max} = 10$, starts with all of them up-to-date, and has one of them fall out of date halfway in time, the client's time-averaged freshness is 7/8, or 87.5% (half the time it was 100% fresh, the other half it was 75% fresh).

freshness the fraction of the client's items that are up-to-date at the end of an interval of time:

$$F_{t_{\max}} = \frac{1}{|\mathcal{R}_c|} \sum_{d \in \mathcal{R}_c} \delta(W_{dt_{\max}} - C_{dt_{\max}})$$

We can use this notion of freshness to "sample" the freshness of a client after its transient initial behavior. For example, in the time-averaged freshness example above, the freshness at time $t_{\max}$ is 75%.

time-averaged age the time since last modification of each of the client's items, averaged over all items over time:

$$A = \frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} \frac{1}{|\mathcal{R}_c|} \sum_{d \in \mathcal{R}_c} (t - \max\{u \leq t | W_{du} = C_{du}\})$$

Unlike freshness, we assign zero age for up-to-date client data items, and the individual data item's age if it is out-of-date. Then we sum the ages and average them for a net score for the client. The age of a client data item is the amount of time since it was last up-to-date.

For example, if again a client is tracking four data items from time 1 to $t_{\max} = 10$, starts with all of them up-to-date, and has one of them fall out of date at time $t = 5$, the client's age is zero during the first half of time, and increases slowly during the latter half. More precisely, the out-of-date item incurs one unit of age at the end of time unit six, two units age at time seven, three at time eight, four at time nine, and five at time ten. This yields an instantaneous client age of $\frac{1}{4}$ at time six, $\frac{2}{4}$ at time seven, and so on to $\frac{5}{4}$ at time ten. Then averaged over time, we say the client's time-averaged age during the period is 0.375 units of time.

These definitions of time-averaged freshness and time-averaged age are designed to be equivalent to the notions of freshness and age in Reference [16].

age the time since last modification of each of the client's items, averaged over all items at the end of an interval of time:

$$A_{t_{\max}} = \frac{1}{|\mathcal{R}_c|} \sum_{d \in \mathcal{R}_c} (t_{\max} - \max\{u \leq t_{\max} | W_{du} = C_{du}\})$$

We define age at the end of time $t_{\max}$ as we did for freshness. In the previous example, for instance, the age at time $t_{\max}$ is simply $\frac{5}{4}$.

Notice that the max expressions in the definitions of age and time-averaged age above require that the client must have started at time 1 with some copy of the data it is tracking, however old. If at some time the client does not have any copy of a data item, then that data item's age at that time, time-averaged or not, is (necessarily) undefined.

We find in practice that it is difficult to evaluate time-averaged freshness, because the effects of initial conditions are averaged into any final results. As a result, we will use a form of the freshness metric in which we specify all subscriber clients to begin at time 1 with all data items fresh. This "freshness-loss" metric tells us, in effect, how good the multicast system is at preserving freshness over time, and allows us to differentiate different multicast systems more clearly in less time. Because we observe in experiments that a metric that performs well under freshness-loss is also good at freshness with other initial conditions, we can use freshness-loss as our form of freshness. As a bonus, this initial condition also ensures that the age of each client's data items is always well-defined.

One could also define other client metrics, but we do not think they are as appropriate for data subscriptions as the ones we have defined so far. For example, we could define a measure of client-image lag in scenarios where the source data does not change often, such as software updates: We would measure the time between each change in the source data and the next time the client becomes fully up-to-date. Alternatively, we could measure a data-item lag for a client, measuring the time between a change in the source data and the next time the client becomes up-to-date on that item. Neither is particularly appropriate for our Web multicast scenario, however, because Web data changes relatively rapidly. A client may never have a fully up-to-date image of its requests, and a data item may change repeatedly before the multicast facility is able to send an update to a client.

3.1.3 Extending Metrics to All Clients

To generalize our one-client metrics to metrics for clients of a multicast facility as a whole, we can compute the metric for each client, then combine the results in two intuitive ways. We assume that, as before, we are considering each metric over a fixed discrete time from 1 to $t_{\max}$. For simplicity, let us say further that there are a fixed number $|\mathcal{C}|$ of clients, $\mathcal{C}$, that are subscribed during the entire time. (In practice, we consider a client's connection or disconnection an instantaneous event, so we can split up the time during which a multicast facility runs into blocks of time during which the number of clients subscribed is constant.) Each client c requests $|\mathcal{R}_c|$ data items, and has score M_c for a one-client metric. We can then define an aggregate metric M in one of two ways:

over clients One way to define the global version of each metric is simply to average (or sum) each one-client metric M_i over all clients:

$$M = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} M_c$$

Defining the global versions of these metrics this way encourages schedulers to favor clients with smaller requests, because the effect of transmitting one data item is bigger on clients making small requests. For example, if we have one client requesting one data item and one client requesting two other items, then sending the first client's request will give it a 100% freshness improvement (a 50% freshness improvement overall), while sending one of the second client's requests will improve one client's freshness only 50% (25% freshness improvement overall). To optimize freshness averaged this way, the multicast facility should clearly favor the smaller request of the first client. We will call this global version averaged *over clients*, e.g. "freshness over clients."

over data There is an alternative corresponding global version of each metric, defined as the *weighted* average (or sum) of the one-client metric over all clients, using the number of items in each client's request as its weight:

$$M = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} M_i |\mathcal{R}_c|$$

(Recall we are assuming each item has equal size; we actually want to weight by the total amount of data in each client's request.) This adds weight to popular data items; that is, a popular data item falling out-of-date has a larger effect on the global metrics than an unpopular item falling out-of-date. For example, suppose two clients request ten items all in common, and a third client requests only one eleventh item. Sending a more popular item increases overall freshness weighted by data by 2/21, while sending the eleventh item increases overall freshness weighted by data by only 1/21. So, a multicast facility optimizing freshness this way should favor any item requested by the first two clients. We will call this global version *over data items* or *over data*, e.g. "freshness over data."

One could extend the above "egalitarian" metrics to favor some clients or some data items as somehow more important, or valuable, than others, but we do not consider those metrics here.

We can 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 such data changes, rather than on any particular server's design.

The choice between freshness and age, however, is more difficult. On one hand, freshness is a metric with a strong intuitive appeal: for clients that subscribe to maintain live indexes to Web data, for example, what matters is the degree to which the index is in-sync with live Web data. A search over such an index is unhelpful if its results are accurate only for the version of Web data just past, just changed, and no longer available. To the contrary, the index is helpful to the degree that its results apply to the version of Web data that is still available for retrieval or reference.

On the other hand, choosing freshness for our performance metric has a unique problem of sometimes encouraging schedulers to abandon some data items completely. This is a problem inherent in the metric, not in any particular scheduler for it; that is, a scheduler may need to abandon data to correctly optimize for the freshness metric.

Example. Here is an example showing why unpopular items should be abandoned to maximize freshness. (Cho [16] demonstrates why too-frequently changing items should be abandoned, so we will omit such an example here.)

Let one client A request a lot of data items; for simplicity, we will only consider two, $\{a, b\}$. Let client B maintain only the data item $\{a\}$. Let both data items change after each slot of time.

In this scenario, the multicast server should always send the changed a, for both freshness-over-clients and freshness-over-data. When the server sends a, two of three data item requests are made up-to-date for that time slot, yielding freshness-over-data of 2/3 and freshness-over-clients of $(\frac{1}{2}+1)/2 = \frac{3}{4}$ for that slot. In contrast, should the server erroneously choose to send b instead, then only one of three data item requests are made up-to-date for that time slot, yielding freshness-over-data of 1/3 and freshness-over-clients of $(\frac{1}{2}+0)/2 = \frac{1}{4}$ for that slot.

In effect, any time the server sends b, its freshness measures are punished and never recovers. So, the server should always send a, and so in this scenario, b is never disseminated. $\square$

Age does not suffer from this problem, but it is insensitive to freshness. A low-age client image may have very few fresh pages at all, and instead merely numerous pages all a few updates obsolete relative to the current hypertext. This is appropriate where old data is valuable in itself, for example, stock tickers whose delayed prices still convey some potentially usable information about a stock's current price.

Clearly, depending on the scenario, a system designer will want to choose the most appropriate metric to optimize and use that as a guideline in the building of a multicast facility. Consequently, we will determine guidelines for all four metrics: freshness over clients and over data, and age over clients and over data. As our particular WebBase clients seem interested in building indexes and conducting research over our data, however, we will focus on freshness as our primary metric to help ensure that our clients' data remain relatively in-sync with our repository for reference.

3.2 Scheduling

Now, we turn to considerations on the server side of the multicast facility. As before, we will consider a single multicast server out of many; in practice, each multicast server offering subscription service must make the same decisions, as its particular circumstances demand. In particular, how should the multicast facility schedule which data item to disseminate next?

We present five possible schedulers in this section. The first serves as our baseline; the next three are adapted from prior work in broadcast and multicast scheduling; and the fifth is designed to optimize freshness over clients in our scenario. Among the schedulers adapted from prior work, we mention the broadcast disk scheduler RxW only for completeness, because it is not a good fit for our large-scale environment.

In prior work on broadcast and multicast scheduling, schedulers were designed for a simpler facility in which clients request a subset of a multicast server's static data items, and then connect to a shared multicast channel until they receive a copy of their data from the multicast server on that channel. The simpler facility is similar to the multicast facility we describe here, except that the data never changes in the simpler facility, so clients only stay connected long enough to get a single copy of the static data they need. We call such clients "downloaders" here, in contrast to the "subscriber" clients who stay connected to keep their data up-to-date.

We adapt heuristics for the simpler multicast scenario to our multicast scenario with subscriber clients as follows: When a multicast server's data item is known to have changed, the server pretends that all the clients subscribed to that data item request the new version of that data item, at the time the server discovers the change. We describe several such heuristics (popularity, R/Q, and RxW) below.

Circular Broadcast We first consider a heuristic that simply transmits each requested data item in rotation. More precisely, we arbitrarily order the data items, and have a pointer to one of them. At every time slot, we move the pointer to the next data item until we reach a data item that some client needs (because the repository's version has changed and a client needs to bring it up-to-date). If we reach the end of the data items, we move the pointer to the first data item and continue; if we consider every data item and find that no clients need any of them, we send nothing. If we do reach a data item that some client needs, we send that data item. This simple heuristic ensures that every requested data item will get broadcast, regardless of its popularity or other characteristics.

This simple heuristic, like the heuristics to follow, also guarantees several advantages over unicast distribution of updates and unscheduled multicast updates. Circular broadcast necessarily subsumes unicast distribution, because for popular data items, a unicast server must divide its outgoing network into many slower streams to disseminate a separate copy of each data item for each client. For example, if two researchers track the Stanford database group's publications on the Web, then a unicast server must send a separate copy of the publication server's contents to each researcher, with each transmission using at most half the server's outgoing network link. A multicast server, on the other hand, can send the contents once to both researchers, using the entire outgoing link and completing the transmission in half the time, granting more time to transmit other requested data.

Further, a scheduled multicast server, even using a simple circular broadcast, provides more flexibility than unscheduled dissemination of a Web crawler's updates. First, a scheduled multicast server is not required to lock its outgoing multicast stream to the same throughput as the Web crawler's throughput, as the direct distribution of a crawler's fetches would require. Second, a scheduled multicast server can decide to disseminate an updated data item other than one being fetched at the moment, as its clients request. Third, a scheduled server aggregates changes to a data item, in effect buffering the updated data before sending it to clients. Clients receive fewer, larger bulk-transfer data items rather than many tiny (page-by-page) updates that they must filter if they received the crawler's download stream directly.

Popularity One simple heuristic for multicast scheduling is to have the server, at each time slot, send a data item that the largest number of clients are requesting at the time. This heuristic is also described in Section 2.2.

R/Q In the R/Q 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 server chooses a data item with maximal R_i/Q_i ratio to send. The heuristic is described more fully in Section 2.2 where it first appears.

In practice, computing each R_i/Q_i ratio for every data item at every time slot proved computationally intensive, so in our study we do not implement R/Q this way. In our study, we maintain a heap of R_i/Q_i values and update those values only when it needs to be changed (for example, because a new client requests the item, or because the client with the smallest pending request that includes this item has just had its pending request changed by new data changes or updates). This makes R/Q fast enough to use, because it needs only remove the top of the heap to make its decision.

In our work on the simpler downloaders-only multicast facility (Chapter 2), R/Q proved an effective way to minimize the average delay of downloader clients.

RxW The *RxW* heuristic is designed for the scenario in which every client requests a single data item. The name comes from the score it assigns to each item i, the product of R_i, the number of clients requesting the item, and W_i, the longest amount of time any client has been listening (waiting) for the item. The heuristic chooses to send a data item with the highest such score. It seeks to favor more popular items and avoid the starvation of any clients. It is described more fully in Section 2.2.

This heuristic requires the computation of many R_iW_i products, because the product changes for every requested data item at every slot of time. This made the heuristic very computationally and memory-access intensive given the scale of our study, much more so than the other heuristics described here. The published way [4] to most substantially increase the speed of RxW is to approximate the heuristic (computing a few, but not all, of the R_iW_i products to find a possible, not definite, maximum), but doing so changes the heuristic and affects its performance. Because of the time it takes to run this heuristic, we did not include it in our study.

RxC We can devise a new heuristic for our freshness-driven subscription scenario: If we want a scheduler that maximizes the freshness metric, then intuitively we should first disseminate the data items that will remain fresh *longest*, and for the largest number of clients.

The reasoning comes from examining each choice of data item as a local decision, with a fixed cost (one time slot) and variable benefit. If a data item we transmit remains fresh for the time we transmit it and one time slot afterwards, for example, then we have gained a "point" of freshness for this data item, times two slots of time that it is fresh, times the number of clients that are maintaining this item. If our freshness is computed over data, then each point contributes directly (and linearly) to the metric. If our freshness is computed over clients, then the connection between points and freshness is less direct but no less important: Every point we gain leads to a higher measure of freshness.

In real life, however, we cannot predict when a data item will next change, and so we are not able to implement such a heuristic directly. Instead, we can try to use the past as indicative of the future: We can compute how often a data item changes, and use that value as an estimate of how long a current data item will remain fresh.

In real life, it is also possible for the change rate of a Web page to vary. For example, a discussion page about a current topic may change rapidly as readers contribute their commentary, but then become static as interest dies off. So, as a matter of implementation we use an exponentially-decaying floating average instead of a strict arithmetic mean computed over all recorded changes. The floating average follows any change in data-item-change frequency.

This leaves us with the heuristic shown in Figure 3.1, which we call RxC ("C" stands for the (estimated) time of next change). For each data item, this heuristic tracks R, the number of clients requesting the data item, and estimates C, the time between updates to the data item.

In the figure, the heuristic is described with three functions. **Initialize** runs first, to initialize arrays whose values are maintained for this heuristic, including C, the estimated time of next change for each data item. **UpdateC** maintains those arrays so that the server has an estimate for C to use when it needs to choose a data item to send. **RxC** actually determines a data item to send by computing the R times C product for each data item.

In Figure 3.1's **Initialize** function, we initialize the array of C_i, which holds the server's estimate for how long the data item i will remain unchanged. Before a data item changes for the first time, the server cannot guess how long the data item will remain unchanged, so we declare it to be infinity. (If the data item never changes, this guess is correct. If the data item changes frequently, this guess will be quickly corrected by the first change in the data item.)

Once a data item changes, we can create an estimate: the first time the data item changes, the time until that change is recorded as the estimate. For subsequent changes to the data item, we take the weighted average of the current estimate and the time interval until the latest change, recording the average as the new estimate. This occurs in Figure 3.1's **UpdateC** function. The time interval until the latest change is computed and stored in $ThisChangeTime$, and used as necessary to update our estimate C_i. The average is weighted with a fixed (predetermined) weight w for the old estimate and $1 - w$ for the new data point. A larger w would yield an estimate less vulnerable to normal variations in the intervals between changes to a data item. A smaller w allows the estimate to more rapidly track changes in the change rate of the data item.

The performance of a RxC heuristic may depend on how the value chosen for w affects the accuracy of the heuristic's estimates for C. To see how RxC is affected by its choice of w, we will also study a hypothetical variant of RxC in which an oracle provides, for each data item i, the ideal (correct) estimate of the average time between data item i's changes, C_i.

Finally, **RxC** actually computes the R_i times C_i product for each data item i, to send a data item with a maximum product. There is a special case to consider in this process: If any C_i are infinity, then item i's corresponding RxC product would be undefined (infinity), making further comparison to other values (in particular, other infinities) impossible. The server therefore considers data items that have never changed specially: For data items that have never changed, a more popular data item is defined to have a "higher" RxC product than a less popular one.

As for R/Q, computing RxC for every data item at every time slot was prohibitively time-consuming, so RxC is also slightly optimized for our study as R/Q was. In our study, we maintain two heaps, one of finite RxC products, and one of the popularity (R value) of the still-unchanged data items. We update values in the heaps only as necessary, moving items from the unchanged-data-item heap to the

Initialize. Sets up floating average.
  $w \leftarrow 2/3$;
  foreach data item i
    $C_i \leftarrow \infty$;
    $LastChangeTime_i \leftarrow 0$;

UpdateC. Updates floating average.
Input: The datestamps of the
  repository's data items.
Output: None, but C and
  $LastChangeTime$ are modified.
  if Initialize has never been called
    Initialize;
  foreach data item i
    if repository's data item i is
        newer than $LastChangeTime_i$
      $ThisChangeTime \leftarrow$
          (time i was last changed) -
          $LastChangeTime_i$;
      if $C_i = \infty$
        $C_i \leftarrow ThisChangeTime$;
      else
        $C_i \leftarrow C_i w +$
            $ThisChangeTime(1 - w)$;
      $LastChangeTime_i \leftarrow$
          time i was last changed;

RxC. Determines what data item to send.
Input: A set of clients and the data items
  they are tracking.
Output: The data item to transmit.
  UpdateC;
  foreach data item i
    $R_i \leftarrow 0$;
    foreach client c
      if c's copy of i is older than
        the repository's
        $R_i \leftarrow R_i + 1$;
    $RxC_i \leftarrow R_i C_i$;
  if any C_i are ∞
    Choose a data item with the
    largest R_i for which C_i is ∞.
  else if all RxC_i are zero
    Send nothing (no items are requested).
  else
    Choose a data item with the
    largest RxC_i.

Figure 3.1: RxC Heuristic

finite-RxC heap as appropriate. This makes RxC fast enough to run, because RxC now needs only examine, at most, the top of two heaps to make its decision.

3.3 Simulating the Heuristics

To compare the heuristics in our Web multicast model, we use a simulation to determine the client data freshness each heuristic would achieve under a variety of conditions. In this section we present the parameters of the simulation, estimate

Description	Web value	Simulation value
Number of simulated Web pages	1 300 000 000	1 285 200
Number of chunks of Web data	86 666 667	85680
Number of simulated Web sites	21 000 000	21000
Large Web sites	840 000	840
Pages in large Web site	990	990
Medium Web sites	3 360 000	3360
Pages in medium Web site	62	60
Small Web sites	16 800 000	16800
Pages in small Web site	15	15
Average Web page change interval	75 days	75 days
Time to transmit each chunk of pages	0.3 seconds	5 minutes
Web sites requested per client (mean)	9000	9

Table 3.1: Simulation Parameters and their Base Values

their values for the Web, then scale those values for our simulation as our base case. In Sections 3.4 and 3.5, we present our results.

In Table 3.1, we enumerate some estimates for the real Web at the turn of the century (2001) in the column "Web value," to form our simulation's *base case*. In this base case, we try to portray a multicast server disseminating data for all but the very most frequently updated Web pages online. (A minority of pages change at least every day, and as such are not practical to crawl constantly, much less disseminate.)

For the number of Web pages and Web sites, for example, we look for possible numbers online [27, 31]. We use these numbers to divide pages into Web sites, creating three sizes of Web sites to capture the effect of the widely different sizes for Web sites in the real world. Similarly, we estimate the time between changes to a Web page, the network resources available to a hypothetical WebBase multicast server, and speculate on an average number of Web sites that a client might request; derivations for all of these numbers in Table 3.1 are detailed in Appendix B. These numbers form a starting point, from which we can create base parameter values to vary for our simulation.

3.3.1 About Chunks

We would like our multicast facility to schedule Web pages individually for dissemination, because individual pages, not entire sites, change at a time. (For example, a front page may be updated with recent news, but background-information pages may remain unchanged.) By scheduling as fine-grained a unit as possible for transmission, we reduce the waste of scheduling and transmitting updated versions of data that has not actually changed.

The numbers we have gathered, however, are all (not surprisingly) large. As a result, a multicast facility on current equipment may be unable to run efficiently when scheduling the multicast of each Web page individually. This is because of two factors in unison: One, it often takes more time to schedule more data objects. Two, the more data objects the server schedules, the smaller each data object is, and the less time it takes to transmit, so a multicast server has less time to schedule each of the data objects.

If a multicast server cannot efficiently schedule each Web page individually, it can gather the pages into "chunks" of u pages for scheduling. Because the larger the chunk, the less precisely we can schedule changed pages for transmission, in practice we would like to choose the smallest number of pages per chunk that still allows the server to run efficiently. For our estimates of the Web, $u = 15$ is a convenient number because all our Web sites are approximately integer multiples of our Web chunks.

We assume that a chunk of pages all belong to the same site, and that clients request and receive Web data in chunks. The server considers a chunk changed if any of its pages has changed, and so schedules entire chunks for transmission instead of individual pages.

For the schedulers that need a count of how many clients need a particular data item, the server tracks how many clients need each page in a particular chunk, and maintains its sum over all the pages in the chunk. The schedulers then use this sum as a count of how many clients need a particular chunk of Web pages. This sum allows us to weight the popularity of each chunk by how much of the chunk is in demand, while still being very easy and fast to maintain accurately.

This approach of grouping pages into chunks allows the server to perform scheduling on fewer, larger units, but at a cost in multicast performance. In practice, one may wish to create as small a chunk as system performance allows.

3.3.2 About the Simulation

First, to form chunks neatly, we round off our numbers, so that small sites are 1 chunk (15 pages), medium sites are 4 chunks (60 pages, not 62), and large sites are 66 chunks (990 pages).

With the actual Web numbers, a detailed simulation takes too much time and memory to run. So, we next scale our numbers for the Web down by a thousand, seeking to preserve the proportions between them, to make our simulation run more efficiently. In effect, we are taking a multicast facility, and splitting it up into one thousand little multicast servers, each with a distinct random slice of the Web (a different subset of the chunks in our Web repository), all sharing the network facilities a big multicast facility would have used. Clients, conceptually, make their requests of a thousand little distinct servers, so that each server disseminates its part of the client requests to all the clients.

This scale-down leaves 21000 sites of 1 285 200 pages in total to be served by a multicast server, not 21 million sites and 1.3 billion pages. This also means that the average client request size to a server goes from 8000–9000 to a simpler 9 sites.

To preserve the time scale from distortion, we rescale the network resources available to the multicast system by the same factor. As a consequence, the Web still takes as long to disseminate as before, and a chunk now takes five minutes to transmit.

As a result of our rescaling of the simulation, the performance of the multicast facility is essentially unchanged to the clients. To see this intuitively, consider a "proxy" that implements the full server by splitting client requests to the thousand little servers behind it, and combines their data streams to the clients. The proxy's measures of client data freshness would be the average freshness over the thousand little servers' subsets of data, matching the average performance of a little server that we simulate. The proxy's measures of client delay would be slightly penalized because

	Fewer than 400 clients	More than 400 clients
< 15 Web sites per client	Circ or Pop (1) (2)	R/Q (1)
15-75 Web sites per client	Pop	R/Q
> 75 Web sites per client	R/Q	R/Q

Table 3.2: Optimizing for Subscribers' Freshness Over Clients

the proxy's net client delay is the maximum client delay of all the subsets of data, but otherwise the comparative results should look similar.

Table 3.1 summarizes all these simulation parameters in the "Simulation value" column.

3.4 Scheduling Subscribers

To determine how to best schedule a multicast facility for subscribers, we run our simulation with various parameters to see how our various server scheduling heuristics compare. We then use this information to develop guidelines for the design and implementation of the multicast facility.

3.4.1 Optimizing Freshness Over Clients

To illustrate, let us first focus on optimizing our facility for the freshness over clients metric. This metric represents the fraction of a client's subscribed data that is expected to be fresh, on average. Based on our experiments, we present in Table 3.2 approximate guidelines for the choice of scheduler in a multicast facility optimizing freshness over clients. The table shows the recommended heuristic depending on the number of clients in the system (columns) and the clients' subscription size (rows), as determined by our experiments. In this and subsequent tables, "Circ" refers to the circular broadcast heuristic; "Pop" refers to the popularity heuristic, and "R/Q" refers to the heuristic of same name.

Note that the guidelines are approximate: in particular, the cross-over points between heuristics are not abrupt, so that if a design scenario falls near an edge between adjacent table cells, the choices in the cells are competitive.

As an example, if we expect a hundred clients tracking fifty Web sites at a time, and care most about freshness over clients, we should choose the popularity heuristic as our multicast facility scheduler.

We believe the cross-over points occur as shown in the table because of the growing overlap in client requests at the higher client loads. In particular, if among client-requested Web data, the average number of clients requesting a chunk grows significantly above 1.0, then more sophisticated heuristics such as R/Q can exploit the varying popularity between data chunks. (At exactly 1.0, by contrast, each client request is unique.) Empirically, we found that an average exceeding about 1.1 represents enough load for a heuristic such as R/Q to outperform circular broadcast.

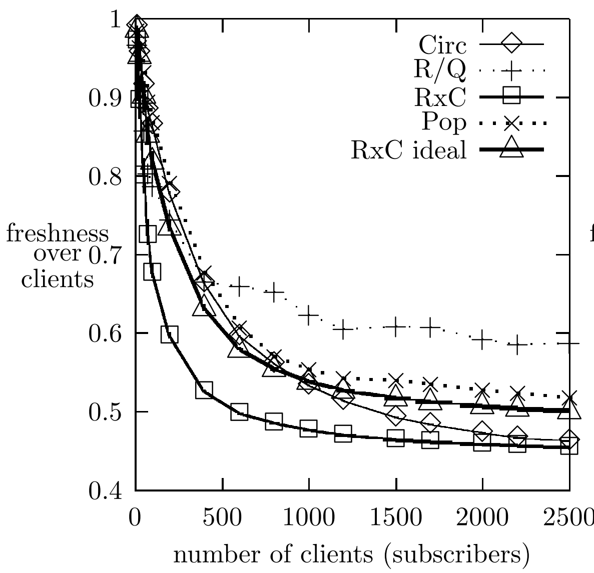

Figure 3.2: Freshness over clients for subscribers

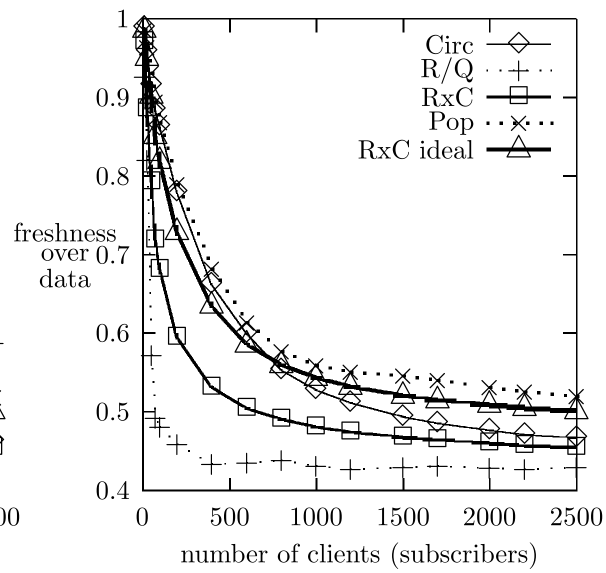

Figure 3.3: Freshness over data for subscribers

An Example Scenario

For lack of space, we are unable to present all the experiments whose results are summarized in the table. So, for illustration, we focus on the parts of the table marked (1), which are plotted in Figure 3.2. In Figure 3.2, we see the performance of our multicast facility using our various scheduling heuristics, for various numbers of subscriber clients. In this and subsequent figures, "Circ" refers to the circular broadcast heuristic; "Pop" refers to the popularity heuristic; "R/Q" refers to the heuristic of same name; and "RxC" refers to the RxC heuristic, with its weight factor arbitrarily chosen as $w = 2/3$. (The effect of w and the plot "RxC ideal" are explained later.) Recall from our simulation values that in our base case, a client requests on average 9 out of 85680 available (about 0.01% of available) Web sites.

Along the horizontal axis, we vary the number of subscribers in the system, our load. On the vertical axis, we plot the freshness the data clients have at the end of 75 days of simulated time (F_{75days}), averaged over clients. To prevent the freshness from being perturbed by transient events right before the end of our simulation, we compute a client's freshness at the end of the simulation as the average of the client's freshness over the last day of simulation time. Choosing the fairly long 75 days allows us to approximate the relative steady-state performance of the schedulers, and determine our most appropriate choice.

Clients start the simulation with ideal freshness (all fresh data), so this plot measures the freshness clients *lost* during their subscription. (Clearly, with more data on the Web changing than the multicast facility can disseminate, clients' freshness cannot remain perfect over the course of their subscription.) Starting with fresh data allows us to differentiate the performance of the schedulers more clearly, as mentioned in Section 3.1.

For example, if our multicast facility has a thousand subscribers, a client starting with fully fresh data would end the simulation with an expected 48% of its data fresh if the server used an RxC scheduler, and 53% if the server used a circular broadcast scheduler. If the server used the popularity scheduler instead, such a client would have an expected 55% of its data fresh. Lastly, if the server used R/Q, a client would have an expected 62% of its data fresh at the end of 75 days. (Because of the rapidly changing nature of the Web, freshness rates are necessarily low: For comparison, an independent crawler would not only apply new load to Web servers online, it could yield freshness as low as 12% compared to the Web if it crawls intuitively but suboptimally, and no higher than 62% relative to the Web if it crawls optimally [16].)

As the number of clients increases to the right in the figure, we see that the freshness over clients performance of all the heuristics degrades; this is to be expected, as we are increasing load on the system as we move right in the figure. As we add more clients, we can expect performance to degrade further, because the server will be unable to keep up with all the requests.

From this figure, we can draw several conclusions. One is that, for freshness over clients, R/Q is a good heuristic to use when the multicast facility is providing data for over 400 clients. (This portion of the figure corresponds to the upper-right cell of Table 3.2.) This is because R/Q uses as a weighing factor the amount of data (effectively, the number of Web pages) a client needs to return to ideal freshness. It turns out this property of R/Q allows R/Q to favor smaller clients and boost their freshness, as we now see.

First, one can see how R/Q's Q factor—the amount of data a client still needs to return to full freshness—can favor clients with smaller requests: The R/Q score (and therefore, the priority for transmission) of a data item is higher if a client that needs the item needs very little other data to return to full freshness. For a client to need very little other data, the client must either have very few requests to track, or be unaffected by changes in the Web (which is equivalent, since our clients select Web data to track independently of the data's change rate).

Next, one can see how favoring small clients helps boost freshness over clients scores. Because for clients tracking few Web pages, the refresh of each Web page represents a relatively high fraction of their freshness, refreshing a page requested by such clients improves the freshness over clients score more than refreshing a page requested by clients tracking very many Web pages. So, when there are a large number of clients, creating a large spectrum in the number of pages clients track, R/Q can perform well by finding the smaller-request clients to benefit.

Observations for Freshness Over Clients

After running many more simulations like the above, varying a number of other parameters, we determine the following guidelines to optimize for freshness over clients, as summarized Table 3.2:

- For our base parameter values and well under 400 clients, circular broadcast or popularity suffice as well-performing schedulers. This is the part of the Table 3.2 marked (2), and shown in Figure 3.2. At 100 clients, for example, circular broadcast and popularity are near 86.5% freshness over clients with circular broadcast doing slightly better, while the next-best performer R/Q is at 78.7% freshness.

- As detailed in the text and figures above, if the multicast facility is providing data for a large number of clients (starting at over 400), R/Q becomes a good choice of heuristic. That is, the results favoring R/Q in the upper-right cell of Table 3.2, as shown in Figure 3.2, extends for the entire the right column of Table 3.2.

- If clients request, on average, larger numbers of Web sites (greater than fifteen, versus the nine-site default), popularity is a good choice. For smaller request averages, circular broadcast does a little better (at base parameter values, for example, circular broadcast edges out popularity by just under one percent freshness), and for very large numbers of sites (above 75 per client) R/Q does just as well, but popularity is competitive at both ends of the spectrum and is the best performer for the values in-between (15-75). At an average of forty Web sites per client, for example, popularity holds about 85% freshness over 100 clients after 75 days, beating R/Q's second-best value of 81%. At an average of a hundred Web sites per client, R/Q begins to take a small lead, at 74% freshness over popularity's second-ranked 73%. Considering the relatively small difference by which popularity loses at such extreme values, it is not much loss to approximate the left column of Table 3.2 as "use popularity."

It appears that circular broadcast is effective in the very light loads of cell (2) (a scenario of under 400 clients and fifteen sites requested per client), where the server does better trying to keep up with everything, than trying to focus on favored data items and abandoning others. In loads higher than that of cell (2), the server cannot hope to keep up with everything, and circular broadcast deteriorates.

	$<$ 400 clients	$>$ 400 clients
$<$ 15 Web sites per client	Pop	Pop
$>$ 15 Web sites per client	Pop	Pop

Table 3.3: Optimizing for Subscribers' Freshness Over Data

We also found that R/Q is a good performer under very high loads (such as when clients request over 75 Web sites per client on average, exponentially distributed). Under higher load, R/Q is consistently able to pick out popular data items requested by small-request clients for dissemination.

The popularity heuristic, then, is a compromise that is effective where R/Q may be unable to make good choices because it is channelled to a few data items with a low Q score, but unable to benefit very many small-request clients by disseminating them, and circular broadcast is insufficient to keep up with the increasingly numerous changes.

3.4.2 Optimizing Freshness Over Data

Let us now turn briefly to freshness over data. Our results for this metric are summarized in Table 3.3. As we can see, the popularity heuristic is a good choice for this heuristic. This is because, as we observed while describing and evaluating the metrics, a freshness metric averaged over data favors popular items, just as the popularity heuristic does. Consequently, the popularity heuristic is a good match for this metric, as we have observed in simulations over a variety of conditions.

Also, we observe that R/Q's favor toward small-request clients is completely unhelpful if the metric to optimize is freshness over data. In fact, if we turn to Figure 3.3, we see that this is exactly the case. In this figure, the vertical axis is now freshness (F_{75days}) over data, so that clients with large requests get more weight in the average. The simulated scenario is the same as in Figure 3.2. This means, for example, that for a system with a thousand clients using the R/Q heuristic, just under 43% of all client data is fresh at the end of 75 simulated days. More notably, R/Q is the worst performer for freshness over data, because of the differences in the two freshness metrics. This underscores the importance of choosing the appropriate metric for each application before making design decisions around it.

3.4.3 General Observations

Here are some more general observations, which are not part of the prior tables:

- The average interval between changes to a Web page has little effect on our design of the scheduler. Whether Web pages change on average once every ten or hundred days, the relative performance of the scheduling heuristics does not change significantly. In particular, the order of preference for the heuristics does not change dramatically. This may be because the heuristics and metrics often use client requests as factors in their computation, and so the component of load that most affects their results comes from the number and size of client requests. By contrast, when the average interval between changes to a Web page is reduced so far that heuristics change drastically in performance, it is because the changes have become so frequent that the multicast system simply begins to collapse, and all the heuristics perform comparably poorly.

- The skew of client requests has little effect on the design of the scheduler. Even when a large majority of client Web site requests are focused on as small a fraction as 5% of all Web sites (instead of the baseline uniform likelihood of request), the relative performance of the scheduling heuristics does not change dramatically. In the experiments we ran to evaluate the effect of client-request skew, each client chooses among large sites, medium sites, and small sites uniformly, as in our base case simulation. Within each group of sites, however, 80% of a client's requests are of a fixed small fraction of the sites in that group. For example, 80% of large site requests fall into 5% of large sites, 80% of medium site requests fall into 5% of medium sites, and 80% of small site requests fall into 5% of small sites. This creates a skew of client requests into 5% of all Web sites, which become "hot."

- Surprisingly, though RxC was designed to optimize for freshness, it does not do so particularly well. In none of the experiments we ran was RxC the preferred heuristic. To determine whether our choice of the heuristic's weight parameter w was hurting its performance, we create a variant of RxC in which an oracle provides the actual average rate at which each Web page chunk changes. (This was implemented by running a simulation, computing the actual average rate at which each Web page chunk changed, then feeding those averages back into a new run of the simulation using the same random seed.) We call this variant heuristic "RxC ideal" in Figures 3.2 and 3.3. As we can see in the figures, the oracle-assisted RxC performs better than RxC, suggesting that there is improvement to be gained in tuning w, but the improved result is still not enough to make an ideal RxC the preferred scheduler for this subscriber scenario.

  One possible reason RxC fails to optimize freshness is that RxC does not take into account the amount of time that has passed since a Web page's last change. RxC strives to compute the expected freshness-over-clients gain in disseminating each data item, but even if the RxC score is computed perfectly accurately, it assumes that the entire time interval until the next change remains for each data item. Clearly, this is not true most of the time; as a server postpones the dissemination of a data item, it should acknowledge that the freshness gain over time of sending the data item falls. RxC does not do this, because doing so would require every RxC score for every data item to be updated at every unit of time, slowing down the multicast scheduler dramatically compared to the other, more efficient to compute heuristics considered here.

  Another reason may come from grouping pages into chunks; RxC schedules chunks, whose change rate is affected primarily by its fastest changing pages. Thus, a chunk may appear to change rapidly, but actually have little change over time because only one page of the chunk is changing frequently, and RxC would misjudge its value.

	< 1000 clients	> 1000 clients
< 20 Web sites per client	Circ (4)	R/Q / Pop (3) (4)
> 20 Web sites per client	Pop	Pop

Table 3.4: Optimizing for Subscribers' Age

3.4.4 Optimizing Age Metrics

For the age metrics (age over clients and age over data), R/Q becomes a poor performer. From our simulations, we conclude the guidelines shown in Table 3.4 instead.

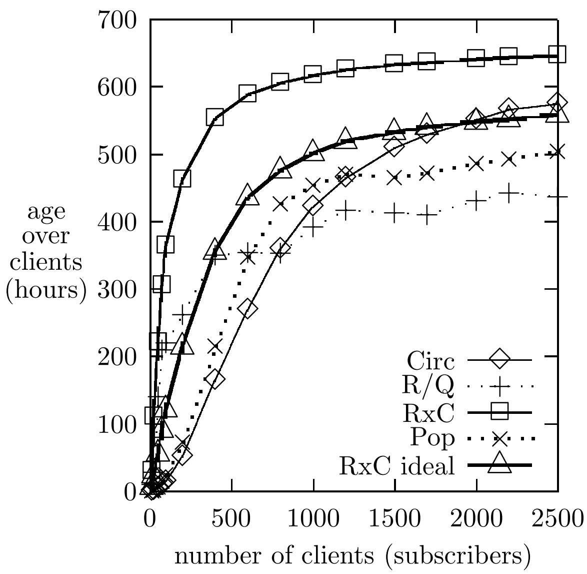

Figure 3.4: Age over clients for subscribers

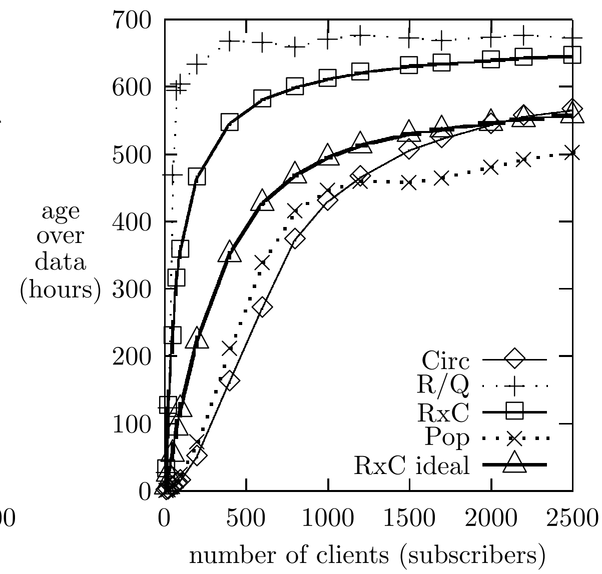

Figure 3.5: Age over data for subscribers

For the age metrics, popularity is a good all-around performer, with the following exception: For age over clients, R/Q is the best performer when the multicast facility is providing data for a very large number (a thousand or more) subscriber clients in our base case scenario. The exception suggests that even in the age metrics, which penalize neglected data with growing age, a strategy of favoring clients with smaller requests is preferable when there is a large variation in client request sizes and a large number of small-request clients to service, as is the case when when a large number of clients have a small average request size.

This means that the cell marked (3) in Table 3.4 represents R/Q if the desired metric is age over clients, and Pop if the metric is age over data. We see this in Figure 3.4, which plots the age over clients for varying numbers of clients, and Figure 3.5, which plots the age over data for varying numbers of clients. These plots span the cells marked (4) in Table 3.4.

Because the age metrics accumulate higher age for neglected data items, the circular broadcast approach, which does not neglect any data, is effective for a larger variety of conditions than it was under the freshness metrics. Still, it is hard to characterize in general the regions where each scheme does best under the age metrics, due to the complex interactions of factors such as update rate and distribution, number of items, number of clients, and size of subscriptions. Consequently, a designer whose scenario diverges radically from the ones we study here may need to empirically determine which cell in Table 3.4 best corresponds to the new situation.

3.5 Scheduling Downloaders and Subscribers

It is likely that in our multicast facility not every client will be a subscriber, able or wanting to stay connected to the multicast facility for updates. As we considered in Chapter 2, clients to a multicast facility may be *downloaders*, clients that connect to a multicast facility, request a subset of the facility's repository, then disconnect as soon as it has received all of its requests (at least once). Here, we would like to consider how the two types of clients—downloaders and subscribers—would interact in a multicast facility, and how we should adjust our scheduler to balance their different interests.

For this section, we define as in Chapter 2 the metric *client delay* or *delay* for downloader clients: the delay of a downloader client is the amount of time between the client's connection to the multicast facility (neglecting the time for the client to issue its request) and the time the client first has a copy (any copy) of *all* the data it requested. This definition is useful for a system like WebBase, in which our clients will typically need to have all the data available before doing batch indexing, mining, or analysis of the data. By definition, a downloader has no data when it first connects to the multicast facility.

3.5.1 Simulating Downloaders and Subscribers

It is tempting to measure steady-state behavior of the complete system to determine its performance. In such a simulation, there would be a fixed number of subscribers, and downloader clients appearing at exponentially-distributed random intervals. The simulation would be run for some time to take the average client delay of downloaders as they leave, and measure the freshness of the fixed subscribers.

Unfortunately, this steady-state simulation turned out to be an inadequate way to assess the performance of the facility. For example, if a scheduler was bad at satisfying downloader clients, a large number of downloader clients would still be in the system (their requests not yet completed) at the end of the simulation, but the actual number of clients would vary not only from scheduler to scheduler, but also depend heavily on transient downloader-client behavior near the very end of the simulation. Worse, a large number of clients at the end of a simulation could either be "normal" for a scheduler, or it could warn of a scheduler that is not sufficiently getting downloader clients out of the system at all, invalidating the computed average client delay for that scheduler. In a steady-state scenario, it would be difficult to draw a line between the two cases without manual review of each simulation. Also, new and deferred downloaders waiting in the multicast system add load to the system over time, making direct performance comparisons for subscribers more difficult.

Rather than to simulate a particular scenario, we want our model to provide a clear way to measure the impact of new downloaders in our multicast facility, and to distinguish between our schedulers in a scenario with downloaders and subscribers. To measure the performance of the multicast facility relative to the client delay and freshness metrics simultaneously, therefore, we arrange to have a number of downloader clients start at the beginning of the simulation in addition to a fixed number of subscribers. This simulates the multicast facility just as downloaders are introduced into the system. Subscribers begin the simulation with ideal freshness, and downloaders with no data. While it may be possible to implement a multicast facility this way—force downloaders to wait until specific times when they enter the multicast facility all at once—we do not choose this model to require this design.

To measure the performance of the system relative to the downloaders, we measure the average client delay of the downloaders in the simulation. To measure the performance relative to the subscribers, we compute the average freshness of the subscribers after a brief, fixed amount of time, regardless of the downloaders' delay. The expectation is that new downloaders add load to the system and distract the server from updating its subscriber clients, so we are measuring the freshness hit subscribers take from the introduction of the downloaders. Choosing a fixed point in time near the starting time, rather than choosing the time all downloaders complete, allows us to measure the penalty subscribers suffer without being biased by the downloader performance of the system. We considered choosing a round one (simulated) day as our fixed time near the starting time, but that proved too short to clearly differentiate the subscriber performance of the different schedulers. So, we choose one week as the time at which we measure freshness.

3.5.2 Results

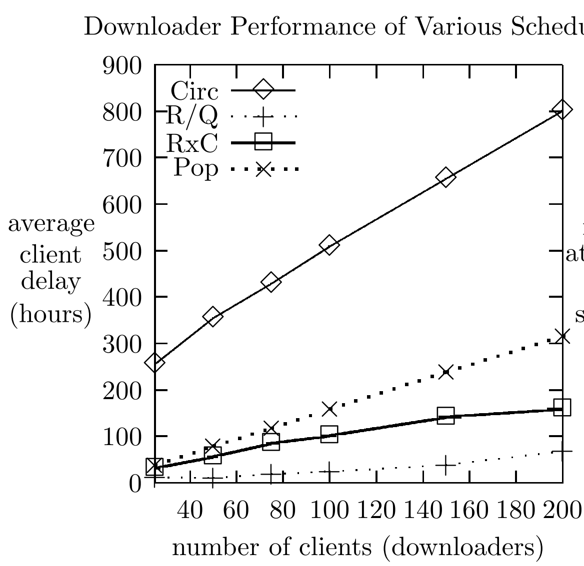

Figure 3.6: Client delay for downloaders in a mixed downloader/subscriber system

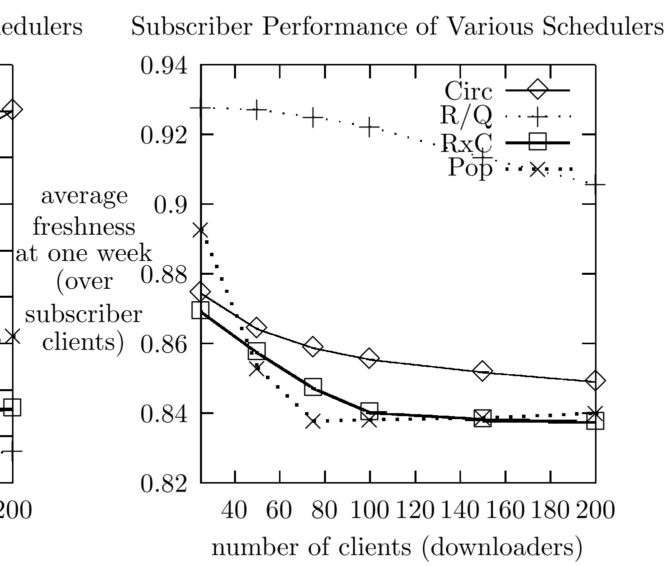

Figure 3.7: Client freshness for subscribers in a mixed downloader/subscriber system

In Figures 3.6 and 3.7, we see the performance of the multicast facility charted for

	< 30 downloader clients	> 30 downloaders
< 15 Web sites per client	R/Q	R/Q
> 15 Web sites per client	R/Q	R/Q

Table 3.5: Optimizing for Client Delay and Metrics Over Clients When Downloaders Are Present

	< 30 downloader clients	> 30 downloaders
< 15 Web sites per client	Pop	Circ
> 15 Web sites per client	Pop	Pop

Table 3.6: Optimizing for Freshness Metrics Over Data When Downloaders Are Present

downloaders and subscribers, respectively, given one hundred subscribers and, along the horizontal axis, a varying number of downloaders at simulation start. On the vertical axis in Figure 3.6 is the downloaders' delay, averaged over clients. On the vertical axis in Figure 3.7 is the corresponding subscribers' freshness at one simulated week.

For example, if we introduce a hundred downloaders to the hundred subscribers already in the system, we find that the downloaders' requests are satisfied with an average client delay of 1.05 days under R/Q, a longer 4.2 days under RxC, 6.6 days under popularity, and a much more painful 21.2 days' wait under circular broadcast. Meanwhile, at the end of one week, the subscribers have an average freshness of 92.2% for R/Q, 85.5% for circular broadcast, and about 84% for popularity and RxC. The R/Q heuristic, in this case, achieves the lowest client delay for downloaders as well as the highest maintained freshness for subscribers.

Perhaps the most striking observation here and in other simulation runs not plotted here is that R/Q performs well for downloader client delay as well as both freshness and age metrics over subscriber clients, as suggested by the numbers above. The results for these three metrics, as summarized from a number of simulations, appear in Table 3.5.

This suggests that when downloaders and subscribers are pooled into the same multicast channel, it is to all clients' advantage to satisfy and clear out the downloader clients as quickly as possible, so that their requests do not linger and confound the service of subscribers when updates to Web data occur.

Again, we run a number of simulations like the above, manipulating various parameters, and from them gather the following guidelines for a multicast facility with both downloader and subscriber clients. We find these conclusions:

- R/Q is a good scheduler for minimal downloader delay, maximal freshness over clients, and even minimal age over clients for varying numbers of downloader clients and average Web site requests per client. This suggests that to optimize these metrics, a server should get downloaders out of the system as quickly as possible to return service to subscribers.

- Popularity or circular broadcast usually optimize freshness or age over data, as shown in Table 3.6. It is difficult to characterize in general the regions where each heuristic does best, due to the complex interactions of factors from update rate and distribution to client request sizes to the proportion between subscriber and downloader clients. Consequently, a designer who needs to optimize for metrics over data, under a mixed subscriber and downloader scenario that diverges radically from the ones we study here, may need to empirically determine which cell in the above table best corresponds to the new situation.

We also found the following observations about this multicast scenario:

- Like the subscribers-only scenario, the average interval between changes to a Web page has little effect on the design of the scheduler. Whether Web pages change on average once every ten or a hundred days, the relative performance of the scheduling heuristics does not change dramatically.

- Circular broadcast consistently incurs very high downloader delay. This is probably because circular broadcast must cycle through most of the union of all requests from all clients before a downloader client's request is satisfied, and this union is usually much larger than the request of any downloader client. In short, circular broadcast makes no consideration—no attempt to speed up—downloaders at all.

- RxC, however, is even worse for downloader delay, in that many of the introduced downloaders did not finish by the end of a fairly long (75 simulated days) run. These downloaders do not have client delay values, and so they were not included in the plotted average. RxC appears inappropriate for a scenario of downloaders and subscribers on a shared multicast facility.

3.6 Related Work

Remarks and references from Section 2.6 also apply here, as we continue data-scheduling work in this chapter: Unlike network caches, we expect clients to be relatively tolerant of delay to receive data at higher network savings.

In broadcast disk scheduling work, clients typically request a single piece of data at a time, and request it only once (rather than requiring updates). We remove the single-data-item-request assumption in Chapter 2, and in this chapter considered subscriber clients. As a result our work extends existing scheduling work in broadcast delivery.

Also, subscriptions are studied in publish/subscribe projects such as SIFT [66], where a server provides subsets of Usenet to subscriber clients who specify their interests. In SIFT, though, the focus is on efficiently filtering data for clients' interests, on the assumption that the system can keep up with and broadcast the amount of information being requested and made available. The assumption remains in multicast work for other publish/subscribe systems including Gryphon, such as [46], which studies the use of IP multicast for dissemination of publish events (updates). It, too, avoids our scheduling dilemma by assuming much higher network resources at its disposal so that all updates can be given to all interested clients. In practice, we find that a large multicast facility, such as one for the Stanford WebBase project, cannot keep up with all relevant updates to its source data (e.g., the Web) as they happen, so the multicast facility must make quick choices to maintain the best performance that it can.

Finally, Reference [16] considers the problem of keeping Web data up-to-date, but does so for only a single client (a crawler) and assuming no knowledge of when updates actually occur in the source data. As such, this work could help keep the source Web repository for WebBase up-to-date with respect to a live Web.

3.7 Conclusion

In this chapter we have modeled a multicast facility which, unlike existing schemes, allows clients to request multiple items from the repository at a time, and either receive all the data they requested or subscribe to updates of the data to maintain its freshness. We consider a number of metrics by which to measure the performance of the facility, and determine from numerous simulations the recommendations for best performance under each metric and under a variety of conditions. With these results to guide the design of the data scheduler for a multicast facility, we now turn to network issues in Chapters 4 and 5, to determine how to deliver the multicast server's scheduled data reliably and efficiently.

Chapter 4

Reliable Multicast Distribution

For our data multicast facility to be useful, clients must be able to receive all their requested data; the facility must successfully deliver all the requested bits to its clients because missing data may render the remaining partial data unusable. Our multicast facility's underlying multicast network, on the other hand, may not guarantee reliable delivery; to the contrary, IP multicast is only a best-effort datagram service and prone to packet loss. One report [65], 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 the scenarios described in Chapter 1, 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 chapter, the first of two chapters focusing on design issues arising from the underlying network, 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 (described in Section 4.1) open up a number of ways to perform reliable transfer that were not available to generic reliable multicast, which we will study here. Chapter 5 will focus on the challenges that result when clients of different network-throughput speeds share the same multicast server simultaneously.

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, effectively punting the problem to the data scheduler. 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 multiple data items at a time, and we 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 chapter, 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 client requests have no fixed latency deadline, 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 4.1 a possible network traffic layout that isolates the network-consuming data transmission itself for further study and optimization. In Section 4.1.1, we propose a simple client delay metric to gauge the performance of our design decisions. In Section 4.2, we describe traditional approaches such as retransmission and forward error correction, and introduce new possibilities that are possible for our multicast facility. In Section 4.3, we describe the simulation we use to evaluate our design options. In Sections 4.4 and 4.5, we examine results from our study to form suggestions for the design of a reliable-multicast facility over an unreliable multicast network.

4.1 The Multicast Facility

For this chapter, we consider the multicast server to have a number of data items (intuitively, static files) ready for dissemination, from which each client will request some 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.

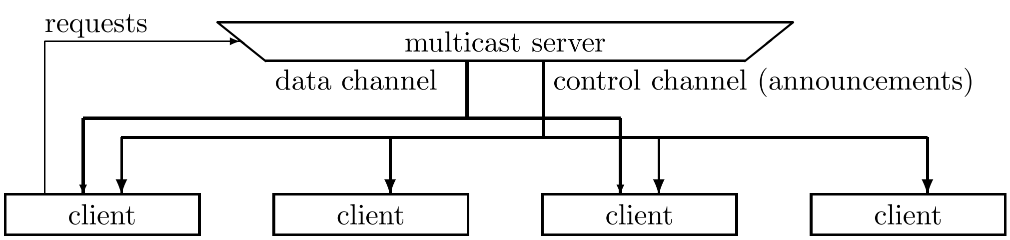

Figure 4.1: Information Flow in the Multicast Facility

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 4.1.

All clients always subscribe to the control channel, because they must receive all the announcements on the control channel promptly. Clients use the announcements to determine when to subscribe to the data channel, so that they receive the data items they requested 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 [23].

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 to broadcast disks, in which a server sends small single-item requests on a local broadcast network to satisfy local clients' requests. 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 4.2.

- 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 until 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 request-to-completion time as our performance metric.

4.1.1 Metric

In a number of scenarios, such as those mentioned in Chapter 1—researchers analyzing subsets of a Web repository, and software updates being saved and applied—or others, such as 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 metric 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.

Therefore, we will focus on client delay as our performance metric. We reintroduce client delay below. It is intuitively similar to the notion of client delay defined in Chapter 2, slightly simplified for this chapter.

For a multicast server with a finite set of data items $\mathcal{D}$, and a finite set of clients $\mathcal{C}$, each client $c \in \mathcal{C}$ is characterized by the data items it requests, $\mathcal{R}_c \subseteq \mathcal{D}, \mathcal{R}_c \neq \emptyset$, and by the time at which the client makes its request, t_c.

We define the *client delay* of client c using the earliest time $t_{c\mathcal{R}_c} > t_c$ when the client has all the data items it requests $\mathcal{R}_c$. We simply say that for this client, its delay is $D_c = t_{c\mathcal{R}_c} - t_c$.

For all clients $\mathcal{C}$, we define the *average client delay* over all clients as $D = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} D_c$.

We might also measure the network usage of the multicast facility in this chapter, and use it as a metric; unfortunately, this is not particularly enlightening because, as we observed in Section 2.3.2, 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.

4.1.2 Multicast Operation

The server operates by gathering the clients' request subsets and using a scheduler, such as R/Q (Chapter 2), 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 4.1.1, 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, a, b, and c; B needs items b and d, and C needs items b and c.

A multicast server implementing R/Q would determine each of the item's R/Q scores, which begin as $\frac{1}{3}$, $\frac{3}{2}$, 1, and $\frac{1}{2}$ for items a, b, c, and d respectively. In absence of other clients, then, the server would send b first, then item c, then the remaining two items in arbitrary order (after items b and c are sent, both items a and d 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 a before d. 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, preferable over the other possible values of $D = \frac{10}{3}$, $\frac{11}{3}$, or 4. □

4.2 Making the Data Channel 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).

4.2.1 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 p data packets, the server would then send an additional $\lceil fp \rceil$ error-correcting packets.

- We can have the server increase 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 p data packets, the server would then send a total of $\lceil (1 + f)p \rceil$ 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 $\lceil (1 + \frac{f}{R})p \rceil$ 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 d, 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.

4.2.2 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.

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.

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.

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 [44].

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 [23], 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 retransmissions 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` [23]). 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.

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 $\mathrm{R}(p)$.

4.2.3 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.

4.3 Simulation

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 4.1: 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 [65]. 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 4.1, 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 4.1, 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 $50 \cdot 1024 / 8 / 512 = 12.5$ times per second. This means one packet is sent every 80 milliseconds.

The NAK delay-time range in Table 4.1 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.

4.4 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.

4.4.1 Retransmission and Rescheduling

We first consider what happens in the simplest case, in which the multicast facility uses only retransmission (Section 4.2.2). In Figure 4.2, 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 4.1. 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 4.2.2.

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 the missing 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 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 retransmission scheme as a backup for forward error correction. For simplicity, we will choose retransmit-always below.

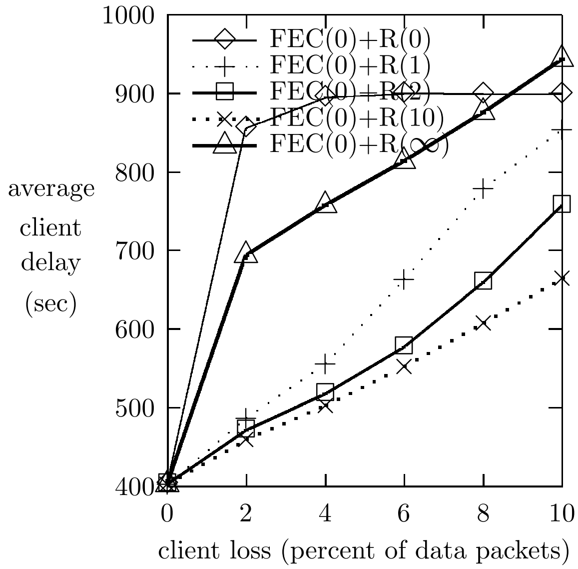

Figure 4.2: Retransmission Schemes

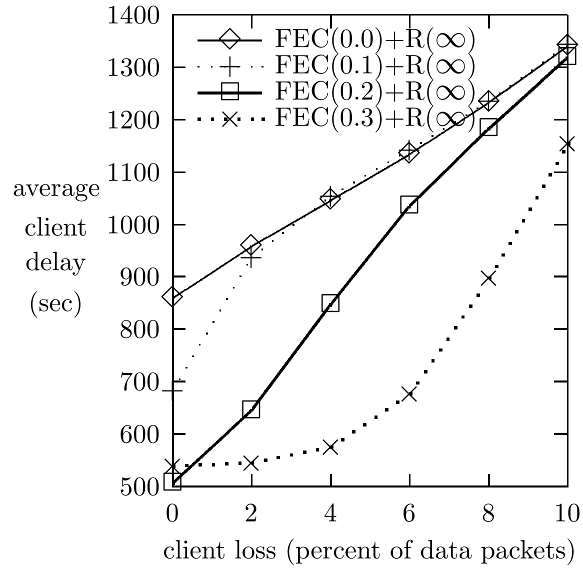

Figure 4.3: Constant Forward Error Correction Schemes

4.4.2 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(∞).

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 4.3. 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. 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 dramatically outperforms FEC with 20% expansion 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.

4.4.3 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. As a result, the client would 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 the stop provides 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 4.4 and 4.5, 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 4.4, the server link loses no packets, a low-loss scenario (as we had used when we considered retransmission earlier). In Figure 4.5, 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 4.5, 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.

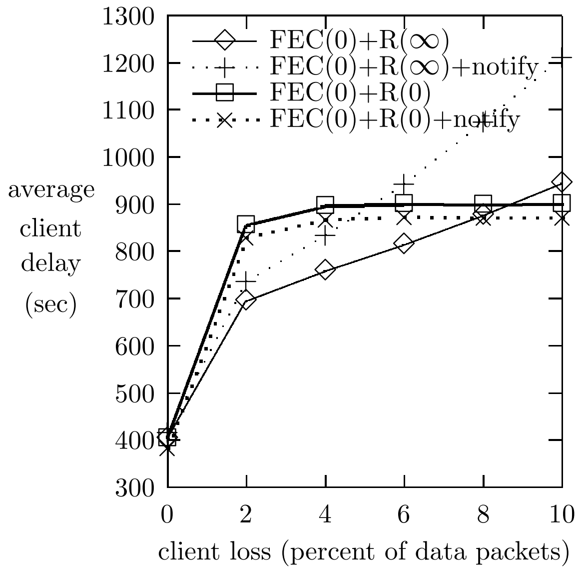

Figure 4.4: Retransmission Schemes

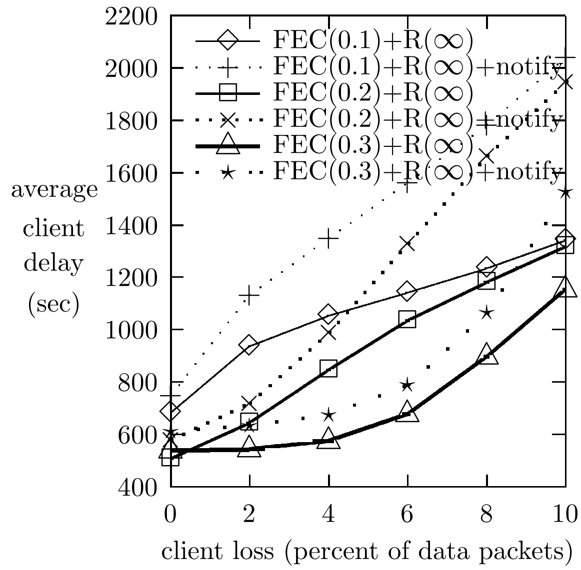

Figure 4.5: Constant Forward Error Correction Schemes

We see, then, that for retransmission-backed schemes, including those using forward error correction (labelled R(∞) in Figures 4.4 and 4.5), 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 4.5 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 4.4), 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%).

4.5 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.

4.5.1 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 4.9, where it is compared to an equivalent scenario without the use of error-correcting packets.

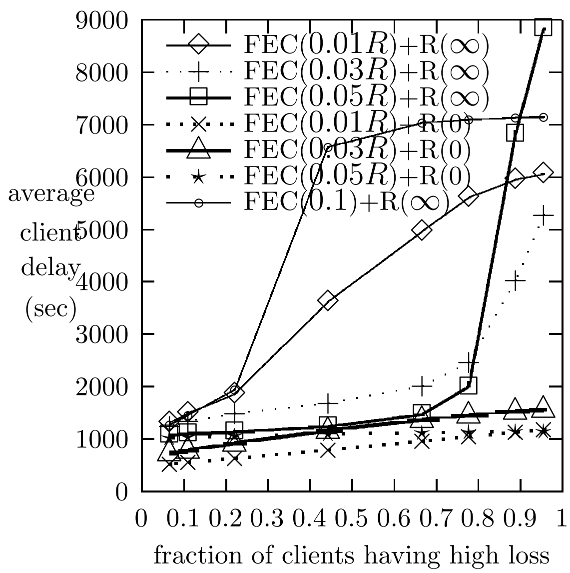

Figure 4.6: Variable Forward Error Correction

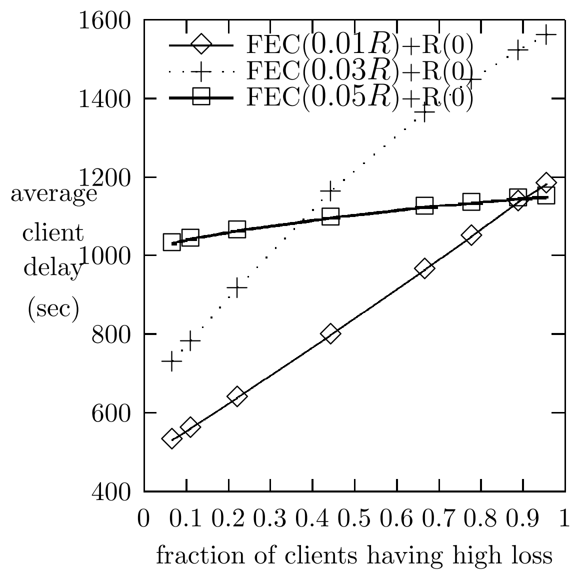

Figure 4.7: Variable Forward Error Correction

4.5.2 Forward Error Correction

In Figures 4.6 and 4.7, a zoomed-in view of Figure 4.6, 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 "fR" indicate a forward error correction scheme whose expansion factor (as defined in Section 4.2.1) 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 4.6. (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 4.7, 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 it 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.

4.5.3 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.

FEC with Client Knowledge

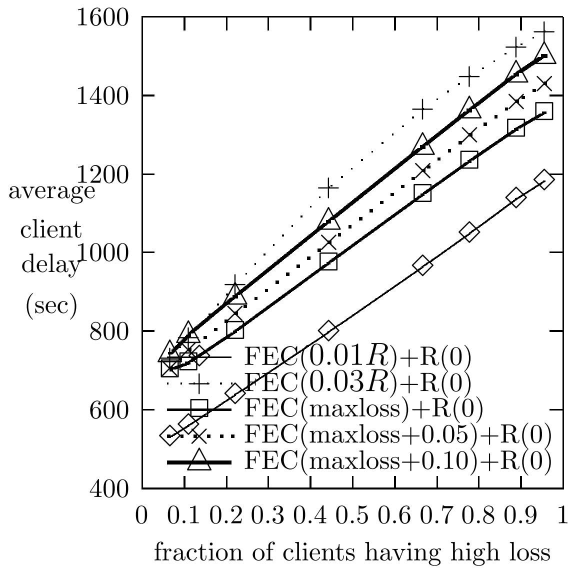

Figure 4.8: Perfectly Loss-Aware Forward Error Correction

In Figure 4.8, 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, for example, "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 packets, 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 deteriorate in a perfect-knowledge system. As a result, this use of client-loss information does not improve the performance of the multicast facility.

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 4.10. 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.

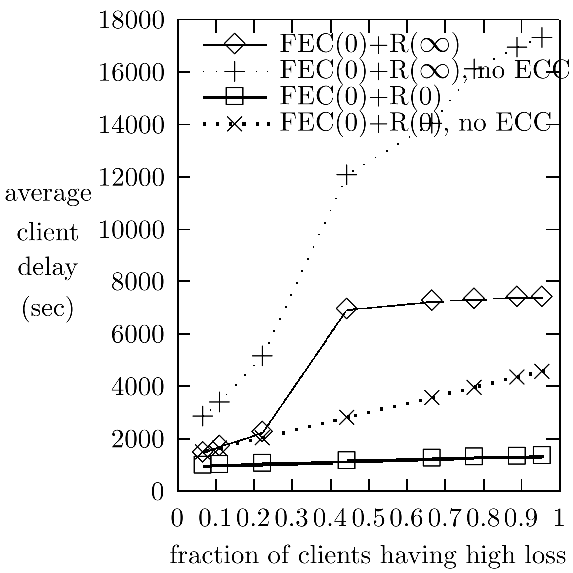

Figure 4.9: Exact Versus Error-Correcting Packets

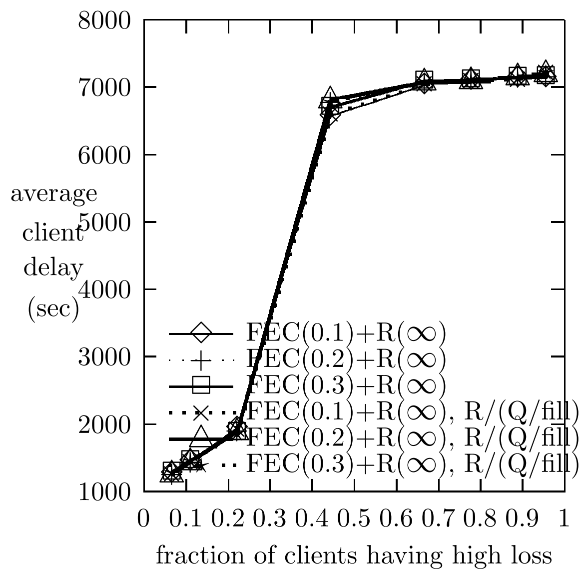

Figure 4.10: Applying a Loss-Sensitive Scheduler

4.5.4 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 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 4.9. 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.

4.6 Related Work

General-purpose reliable multicast protocols have been widely studied, with numerous variations proposed. A networking text [59] details the design of IP multicast; a multicast text [43] surveys a number of such protocols, such as [23] and [44]. In this chapter, we extended 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 chapter is designed to complement reliable multicast data dissemination, such as in broadcast disks [4] as well as our own multicast facility. 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, Reference [60] 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 systems (e.g., Gryphon [46]) and Web caching (e.g., [54]), but there, client latency is expected to remain very small, so reliability mechanisms requiring longer completion time (such as our option to reschedule a request) may be inappropriate for such systems.

4.7 Conclusion

In this chapter 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.

Chapter 5

Slicing Server Bandwidth

In this second chapter addressing networking challenges in a data multicast facility, we consider the problem of managing the available bandwidth (the server's available outgoing network throughput) when clients may have different capacities, each of which may be less than the total bandwidth available to the multicasting server. In particular, the server must be able to *split the available bandwidth* into multiple multicast channels, instead of using a single channel that all clients must receive. With multiple channels, channels may be configured for the bandwidth of particular sets of clients. It is especially important to have this flexibility in an Internet environment where nodes have widely different throughput links, as opposed to (say) the telephone network, where every node has the same bandwidth.

In addition to letting us cope with heterogeneous clients, splitting the available bandwidth into multiple channels also introduces new opportunities for improving performance. For instance, a server administrator may *split the data* to distribute across multiple servers, with each server providing its own subset of the data over a separate channel (or sub-channels in order to deal with heterogeneous clients interested in its partition of the data).

Splitting the server's network capacity into smaller channels, however, has potential disadvantages. For instance, clients that are distributed into multiple channels are less likely to have their requested data items coincide with those of other clients, simply because there are fewer clients on the same channels with which to share requests. As a result, each transmission reaches fewer clients on average, reducing the efficiency of the multicast. Also, smaller channels take longer to send each data item to its clients than one big channel, simply because smaller channels have less throughput.

In summary, managing the overall available bandwidth presents important tradeoffs that we believe have not been fully addressed in the literature. Thus, in this chapter we consider different ways to use the bandwidth to serve heterogeneous clients, and consider their impact on system performance. As we will see, the choices can significantly affect performance. In our experiments, we find that the delay clients experience retrieving data from a multicast server can vary up to five-fold depending on the bandwidth partitioning, and by three-fold or more depending on how clients are assigned to the channels that fit within the clients' download limits.

In the rest of this introduction we will illustrate in more detail the bandwidth and data partitioning schemes we will consider, and discuss the tradeoffs involved. As our running example, consider four clients requesting data as shown in Table 5.1. Of three data items $\{A, B, C\}$, the first three clients request $\{A, B\}$, $\{B, C\}$, and $\{A, C\}$ respectively. A fourth client requests all three data items. The server, we assume for this example, has as much network capacity as the fourth client, and the fourth client has three times the network capacity of the other three clients.

Table 5.1: Example Clients

Client	Requested Items	Network Capacity
1	A B	1
2	B C	1
3	A C	1
4	A B C	3

Single-Channel Multicast

In a single-channel multicast facility as shown in Fig. 5.1, the "base case," a single server with all the data items disseminates the data over a single multicast channel. Each client uses a reliable unicast channel (TCP) to send the server its requests, then

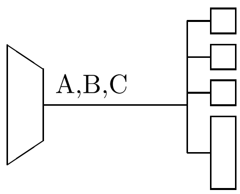

Figure 5.1: Single-Channel Multicast

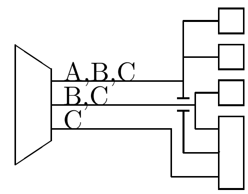

Figure 5.2: Multiple-Channel Multicast

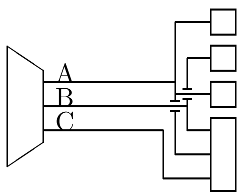

Figure 5.3: Split-Channel Multicast

waits for the data to appear on the multicast channel. The server gathers its requests from all its clients, chooses which data item to send next based on its demand, and sends it. The server chooses another item, and repeats until clients have received their requests and disconnect.

In the four-client example, all four clients would be forced to connect to the same channel, and therefore receive data at the same rate. For this to be viable, the channel transmits at a speed no higher than the capacity of the three slower clients. Because each data item receives the same pattern of client requests, the scheduler's transmission ordering is arbitrary; suppose it simply decides to send them in order A, B, C, as shown in Fig. 5.1. When the first client receives its two data items, A and B, it disconnects. After the third data item is sent, the three remaining clients disconnect, because they have also received all their requested items.

We see that this conserves network resources for the server, because putting all clients on the same channel ensures that any overlap in client requests is effectively exploited. We see also that the above example assumes that the multicast server chose its single channel to be a speed that the slower clients can follow. This choice clearly underserves high-throughput clients, such as client 4, which could receive data more quickly than the chosen throughput. In this example, this choice also underutilizes the multicast server's network, which has three times the throughput it is actually using for its one channel.

Multiple-Channel Multicast

In a multiple-channel multicast facility, as shown in Fig. 5.2, a server disseminates its data over multiple channels of the same speed. Clients could be directed to any channel(s) by the server as their (declared or measured) network capacities allow.

In our four-client example, the server has created three equivalent channels, each able to disseminate all data. Each client is randomly assigned to a channel, until the client's network capacity is filled. (Of course, in other designs, the server could have created a different number of channels, and assigned clients differently.)

For this example, we assume that the first two of the slower clients are assigned to one channel, the third to another, and the fourth client assigned to all three channels concurrently, as shown in Fig. 5.2.

The first channel, then, would send data items A, B, C; the second channel, having only the third and fourth clients, would send B, C. The third channel would send only C. As a result, the third client could disconnect from the system after two data-item transmissions (rather than three as in the single-channel version of this example), and the fourth client could disconnect after just one data-item transmission's time (rather than three, as in the single-channel example). The improvements in performance for the third and fourth clients are possible because they are connecting to new channels that did not exist in the single-channel version.

We see, then, that by tapping more of the server's network capacity in some way, we are able to improve multicast performance for the server's clients. On the other hand, we see that there is less data shared between clients; the third client was not able to use any of the traffic sent to the first two clients, because it did not have the network capacity to receive data from the additional channels as well as its own.

Split-Channel Multicast

In another version of the multiple-channel multicast facility, shown in Fig. 5.3, a server could split its *data* into disjoint equal-size pools, each of which it disseminates over a separate channel of the same speed. Clients would be directed to the channels by the server as their requests demand and (declared or measured) network capacities allow.

In our four-client example, the server could split the data so that each of three channels has a separate data item. (Usually, the server will have many more data items on each channel, but our example is small for demonstration.) In this case, the fourth client can get all three data items at once and leave in one item's transmission time. The other three clients, however, must choose to listen to one data item, then request the rescheduling of the other data items of interest over each item's respective channel. Fig. 5.3 shows how this example would appear during the transmission of the first item on the server's channels.

This approach helps exploit the overlap of client requests somewhat, but not always as well as one would expect, as we see in the example. Further, this approach requires each channel to be accessible to the slower clients, in case such a client requests the data on that channel. This limits the flexibility with which the server could split its available network capacity. For example, even the fastest clients must wait for the time it takes to transmit one data item for the slowest clients, even if it can otherwise receive its requests more quickly.

Outline

We will explore in this chapter some of the ways that a multicast facility can use its outgoing network capacity to serve heterogeneous clients, as well as its own load distribution, with multiple concurrent multicast channels.

We consider a number of pressing design questions, such as how many multicast channels a multicast server should run concurrently, how the transmission rate should be chosen for those channels, and how clients should be assigned to those channels.

5.1 The Multicast Facility

Recall that in our multicast facility, the multicast server has a fixed outgoing network throughput available for its use, and a number of data items (intuitively, files) ready for dissemination, from which each client will request some subset. A new client requests a subset of the server's data items using a reliable unicast connection to the server. Unlike the previous chapter, the multicast facility may split its bandwidth into multiple multicast data channels. Because each channel behaves similarly, we will first review the operation of a single data channel. Then we will consider how a multicast facility might decide to split its bandwidth into multiple channels.

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.

To minimize the consumption of clients' download links, we adopt a well-known technique as described in Chapter 4 and separate the server's traffic into a low-bandwidth *control channel* announcing data items, and a *data channel* (consuming the remaining allocated bandwidth) to carry the data items themselves.

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

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 as before we will use client delay as our performance metric.

5.1.1 Scheduler

The server conducts a data multicast channel by gathering the clients' request subsets that can be sent on the channel and using a scheduler to decide which requested data item to send on the channel.

There are a variety of schedulers that could be used for a multicast facility. For example, a circular-broadcast scheduler could consider each of the data items in turn, sending a data item if there are any outstanding requests for it at the time. When the circular-broadcast scheduler runs out of items to consider, it considers all the items all over again. Alternatively, a scheduler such as R/Q (Chapter 2) operates quickly even when scheduling large numbers of data items, and can help reduce client delay. The R/Q scheduler assigns a score to each data item by computing the number of clients requesting the data item divided by the smallest remaining request size of a client requesting the data item. The scheduler then chooses to send a data item with highest score.

As we increase the number of clients making requests on a multicast channel, we expect the client delay on the channel to increase with the added load. On the other hand, as the number of clients increase, the likelihood of two clients requesting the same data item increases also, so that the overlap of client requests also increase with the number of clients. As a result, regardless of scheduler, we could expect that the client delay of a well-behaved system have an upper bound even as the number of clients increases indefinitely. An upper bound is reasonable because we know that a multicast scheduler can, in the worst case, resort to acting as a traditional "broadcast disk," and disseminate each data item in turn, cyclically. In this worst case, the client delay must be no higher than the time it takes the multicast channel to disseminate every available data item once; this forms the upper bound.

5.1.2 Managing Multiple Data Channels

In this section, we consider how a server, not knowing its clients' network capacities in advance, can split its available outgoing network into multiple multicast channels.

We assume that the server does not know its clients' bandwidths in advance because this is the case for many services in a heterogeneous environment such as the Internet. (For example, Web sites that offer "high-bandwidth" and "low-bandwidth" versions of their material often require viewers to choose which version to see by following a particular hyperlink.) Though many clients may have a fixed bandwidth for downloads, a server would not know this until such clients connected to it. Hence, a server may not know even the distribution of its particular clients' available bandwidth in advance.

Also, not only can clients be connected to the Internet using different-bandwidth links, even if they connected using the same bandwidth links, clients may have different bandwidths actually available for multicast download after subtracting the varying consumption of other network applications. Consequently, a server must assume a minimum supported client bandwidth—a bandwidth that all its clients are expected to meet to receive service—but the server must then decide how it will split its bandwidth into data channels in advance of actually running the service and receiving client requests.

There are a number of ways for a server to decide how to divide its outgoing network bandwidth into a number of data multicast channels. To better sample the large space of possible options, we want to choose several techniques that are noticeably different from each other. We describe them below, and assign each one an identifier in **boldface**.

- **eq-bw** A server can simply partition its throughput into a fixed number of equal-bandwidth channels, each offering all data. The fixed bandwidth is assumed to be no higher than the minimum client bandwidth, so that any supported client can connect to at least one of the channels. A faster client can subscribe to as many channels as its wishes, until the sum bandwidth of the subscribed channels reaches the client's network capacity. The client would then split its data requests among them.

  A client connecting to such a system needs to choose which channels to which to connect:

  - **eq-bw-rand** It could choose randomly.
  - **eq-bw-R** It could choose the channels with the fewest other clients, as determined by the server.

  - **eq-bw-Q** It could choose the channels with the smallest number of clients times average-request-size-per-client product, as determined by the server. (Let us call this product the *load factor*.)

  The first is the simplest to implement, but the latter two, with server intervention, may allow clients to distribute their load across the server's channels more effectively, improving service on the channels overall.

- **eq-part** A server exporting its set of data items $\mathcal{D}$ can simply partition its throughput into a fixed number C of equal-bandwidth channels, each offering a partition of distinct $|\mathcal{D}|/C$ data items. As with the prior option, the fixed bandwidth is assumed to be no higher than the minimum client bandwidth.

  Because each client may make requests across the entire pool of $|\mathcal{D}|$ data items, each client may need to connect to multiple channels. To ensure that it can receive the data it requested when it is sent, a client should connect to no more channels than its bandwidth allows at any time. As each channel's request is fulfilled, the client could disconnect from that channel and subscribe to another channel that it still needs. A client connecting to such a system would choose which channels to connect randomly, when it first connects and each time it switches from one channel to another.

  - **eq-part-all** An aggressive client, on the other hand, can attempt to connect to all data channels whose data it needs, and gamble that the requested items are not sent concurrently. If too much data happens to be sent at the same time, the client would listen to only a subset of channels that it can receive with its bandwidth, disconnect, then reconnect to rerequest the data items it missed. We mention this option for completeness.

- **lin-bw** A server can partition its throughput into a number C of linearly-increasing-bandwidth channels, starting from a base throughput B_0 (no higher than the minimum client bandwidth) and incrementing by some constant factor k, i.e., $B_0, (1+k)B_0, (1+2k)B_0, (1+3k)B_0, ...(1+(C-1)k)B_0$. Because some clients are not fast enough for the fastest channels, all channels must offer all data. This arrangement of channels bandwidths allows clients to easily subscribe to one channel near their capacity.

A client could connect to such a system in a number of ways:

- **lin-bw-hionly** A client may connect to the highest-bandwidth channel it can use, and request everything from it.
- **lin-bw-hidown** It can add lower-bandwidth channels until its network capacity is reached.
- **lin-bw-R** It can connect to channels in order of increasing popularity. That is, the server determines the subset of channels whose bandwidths are no higher than the client's available bandwidth, then assigns the client to a channel in the subset that has the fewest other clients subscribed. If the client has any bandwidth remaining after the subscription, the server determines a new subset and repeats until the client's bandwidth is exhausted or the server's channels have all been enumerated.
- **lin-bw-Q** It can connect to channels in order of increasing load factor (recall, clients times the average request size per client) in the same way as above.

• **exp-bw** Users can connect to the Internet using technology of widely varying speeds. For example, telephone-line (POTS) modem connections are tens of kilobits per second (kbps); "broadband" SDSL connections are hundreds of kilobits per second; "broadband" ADSL and cable-modem connections are currently able to receive, like 10Mbps Ethernet, a few thousand kilobits per second. Directly networked users on 100Mbps Ethernet can send and receive tens of thousands of kilobits per second, and so on. This order-of-magnitude progression in bitrate from one connection type to the next suggests channels of exponentially-increasing bandwidth.

A server can partition its throughput into a number C of exponentially-increasing-bandwidth channels, starting from a base throughput B_0, i.e., $2^0 B_0$, $2^1 B_0$,

2^2B_0, ... $2^{C-1}B_0$. Like the prior option, all channels must offer all data, and the same methods for clients to connect to these channels apply:

- **exp-bw-hionly** A client may connect to the highest bandwidth it can use, and request everything from it.
- **exp-bw-hidown** It can add lower-bandwidth channels until its network capacity is reached.
- **exp-bw-R** It can connect in order of increasing popularity.
- **exp-bw-Q** It can connect in order of increasing load factor.

- **perf-bw** If a server should happen to know, for example, that its clients are all of one of three download speeds, then it could choose to partition its bandwidth into channels of these three speeds, perfectly matching its clients' bandwidths. More generally, if the server knows the bandwidths of all its clients in advance, then it could choose "ideal" channels of bandwidths that match the most common client speeds. Again, in this scenario all channels must offer all data.

In scenarios except eq-part and -hionly, where a client connects to more than one channel, the client distributes its requests proportionately among its subscribed channels by bandwidth, rather than making all its data requests on the highest-bandwidth channel it can use. For example, if a client subscribes to two channels, one twice the bandwidth of the other, then it would send two-thirds of its requests onto the faster channel, and the remaining third of its requests to the slower channel.

Of course, these mechanisms for splitting server bandwidth may cause the server to have "left-over" bandwidth not allocated to a channel, which in turn causes some performance loss. Similarly, the above mechanisms may cause clients to have "left-over" download bandwidth not used to receive data from a channel; the performance impact of this effect will contribute to the evaluation of the above techniques.

5.2 Method

To evaluate some of the ways a server can partition its outgoing network bandwidth, we turn to a simple model to estimate our performance metric, client delay. Client delay measures the time between a client's connection to the multicast system and the earliest time the client has received a copy of all the data it requested—intuitively, the end-to-end time used by the client.

Given the large number of scenarios we wish to consider in our experiments (as detailed in Sections 5.3 and 5.4, and the varied number of multicast channels in our scenarios, it would be prohibitive to compute the delay on each channel via simulation. Instead, we develop a function to estimate the client delay on a channel, and use that function in our experiments. As we will see, the function is only a rough approximation for client delay, but it captures the essential behavior of the multicast channel. The function also provides some useful insight as to why the channel operates as it does, something a simulation does not provide.

Variable	Description
$D()$	Average client delay
N	Number of clients awaiting data
$\lvert\mathcal{D}\rvert$	Number of data items server offers
$t_{\mathcal{C}}^{-1}$	Client arrival rate
r	Request size for client on a channel
b	Throughput of channel ("bandwidth")
β	Bandwidth split factor

Table 5.2: Notation for Computing Client Delay

Our model assumes that the expected average client delay (D) is a function of three principal factors: the rate at which clients arrive to make requests of the server ($t_{\mathcal{C}}^{-1}$, the inverse of the time between clients); the number of data items each client requests on the multicast channel (r); and the throughput of the multicast channel (b). Other variables are described as they are introduced, and all are summarized in Table 5.2.

$$delay = D(t_{\mathcal{C}}^{-1}, r, b)$$

With this assumption, we are able to consider different ways of using a multicast server's available bandwidth by computing their effect on these three factors, and observing their result on client delay D.

To determine D, we first consider a low-load approximation for $D(t_{\mathcal{C}}^{-1}, r)$ based on M/M/1 queueing, for fixed b. Next, we recall that a multicast system can have an upper bound on its delay because in the worst case it takes finite time to disseminate all the available data once. We can establish this upper bound, which depends on the clients' average request size, r. Then, we determine the effect on the upper bound from the number of clients in the system, which we will translate into the client arrival rate $t_{\mathcal{C}}^{-1}$. This information will allow us to approximate the client delay function $D(t_{\mathcal{C}}^{-1}, r)$ at higher loads using the collector's problem. Finally, we merge the results of the client delay for low and high loads to determine a $D(t_{\mathcal{C}}^{-1}, r)$, and apply a factor from b to determine $D(t_{\mathcal{C}}^{-1}, r, b)$.

5.2.1 Low-Load Approximation

To determine a function of how client delay varies by the number of clients in the system, we first review some results of classical queueing systems (M/M/1 or G/M/1), then adapt it to our multicast scenario (which is not strictly a queue, as we see in detail below).

In an M/M/1 or G/M/1 queue, clients arrive to request a single job of a server in first-in-first-out order. Unlike our multicast system, the server can provide service to only one client at a time, after which the client departs. In this queue, if clients appear at exponentially-distributed intervals, it is known that the expected number of clients waiting in the queue at a time is $N = \frac{s/t_{\mathcal{C}}}{1-s/t_{\mathcal{C}}}$ for s the service time of a single client and $t_{\mathcal{C}}^{-1}$ the rate at which clients appear in the system. (Clients may not appear, on average, more quickly than they can be serviced, because this would overload the queue; $s/t_{\mathcal{C}} < 1$.) Also, by Little's result, the corresponding expected client delay is $D = Nt_{\mathcal{C}} = \frac{s}{1-s/t_{\mathcal{C}}}$.

Because our multicast facility is not a strict queue—a server can assist multiple clients, without regard to order, when it disseminates a data item—we need to adjust the queue delay formula above to better fit our system. To make this adjustment, we approximate the effect that clients' possibly-overlapping multiple-item requests would have on the multicast system's service time per client, and substitute the adapted service time into the delay function of an M/M/1 system. Though our multicast facility does not have a service time independent of the number of clients as the M/M/1 model assumes, we find that the resulting approximation remains similar to simulation results, as we see later.

For this approximation, we suppose that we have a large number of data items $|\mathcal{D}|$, from which the clients make their requests of r items. Further, let us say that there are already N clients in the system awaiting service, in the steady state. If a new client makes a request at random, it has a probability of (approximately) $(1 - \frac{1}{|\mathcal{D}|})^{Nr}$ that one item it requests is not already being requested by the N clients in the system before it. Because the client requests r items this way, it has an expected value of approximately $r(1 - \frac{1}{|\mathcal{D}|})^{Nr}$ requested items that have not already been requested by the N clients before it.

For simplicity, let us think of a unit of time as the time it takes to send a data item: Then, the additional service time that this new client requires is also $s = r(1 - \frac{1}{|\mathcal{D}|})^{Nr}$. With our substitution for s, we get $D(t_{\mathcal{C}}^{-1}, r) = Nt_{\mathcal{C}}^{-1} = \frac{r(1-\frac{1}{|\mathcal{D}|})^{Nr}}{1-r/t_{\mathcal{C}}(1-\frac{1}{|\mathcal{D}|})^{Nr}}$.

We cannot get a closed form for $D(t_{\mathcal{C}}^{-1}, r)$ from the above, because we do not have a closed form for N, but we are able to determine numerical approximations of this function. We can approximate N even without its closed form (using an implementation of Newton's method), then use the value of N to obtain $D(t_{\mathcal{C}}^{-1}, r)$.

This approximation of $D(t_{\mathcal{C}}^{-1}, r)$, however, is restricted to regions of relatively low load ("queue-like" behavior), so we must next approximate the client delay function at higher load.

5.2.2 Upper-Bound Delay

As with number of clients, we expect that client delay would increase with the size of the clients' requests. Similarly, we expect that as the request sizes grow, clients' requests will increasingly overlap, leading the increase in client delay to taper off.

More precisely, again we consider the worst case, in which the multicast system is reduced to cyclic broadcast. For one client just connected and waiting for r data items, randomly selected, we expect the client to wait until the last of its r random selections is broadcast. When the r data items are selected using a uniform distribution, the mean fraction of the data that passes before the client receives all its data is $\frac{r}{r+1}$ [22]; therefore, for $|\mathcal{D}|$ data items total, this is $|\mathcal{D}|\frac{r}{r+1}$ data items.

We see that this function, $\frac{r}{r+1}$, captures the desired behavior for the upper bound:

- For $r = 1$, the upper bound is approximately $|\mathcal{D}|/2$, which is what we would expect as the client delay for a very large number of clients each requesting a data item. In a multicast system reduced to cyclic broadcast, a client should expect, on average, about half the system's data items to be broadcast before receiving the one it wants.

- For large r, the upper bound must approach but not exceed $|\mathcal{D}|$, the time to transmit all the data once.

With this expression for the client-delay upper bound, we now determine how the number of clients affects the use of this bound.

5.2.3 High-Load Approximation

To determine how the number of clients in the system affects client delay, we observe that the above computation for upper bound determined the *fraction* of total data that a client should expect to receive before it gets the r items it requested. With sufficient load, we would expect that all data items the server makes available is being requested, so the total data is $|\mathcal{D}|$. At slightly lower load, by contrast, the clients in the system may collectively request fewer than $|\mathcal{D}|$ items from the server.

The number of items that a set of clients request can be approximated using results for the classic collector's problem, or occupancy problem [22]. In the collector's problem, a collector repeatedly purchases items, each chosen at random from an infinite bag of $|\mathcal{D}|$ distinct items so that each of the $|\mathcal{D}|$ items is equally likely to be acquired from any purchase. For a large number of purchases n and not-as-large $|\mathcal{D}|$ (such that $|\mathcal{D}|e^{-\frac{n}{|\mathcal{D}|}}$ remains bounded), the collector can expect to acquire approximately $|\mathcal{D}|(1 - e^{-\frac{n}{|\mathcal{D}|}})$ of the $|\mathcal{D}|$ distinct items at least once.

To map our problem to the collector's problem, we approximate our client requests so that each client requests one data item at a time out of $|\mathcal{D}|$ total, and does so r times on average. (This mapping is an approximation because it allows a client to choose the same item more than once.) Then, N clients would make Nr requests in total, and can expect to request $|\mathcal{D}|(1 - e^{-\frac{Nr}{|\mathcal{D}|}})$ distinct items in total.

Because the total data requested by N clients is $|\mathcal{D}|(1 - e^{-\frac{Nr}{|\mathcal{D}|}})$, and each client needs to receive $\frac{r}{r+1}$ of that before it has the items it chose, each client can expect a client delay of $D(N, r) = |\mathcal{D}|(1 - e^{-\frac{Nr}{|\mathcal{D}|}})(\frac{r}{r+1})$ for N clients requesting r data items each.

To obtain our high-load approximation $D(t_{\mathcal{C}}^{-1}, r)$, we have to determine what client arrival rate $t_{\mathcal{C}}^{-1}$ would generate a number of clients N requesting data from the server in the steady state. Fortunately, we have the resulting client delay $D(N, r)$, so we know how long each new client expects to spend in the system. Again invoking Little's Law, we observe that in the steady state, the rate of clients arriving must equal the rate of clients completing their requests, which on average is $N/D(N, r)$. So, $t_{\mathcal{C}}^{-1} = N/D(N, r)$.

In short, given some N and r, we can determine $D(N, r)$, then $t_{\mathcal{C}}^{-1}$. These values determine a domain-range pair of the delay function $(t_{\mathcal{C}}^{-1}, r) \mapsto D(t_{\mathcal{C}}^{-1}, r) = D(N, r)$.

5.2.4 Merging High- and Low-Load Approximations

To merge our two approximations, we observe that the low-load approximation tended to slightly underestimate our simulation results at low loads (as we see in the next section), an error which widens in high loads. So, we combine the high-load and low-load approximations by defining our delay function as the larger of the two. In the high-load approximation, not all values of $(t_{\mathcal{C}}^{-1}, r)$ are represented for integer N, so we use the delay of the nearest $t_{\mathcal{C}}^{-1}$ and r for which a value is available. In the lowest loads (ones for which there is no computable client delay at any lower $t_{\mathcal{C}}^{-1}$), we say that the high-load approximation is zero client delay, which ensures that the low-load approximation prevails when the high-load approximation offers no estimate.

5.2.5 Multicast Channel Throughput

Lastly, we approximate the contribution to client delay from the network resources of a multicast channel. To do so, we can begin by considering $D(t_{\mathcal{C}}^{-1}, r, b) = \frac{1}{b} D(t_{\mathcal{C}}^{-1}, r)$. That is, if a multicast channel of half the network resources of another has half the bandwidth and takes twice as long to send its data, then the contribution of the network cost to client delay is simply an independent linear factor.

Because of constant (nonlinear) overhead on a multicast channel, however, it is possible that the multicast channel of bandwidth $b/2$ incurs a delay factor different from $2/b$. Control packets (such as ping packets, if any) may be a constant overhead that consumes a larger fraction of low-throughput channels; overhead from transmission and retransmission for reliable data transfer may vary nonlinearly in the throughput of the channel (and its clients). On the other hand, it is possible for the bandwidth of a network connection to be nonlinear in its cost. For example, because of economies of scale in commodity connections and equipment, it might be less expensive for a server administrator to purchase two smaller network connections, and have a data channel over each of them to the outside world, than to pay for a single larger connection that may require the installation and maintenance of new network hardware and media. In such a case, spending $b/2$ units of resources on one channel may get more than half the effective bandwidth of what b units of resource would buy in one channel.

To consider such nonlinear contributions to delay, we consider slightly nonlinear factors, such as $(\frac{1}{b})^{\beta}$ for β near 1. If $0 < \beta < 1$, a linear increase in network cost buys a less-than-linear reduction on client delay (i.e., bandwidth is expensive). If $\beta > 1$, a linear increase in network cost buys a greater-than-linear reduction in delay (i.e., bandwidth is cheap).

As a result, we now have as our client delay function $D(t_{\mathcal{C}}^{-1}, r, b) = (\frac{1}{b})^{\alpha} D(t_{\mathcal{C}}^{-1}, r)$.

5.2.6 Comparison to Data

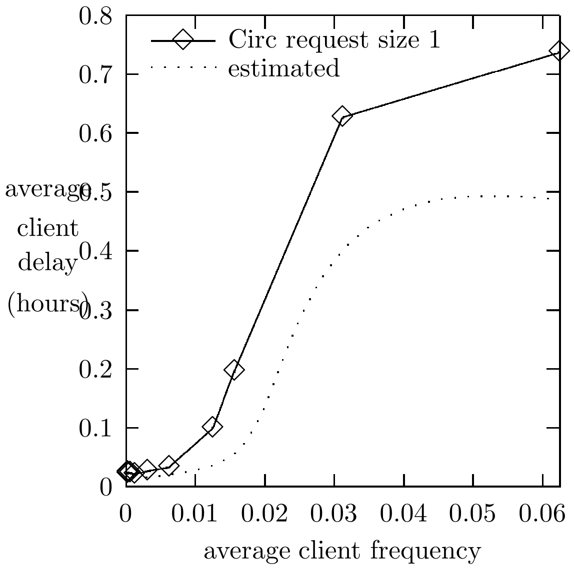

Figure 5.4: Delay versus Client Frequency

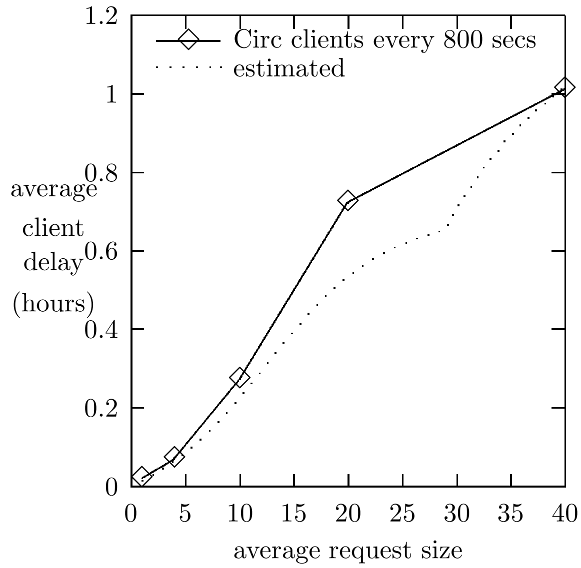

Figure 5.5: Delay versus Request Size

Lastly, for verification, we compare our analytic model to data from a detailed event-driven simulation. We note that our model omits details implemented in our simulation, which can increase the simulation's client delay. For example, in our detailed simulation, a client that first connects to the multicast system does not actually receive data immediately from the server; instead, a data-item transmission is probably already in progress, and unable to capture the entirety of the transmission, the client waits for the transmission to finish before it begins listening for data. This initial delay contributes to a client's total delay, but is not represented in our analytic model. Also, both our analytic models, based on the M/M/1 queue and the collector's problem, use a simplification that allows clients to choose a data item more than once as part of its "set" of requests, but duplicate selections in one client do not occur in simulation. As a result, our analytic model underrepresents the actual size of clients' very large requests, leading to an underestimate of their delay.

In Figures 5.4 and 5.5, we plot the simulated client delay of a multicast facility and our analytic model's results, calibrated to the same units of client delay, for comparison. In the two figures, both are plotted as a function of client-arrival frequency and as a function of the clients' average request size, respectively. The simulation uses a circular-broadcast scheduler (labelled "Circ"),

For both figures, clients arrive at exponentially-distributed random intervals, and request data items from a pool of 100 available data items, each of which takes 48 seconds to transmit. In Figure 5.4, these clients arrive with an average request size near one data item and vary the frequency of new clients along the horizontal axis. In Figure 5.5, we vary the average request size of the clients, fixing the clients' average interarrival time at 800 seconds.

In both figures, we see that the analytic model (labelled "estimated," and appearing as a dotted line) captures much of the behavior of the circular-broadcast scheduler ("Circ"), as intended. In particular, we find that many values from the analytic model are similar to the corresponding values from our detailed simulation. Also, the estimate captures the expected behavior of the system, increasing with client load and request size as expected.

We notice, though, that our model tends to underestimate the simulation's client delay values, especially in more heavily loaded scenarios. Also, we see an artifact of our merging two separate estimates, a low-load and high-load approximation; in Figure 5.5 we see a nondifferentiable "corner" at the point where we switch from one approximation to the other. As a consequence of the switch, though, we get estimates much closer to our simulated values.

Although our analytic model provides approximate results, we have verified that the conclusions reached using the model (detailed in subsequent sections) are not sensitive to the approximation. For example, in one test we change the predictions of the analytic model by up to 20%. Even with this large change, the relative performance of the schemes we studied did not change.

5.3 Results for Varying Numbers of Channels

In this section, we examine how server performance is affected by the number of multicast channels it uses. To do so, we consider, for this section only, a simple initial scenario in which clients are as fast as the server, allowing them to receive whatever

Parameter	Description	Range
$\lvert\mathcal{D}\rvert$	Number of data items available	100
r	Number of data items requested per client	1–50
C	Number of channels	1–5
B_0	Base throughput (minimum supported bandwidth)	1 (20 kbps)
β	Bandwidth split factor	1.0
	Server bandwidth	50 (1 Mbps)
	Client bandwidth	50 (1 Mbps)

Table 5.3: Some Study Parameters

data the server can send.

In this initial scenario, we run one server for each of the multi-channel techniques we consider in Section 5.1.2. Each server is given fifty atomic, discrete "units" of bandwidth which correspond to approximately one megabit per second of outgoing bandwidth. (As in the prior section, each data item takes about 48 seconds to transmit each data item at this bandwidth, corresponding roughly to the minute it takes to send a Web site in at this bandwidth in Chapter 2.) Each server uses as many of its fifty units as its bandwidth-allocation technique allows.

We are able to test all the servers using one to a maximum of five channels. At six or more channels, the exp-bw servers cannot split their bandwidth without making their slowest channel slower than the base unit of bandwidth, about 20 kilobits per second (compare to a 28.8kbps modem).

To simplify the results, we present here only four of the servers, reserving the remainder for a following section. In particular, we present exp-bw-hionly instead of all the exp-bw servers, and lin-bw-hionly instead of all the lin-bw servers. The -hionly servers are not only the simplest implementation for each technique of splitting server bandwidth, assigning each client to exactly one data channel, but as we shall see, they often perform well within their group, and their results help us understand the multicast system better. Similarly, for simplicity, we will show eq-bw-Q and not eq-bw-R, and we will show eq-part, giving us one server from each group of techniques for splitting server bandwidth.

Because clients have as much bandwidth as the server, they can listen to as many channels as each server scheme allows. Consequently, we will be able to link changes in performance directly to the server scheme and the different numbers of channels, rather than to the varying degree that clients' bandwidths are supported, as could be the more general case if clients were slower than the server.

We also consider a varying average client request size, of up to half the available repository, to see how request size affects our results. Table 5.3 summarizes our space of parameters. For simplicity and clarity, we have chosen to have 100 data items ($|\mathcal{D}|$) in the full repository, as the granularity by which clients make requests. Given our model, this means that we can vary clients' request size as finely as units of one percent of a repository's data pool. Request size maps conveniently to one percent of a server's repository.

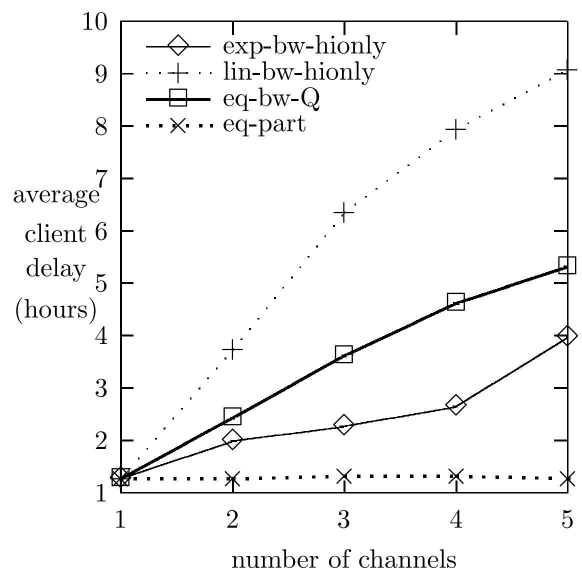

Figure 5.6: Varying Number of Channels for Request Size 9

In Figure 5.6, we vary the number of channels from one to five along the horizontal axis, and plot the average client delay on the vertical axis. Clients request nine data items each. At one channel, all of the server schemes become the same single-channel multicast, so all schemes have an identical client delay.

From this figure and other related results, we highlight some conclusions and intuition about our system exposed in this scenario.

One, we find that increasing the number of channels itself creates inefficiency that slows down the system overall for most schemes, so to optimize performance, a system designer should use as few channels as possible to meet client bandwidth requirements.

Two, we find that eq-part holds its performance rather than degrading in this scenario. This indicates that where we expect our clients to have uniformly high bandwidth, a data-partitioning scheme can best limit the performance penalty that would normally accompany splitting the server's bandwidth into smaller channels to accommodate lower-bandwidth clients. (In following sections, we shall see how the schemes compare for other client loads.)

Three, we find that of the schemes tested above that do not require data to be partitioned across channels (the -bw- servers, as opposed to the eq-part servers), exp-bw-hionly degrades relatively little as we increase the number of channels and the size of client requests. Surprisingly, this is true even though clients use less of the server's bandwidth in the exp-bw-hionly than they would in other schemes in which clients can use more than one channels' bandwidth.

Studying other results in this scenario, we find that the most significant factor in the performance of these systems is the average client delay of the lowest-bandwidth channel. Because a client listening to data items on all of its subscribed channels must wait for the data items on its slowest channel to finish transmission before the client is done, it must wait for whichever channel takes the longest to complete. In this scenario, the channel of highest delay is generally the lowest-bandwidth channel. Though a low-bandwidth channel has proportionately fewer data-item requests than higher-bandwidth channels, the cut in bandwidth itself hurts client delay dramatically. For example, we find that as client request size grows, lin-bw-hionly, which uses only its fastest channel, improves relative to other lin-bw schemes as the request sizes increase. Also, the lin-bw schemes that use all their channels (that is, except -hionly) improve relative to the exp-bw schemes using all their channels because lin-bw allocates more bandwidth to its lower-bandwidth channels than exp-bw does.

The eq-part server, on the other hand, demonstrates that a lower maximum delay can compensate for lower channel bandwidth where a smaller request size does not. In an eq-part server, partitioning forces the requests of the clients on each channel to coincide more frequently over smaller sets of requested data items; the smaller sets become key because they reduce the *maximum* client delay on that channel by the same factor that the decrease in bandwidth increases the delay. In the other schemes, where the maximum delay is unchanged on each channel, the reduced request load on each channel has less effect on the delay in that channel.

The fact that even eq-part can only maintain the same level of performance across varying number of channels, when clients are ideal and able to receive all of the server's channels concurrently, warns us that in practice, with some clients required to listen to subsets of the channels at a time, eq-part will perform worse than we see here. We consider that next, with a more heterogeneous mix of clients.

5.4 Results for Two-Tiered Clients

Parameter	Description	Range
$\lvert\mathcal{D}\rvert$	Number of data items available	100
r	Number of data items requested per client	1–50
C	Number of channels	varies by technique
B_0	Speed of slow clients	1
	Speed of fast clients	1–50
β	Bandwidth split factor	1.0
Fast and slow clients appear in equal numbers.		

Table 5.4: Some Two-Tiered Study Parameters

In this section, we determine how various multiple-channel multicast schemes compare for clients of different bandwidths. To do so, we consider a scenario in which clients requesting data of a server are equally likely to be low-bandwidth ("slow") or high-bandwidth ("fast"). Though we also considered another scenario, in which clients have exponentially-distributed levels of bandwidth (motivated by the same observation behind the eq-bw techniques—that many connections to the Internet fall into "tiers" of technology whose maximum bitrates are orders of magnitude apart), we found that the results from this simpler two-tier scenario are similar and easier to describe. Therefore, for clarity, we use the two-tier scenario below, and omit the corresponding exponentially-distributed-bandwidth scenario.

In this scenario, we vary the relative bandwidths of the slow and fast clients by keeping the slow clients at one base unit of bandwidth, and varying the bandwidth of the fast clients. We vary the bandwidth of the fast clients across its entire allowable range, from the one unit of the slow clients up to the fifty units given to the server. We also vary the request size of a client, in data items, from 1 to 50. The server uses as much of its fifty units of bandwidth as it can, as its bandwidth-splitting technique dictates. This allocation yields 50 one-unit channels for eq-bw and eq-part bandwidth allocation, nine channels of bandwidths one through nine for lin-bw bandwidth allocation, and five channels of bandwidths 1, 2, 4, 8, and 16 for exp-bw bandwidth allocation.

Table 5.4 summarizes the parameter space described above, in which we evaluate our various multicast schemes.

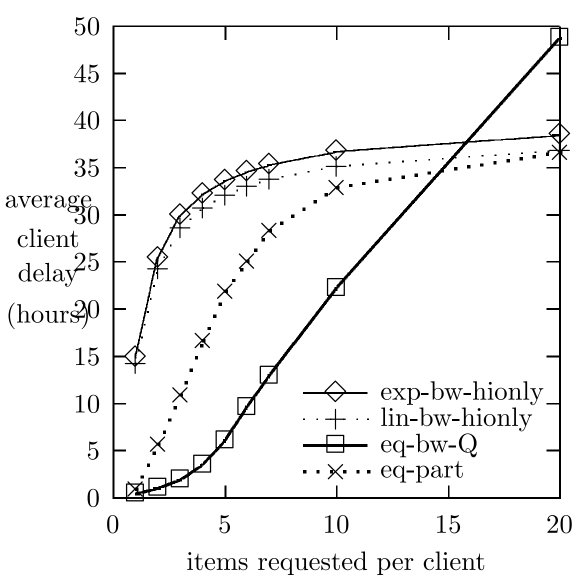

Figure 5.7: Various Request Sizes for Clients Having 5:1 Bandwidth

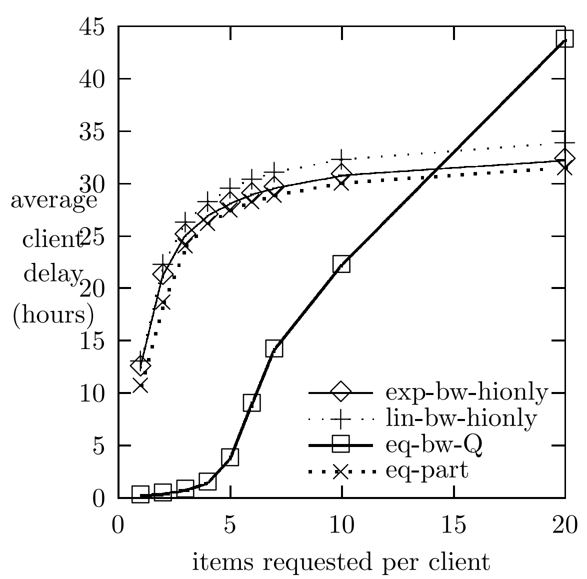

Figure 5.8: Various Request Sizes for Clients Having 30:1 Bandwidth

In testing our space of parameters and comparing the relative performance of numerous techniques, we find that a good technique can often provide a two- to four-fold improvement in delay over a bad one, suggesting that under most circumstances, there are clear choices that must not be overlooked.

For example, in Figures 5.7 and 5.8 we plot the client delay, averaged for all clients, as a function of the clients' request size in data items. In Figure 5.7, the fast clients have five times the bandwidth of the slow ones. In Figure 5.8, the faster clients have thirty times the bandwidth of the same slow ones. Because there are as many fast as slow clients, the average client delay plotted in the figures show the midpoint between the delay of the fast clients and the delay of the slow ones. As client-request size increases to the right of each plot, request load grows in the system and client delay goes up. As they approach the high-load steady-state of the multicast system, the increase in client delay tapers off. Because the slow clients have only a small fraction of the bandwidth that the server has, they require forty minutes to receive a single data item. Hence, slow clients can take nearly seventy hours (forty minutes times one hundred items) to receive all their data from a server's transmission of its data items.

First, we can see that in this scenario, the choice of server technique to provide the lowest client delay is between eq-bw-Q and eq-part. In particular, eq-bw-Q does well for lower request sizes (up to about 15% of the total data pool), and eq-part becomes the lower-delay choice after that.

For low request sizes, both eq-bw and eq-part schemes are able to distribute clients across a large number of equal-bandwidth channels, but eq-part requires clients to hop across channels to receive all of their data and eq-bw does not. As a result, each channel in both scenarios has a fairly small number of data items to distribute (and hence, fairly low client delay for clients on that channel), but eq-bw saves clients the time of waiting on multiple channels in sequence.

As request size increases, though, the number of data items requested on each eq-bw channel grows up to the size of the entire repository; eq-part channels, on the other hand, have a much lower maximum request size within each channel, so it scales to larger request sizes with less degradation. So, for higher request sizes, eq-part does better than eq-bw.

5.4.1 Results Comparing More Techniques

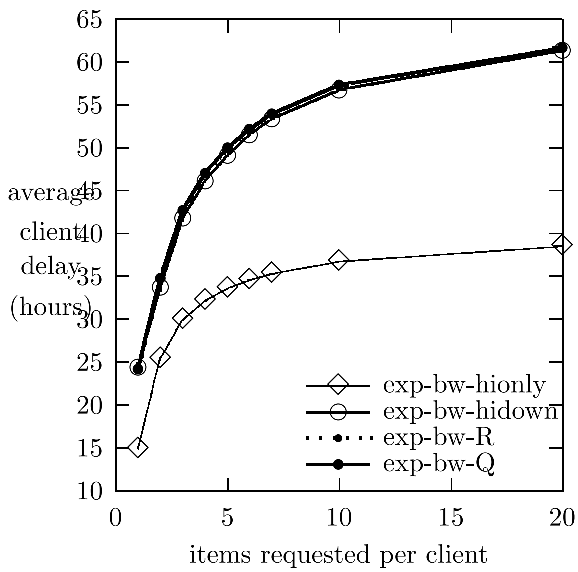

Figure 5.9: Various Request Sizes for Clients Having 5:1 Bandwidth (exp-bw only)

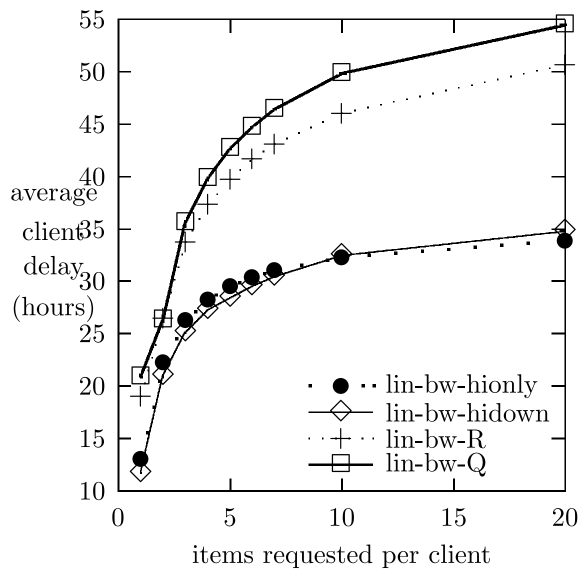

Figure 5.10: Various Request Sizes for Clients Having 30:1 Bandwidth (lin-bw only)

We can also use the scenario of Table 5.4 to study the other variants that have not been considered. In Figure 5.9, for example, we chart all the exp-bw techniques in the scenario corresponding to Figure 5.7. (That is, the exp-bw-hionly points in the two figures are identical because they represent the same scenario.) In Figure 5.10, we chart all the lin-bw techniques in the scenario corresponding to Figure 5.8.

We observe that the -hionly servers perform well relative to other client-channel assignment techniques. For example, in Figure 5.9 exp-bw-hionly outperforms the other exp-bw servers, and in Figure 5.10 lin-bw-hionly is an optimal or near-optimal performer among lin-bw servers. We find that -hionly schemes counterintuitively often do better, not worse, than -hidown schemes, even though the latter use more of the server's and clients' bandwidth for transmitting data. Consequently, exp-bw-hionly and lin-bw-hionly servers are useful choices to represent the exp-bw and lin-bw

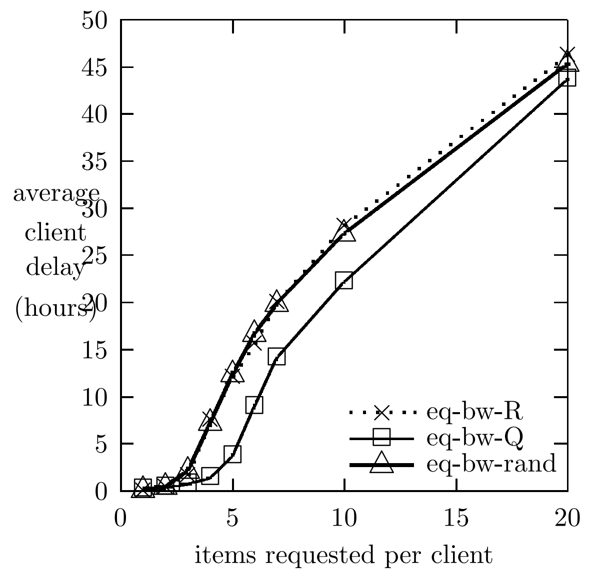

Figure 5.11: Various Request Sizes for Clients Having 30:1 Bandwidth (eq-bw only)

groups of servers; not only is their client-channel assignment technique fairly easy to implement, we see that their performance within their groups is fairly good.

To understand why -hionly servers perform well, we recall that client delay is determined by the "slowest" channel, the connected channel that has the highest average delay. Here, we find a demonstration of that that observation. The -hionly server allocates all clients to the top-bandwidth channel only, and so is better able to minimize the effect of slower lower-bandwidth channels. As a result, clients are often better off listening to high-bandwidth channels for data, even though clients have to wait through more data-item transmissions to get the data they need. In lower-bandwidth channels, the waiting time for a data item imposed by conflicting requests affects client delay more than the transmission time for those data items. Clients are often better off listening to high-bandwidth channels for data, even when they are able to add lower-bandwidth channels using their remaining download capacity.

To understand why -hidown does particularly well in Figure 5.10, let us examine how the -hionly and -hidown systems actually behave given the scenarios plotted in

Figures 5.9 and 5.10. For the scenarios in both figures, all the exp-bw and lin-bw schemes have a unique bandwidth-one channel, and the low-bandwidth clients are all on it. The high-bandwidth clients, on the other hand, have other options. In particular, in exp-bw-hidown, bandwidth-30 clients can be assigned to channels of bandwidth 16, 8, 4, and 2 (for a total of 30), channels shared only with each other. This arrangement allows high-bandwidth clients to use most of the server's available bandwidth, all of their own bandwidth, and most importantly, not have to depend on the one channel where all the low-bandwidth clients are already loaded. Hence, exp-bw-hidown does much better relative to other exp-bw servers in the 30:1-bandwidth scenario than in Figure 5.7, where the faster clients do wait on the low-bandwidth channel (because they are allocated to channels of bandwidth 4 and 1). Similarly, lin-bw-hidown is able to assign bandwidth-30 clients to high-bandwidth channels of bandwidth 9, 8, 7, and 6 (for a total of 30), channels shared only with each other and not with low-bandwidth clients, so we see relatively good performance for lin-bw-hidown in Figure 5.10. We see, then, that -hidown does well for fast clients whose bandwidth happens to split into higher-bandwidth channels not shared with slow clients, and it does poorly if those fast clients have any bandwidth left over that causes them to be assigned to the bandwidth-one channel. The -hionly servers, on the other hand, are not affected by such small differences in bandwidth, and always separate fast clients from slow ones.

Lastly, we find in our results that eq-bw-Q often outperforms the other two variants of eq-bw slightly, suggesting that for nontrivial client request sizes, the Q "load factor" is a useful channel-assignment heuristic, all else being equal. In Figure 5.11 we see a representative result from comparing the eq-bw options. We see that eq-bw-R does not significantly outperform random channel assignment, but eq-bw-Q performs slightly better.

5.4.2 Fast and Slow Clients

With two tiers of clients, we can further examine how our various techniques treat fast and slow clients separately. In Figures 5.12 and 5.13, we consider a scenario in

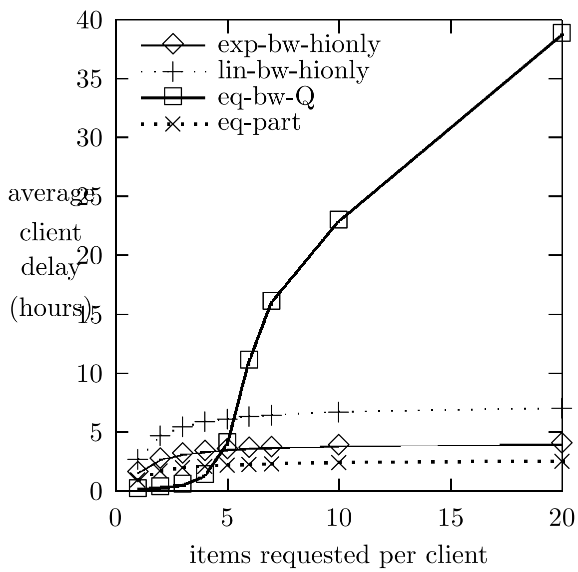

Figure 5.12: Various Request Sizes for Fast Clients Having 40:1 Bandwidth

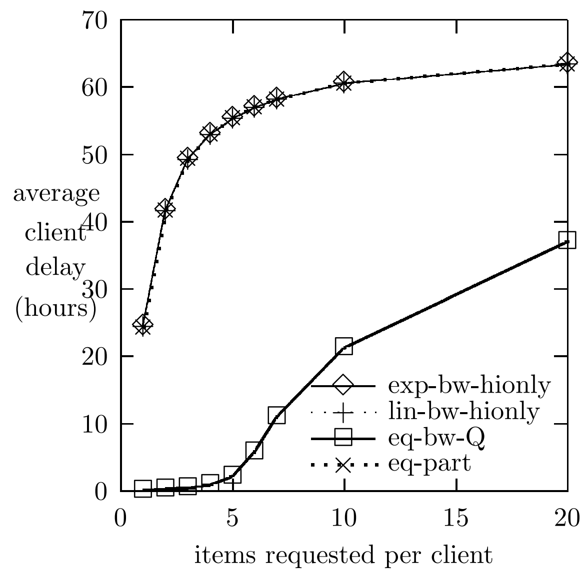

Figure 5.13: Various Request Sizes for Slow Clients Having 40:1 Bandwidth

which the fast clients have forty times the base bandwidth of the slow clients, again with both classes of clients appearing in equal numbers. In Figure 5.12, we show the average client delay of the fast clients. In Figure 5.13, we show the average client delay of the slow base-bandwidth clients.

We find two particularly noteworthy features when comparing these plots. One is that the -hionly techniques and eq-part strongly favor fast clients over slow ones. For fast clients, the -hionly techniques' performance approaches that of eq-part; in a few scenarios (with larger request sizes, not plotted here), they can even beat eq-part for the best all-client average delay. For slow clients, on the other hand, their performance is relatively poor. In effect, the -hionly and eq-part techniques are "fair" in the sense that fast clients with forty times the download bandwidth get their data about forty times as fast: they are proportionate.

By contrast, we see that eq-bw techniques offer exactly the opposite behavior. Their performance for both fast and slow clients is comparable, despite the difference in bandwidth between them. Because eq-bw has channels that are all alike, all channels' performance are similar, and the maximum time a client may have to wait to receive the last item it needs does not vary much by the channel(s) to which it is subscribed. In effect, the eq-bw techniques are “fair” in the sense that they strive to provide comparable service to all clients, regardless of bandwidth: they are egalitarian.

In summary, we find that the performance of exp-bw and lin-bw systems depends significantly on which streams clients get assigned. If fast clients get assigned to slow streams with slow clients, their overall average client delay suffers. On the other hand, systems in which fast clients are separated from slow ones perform better than ones where all clients coincide on a channel, even if the fast clients use more download bandwidth in total in the latter case.

More importantly, however, the best performers create channels of equal bandwidth. For relatively small request sizes (up to about 15% of the data pool), eq-bw-Q provides minimal average client delay, and for the larger request sizes (greater than 15% of all data items), eq-part is the best performer of the techniques we considered.

5.5 Bandwidth Factor

In Figure 5.14, we consider the effect of nonlinear bandwidth factors β on the client delay of our techniques. Recall from Section 5.2 that a low factor ($\beta < 1$) represents expensive bandwidth, so that an increase in bandwidth use (cost) for a channel acquires a less-than-linear increase in throughput to clients, and that a high factor ($\beta > 1$) represents cheap bandwidth, so that an increase in bandwidth use (cost) for a channel leads to a greater-than-linear increase in throughput to clients on the channel.

In the figure, we consider clients of exponentially-distributed bandwidth, chosen to approach an average 40 units but capped at the extremes of 1 and 50 units. The clients request 20 data items each from a multicast server. We vary the bandwidth factor α along the horizontal axis to observe its effect on average client delay, plotted on the vertical axis.

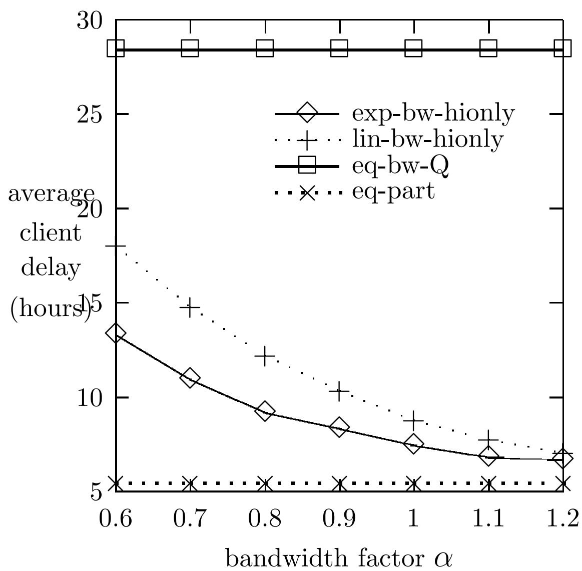

Figure 5.14: Effect of Nonlinear Bandwidth Factors on Exponentially-Distributed-Bandwidth Clients

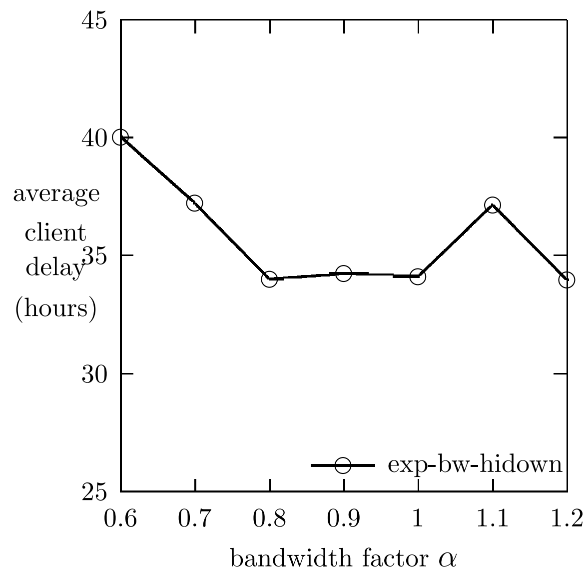

Figure 5.15: Effect of Nonlinear Bandwidth Factors on Exponentially-Distributed-Bandwidth Clients

We find that the effect of nonlinear bandwidth cost varies by the bandwidth-splitting technique used by the server. For eq-bw-Q and eq-part, which use channels of base (unit) bandwidth, there is no effect because all the channels are unchanged. For the -hionly schemes, though, we see an improvement in performance as β rises. Because higher-bandwidth clients are directed into the one highest-bandwidth channel to which they can subscribe, an improvement in that channel's throughput reduces their client delay directly. As we can see in Figure 5.14, the client delay of the -hionly schemes approaches that of the lowest-delay eq-part scheme as the bandwidth of the high-bandwidth channels rises.

To see why the gap in performance between -hionly and eq-part narrows, let us compare how long it takes the two schemes to transmit the entire repository once. In this scenario, the eq-part server offers fifty channels with 2% of the repository each, so let us suppose that it takes two units of time to send 2% of the repository over a one-unit-bandwidth channel. For clients that can subscribe to a sufficient number of channels simultaneously, those two units of time are sufficient to receive all the data they need. For comparison, it would take a -hionly server about the same amount of time—only about two and a half units—to send the entire repository at bandwidth 40, if such a channel were available to clients having sufficient capacity. In this scenario, clients would have sufficient capacity on average, but their actual delay will vary by their actual bandwidth and the bandwidths of the channels available to them. So, we see that with increased bandwidth made available by larger β, the -hionly servers' performance improves as fast clients get faster channels over which to request their data.

In other cases, where the effect of changing channel bandwidths affects not just which channels a client subscribes, but also how many, the benefit of slightly increased channel bandwidths is less clear. Where clients subscribe to fewer channels to fill their download bandwidth, for example, benefits from higher bandwidth channel are offset by the larger request load on each channel that results, from an increase in request size per client and from an increase in number of other clients. We see an example in Figure 5.15, in which we chart the performance of exp-bw-hidown under the same scenario. Though the increase in β from 1 to 1.1 increases the bandwidth of two channels, from 8 to 9 and 16 to 21, performance actually deteriorated. At β 1.1, clients of numerous bandwidths (such as clients of bandwidths 16–20) were all subscribed to the same channels (of bandwidth 1, 2, 4, and 9), including the one-unit low-bandwidth channel, even though that arrangement underutilized some clients' available download bandwidth. Fortunately, we find that these less predictable servers do not outperform all the schemes we have already considered in Figure 5.14, so we need not worry about their more chaotic behavior.

5.6 Client Bandwidth Information

Lastly, we consider whether a server benefits from knowing its clients' exact bandwidth. In Figure 5.16, we consider a scenario in which clients have either one base unit of bandwidth or forty units of bandwidth. In this case, the two tiers of clients do not necessarily appear in equal numbers; instead, we vary the proportion between them along the horizontal axis, with mostly slow clients at the left of the chart, and mostly fast clients at the right. Along the vertical axis, we plot the client delay of all

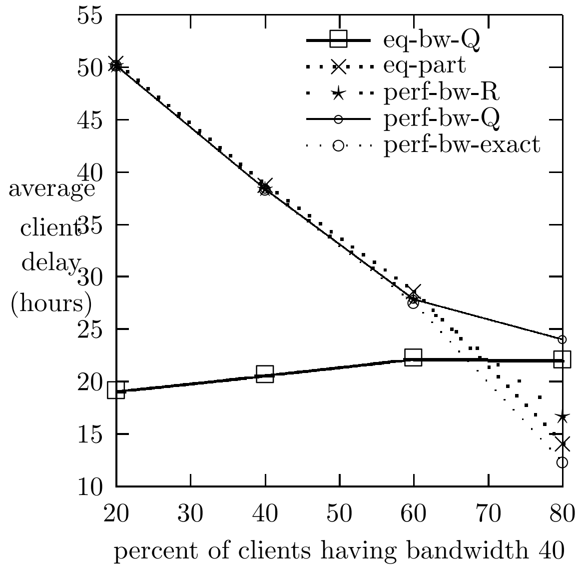

Figure 5.16: Various Numbers of Fast Clients Having 40:1 Bandwidth

clients in the system.

In this test, we add new servers using techniques that know and use information about the clients' bandwidths. Unlike lin-bw and exp-bw techniques, these new *perf-bw* servers do not have to create channels of varying bandwidths because they serve only the two given classes of clients, and can do so knowing exactly how fast the fast clients are. Instead, perf-bw servers create only data channels of the two known tiers of bandwidth, in equal numbers, until the server's available bandwidth is exhausted. For example, in our scenario, we have clients of one and forty units of bandwidth, so the server creates one channel of one unit for the slow clients and one channel of forty units for the fast clients. (If the server had more than eighty-two units of bandwidth at its disposal, it would create another pair of such channels.) The perf-bw-R and perf-bw-Q servers use the same client assignment techniques (client popularity and load factor) as described in Section 5.1.2. Below, we describe and consider perf-bw-exact.

We see in the figure that, surprisingly, the server's knowledge of the clients' exact bandwidths does not dramatically improve the performance of system beyond what we are already able to do with eq-bw-Q and eq-part. If server knowledge improved the performance of the system, we should see a perf-bw server have lower client delay (a lower curve) than the existing server eq-bw-Q where its delay is lowest, or lower than eq-part where its delay is lowest. Instead, we see that all the perf-bw servers have fairly similar performance to eq-part in Figure 5.16.

Also, surprisingly, perf-bw-Q does worse than perf-bw-R for fast clients (as shown toward the right). In contrast, this observation is the opposite of what we found for eq-bw techniques, where load factor (Q) was a better channel-assignment than popularity (R). It turns out that, as we run out of slow clients in the system, both systems start assigning fast clients to apparently-underutilized slow channels. In this scenario, perf-bw-Q distributes more fast clients to slow channels than perf-bw-R, and the higher delay of these fast clients hurts client delay.

Hence, we see that we can do better than perf-bw-R and perf-bw-Q by having a new perf-bw server assign fast clients only to fast channels, and slow clients only to slow channels. This behavior is plotted as perf-bw-exact. As we can see, its client delay performance does not deteriorate even when fast clients strongly outnumber slow ones, and this server is able to slightly outperform eq-part, including scenarios where eq-part had the lowest delay of the techniques tested. (The exp-bw-hionly and lin-bw-hionly servers, omitted from the plot for clarity, have performance slightly worse than that of eq-part in Figure 5.16.)

5.7 Related Work

There is much work on multiple-channel systems for disseminating data [9, 61, 13, 39, 10], but such work typically assumes the transmission of only one data item to all clients, rather than disseminating possibly-overlapping selections of data items from a large repository. In a scenario with only one data item, all data being transmitted is interesting to all clients, unlike our scenario, where reducing client delay often requires reducing clients' waiting time for their data in the presence of conflicting requests. Consequently, this chapter studies how to allocate different data items to different channels, as well as how to assign clients making varying requests to such channels.

There is a large body of work for layered multicast of a single audio or video stream (such as References [42, 62]). This work often assumes a scenario that, unlike ours, allows sending only partial data to lower-bandwidth clients. Variants on layered multicast adjusts the bandwidth of each data channel over time (e.g., using equation-based bandwidth estimation [34]), sometimes to optimize a client-utility metric ("Fairness Index" [38] or "social welfare" [33]). Reference [37] considers bandwidth allocation for video streams, between different streams of different importance and within streams for different clients, assuming the streams are sent simultaneously and clients can accept fewer bits for less quality.

Collectively, this related work helps clients determine their available download bandwidth for multicast without starving TCP and TCP-friendly connections under congestion, an issue we do not consider here except to assume that clients correctly determine their bandwidth. (Readers interested in this area may also wish to pursue work on ensuring clients remain TCP-friendly even when motivated to oversubscribe to data channels for more throughput, such as [28].) Our present work, on the other hand, considers the distribution of many data items across different channels, without degrading the data for slower clients.

While not designed for large-scale data dissemination, multicast and broadcast protocols are also often considered in the context of group communication protocols [21], which provide a message-passing protocol that guarantees delivery and ordering. For example, Reference [21] considers probabilistic group communication in which both reliable delivery and consistent ordering can be probabilistic, and Reference [6] uncovers and studies failures in the Ensemble group-communication system.

Though we assume multicast exists as a network-level service, it is possible to implement multicast at the application layer using host-to-host unicast connections. For example, unicast can link disconnected multicast-capable networks [67]. Alternatively, epidemic/gossip protocols can connect the server and data-requesting clients, using a hierarchical (rather than flat) structure to better fit wide-area networks [29]; to provide reliability for an unreliable-multicast service for mobile clients [15] (data is sent by multicast, then gossip, to maximize the number of clients that the data successfully reaches); or alone to provide probabilistically reliable multicast service in an ad-hoc network [40]. TCP has network-friendly congestion control, so with proper buffer management to optimize the usefulness of deliverable data, it is a viable option for application-layer multicast [47]. For more information, Reference [14] compares several approaches to implementing gossip protocols.

5.8 Conclusion

In this chapter we studied how to use a multicast facility to reliably disseminate data to interested clients of different download capacities. To accommodate clients of different network speeds, a multicast server can slice its available outgoing network connection into channels of equal or differing bitrates, varying linearly or exponentially. If the server uses channels of equal rates, it can offer its entire data repository on each channel, or it can partition its repository across its channels, requiring clients to hop from channel to channel as needed.

To study channel slicing, we developed an analytic model that effectively captures performance for one multicast channel. This model makes it possible to evaluate the large number of configurations that had to be considered in our evaluation.

Studying these different server techniques under a variety of conditions, we find that having equal-bandwidth channels offers the best performance for a variety of client loads. Further, if clients request fairly small portions of the available data—up to 15% of the repository—then the server should offer all of its data on all channels, and balance client load across the channels using a load factor combining the number of clients on a channel and the number of data-item requests they are making. On the other hand, if clients request larger portions (over 15%) of the repository, then the server should partition its data into disjoint channels. Clients should subscribe to channels offering data of interest as their download speed allows, migrating from one channel to another as it receives the data it needs.

We find that the proper split of server network capacity into channels can improve performance very significantly (up to five-fold in the scenarios we studied). By correctly determining how clients are assigned to the channels that fit within the clients' download limits, we can again improve performance very significantly (three-fold or more in our studies), suggesting the importance of careful design choices. With the results from these last four chapters in hand, let us now turn, in Chapter 6, to the actual implementation of a multicast data facility.

Chapter 6

Implementation

As we can see from the text up to this point, the careful design of a data multicast facility requires research into many issues. To be useful, the multicast facility must successfully handle many challenges:

1. It must transmit requested data in a way that minimizes wait time for the clients requesting data (Chapters 2 and 3).

2. It must efficiently guarantee complete delivery of all the requested data to all its clients, even though the IP multicast networks it uses are unreliable (lossy) by design (Chapter 4).

3. It must be able to transmit to clients of different download speeds without flooding links of slow clients or providing needlessly slow service for fast ones (Chapter 5).

4. It must not use excessive numbers of IP multicast addresses, a finite resource that must be shared with other running implementations and other multicast services.

In this chapter, we describe how we implemented a multicast facility designed to meet the above challenges, and measure its performance when run over low-end commodity computers. In Section 6.1, we review the high-level design of our multicast server, and the approaches we considered to attack each of the challenges listed above.

In Section 6.2, we describe some design decisions we made for our implementation. Next, in Section 6.3, we run our implementation to measure its performance, test its scaling limits, and compare them to that of a traditional unicast file-server approach.

6.1 Basic Design

In our multicast facility, the server offers a set of data items, each one of which is assigned a unique numeric identifier (cf. files and inode numbers). To create this set of items, the server splits each file it offers into disjoint fixed-size pages, making each page a data item. Clients request files from the server, which the server translates into requests for all of the pages created from those files. Clients may receive items in any order, and rearrange them back into the original requested files. In this way, our multicast server can honor the assumption from previous chapters that data items be of similar size, and still offer files that can vary widely in size. For example, a crawl of one Web site in the Stanford WebBase can be offered as one file, without regard for precisely how much data WebBase was able to obtain from that site (up to the limits imposed by the local filesystem).

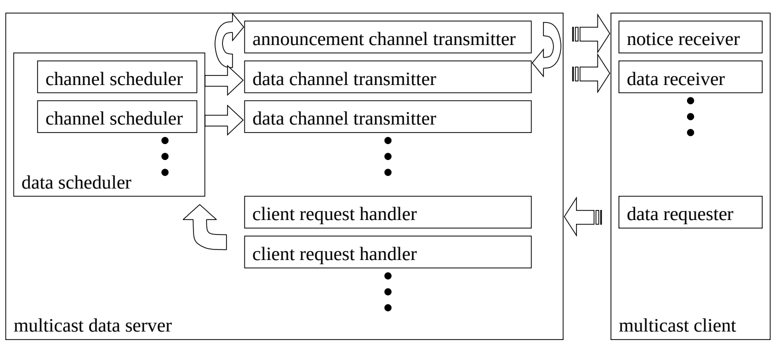

Figure 6.1: Overview of Multicast Facility Design

In Figure 6.1, we see the basic design of our multicast facility. The box on the left represents the server process, running on a server offering data over multicast. The box on the right represents a corresponding client, which a user would invoke to fetch data from the server. With this figure as reference, we will first outline the basic flow of operation of the multicast facility. Then, we can focus on the server's various components in more detail.

6.1.1 Normal Flow of Operation

To begin a data request, a client must first know the address and port number where the data server is listening, and connect to it using a unicast (TCP) connection. Because this connection is unicast, each such connection connects only two peers (as defined by TCP), one client and the server. This unicast control (request) connection provides the channel for all client-to-server communication, and allows the server to send responses specific to one client, for the duration of the client's use of the server. Using this request connection, the client sends the list of files it wishes to receive from the server, as indicated in Figure 6.1 by the arrow from the multicast client's data requester to its server's client-request handler.

In IP multicast, the server sends a transmission to multiple recipients together by sending the transmission over UDP to a destination IP address reserved for IP-multicast use. (The addresses reserved for multicast are defined in the IP specification, and are unassigned and unassignable to any *individual* host on the network.) This multicast IP address represents a *multicast group* of clients who are interested in receiving the data sent to that group address. The exact set of clients in this group are defined by the clients themselves, subscribing and unsubscribing from the group address by sending group membership announcements to their local routers.

The server sends the client the server's multicast group (IP) address information, so that the client is now able to receive the server's multicast transmissions. The client uses this information to subscribe to the multicast group for the server's single announcement (control) channel. (The arrow from announcement-channel transmitter to multicast client at the top of Figure 6.1 is a multicast transmission from the server to all its clients over the announcement channel's multicast group address.)

The server's client-request handler receives the list of requested items from the client, and adds the client's request information to the server's data scheduler (as suggested in the arrow in the server from client-request handler to data scheduler). The data scheduler accumulates request information from all clients and decides which of the (typically numerous) competing requests should be sent to optimize the performance of the system, such as the average delay for clients to receive the data they request. The scheduler is also responsible to maintaining the request information and ensuring that requests are eventually scheduled for transmission, so that clients ultimately receive all their requested data. In our implementation, the data scheduler consists of an array of individual data-channel schedulers, each of which decides the transmissions of a single data channel as described next.

The multicast server must be conscientious of its clients' possibly varied download speeds, because clients may be connected via network links of differing throughput. For this reason, the multicast server splits its outgoing network connection (data-item transmissions) into a number of data channels, each with its own multicast group (IP) address. The server uses each data channel to disseminate data items at a particular rate (bandwidth).

Each data channel's transmissions are sent by a separate data-channel transmitter, a component of the multicast server that maintains a small queue of data packets it is about to send. When this queue runs low, the data-channel transmitter prompts the scheduler to choose which data item that data channel should send next. The scheduler makes the decision (in Figure 6.1, the arrow from data scheduler to data-channel transmitter), and the announcement channel then is given notification of the decision so that it can send an announcement of the imminent data item to all clients. The announcement (in Figure 6.1, the arrow from the multicast server's announcement-channel transmitter across the network to the multicast client) indicates the data item soon to be sent, and which data channel will send it.

A client interested in the data item just scheduled can join the multicast group for the data channel. By relying on the well-known technique of using a separate announcement channel to declare data-channel transmissions, clients can subscribe to relatively high-bandwidth data channels only when requested data appears, and minimize their download of unrequested data. To provide time for the client to join the data channel, the server imposes a delay between the time a data item is announced and the time the data channel can first start to send it.

The data channel sends the data item to its multicast group (IP address), representing all the clients interested in that item (in Figure 6.1, indicated by the arrow from a data-channel transmitter across the network to the multicast client). Because of packet-size limitations, the data item is sent as a sequence of small datagrams, each self-describing using a packet header that includes the unique identifier for the data item and an offset into the data item that represents the range of bytes in the data packet. Clients receive and save the packets for the data item, and then can disconnect from the data channel (remaining subscribed to the announcement channel) until another request item is announced. When clients receive all the items they requested, they disconnect from all of the server's multicast channels and close their unicast request connection. Clients may be subscribed to more than one data channel at a time, as their download speeds allow.

Clearly, it is possible for clients to remain connected to data channels even when they are not receiving a requested data item, eavesdropping on the other data items being sent (not knowing who or how many requested them). Similarly, it is possible for the client or server to break the unicast request connection between them as soon as the client receives the server's multicast-address information; this behavior is one way to use the multicast facility to implement a more traditional broadcast disk [1], in which the server sends data items without specific knowledge of each client's requests. Though our design allows these extensions to be implemented trivially for servers facing very large client loads—i.e., more concurrent clients than it can maintain request connections for—we will focus on basic operation without these modifications.

6.1.2 Announcement and Data Channels

In our existing research for this system (Chapter 5), we considered a variety of ways to allocate the server's bandwidth among the data channels and ways to offer the data across them. We discovered that the server should use as few channels as necessary to support its clients' range of download speeds. Also, we found that one good way to split the server's bandwidth is to provide each data channel uniform bandwidth, rather than varying the channels' bandwidths dramatically. Further, the server should partition its data items so that each data channel disseminates a smaller, distinct subset of the server's items. In effect, each data channel represents a mini-data-server, with an independent but equal fraction of the server's resources. For servers of lower loads, there is another good way to split the server's resources, which we consider in Section 6.2.

Clients in this scheme would send their requests and their download speed to the server, and subscribe to data channels carrying requested data items as their download speed allows.

6.1.3 Data Scheduler

The scheduler determines which data item to send to the multicast clients on each data channel, given information about the currently listening clients and their requests. As we observed in Chapters 2 and 3, we found that a carefully chosen scheduler can significantly improve the performance of a multicast facility to its clients.

As a default, we implemented a "null" circular-broadcast scheduler, which transmits each data item in turn as long as it is requested. This scheduler is applied to each data channel independently for its subset of the server's data items. (As described in Section 6.1.2, each data channel has its own disjoint partition of the server's data items.)

To improve performance for our clients, we strive to reduce their client delay, the average time it takes a client until it receives all the data items it requested. Recall in Chapter 2, we devise one way to do so, a data scheduler we call the R/Q heuristic. In this heuristic, the scheduler computes a score for each data item using two components: R counts how many clients are requesting (waiting for) the data item, and Q holds the size of the smallest pending client request including that data item. The R/Q heuristic divides R over Q for each item, and chooses a data item with the highest such value. This approach strives to favor data items that are much requested, and that are about to help complete a client's request, a combination that outperformed using data-item popularity alone.

The R/Q scheduler can be implemented by maintaining a heap of R and Q values for each data item and updating them only when requests affecting those particular values change. Thus, it is possible to maintain a R/Q heap whose values change only when its R or Q components change.

As with the circular-broadcast scheduler, R/Q could be applied independently to each data channel for its subset of the server's data items.

6.1.4 Transmission Reliability

The packets of data the server sends over multicast are unreliable datagrams (UDP), which may be dropped on their way to clients. To ensure the successful delivery of announcements and data, the server must cope with possible losses.

For the announcement channel, clients must receive all its 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 with high probability. Protocols, such as SRM [23], have been proposed for exactly this type of application; for simplicity, we have the server simply send the data redundantly, with the expectation that the client will receive at least one copy of each announcement before its value expires. Also, the announcement-channel transmitter must send regular "heartbeat" packets onto the announcement channel even if the server has nothing to announce; this is necessary to ensure that clients can detect if they have lost packets. For example, the presence of heartbeat packets is useful to distinguish between a quiet announcement channel with no recent notices, and a busy announcement channel whose packets to the client have all been lost.

For the data channels, we observe that the clients do not have specific time-sensitive deadlines by which they must successfully have a particular data item, which allows us more flexibility in our reliability scheme. Further, clients need receive only the data items they are requesting, a subset of the data being sent on the data channels. Conversely, each of the data channel's items can have different client request characteristics.

We seek to exploit these properties of our multicast facility to improve performance for the system's clients. To find such a reliability technique for the data channels, we turn to our study of a variety of reliability techniques for a multicast facility like this one (Chapter 4). The techniques we study include

- traditional retransmission and redundant transmission (forward error correction) techniques;
- novel twists on these traditional techniques, such as selective retransmission for small losses and variable erasure-coding redundancy for different data items within a single data channel; and
- combinations of these techniques and changes to the data scheduler to make it more loss-aware.

In our implementation, the data-channel transmitters apply variable forward error correction to the data item before breaking it into network packets for transmission. In particular, for each additional client that is requesting a data item at the time the item is sent, the data-channel transmitter adds a fixed fraction more data to the transmission in erasure-coded packets. For example, this means that a data item requested by fifteen clients gets three times the redundant packets of a data item of the same size requested by five clients.

Using erasure encoding on the data being sent allows a client to receive any subset of the packets to reconstruct the original data item, so long as the client's subset of transmitted data is at least as large as the size original data item.

Further, the code allows clients to send negative acknowledgements (NAKs) to the server through the unicast request channel to indicate additional lost packets beyond those which can be reconstructed using redundant data. Each data channel can honor NAKs from clients by requesting announcement for and appending of additional erasure-coded packets to its own transmission queue. Alternatively, the server can use the NAK to re-enter the client's request back into the data scheduler as a new request for the incomplete data item. This latter approach ensures that the server knows that the client still needs the data item, without committing any resources to its transmission immediately.

6.2 Implementation Issues

In implementing our multicast server, we encountered a number of design issues we had not previously addressed. We consider several in detail below.

6.2.1 Channel-Partitioning Schemes

In Section 6.1.2, we described our multicast server as one which partitions its outgoing bandwidth into data channels, partitioning its data into disjoint sets, one for each channel. In this section, we revisit this decision—how the server should best use its available outgoing bandwidth—and describe the options we considered in the design phase.

Data-Partitioning Server

The multicast server described in Section 6.1.2, which we call here the *data-partitioning* or *partitioning* server, starts by partitioning the bulk of the server's outgoing bandwidth into data channels equally. Similarly, the server partitions its data into as many disjoint sets as the server has data channels. The server assigns one set to each channel, so that each data channel only carries data from its assigned set.

Load-Balancing Server

Our research into bandwidth partitioning (detailed in Chapter 5 and mentioned in Section 6.1.2) suggests a method of handling clients of varying download speeds when client load is very low: If the server faces relatively few clients and small requests, split the server's bandwidth into data channels of uniform bandwidth, as in the data-partitioning server. Instead of partitioning the server's data, though, the server offers all of its data items on all of the data channels. Instead of having clients hop from data channel to data channel to receive all of their requested data items, the clients would be assigned to certain data channels when they first make their requests, and have their entire requests provided on those data channels.

To assign the clients to their data channels, the server assigns a client-load score to each data channel, and steers clients to the data channels with lowest score. To determine the load score for a data channel, the server uses the cardinality of the bag of all data items all clients on the data channel are requesting in total. Equivalently, the server multiples the number of clients already currently assigned to the data channel by the average number of data items those clients are still requesting. Because of the data-channel assignment technique, let us call this alternate approach a *client-load-balancing* or *load-balancing* server.

Example.

For an example to illustrate the partitioning and load-balancing servers, let us consider two separate multicast servers, one implementing each approach, but otherwise identical. We can see how these two servers would share six data items to the same clients.

Both the data-partitioning and client-load-balancing servers would split most of their transmission bandwidth into data channels; for this example, say each server chooses to have three data channels. The data-partitioning server then assigns two data items onto each data channel: data items a and b are assigned to the first channel, data items c and d to the second channel, leaving items e and f to the third channel. On the other hand, the client-load-balancing server would offer all six data items on each of the three channels.

Now suppose a client shows up, requesting items $\{a, c, d\}$, but has only enough bandwidth to download two data channels simultaneously. For the data-partitioning server, the client would have to download a from the first channel, c and d from the second. For the client-load-balancing server, the client would be assigned to two of the data channels depending on the load already on them. Say the first channel has clients still waiting for $\{a, e\}$ and $\{a, c, f\}$: its client-load score is 5. Say the second channel has clients still waiting for $\{c\}$ and $\{b, f\}$: its client-load score is 3. Say the third channel is idle: its client-load score is zero. In such a case, the new client would be assigned to the second and third data channels, because they have the lowest client-load scores. □

Implementation Rationale

Our choice of design for our implementation depends principally on the guidance from our research, and the simplicity of implementation. First, we mention the research; then we consider the interests of expedient implementation.

In Chapter 5, we found that the data-partitioning technique we choose for our implementation performs better at high client loads, and that the client-load-balancing technique performs better at very low client loads. Therefore, ideally we would like a server that is able to do either, depending on its client load at the time.

Our research also considered several ways to split the server's bandwidth into channels of different bandwidth, but found that those techniques did not perform better than the uniform-bandwidth data channels already described. Therefore, we do not consider those varying-bandwidth data-channel techniques further.

For our implementation, we notice that implementing only one of the techniques simplifies our implementation significantly. Obviously, such a server needs to implement only one, not both, techniques completely. More importantly, though, a server that switches seamlessly from one technique to the other would also need a way to notify clients of each switch, giving each client its channel assignments when the server moves to load balancing. The clients, in turn, must be able to receive and handle such notifications gracefully. For example, when switching to the client-load-balancing technique, the server must notify each client of its data-channel assignments so that the client will follow.

Between the two techniques, we notice that for the load-balancing technique, the server's data scheduler must maintain a client-to-channel mapping (the one the server suggested to each client individually) so that different client requests for the same data item can be mapped to the different data channels correctly. For example, clients' possible retransmission requests for lost data packets must be directed to the correct data channel. By contrast, in the data-partitioning scheme, the data-item-to-channel mapping is static for each server; it can be a fixed hash function from items to channels. There is only one channel where requests of a particular data item can go.

As a result of these implementation issues, we chose to implement only one of the two techniques described in this section, a solely data-partitioning server as described in Section 6.1.2. We consider the addition of the other technique a possible future enhancement as usage patterns and client performance dictate.

6.2.2 Announcement Channels

A server is likely to transmit over multiple data channels, so that it can accommodate clients of low download speeds (able to receive only one channel's worth of data at a time) as well as clients of higher speeds (which can receive multiple channels' transmissions simultaneously). In the case of our data-partitioning server, each data channel transmits only a disjoint subset of the server's files. Therefore, a low-download-speed client would need to receive different data channels at different times in order to accumulate its requested data from all the different channels where the requests appear.

Each of these data channels must have its transmissions announced so that clients know to subscribe as needed. To provide announcements, the server can operate as many announcement channels as a separate one for each data channel, or as few as a single one for all of its data channels. (Figure 6.1 shows a multicast server using a single announcement channel.) In this section, we will describe the operation of, then consider the tradeoffs for, these two extremes in particular: one announcement channel in total and one announcement channel per data channel.

Single Announcement Channel

To operate a single announcement channel in total, the server must hold one multicast group (IP) address for it, and transmit announcements for all the data channels on it. The server provides this announcement channel's address to clients when they connect, and clients remain subscribed to the announcement channel to receive all its announcements the entire time they are using the server.

A client in this system may filter out all but the announcements for transmissions on a small subset of data channels—no more data channels than the client is able to download simultaneously. The client would change this subset as its set of pending requested items shrinks. In effect, the client can filter out announcements to pretend that it is still subscribed to no more data channels than it can download simultaneously, thus ensuring that it cannot violate its download-speed limits.

For example, suppose a server has partitioned a client's requested data items across five data channels, but the client has only the download speed for three data channels. The client would filter out all but the announcements for three of the five data channels until it is done receiving all the requested data on one of those channels. At that time, the client would ignore announcements for the no-longer-interesting channel, and start filtering in announcements for another of the two remaining channels.

While such operation is relatively easy to model and study, in practice, the client can do better. In particular, it is easy for a client to "subscribe" to all data channels aggressively as an optimization: The client can monitor announcements for more data channels than it is able to download simultaneously, on the expectation that it will not have to subscribe to all of those data channels simultaneously to receive requested data items. If the expectation is correct, and most of the time a client is monitoring a data channel is spent waiting for a requested data item rather than downloading one, and conflicting transmissions will be rare. If the expectation is incorrect, and a client is frequently unable to fetch data items scheduled for it, then system performance may suffer.

With this optimization, our example client would no longer filter out announcements for any of the five data channels carrying requested data. If no more than three channels send a requested data item at the same time, the client can benefit from fewer missed transmissions (less waiting). If four or five channels are transmitting a requested data item at the same time, the client must choose to download only three of them to stay within its download-speed limits, and lose the transmissions of the remainder.

This optimization also provides us another benefit, by allowing us to simplify how the server detects lost transmissions. In our implementation, whenever the server transmits a data item relevant to a client, the server assumes by default that the client subscribes to the data channel to receive the item. If the client instead fails to receive the item, it must indicate so to the server. The client indicates its loss of the transmitted data item as if all the packets of the data item being skipped were lost on the network during transmission. Consequently, the server would know that the client still needs the missed item after its scheduled transmission. Because we think of clients as aggressively subscribed to all data channels, we know clients are aware of, and try to pursue, every relevant transmission. We would not have to make a special case of determining transmissions lost to clients because clients were not subscribed to the data channel at the time.

If our example client were faced with four simultaneous requested-data-item transmissions, for example, it would have to send a negative acknowledgment indicating the total loss of one of those data items. If the client failed to send the notice of data loss, the server would have assumed that the client received all four items, and the server would no longer know that it needs to schedule the lost item again for this client.

Separate Announcement Channels

Turning to operating a separate announcement channel for each data channel, we observe that each announcement channel must be assigned its own multicast group (IP) address. One way to implement multiple announcement channels is to (surjectively) map data items to data channels for transmission, and (surjectively) map data channels to announcement channels for its announcements. The server would acknowledge each client data-item request by identifying the announcement channel where announcements of that data item are ultimately assigned. Given its set of relevant announcement channels, the client joins a subset of them at a time as its download speed allows.

For example, suppose a server has six data items partitioned into three data channels, and each data channel has a corresponding announcement channel. If a client's requested data items span all three data channels but the client has only the download speed for two, the client would simultaneously subscribe to only two of the announcement channels at a time.

For each relevant announcement channel to which the client is subscribed, the client knows which data items to expect from that channel given the information the server has already provided. When the client has received all of those items for a channel, the client knows the announcement channel has become irrelevant, and disconnects from it. In its place, the client adds another relevant announcement channel until it runs out. Eventually, the client exhausts all of the relevant announcement channels, and its request is complete.

For example, suppose our example client requests all six of the server's items. It received channel information for each data item in response to each item request, so it knows that the first two data items are sent on the first data channel and announced on the first announcement channel. Consequently, when the client listens to the first and second announcement channels (tentatively ignoring the third), it will know that it can unsubscribe to the first announcement channel and subscribe to the third as soon as it has successfully received the first two data items. Similarly, the client knows which data items the second announcement channel can announce, so if it receives those items first, it can unsubscribe from the second announcement channel to pick up the third.

Design Tradeoffs

In choosing how many announcement channels to have, we consider the overhead of the announcement channels to clients, their consumption of IP address space, and how the server and client perform should the client skip some announcement channels:

If the server implements only one announcement channel in total, clients not able to subscribe to all data channels may receive announcements that are useless to them, wasting slower clients' download capacity with additional overhead.

If the server implements a separate announcement channel for each data channel, the server uses more IP multicast addresses for its service, a shared resource that is much smaller than the space of IP addresses in general. A single announcement channel requires 1 IP multicast address for C data channels (for a total of $C + 1$ addresses); separate announcement channels require C addresses for C data channels (for a total of $2C$ IP multicast addresses).

To exploit the benefit of lower-bandwidth announcement overhead using disjoint announcement channels, a client must receive only the announcements for a subset of the available data channels at a time. Unfortunately, doing so risks the client missing valuable announcements from other data channels whose announcements it is missing.

For example, if our example client has subscribed to two of three announcement channels on a server, it is missing the announcements on the third channel. While the client is waiting for a requested item to appear on the first two channels, the third data channel may happen to be scheduled to send a requested data item that the client will miss, but could have downloaded, simply because the client was on the wrong channels at the time.

It is tempting to have the server dictate the order in which the client subscribes to its data channels, to avoid the suboptimal lost transmission just described. Unfortunately, for the server to be able to suggest a channel ordering to the client effectively, the server itself must know the "right" ordering of relevant channels at the time the client makes its requests. For most data schedulers, which adjust their decisions to client loads as they appear, such knowledge is impossible because the server does not know what client loads the future brings.

On the other hand, for a system using a single announcement channel in total, the server need not make any such predictions, because the client receives all the announcements and can make its locally optimal decision at the time its requested data items are transmitted. As a result, the single-announcement-channel server needs not manage or track which subset of data channels a client is or should be monitoring; similarly, the server need not compute which data-item transmissions the client could reasonably receive at each moment in time. (If we did not allow clients to monitor all data channels at once regardless of download speeds, then this complexity would again be necessary.)

In light of these considerations, we expect a single-announcement-channel servers to consume fewer IP resources and require less foresight of its data scheduler to perform well for its clients. Therefore, we use a single announcement channel in our implementation.

6.2.3 Verifying a Successful Transmission

When a server is finished sending a data item on a data channel, it needs to know whether the clients interested in that data item received it successfully. Our implementation uses negative acknowledgments (NAKs) from client to server to indicate data loss, and silence to presume success. In this section, we revisit the ways we considered for the server to verify client receipt of a data item. Should clients send notice if they successfully receive the item, or should they send notice if they fail to do so?

In either design, to ameliorate the implosion of responses to the server (either indicating many successes or many failures), clients must wait a random delay before sending a response to spread their responses out over time. Because the client responses affect the scheduling of the data item for individual clients, the server must receive all of the responses to behave correctly. If the server fails to receive a client's notice, the server can incorrectly conclude whether the client has received a data item; if the server thinks a client has a data item when the client does not, the server can fail to schedule the data item entirely in the future, leaving the client unable to complete its request. If the server thinks a client does not have a data item when it does, the server will provide incorrect information to its data scheduler and transmit the data item again prematurely or needlessly; this scenario becomes a problem if the server draws the incorrect conclusion often, either for numerous clients or for numerous data items per client. So, to ensure that the server's image of its clients' state is remains in sync with reality, the clients send their responses using their reliable unicast connection to the server.

Acknowledgments Only (ACK)

In the case of client acknowledgments (ACKs) on success, the client would send an ACK when it has successfully received or reconstructed the data item. Until then, the client remains silent for that data item. Until the server receives an ACK from a client, the server assumes that the client does not have the data item.

Negative Acknowledgments (NAK)

In the option where a client sends negative acknowledgments (NAKs) on failure, the client would send a NAK for a data item when the item's transmission ends but the client did not receive enough data from the transmission to reconstruct the data item. The client can determine that an item's transmission has ended either because no packets for the item have been sent for some time. Because the NAK must be sent reliably, success can be indicated using silence.

Because our multicast facility uses erasure-coded packets to allow any packet from a data item to substitute for any other packet for the item, having a count of lost packets is sufficient for the server to send more erasure-coded packets. The NAK itself can therefore send information indicating the particular packets the client lost, or simply the number of packets lost. In our multicast facility, the latter form of client NAKs is sufficient for our purposes and yields smaller transmissions, so we will not consider the former version of client NAKs further.

Finally, it is possible for the client to send one ACK or NAK as appropriate at the end of every transmission. Because this option requires a client to always send some response to a transmission of a requested data item, it requires the most network transmission (and implosion of responses) of all. For this reason, we will not consider ACK+NAK further here.

Design Tradeoffs

To decide how to best indicate the client's status to the server, we wish to consider for each option its volume of network transmission (overhead), its impact on other aspects of the multicast-facility implementation, and its behavior during a network failure between client and server.

To minimize client-to-server transmissions, ACKs are preferable if success is rare, and NAKs if failure is rare.

Because send-on-success is advantageous only if clients that fail to receive their data are not also required to send a NAK, a server implementing ACKs cannot provide on-demand retransmission of the needed number of packets: the server would not receive a notice indicating the needed number. Instead, such a server would need to reschedule the entire data item for the clients that do not indicate their success. On the other hand, ACKs can be implemented easily on a client; the client need not actually know when a data item transmission ended to implement ACKs correctly. It need only know when it has successfully received an item.

Because a client might be disconnected from the server by a packet-dropping network fault long before their unicast connection indicates the disruption (even when the TCP "keepalives" option is enabled), the server might not immediately know that a client is disconnected and should no longer be serviced. Using NAKs, a client subjected to such a disruption is likely unable to send a NAK for its lost packets, so the server would not waste time trying to send retransmissions to the disconnected client. By choosing to optimistically assume success until a NAK is received, we ensure by design that the server will not needlessly attempt to handle lost transmissions for a client that silently disappears from the network. (Unfortunately, this benefit only works to avoid needless retransmission or rescheduling for a silently disconnected client; such a client may otherwise have requests still or already waiting in the scheduler, and those requests will remain in the scheduler until the network disconnection is explicitly detected.)

Given the above, we choose to use NAKs only in our implementation to avoid needless network transmission from either client or server, and retain the flexibility to implement on-demand retransmissions if needed.

6.2.4 Identifying Data Items

In typical use, the multicast server shares a set of files to requesting clients, and those files are uniquely identified by filename. In this section, we consider briefly how the multicast server might use or adapt filenames to mark its files internally and on the network protocol between server and client.

In particular, we consider three ways the multicast server can easily identify its files:

1. The server can use the name of file directly. An announcement would indicate

the file being announced by placing the filename in an announcement packet.

2. The server can sequentially number each file in some arbitrary order, then use the assigned integer to represent the file. An announcement would indicate the file being announced by placing its assigned integer in the announcement packet. A client would request a file by name from the server, and the server would reply with the assigned integer.

3. The server can hash the filename into an integer, then use a fixed number of bits from the hash to represent the file. An announcement would indicate the file being announced by placing the hash value in the announcement packet. A client would request a file by name from the server, both of which use the same function to determine its hash value.

The tradeoffs between these options become clear when we consider the constraints they impose on the files being shared, and the overhead that each option requires.

The first option effectively restricts the length of the filenames that can be shared over the multicast facility. The length restriction is required so that any multicast (UDP) transmission involving the identifier can be guaranteed to fit within packet-size limits. If the identifier were simply an unbounded-length filename itself, the length may cause UDP datagrams including it to exceed their guaranteed-supported size. Such large datagrams may be dropped by any device on the network. The result of such packet loss would be a difficult-to-diagnose problem which affects filenames exceeding certain lengths when announced or distributed over certain network routes (to certain, but not necessarily all, clients).

To avoid this problem, the multicast server can restrict the length of filenames that it is willing to share. For filenames to be included as identifiers in the headers of each data packet, they must be short, regardless of the filename lengths allowed on modern filesystems; as a result, a multicast server using filenames exclusively may be unable to share many files simply because their filenames are too long.

The last option requires a hash function that is unlikely to collide. In particular, a multicast server cannot reliably serve two files whose names have the same hash value if the server cannot otherwise distinguish the two files in their individual data packets. Hence, it is possible to have files of same length, same timestamps, different filenames, and different contents that could reside independently on the server but cannot be successfully shared.

The first option, using filenames themselves, completely sidesteps the need to maintain a separate mapping between files and identifiers that the other two options would require.

The second option, enumerating the files, requires both client and server to maintain a filename-to-integer mapping in both directions, and requires the server to communicate its integer identifier for each file to its clients, since a client cannot determine the values on its own. The mapping may not be preserved between invocations of the multicast server, so the file identifier may be neither global nor persistent, for either server or client.

The third option, hashing the filenames, is convenient for both server or client to do to get identifiers from filenames. On the other hand, this option still requires a reverse mapping from hash to filename for clients, which must successfully save received data under the correct filename. (The server may require the reverse mapping as well, to provide logging or notification of its operations using human-readable filenames rather inscrutable file identifiers.)

In our implementation, we use (the lower) 64 bits of the SHA-1 hash in the server, which we expect should make filename collisions unlikely.

6.2.5 Data Items

A data item represents the atomic unit of request and transmission for our multicast facility. In Section 6.1, the multicast server is described as offering fixed-size pages of the server's shared files as data items. In this section, we reconsider that design and consider another approach: to offer each variable-size file in its entirety as a data item.

In the former design, each page must be assigned a unique identifier. To do so, we have the server hash the filename into a (presumed unique) identifier, and append the starting file offset of the page. The result is an identifier that we expect to be unique for each page (barring filename-hash collisions). When a client requests a file, the server can enumerate all the identifiers that the client is requesting because the identifier is determined from the filename and from page offsets that increment from zero in fixed units until the end of file. When a client receives its data items, it must arrange them in page-offset order to reconstruct the original file it requested.

In the latter design, each file is a data item, so the hash of the filename itself provides a unique identifier for the data item (barring filename-hash collisions). Clients request the file directly, receive it as a single item, and save it.

Clearly, the simpler choice is to consider each file a distinct data item; for a given filename-hash function and set of files, this choice also yields shorter data-item identifiers. On the other hand, a decision to use unbounded variable-size data items does have its drawbacks.

To implement reliability, the server must ensure that each client receives every data item it requests, so the erasure code used to protect a transmission against losses is computed over an entire data item at a time. The computation of erasure codes on a data item is fastest if the data item is entirely in main memory, something that may be difficult for very large items because they may not fit. Also, the computational cost of erasure codes grows with the size of the data item being protected, so using larger data items can create additional CPU usage. If the filesize exceeds any limits of the erasure code implementation, the file may not be protected using erasure encoding at all. Lastly, with the assumption of uniform-size data items removed, the data scheduler will need to take item size into account to optimize system performance.

By breaking each shared file into small fixed-sized pages, we ensure consistently speedy erasure-code computation of each data item before its transmission, and allow clients to receive, decode, and write each page as the client successfully receives it. Using larger-sized pages (data items) would increase this computation cost, but reduce the number of data items the data scheduler must manage. In effect, varying page sizes allows us to sample the spectrum between the extremes of using small pages versus entire files. In Section 6.3.3, we consider varying page sizes in our implementation and their effect on the multicast server.

6.3 Evaluation

Description	Value
Number of files offered	22
Total size of files	20 533 768 bytes
Largest file	1 885 394 bytes
Smallest file	462 732 bytes
Number of data channels	1
Throughput permitted on data channel	10 000 bytes/sec
Expected packet-loss rate [see text]	0 %
Scheduled unit of transmission	30 000 bytes (max)
Data transmitted per data packet	500 bytes
Data-channel retransmission policy	no NAKs sent
Data-channel error-correction redundancy	0.01R (up to 1.20)

Table 6.1: Default Parameters for the Multicast Server

In this section, we examine the performance of the running multicast implementation. First, we describe parameters of the running implementation common to most of the runs that we use. Next, in Section 6.3.1, we compare our implementation to our prior simulation results. In Section 6.3.2, we attempt to find the scalability limits for our implementation. Finally, we examine the profile of the running server to determine how much time it spends on critical tasks, and determine ways to further improve its performance.

The multicast server runs on a 450 MHz Pentium II with 256 MB of RAM running FreeBSD 4.9; most of the data presented in this section has the server running on such a machine. (Some data comes from the server running on GNU/Linux machines with more hardware, but as we shall see below, the multicast server needs neither as much CPU nor RAM as the server already described provides.) The multicast clients run on a mix of FreeBSD and GNU/Linux machines of various CPU speeds and RAM sizes, either running on the same host as their multicast server or all sharing a local network with each other and their server in the same building. The network is linked with IPv4, though the code is intended to operate in IPv6 multicast networks as well. The evaluation runs use parameters summarized in Table 6.1 unless otherwise noted;

selected values from the table are explained in more detail below.

In our implementation, we impose a maximum network bandwidth that each data channel is to honor; the multicast server monitors the bandwidth each data channel consumes, throttling it as needed to prevent its transmissions from exceeding the maximum bandwidth. Because the building network linking the multicast servers and clients is in general use, we chose to use a relatively small fraction of the network's bandwidth for our data dissemination to avoid slowing down other network traffic, avoid interference from congestion with other traffic, and avoid alarm from network administrators. For the same reasons, most runs occurred at relatively quiet hours on the network (between 11pm and 5am local time). The expected packet-loss rate is zero for such low throughput over a local network, but we did see some UDP packet reordering between server and clients. If the number of clients running on a machine was very large (near 200), individual clients could also lose UDP packets before they get a chance to process them, making the effective packet-loss rate for such clients nonzero.

The files used for testing were chosen to require a measurable, but not long, time to transmit using the relatively low network throughput; as the numbers in Table 6.1 show, each file takes about 1 to 3 minutes to transmit. The actual file sizes are not round numbers because the files are not synthetic; they happen to be low-bitrate compressed music of an audio compact disc.

As we recall from Section 6.1, the multicast server actually schedules and transmits fixed-size pages of shared data files. In Table 6.1, the size of the fixed-size page is listed as "Scheduled unit of transmission." The last page of each file may be smaller than the given page size, chosen to be just large enough for the last bytes of the file.

As we recall from Section 6.1.1, there are practical limits on the size of packets that the multicast server sends. Because UDP/IPv4 only requires the ability to receive packets of 576 bytes - 20 bytes (of IPv4 header minimum) [51] - 12 bytes (of UDP header [50]) = 544 bytes, we choose a small per-packet payload of 500 bytes to ensure that the server's data-channel packets can be delivered over IPv4. [11] (A similar computation with UDP/IPv6 yields 1280 bytes - 40 bytes - 40 = 1200 bytes maximum [19]; any packet that fits in UDP/IPv4 will also fit in UDP/IPv6.) (The data channel introduces 34 bytes of packet header preceding file data, so the server's data-channel packets are 534 bytes $<$ 544 bytes.)

As currently implemented, the data channel does not use retransmission requests to transmit more data for a page. Instead, it uses forward error correction to add redundancy to a page in advance, then requires any clients suffering excessive loss to request that the page be rescheduled for future transmission. The variable forward error correction adds 1% more packets (rounded down) of redundancy per client requesting the page, so that, for example, it requires at least two clients requesting a 30,000-byte page for the server to add the 61st packet to the page's transmission. (Without redundancy, 30,000-byte pages / 500-byte packets = 60 packets per page.) Because the error-correction code traditionally requires its parameters (N, k) in advance, we set a maximum amount of redundancy we will request the error-correction code to generate. For the unmodified error-correction code used in the implementation [53], we set a maximum redundancy of 1.2 times the page size, regardless of the number of clients. This maximum setting yields 2.2 times the page size in transmission in total (1.0 of page data plus 1.2 of redundancy). Consequently, no page requires more than 132 packets to transmit given the parameters in Table 6.1.

6.3.1 Comparison to Simulation

Here, we compare the performance of our implementation to the behavior predicted by our work using simulation. Though there are significant differences in the models and parameters between simulation and implementation, we find that the resulting behaviors are substantially similar, suggesting that our simulations capture the essential behavior of the system.

One difference is that in the simulation, clients can request each scheduled data item individually and independently. Another difference is that the implementation run uses pages of files for the scheduled data items and has clients request entire files of pages at a time rather than individual pages of files. Further, the implementation includes delays imposed by the control channel to ensure that clients can subscribe to the data channel in time to receive their requested transmissions.

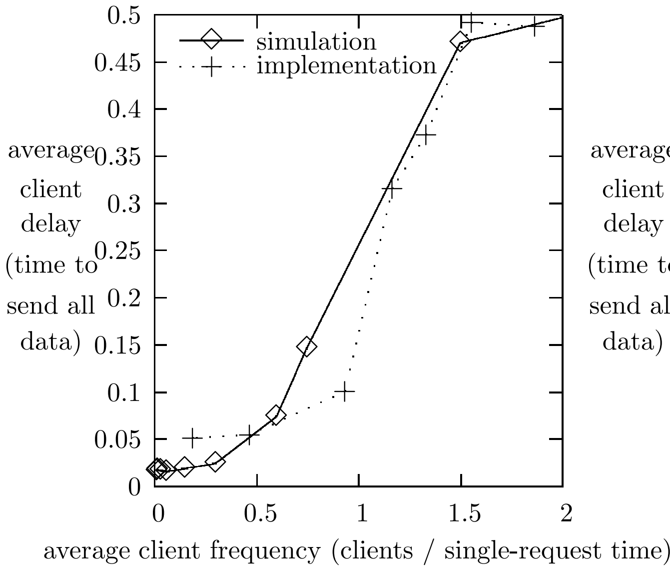

Figure 6.2: Delay versus Client Frequency

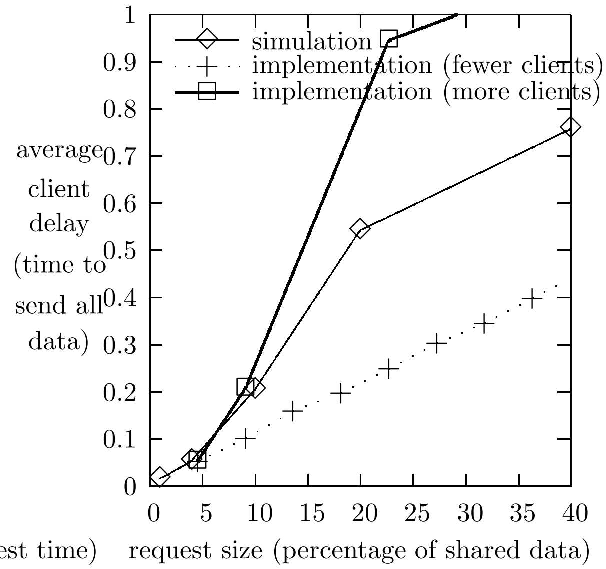

Figure 6.3: Delay versus Request Size

Despite these and other differences, we are able to compare a simulated server using a circular-broadcast scheduler with our running system which implements it. In Figures 6.2 and 6.3, we see the results of the comparison.

In Figure 6.2, we vary the rate at which clients appear in the system along the horizontal axis, and plot the average client delay on the vertical axis. Because the transmission time for an individual request is very different in the simulation and implementation runs, we compare the performance of the systems directly by plotting the horizontal axis as a function of the corresponding single-request transmission time. That is, the unit of time for the client arrival rate is the amount of time it would take the server to transmit a single request's amount of data (for the simulation, the 48 seconds to send one fixed-size data item; for the implementation, 93.3 seconds, the approximate average time to send one filesize of bytes over the network). For the remainder of this section, let us call this unit the *normalized client-arrival rate*. This normalization allows us to compare similar client loads on the system directly.

Similarly, the total size of shared data is very different between our past simulation and current implementation, so we normalize the vertical axis using the total time it would take the server to transmit all its bytes of data given its data-channel throughput. This normalization helps us compare client delay using the factors key to the behavior of the system: the time to send one piece of data and the time to send all of it. For example, along the horizontal axis, 1.5 indicates that three clients are added to the multicast server in the time it takes the server to send two data items. For the implementation in particular, three clients every two items means clients are arriving every (93.3 / 1.5 = 62.2) sixty seconds. On the vertical axis, an average client delay of 0.5 means that the client delay is about half the total time the server would require in principle to send all of its data items once, sequentially and without protocol overhead. For the implementation, which requires 20,533,768 / 10,000 = 2053 seconds to send all the bytes in its data items at 10,000 bytes per second, we see that an arrival rate of one client every 60 seconds yields a client delay of about 1010 seconds (just below half of 2053). On the figure, this is the data point at approximately (1.5,0.5).

We notice that the curves in Figure 6.2 are very similar; each is an S-shaped curve whose bottom is near the transmission time for one request ($\frac{1}{100}$ of the simulated data, $\frac{1}{22}$ of the implementation data) and whose top is near the transmission time for one half of the server's available data. The transition between these two levels occurs, as we would expect, as clients requesting one item/file begin to arrive more rapidly than the server can transmit it. Both curves approach the transmission time for half the server's data because at high load, each client will need to wait until half the server's data has been sent before its data item is scheduled for transmission (on average).

We also notice that the right half of each curve represents a region in which client requests exceed the rate that a strict queue can service; when more than one client arrives every item-transmission, the clients accumulate in an unstable queue and the client delay time for unicast transmission at the data channel's bandwidth becomes unbounded (until the server runs out of network sockets). Similarly, if a unicast server provided data to all its clients simultaneously using the same amount of network bandwidth by splitting the throughput across the clients, the server would similarly slice its bandwidth into unboundedly-many slices. As a result, our multicast server is able to provide service for client loads over a given network that would overload corresponding unicast servers—as intended for a multicast service.

In Figure 6.3, we vary the request size of new clients appearing in a system at a particular rate. The client-arrival rate for both simulation and implementation (fewer clients) is one client per 16.67 items' transmission time, or a normalized client-arrival rate of $\frac{1}{16.67} = 0.06$. (We will consider the more-clients curve below.) The request size is expressed as a percentage of the server's total data.

In the plots, we see that the system load scales in the direction we expect; as load increases, client delay grows as a fraction of time it takes the server to send all data once. Further, the growth should approach the time it takes to send all the data once, then reach a ceiling. In the higher-load implementation run and simulation, we see this behavior in the figure. In the lower-load implementation, the delay is lower and linear because of remaining differences between simulation and implementation.

In implementation, clients request more scheduled pages as fewer files than in simulation, leading to two differences. First, a client arriving every 16.67 items' transmissions is very low load for the implementation, which has much fewer files in total than the simulation has data items. Second, as a result, implementation clients can request much more data of the server before interfering with subsequent clients' requests, and we see the resulting nonoverlapping client load as client delay linear in the client's request size.

To our explanation of the differences we see between simulation and lower-load implementation, we consider a higher-load implementation that increases client load, to create overlapping client requests similar to that of the simulation. The rate for the curve labelled "implementation (more clients)" is about one client per 2.14 items' transmission time, a normalized client-arrival rate of about 0.47. With this higher client-arrival rate, client requests overlap at request sizes of about 10% of the server's data, which is much closer to the simulation in which clients overlap at request sizes of just over 15% of server data. These two curves show the same growth-towards-maximum-delay behavior in response to the overlapping client requests.

The differences between these two curves, then, are a matter of scale rather than behavior. Several factors combine to make the high-load implementation delay higher:

1. The load for the implementation curve is higher than that of the simulation curve.

2. Forward error correction in the implementation adds redundancy to each transmission as the number of waiting clients increases. As a result the server takes longer to send the same number of data items.

3. Protocol overhead for data-channel packets reduces the effectiveness of the implementation's network usage, so that an implementation's actual time to transmit all its data exceeds the computed time for any given network throughput limit for the data channel.

With the high-load implementation curve, we see the performance of the multicast implementation in regions of client load and network bandwidth that again would overload unicast delivery, and how multicast allows a server to handle such request loads in a clear and controlled way.

6.3.2 Scaling Performance

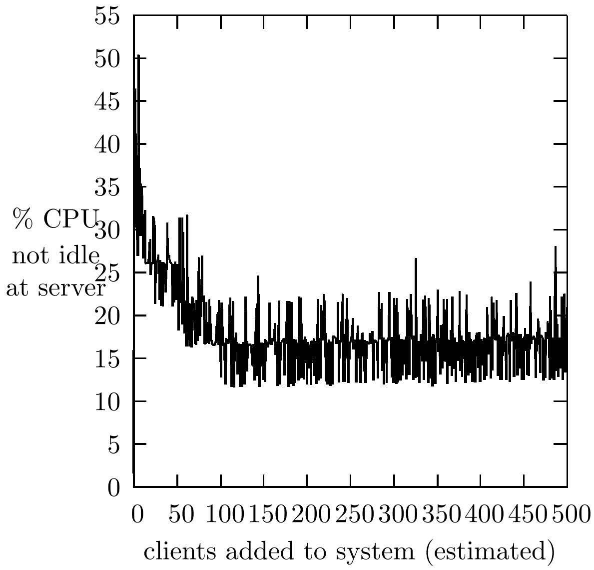

Figure 6.4: CPU usage at multicast server

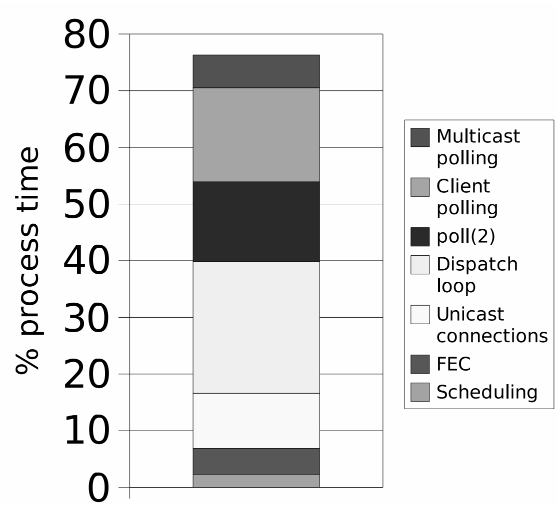

Figure 6.5: CPU consumption of various server tasks

To determine how well the multicast server scales, we connected a large number of multicast clients to the server, each requesting all files available at the server. Starting with an idle server, we added a new client to the system every twenty seconds and measured all CPU use on the multicast server machine. To measure all CPU use, we ran iostat(1) and subtracted the idle CPU percentage from 100%; this includes all CPU usage directly and indirectly required by the multicast server, and a small amount to record the CPU usage itself. Each client requests all of the files the server has available. We kept adding clients in an attempt to find the multicast server's scalability limit—how the server fails, and what resources the server exhausts first. Unfortunately, we did not hit any limits in the server even after adding over 1000 clients, by which point we had run out of CPU and disk-access time for the clients scattered over several machines (each 2.8 GHz Pentium 4).

In Figure 6.4, we plot the CPU usage over time, marked on the horizontal axis as the number of new clients added to the system. Because the client requests are large, clients accumulate in the system over time; the client delay time to receive all data exceeds 2053 seconds, while clients are added every 20 seconds. As we can see, the multicast server's CPU usage of the 450 MHz Pentium II is low overall. Further, the CPU usage actually drops as we first add clients into the system. This surprising behavior occurs because new clients requesting each transmission increases the error-correcting redundancy added to the transmission. As the redundancy grows, the time spent on each transmission grows, so scheduling and file reading occur less frequently.

Finally, when we have added 120 clients to the system, we reach the limit of redundancy that forward error correction adds to each transmission. After this point, we see the effect of increased client load on the server as a slow increase in CPU usage. Though subtle as plotted in the figure, the increase, about 1% additional CPU usage per 400 clients, continues visibly over the addition of over 1200 clients to the system until the client machines are unable to keep all their clients running. (As the number of clients increase on each client machine, the machine becomes unable to deliver key control-channel packets to all of its clients reliably, effectively disconnecting them from the multicast.)

As the number of clients and client requests grow, the data structures in the server that track the clients and requests get larger and take more time to process. At the slope plotted, though, it would take tens of thousands of simultaneous clients in the system for the server to use all of a 450 MHz Pentium II.

In Figure 6.5, we see some results of a separate profiling run of the multicast server under load. We manually gathered the CPU usage attributed to various portions of the server code into related groups we call tasks. With the resulting summary, we can see how much CPU is consumed performing various key tasks for the multicast scheduler, and hint how the server implementation could be improved. In this profiling run, we attempted to add over 1500 clients to an idle server, several per second, to build a high client load quickly for profiling.

The percentage of total server-process time spent on each of the listed tasks is shown; the total is not 100% because numerous minor tasks were omitted for brevity.

We start with a brief tour of the implementation of the multicast server's operation. Then, we can see how the server's CPU time is consumed in its key tasks.

The multicast server implementation uses a traditional event loop:

1. Request notification when an open network connection is ready for (or with) more data, or when a given timeout lapses. Each network connection is a multicast channel, a client control channel, or the listening socket that receives new client connections.

2. Check which connections are ready, and dispatch the handler for each channel ready for activity.

3. Call an idle function for each connection, so that code managing each connection has an opportunity to run even if it has no network activity. Currently, data channels use this opportunity to track their bandwidth usage over time to determine when they can send a new data packet without exceeding their prescribed bandwidth limit.

4. Check for any quitting or disconnected clients. Clean up connection for departing or departed clients.

The individual channel handlers (invoked as part of step 2) are responsible for the activity that drives the server's transmissions:

a. When a data channel handler notices that it is about to run out of data to send, it requests that a new page be scheduled.

b. The scheduler chooses a new page on demand. The control channel is asked to announce the page when it is able to do so.

c. When the control channel announces the page, it notifies the data channel when the data channel can begin sending the new page.

We find from our profiling data that the CPU-consuming tasks of the server are

- **Multicast polling.** (invoked in step 3.) At regular time intervals, code for each multicast channel computes the channel's current bandwidth to determine whether a data channel can send data without exceeding its bandwidth limit.

- **Client polling.** (invoked in step 3.) At regular time intervals, code for each client connection is run so that it can ping its client if needed to verify the client's continued connection. Because the code currently does not implement the client ping, the code is a no-op and the server actually relies on TCP keepalives to verify a client connection.

  TCP keepalive timeouts are typically set in the kernel to time out broken connections after a fairly long time (from many minutes to hours). To detect disconnected clients sooner (e.g., to unschedule their requests), client polling must be modified to perform its own ping.

- **poll(2).** (step 1.) This measure is the CPU time directly attributed to the library call and kernel service poll(). This measure does not include the time to set up the parameters of the call, or the time to scan its results.

  The poll call, like select(2), requests that the operating system return when any of the provided file descriptors (for the multicast server, network connections) are ready for more data. Unlike select, poll does not have a compile-time limit on the number of connections the kernel can track. The call to poll includes an array linear in size in the number of file descriptors to be monitored, which the system must scan as input and set as output.

- **Dispatch loop.** (steps 2–4.) The dispatch loop is the event-loop code (described above) surrounding but not including the poll call, i.e. the overhead for steps 2–4.

- **Unicast connections.** (invoked in step 2.) The server must receive and parse client requests, and send individual responses to each client. For example, the server must provide each client the address of the control channel so that a client can subscribe to the multicast channel. "File not found" errors are also sent individually over the unicast TCP connection.

- **FEC.** (invoked in step 2.) This measure represents all time attributable to forward error correction. The measure includes setup time for the facility, and encoding time for each packet. The setup occurs at the time a page is scheduled (step (2)b.), and the encoding occurs for each packet at the time that packet can be sent (invoked in step 2.).

- **Scheduling.** (step (2)b.) The server must determine, as needed, which of its requested data to send next.

As we can see, the server suffers principally from the enumeration of its clients. Both "client polling" and "dispatch loop" refer to operations that are dominated by simple operations on every one of a large number of clients. For client polling, the simple operation is currently a no-op, to be expanded to perform client pings or to be removed to save additional CPU time. For the dispatch loop, the simple operation is checking whether poll() indicates new activity on the corresponding unicast client connection. Similarly, the poll() call itself is dominated by the number of clients, because there are many more client connections than multicast channels in the parameters.

On the other hand, we find that some key computation for the server, data scheduling and FEC computation, take relatively little time overall. As the server code is optimized, these two operations would take a larger fraction of the total server time, but currently they represent only about 2% and 5% of the server's CPU time respectively.

Lastly, we notice opportunity for performance optimizations in the multicast server. As mentioned above, we can omit the code flow for client-connection pings. We can also merge the last two steps of the event loop described above to minimize the number of traversals through the data structure storing all the client-connection handlers. Based on their current CPU usage, we find from our profiling data that these two improvements can remove about 16% of the CPU time shown in Figure 6.5.

Surprisingly, we also find that about 3% of the process time is spent in gettimeofday(2), the operating system service for the current time of day. As such, the server spends slightly more CPU time retrieving the current wall clock on demand than the server spends scheduling data, because the current time is very frequently used to measure bandwidth and detect timeouts. A small CPU saving is possible if the server caches its retrieved time for use throughout an entire iteration of the event loop, rather than retrieving the most up-to-date time every time it is needed. (The wall-clock time is requested at 22 points in the server code, several of them occurring inside small loops.)

6.3.3 Varying Page Size

In this section, we consider how the multicast server performs under a variety of maximum page sizes. We vary the default 30,000-byte page size over a range from 1000 bytes to 58,000 bytes (a limit imposed by error correction as used), and find that large page sizes lead to better performance than small ones. (Recall that the server schedules, announces, and transmits pages of a fixed maximum size, except at the end of a file where the remaining bytes may form an even smaller page. For example, smaller-than-one-maximum-page files may be sent as a single smaller page.) Using our implementation, we are able to measure the actual CPU usage and memory usage of our server under various page sizes, something that could not be realistically predicted with an abstracted simulation.

In Figures 6.6 and 6.7, we see two aspects of the multicast server's performance as we vary the system's page size, the size of the unit of scheduling and transmission. In both figures, we vary the page size along the horizontal axis. In Figure 6.6, we plot

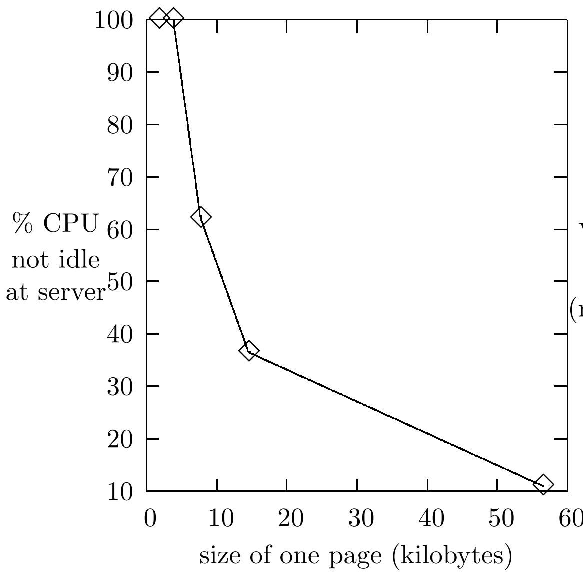

Figure 6.6: server CPU usage and page size

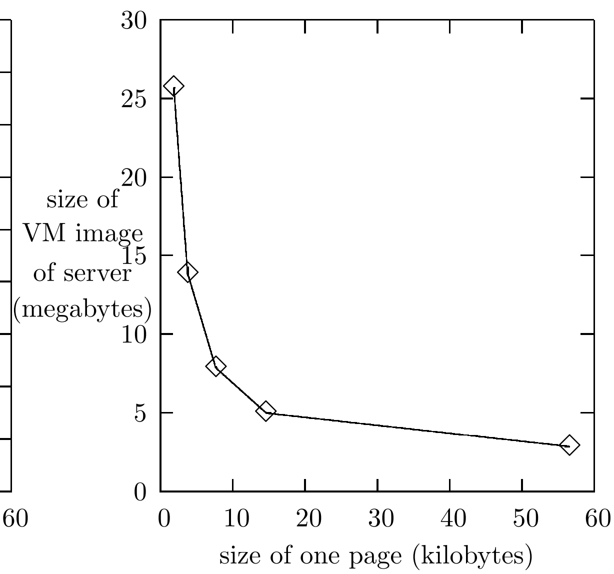

Figure 6.7: server virtual-memory usage and page size

the corresponding CPU usage of the multicast server (measured as non-idle CPU time). In Figure 6.7, we plot the corresponding virtual-memory image size of the server process. Because the code at each data point is the same except for the integer representing the page size, differences in the virtual-memory image help profile the memory consumption of data structures in the server. To provide some load to the system, we add 100 clients to the idle server, each requesting all available data.

For example, if we have the multicast server use 15,000-byte pages, we see that it would use about 36% of a 450 MHz Pentium II and less than two megabytes of memory to provide service to 100 simultaneous clients. (For a single client, the memory usage ranges from below two megabytes to seven megabytes.)

We see that the dominant effect of page size on the server is to reduce the number of pages to be handled in the server. As we observed in the previous section, the CPU cost of FEC encoding and data scheduling are small, so they has similarly small effect on CPU usage here. Instead, we benefit in computation time and memory usage from using fewer larger pages, which means fewer pages' requests to track and smaller data structures as a result. Further, this benefit also extends to the control channel, which has fewer pages to announce per unit time at a given data-channel bandwidth usage. Consequently, the control channel bandwidth consumption also goes down as the page size increases. We observe that, for a data channel using an IPv4 multicast address (as opposed to a larger IPv6 multicast address), the control channel bandwidth can drop from up to 2700 bytes/second (for the worst-case 1000-byte pages—i.e., two data-channel packets per page) less than 100 bytes/second (for page sizes over 15,000 bytes) in announcements.

In light of these results, we conclude that it is desirable for the server to have large page sizes, as far as the forward error correction allows. Because the FEC implementation we use has a maximum number of packets it can generate, increasing the page size (up to that limit) reduces the maximum number of redundant packets we can create and transmit. Therefore, we expect the effective page-size limit in practice will be imposed by the expected data popularity and expected network-loss rates for clients. Also, though larger page sizes clearly benefit the server, they may affect clients at run-time by requiring them to successfully receive and decode larger units of data at a time. (The current implementation is able to handle, without modification, servers using page sizes that are a multiple of 500 bytes, i.e., pages that are an integer number of data-channel packets.)

6.4 Conclusion

In this chapter, we considered a number of design and implementation issues that affect our prototype. We tested our implementation over a local IP multicast network to validate our results and measured the performance of our working system. We found that our implementation validates the simulation results we used to guide the design and implementation of this multicast facility, and that our implementation is able to scalably disseminate data using very little bandwidth and CPU power to thousands of concurrent clients.

Chapter 7

Conclusion

Large-scale unicast data dissemination is network intensive because it requires a server to send many copies of requested data items, once to each client that requests them. To use the server's network connection more efficiently, we designed and implemented a multicast data facility that is able to exploit the properties of unreliable, one-to-many multicast transmission. As we have seen in this dissertation, such a facility requires study of many questions.

In Chapters 2 and 3, we considered the design of the multicast server's data scheduler, which decides how to order the transmission of the server's requested data items in order to optimize client performance. In Chapter 1, we noted how clients often care about the performance metric we call client delay (unlike the metrics studied in much existing multicast dissemination and broadcast disk work); so, in Chapter 2, we studied how a server's data scheduler impacts the client delay of the resulting multicast system. We also considered how performance is affected by varying client request overlap, having clients wait to form batches, and other factors. In Chapter 3, we considered subscriber clients that are interested in maintaining high freshness or low age in their collections. After uncovering guidelines for schedulers to improve performance for subscribers using these metrics, we studied the performance of subscribers and one-time downloaders sharing the same multicast system.

In Chapters 4 and 5, we studied network issues for the multicast facility. In Chapter 4, we considered how to adapt traditional reliability techniques to exploit the properties of our multicast system, and in so doing provide a data transmission reliability guarantee with improved performance. We also considered the potential benefit of a server that exploits knowledge of its clients' loss rates in the data scheduler, and of notifying new clients of a data item transmission already in progress. In Chapter 5, we considered how to provide service for clients of different network speeds by dividing the multicast server's data and network bandwidth into multiple data channels. We studied how performance is affected by the number of channels the server uses; the way the server divides its bandwidth, its data, and its clients among the channels; and the cost of channel bandwidth. We also studied for comparison a server that knows its clients' connection speeds in advance and sets its channel bandwidth to them.

Lastly, in Chapter 6, we described how we implemented a multicast data-dissemination facility. After considering the tradeoffs of several implementation issues, from the number of announcement channels to how the server breaks files into atomically scheduled data items, we validated the results of our simulation study, instrumented the server's performance for an increasing number of clients, and profiled its CPU usage for key tasks including scheduling, erasure coding, and network operations.

7.1 Future Work

Despite our work so far in the design and implementation of a multicast data dissemination system, significant challenges remain to improve its performance and usability. These challenges range from implementation issues to functionality and performance issues. This section mentions several examples, including run-time improvements to bandwidth management, increased data scheduler integration, and generalizations of the multicast facility considered here.

7.1.1 Run-Time Improvements

The multicast facility described in Chapter 5 is able to create a set of data channels and disseminate data over them at a fixed bandwidth for each channel. As we have seen in Chapters 5 and 6, this approach allows a relatively straightforward implementation with good performance characteristics. This design helps a server guarantee that it is always able to service a client of a minimum supported bandwidth, even though the server may not know when such a client may appear.

One improvement to the system would be for the server to receive information about each client's estimated download bandwidth, and use it to determine the actual minimum client bandwidth from moment to moment. With this information, the server can adjust its bandwidth allocation to try to improve client performance.

For example, the server may decide to stop assigning new clients to several eq-bw-Q channels, and when those channels fall idle, increase the bandwidth of all remaining channels. When a low-bandwidth client appears, the bandwidths of all channels can be reduced to their original values without harm. The effectiveness of this example approach would depend on a tradeoff between the delay degradation when clients are assigned to fewer channels, and the improvement when those channels' bandwidth is increased.

7.1.2 Data Scheduler Integration

Our results for the data scheduler are independent of the other issues in the design of a multicast facility. Further improvements in the scheduler may be possible if the scheduler is integrated more tightly with the rest of the system. For example, each data channel is expected to run its own data scheduler to handle the dissemination of that data channel's items. Not yet considered is the possibility that the different channels' schedulers can cooperate. Cooperation can provide an explicit mechanism to reduce or eliminate the chance that a client will have more requested data items transmitted simultaneously than the client is able to receive at once. Alternatively, cooperating schedulers may judiciously force some clients to skip data transmissions in this way in order to improve client performance for others. A joint scheduler for all the channels might also better track the entirely of a clients request, rather than the subset of a client's request on a particular data channel, when weighing each client's remaining request size or time to completion.

7.1.3 Exploiting Data Client Request Patterns

A multicast server receives and processes explicit requests for data from its many clients in order to provide the information they need. One consequence of this model, which we have not considered here, is that the server accumulates information about client requests that it can exploit. For example, a server applying data mining techniques to the client requests, treating each client's requests as a "market basket," can find correlations in data-item requests that are not obvious at first glance, or automate the enumeration of multiple such correlations.

This additional information can be used to improve service for clients in various ways. For example, the server can use the information to determine how to reorganize its data items into atomically scheduled chunks. Also, the server can attempt to improve performance by favoring the dissemination of correlated items near each other in time, when the scheduler finds multiple items to be of comparable priority under the items-requested-independently assumption.

7.1.4 Different Perspectives of Data Clients

We made certain assumptions about our clients in the design of this multicast facility, which lead to the metrics that we consider. As we observed in Chapter 2, the choice of metric greatly affects our choice of data scheduler, a key component of the multicast facility. Just as this dissertation extended and generalized broadcast disks work in considering a more general class of clients, multicast work can be generalized again by considering more types of data clients.

For example, instead of client delay, our request-to-finish time for a downloading client, one can consider other relevant metrics, applicable to different kinds of clients. A normalized client delay, sometimes called stretch, measures the client delay divided by the client's request size. Though the stretch metric is not considered here, it may be particularly relevant for clients that are able to postprocess their requested data as it arrives, regardless of order.

Also, clients may have their delays weighted by some measure of precedence. In this scenario, clients with more heavily weighted delays are more "important" than clients of low weight, and an appropriate data scheduler would favor minimizing delay of important clients over the delay of unimportant ones. Such a scenario may arise for a data server with VIP clients, with clients paying different amounts for access, or with clients being profiled by their requests ("I want to favor clients who are requesting the data I am promoting;" "I want to favor clients with high bandwidth.").

More generally, "opportunistic" clients may be willing to terminate their requests prematurely depending on data availability. Such a client requests all the data it *wants* but more data than it *needs*, and tries to balance its total delay with the amount of additional wanted data it receives. For example, a client may need archives of a particular strongly-connected subset of the Web at minimum to conduct an experiment, and therefore it waits as long as necessary to receive this subset. ("I need a data sample.") It may want more Web data outside this subset, but because this additional useful data is not required, the client is willing to disconnect rather than wait once it has the required data. ("I want to reduce experimental noise with more samples, until they start taking too long to arrive.") In this case, a new performance metric for the multicast facility must measure both client delay and a (possibly weighted) fraction of desired data actually sent to the client. (Each requested data item may be weighted by its utility to the client.)

Also, order-sensitive clients may require their data to arrive in a particular order. Such a client requests a set of data items to be distributed consistent with a given partial (or full) ordering. For example, a client that is unable to store all the data it is requesting for postprocessing may need to specify an ordering for the data in order to effectively postprocess the data as it arrives. In this scenario, a data item sent out of order for a client would be ignored by that client, requiring the item to be sent later.

Finally, a multicast server may need to consider clients of all these possibilities, just as Chapter 3 considered a multicast facility that services both client-delay downloaders and subscribers. A data scheduler that is able to balance the needs of many competing clients with many different metrics can further generalize the scheduler work presented here.

7.1.5 Distributed Multicast Data Dissemination

Though the multicast data facility is presented as a way for a single server to distribute data more widely and efficiently, multicast dissemination is not restricted to a single data source. One promising approach to data dissemination is the *distributed* distribution of data using multicast. This approach is particularly important, for example, when single-source data dissemination becomes impractical.

- For large numbers of channels and small data items, the server's performance may be limited by the seek rate of the disk from which it fetches data to fill each of its data channels. For example, if each data channel schedules a new data item every second, and an average disk seek takes 10ms, then the server can reliably serve up to 100 channels in total, regardless of its computational power and available network bandwidth.

- Large collections of related data may be stored on separate sites, and may be difficult to gather into one place for a central server. For example, Stanford WebBase Web-crawl data is (as of this writing) available in part from the Stanford University campus and in part from the San Diego Supercomputer Center (SDSC). The size of this crawl data makes it difficult for us to hold all the data at Stanford, but the crawls occur at Stanford, so the most recent data is usually available at Stanford first and migrates to SDSC.

- Clients may have more bandwidth than is available to a particular server, but less than the bandwidth available to multiple cooperating servers. That is, a single server's outgoing bandwidth may be less than the download bandwidth of a particular client, but two servers' outgoing bandwidth may be more than sufficient.

Creating a distributed multicast data facility requires many design decisions to be studied. For example, should the coordinated servers each control separate data channels, or should each server share data items on the same set of channels? Should the servers load-balance clients by distributing the clients among the servers, or should the clients contact each server and request subsets of data items from each one?

A distributed multicast facility requires many past considerations to be revisited. An effective data scheduler, for example, may need to be distributed across multiple servers, coordinating their decisions. One version of a particular data item may be available at a particular server, but the updated item may become available on a different one, requiring a different model for subscribers to keep their data up-to-date than described in Chapter 3. If the data items in the distributed facility can migrate or be copied from server to server over time, the multicast system's model, design and protocol must accommodate the migrations without fault.

With careful design, such a distributed multicast facility can provide data service that is scalable and flexible in ways beyond that of a solitary multicast data server.

Though multicast has been proposed for the Internet since 1985 [17], network-level multicasting is currently a service available to only a minority of Internet users; as of April 2001, Multicast Technologies, Inc., estimated that "the penetration of multicast into the commodity Internet seems to be between 3 and 5 per cent."[45] Such availability seems rather low for a service more than 15 years after its proposal, but it turns out that designing and implementing a deployable multicast, like the one we assume in the network in Chapter 1, required a great deal of work: For example, PIM-SM [20], a multicast-packet routing protocol being developed and deployed for inter-domain networking now, bears little resemblance to the early approaches from the time multicast over IP was first conceived (cf. DVMRP [63]). Continuing work (such as source-specific multicast [8]) may again change how packets are multicast on the Internet in the years to come.

Because of the continuing work on multicast design and (now-pervasive) implementation, technical deployment is now considered feasible enough to be a mandatory part of the Internet Protocol, version 6 (IPv6) [18]. Indeed, Perey and Eubanks report that the challenges that remain to pervasive deployment of multicast are no longer technical: "From extensive talks with ISP operators, the consensus is that [most Internet service providers do not offer multicast to end users] is due to a lack of content, not due to a lack of technology."[48]

So, the research and implementation challenges to working, useful multicast are being addressed. As a result, Whether network users can enjoy multicast services (such as the scalable data dissemination described in this dissertation) will depend increasingly on advances for multicast economically and politically. Indeed, the inertia to leave a network untouched may be difficult to overcome. If the inertia can be overcome, though, to make multicast networking ubiquitous, the benefits of multicast—faster, more scalable delivery at much lower network cost—are tremendous.

Appendix A

Scheduling for a Special "Partitionable" Case

Under a fairly restrictive special case, one can efficiently generate a schedule to minimize client delay. The special case is this: If all clients of a multicast server appear at the same time, and are *perfectly partitionable*, we can arrange the data items requested in an order that produces the provably minimum client delay.

A set of clients and their data item requests are *perfectly partitionable* if and only if clients and requested data items can be placed in partitions that satisfy the following conditions:

- each data item requested by some client belongs to exactly one partition
- each client making a data request belongs to exactly one partition
- each client in a partition makes a request of *exactly all* the data chunks in the same partition

More formally, we define a perfect partition thus:

For a set of clients $\mathcal{C}$ and a set of data items $\mathcal{D}$, a *perfect partition* is a mapping $\mathcal{P} : \mathcal{C} \cup \mathcal{D} \mapsto \mathbb{Z}$ such that $\forall$ clients $c \in \mathcal{C}$, $\forall$ data items $d \in \mathcal{D}$, c requests $d \Leftrightarrow \mathcal{P}(c) = \mathcal{P}(d)$.

Without loss of generality, we will suppose that any unrequested data item is mapped to zero, and that $\forall i > 1, \exists d \in \mathcal{D}(\mathcal{P}(d) = i) \Rightarrow (\exists d' \mathcal{P}(d') = i-1)$. Intuitively, we are simply taking up the slack in the range of the mapping, which allows us to enumerate our partitions from 1 to P (where P is the largest natural number for which $\exists d \mathcal{P}(d) = P$).

For perfectly partitionable collections of client requests, we can define

$$\begin{aligned}
\forall 1 \leq i \leq P, P_i &= \{\text{ client } c | \mathcal{P}(c) = i\} \cup \{\text{ data item } d | \mathcal{P}(d) = i\} \\
|\mathcal{C}_i| &= |\{\text{ client } c | \mathcal{P}(c) = i\}| \\
|\mathcal{D}_i| &= |\{\text{ data item } d | \mathcal{P}(d) = i\}|
\end{aligned}$$

Example. The following collection of client requests is not perfectly partitionable:
client A requests $\{a,\ b,\ c\}$
client B requests $\{a,\ b\}$

Proof by contradiction: Suppose $\mathcal{P}$ exists. Then $\mathcal{P}(A) = \mathcal{P}(a)$ and $\mathcal{P}(A) = \mathcal{P}(c)$, therefore $\mathcal{P}(a) = \mathcal{P}(c)$. But, $\mathcal{P}(B) = \mathcal{P}(a)$, so therefore $\mathcal{P}(B) = \mathcal{P}(a) = \mathcal{P}(c)$. This is a contradiction, since client B does not request data item c as $\mathcal{P}(B) = \mathcal{P}(c)$ would require. $\square$

Example. The following collection of client requests is perfectly partitionable:
client A requests $\{a\}$
client B requests $\{b,\ c,\ d\}$
client C requests $\{b,\ c,\ d\}$

One perfect partition is simply $\mathcal{P}$, defined by $\mathcal{P}(A) = \mathcal{P}(a) = 1, \mathcal{P}(B) = \mathcal{P}(C) = \mathcal{P}(b) = \mathcal{P}(c) = \mathcal{P}(d) = 2$. The other is the "same" partition enumerated in the other order. In the future, we will simply choose one perfect partitioning arbitrarily and say

$$\begin{aligned}
P_1 &= \{A, a\}, |\mathcal{C}_1| = 1, |\mathcal{D}_1| = 1 \\
P_2 &= \{B, C, b, c, d\}, |\mathcal{C}_2| = 2, |\mathcal{D}_2| = 3 \quad \square
\end{aligned}$$

To arrange the data items into a schedule with minimum client delay, we first find the partitions as defined above. Then, we sort the partitions in decreasing order of the ratio ($|\mathcal{C}_i|/|\mathcal{D}_i|$). Finally, we take each partition in order, one at a time, and send all of that partition's data items in arbitrary order. (Since each data item in a partition is requested by the same clients, it clearly no longer matters in which order the data items of the partition are sent.)

This algorithm always determines a schedule with minimum client delay because of two reasons: The first is that there is always an optimum (minimum delay) server schedule that sends each partition's data without interruption. The second is that the sorting of the partitions defined above provides the optimum arrangement of partitions. Therefore, the resulting schedule of data from the process above must be an optimum server schedule. (There is technically a third point, that the ordering of data within a partition does not affect client delay, but this is presumed obvious enough not to address.) Each point can be considered in turn.

The first claim is that there is always an optimum server schedule that does not "split" partitions, sending some but not all data items of one partition before sending data items from another partition.

Intuitively, this claim is because if we had a schedule that sends some data items of one partition without sending all of them, we could "unsplit" that split partition to get a new server schedule that incurs no more client delay than before; we do this by taking the data from the partition that was interrupted by another partition's data, and schedule the interrupting data before the data from the split partition. This process of "unsplitting" partitions could be repeated (each time leaving one less split in the schedule than before) until no partition splits remain. Because the total client delay does not go up with each "unsplitting," an optimum schedule could be unsplit this way and remain an optimum schedule.

We can express this more formally:

For a perfectly partitionable collection of client requests in which all clients appear at the same time, we can describe a server's schedule of the requested data items (partitions) as a finite sequence $S = \langle s_1, s_2, s_3, \ldots, s_{\dim(S)} \rangle$ of length $\dim(S)$, the number of data items requested. At time t, the server sends a previously unsent data item from partition P_{s_t}. Since every requested data item needs to be sent once to satisfy the requests for it, clearly for $1 \leq i \leq \dim(S)$, i appears in S exactly $|\mathcal{D}_i|$ times, once for each of the $|\mathcal{D}_i|$ data items in partition P_i.

A partition P_i is *split* at t_2 in S if $\exists t_1 < t_2 < t_3$ such that $S_{t_1} = S_{t_3} = i$ and $S_{t_2} \neq i$, and there is no smaller t_2 for which this is true.

We can "unsplit" S's split at t_2 by finding the smallest t_3 for which the above expression is true, and doing the following:

Let $S_0 = \langle s_1, s_2, \ldots, s_{t_0} \rangle$, the longest prefix of S in which the scheduled partitions are not split. Let $S_1 = \langle s_{t_0+1}, s_{t_0+2}, \ldots, s_{t_2-1} \rangle$, $S_2 = \langle s_{t_2}, s_{t_2+1}, \ldots, s_{t_3-1} \rangle$, and $S_3 = \langle s_{t_3}, s_{t_3+1}, \ldots, s_{\dim(S)} \rangle$, so that $S = S_0$ concat S_1 concat S_2 concat S_3.

Replace S with $S' = S_0$ concat S_2 concat S_1 concat S_3.

Claim. S' incurs no more client delay than S.
Only the data items scheduled in S_1 are delayed by S'; all other data items are scheduled at least as early in S' as they are in S, and so cannot contribute to additional client delay. This means only the partition scheduled in S_1 needs to be considered.

The data items in S_1 are of the partition that is split first in S, and this partition's data items resume transmission in S_3. Further, S_3 begins at the same time in S' as in S. Therefore, the client delay for the partition scheduled in S_1 has not changed either. $\square$

Because this unsplit operation modifies S to remove the earliest split (all splits in S_2 and S_3 are preserved and the one at t_2 is removed), and there are a finite number of splits in S, this operation can be repeated until all the splits are removed and would necessarily terminate. This would yield an alternative schedule that has no more client delay than the schedule from which it is derived, so if there exists a schedule with splits that has minimum client delay, then this process will create a schedule with minimum client delay that has no splits.

The second claim is that, for server schedules that do not split partitions (so in effect, for server schedules of *partitions*), an optimum ordering of partitions must have ratios ($|\mathcal{C}_i|$ / $|\mathcal{D}_i|$) in nonincreasing order. Here is the proof of this second claim.

Claim. An ordering of partitions with minimal client delay $(P_{i_1}, P_{i_2}, \ldots, P_{i_P})$ must satisfy

$$\forall 1 < i \leq P, \frac{|\mathcal{C}_{i-1}|}{|\mathcal{D}_{i-1}|} \geq \frac{|\mathcal{C}_i|}{|\mathcal{D}_i|}$$

Proof by contradiction. If not, there exists a schedule of partitions $S = (P_{i_1}, P_{i_2}, \ldots, P_{i_P})$ with minimum client delay for which $\exists k \frac{|\mathcal{C}_{k-1}|}{|\mathcal{D}_{k-1}|} < \frac{|\mathcal{C}_k|}{|\mathcal{D}_k|}$.

The client delay of S, D_S is

$$\begin{aligned}
\text{Delay of } S = D_S = \sum_{i=1}^{P} \sum_{j=i}^{P} |\mathcal{C}_j||\mathcal{D}_i| \quad &= \quad \sum_{i=1}^{k-2} |\mathcal{D}_i| \left[\sum_{j=1}^{k-2} |\mathcal{C}_j| + |\mathcal{C}_{k-1}| + |\mathcal{C}_k| + \sum_{j=k+1}^{P} |\mathcal{C}_j| \right] \\
&+ |\mathcal{D}_{k-1}| \left[|\mathcal{C}_{k-1}| + |\mathcal{C}_k| + \sum_{j=k+1}^{P} |\mathcal{C}_j| \right] \\
&+ |\mathcal{D}_k| \left[|\mathcal{C}_k| + \sum_{j=k+1}^{P} |\mathcal{C}_j| \right] \\
&+ \sum_{i=k+1}^{P} |\mathcal{D}_i| \left[\sum_{j=i}^{P} |\mathcal{C}_j| \right]
\end{aligned}$$

The double summation by which D_S is defined comes from the observation that when partition i of $|\mathcal{D}_i|$ data items is being transmitted over the multicast tree, exactly the clients in partitions i through P are still waiting for their requests (and therefore accumulating client delay). The double summation is then expanded for detail.

If we switched S_{k-1} and S_k to form a revised schedule S', we can compute its (different) total client delay $D_{S'}$ the same way:

$$\begin{aligned}
\text{Delay of } S' = D_{S'} \quad &= \quad \sum_{i=1}^{k-2} |\mathcal{D}_i| \left[\sum_{j=1}^{k-2} |\mathcal{C}_j| + |\mathcal{C}_k| + |\mathcal{C}_{k-1}| + \sum_{j=k+1}^{P} |\mathcal{C}_j| \right] \\
&\quad + |\mathcal{D}_k| \left[|\mathcal{C}_k| + |\mathcal{C}_{k-1}| + \sum_{j=k+1}^{P} |\mathcal{C}_j| \right] \\
&\quad + |\mathcal{D}_{k-1}| \left[|\mathcal{C}_{k-1}| + \sum_{j=k+1}^{P} |\mathcal{C}_j| \right] \\
&\quad + \sum_{i=k+1}^{P} |\mathcal{D}_i| \left[\sum_{j=i}^{P} |\mathcal{C}_j| \right]
\end{aligned}$$

It turns out that D_S and $D_{S'}$ have many terms in common which cancel when we examine their difference:

$$\begin{aligned}
D_S - D_{S'} \quad &= \quad (|\mathcal{D}_{k-1}| \left[|\mathcal{C}_{k-1}| + |\mathcal{C}_k|\right] + |\mathcal{D}_k| \left[|\mathcal{C}_k|\right]) \\
&\quad -(|\mathcal{D}_k| \left[|\mathcal{C}_k| + |\mathcal{C}_{k-1}|\right] + |\mathcal{D}_{k-1}| \left[|\mathcal{C}_{k-1}|\right]) \\
&= \quad |\mathcal{D}_{k-1}||\mathcal{C}_k| - |\mathcal{D}_k||\mathcal{C}_{k-1}|
\end{aligned}$$

To prove our claim, we are supposing that S is a schedule of minimum client delay and that $\frac{|\mathcal{C}_{k-1}|}{|\mathcal{D}_{k-1}|} < \frac{|\mathcal{C}_k|}{|\mathcal{D}_k|}$. The latter yields

$$\begin{aligned}
|\mathcal{C}_{k-1}||\mathcal{D}_k| \quad &< \quad |\mathcal{C}_k||\mathcal{D}_{k-1}| \quad \text{Since all } |\mathcal{D}_i| > 0 \\
|\mathcal{D}_k||\mathcal{C}_{k-1}| - |\mathcal{D}_{k-1}||\mathcal{C}_k| \quad &< \quad 0 \\
|\mathcal{D}_{k-1}||\mathcal{C}_k| - |\mathcal{D}_k||\mathcal{C}_{k-1}| \quad &> \quad 0
\end{aligned}$$

which means that $D_S - D_{S'} > 0$, or $D_S > D_{S'}$. That is, S is not the schedule of minimum client delay, since S' has a lower client delay than S. With the contradiction our proof is complete. □

(As a side note, one can see fairly quickly that this perfectly-partitionable special case is equivalent to a situation in which a set of clients each request a single data item of variable size $\mathcal{D}_i$.)

It turns out that if we relax the notion of perfect partitions even slightly, so that $\forall$ clients $C \in \mathcal{C}$, $\forall$ data items $d \in \mathcal{D}$, c requests $d \Rightarrow \mathcal{P}'(c) = \mathcal{P}'(d)$, we can no longer apply this algorithm. Let us called this weaker $\mathcal{P}'$ an ordinary partition. (The example immediately following the definition of perfect partitions is not perfectly partitionable, but is ordinarily partitionable.) This is because the optimum (minimum delay) schedule for ordinarily partitionable requests may split partitions, as shown below.

Example. The following requests, appearing at the same time, form two ordinarily partitions.

a	b	c	d	e	f
				$\star$	$\star$
				$\star$	$\star$
$\bullet$		$\bullet$	$\bullet$		
$\bullet$	$\bullet$				
$\bullet$	$\bullet$				
$\bullet$	$\bullet$				
$\bullet$	$\bullet$				

Delay as scheduled: $0 + 8 + 4 + 12 = 24$.

The requests denoted by "$\star$" form one ordinary partition, and the requests denoted by "$\bullet$" form the other. Both the schedule above and its reverse, $\langle f, e, d, c, b, a \rangle$, have no splits. Neither schedule, however, minimizes client delay. The schedule below splits the partition denoted by "$\bullet$" at time 3, but has a lower client delay than the schedule above:

a	b	e	f	c	d
		$\star$	$\star$		
		$\star$	$\star$		
$\bullet$				$\bullet$	$\bullet$
$\bullet$	$\bullet$				
$\bullet$	$\bullet$				
$\bullet$	$\bullet$				
$\bullet$	$\bullet$				

Minimum delay, as scheduled: $0 + 8 + 8 + 6 = 22$. $\square$

Clearly, with no guarantee that the minimum-delay schedule will not force ordinary partitions to split, the algorithm described above would fail.

Appendix B

Simulation Parameters for the Changing Web

In Section 3.3, we enumerated a number of parameters to form a base case for our simulations. We detail the derivation of our base-case values here.

B.1 About the Web

To begin, we determine a few statistics about the Web. Because it is difficult to capture the structure and nature of the large and growing Web, we will complement the measured numbers with some estimates to create our model for the Web. The values we use for the Web appear as "Web value" in Table 3.1.

We find from [16] that a large fraction of pages (nearly a quarter) change at least every day, pages that we will not attempt to track in this system. (It is not practical for us to constantly crawl a large number of popular pages, much less repeatedly disseminate them over multicast as they change throughout a day. A client that wishes to track such pages will have to crawl them independently, as it needs them.) We will estimate the change frequency of the remaining pages, then—the ones we can attempt to track. We can observe from [16] that the mean change interval for the remaining Web pages is roughly 75 days.

Estimates of the number of pages on the Web varies depending the source of the statistic, but a fair number is around 1.3 billion (thousand million) hypertext pages, as of the turn of the century. Google [27], in particular, claims to have seen and indexed around that many pages for its search engine according to its root page. Meanwhile, Hobbes' Internet Timeline [31] claims there were 21 million Web sites around that time, where a Web site is defined by domain name and port number (that is, a Web site is defined by shared elements in the URL, rather than any judgments about the content or authorship of the available Web pages themselves).

From these numbers, we can do a quick division (1.3 billion over 21 million) to see that though Web sites may vary widely in size, we should expect an arithmetic mean of about 62 pages per Web site. To loosely approximate a distribution of varying-size Web sites, we invoke the popular 80/20 split as follows:

Eighty percent of Web sites account for twenty percent of Web pages (that is, most sites are very "small"), yielding 16.8 million small Web sites accounting for 260 million Web pages. Therefore, on average, a "small" Web site has 15 pages each. Unfortunately, the remaining (bigger) sites still have a very small average number of pages, so to widen the divide between small and large sites, we apply the 80/20 split again to the remaining pages:

Of the remaining Web pages, we again do an 80/20 split to get 840 000 "large" Web sites accounting for 831 million pages (yielding about 990 pages each), and 3.36 million "medium" Web sites accounting for 3.36 million pages (of 62 pages each).

In this way, we get a small number of large Web sites (of 990 pages each), a pool of medium sized Web sites (of 62 pages each), and many small Web sites (of 15 pages each).

B.2 About the Multicast Server

Assuming that we have 10BaseT Ethernet as a bottleneck connection speed for our wide-area multicast, we might safely consume 1 megabit per second of network usage to disseminate our data. Given this number, we are able to send about 3000 compressed Web pages per minute. (This is calculated in Chapter 2 as follows: 2.6 kilobytes per page compressed, times 3000 pages, yields 7800 kilobytes of data to send over 1 megabit per second. Such a transmission takes about 61 seconds.) As such, the multicast server can disseminate a compressed version of its image of the entire Web in about 300 days.

B.3 About the Clients

How much data does a client request? Because such a multicast service does not exist in wide deployment, it is difficult to know for sure. Instead, we must make an educated guess to get an initial value.

We consider a hypothetical scenario in which clients are interested in maintaining topical subsets of the Web for search or data mining. To this end, we turn to the Lycos 50 to "nominate" some topics that might be popular enough for topical indexing or mining. Then, we find an in-depth (topical) link directory site for these topics to estimate the topic's size in Web sites.

For a randomly chosen day (17 Mar 2001), Lycos 50 [41] reported that the top five searches on the Lycos search engine were NCAA basketball, Dragonball, Napster, St. Patrick's Day, and Britney Spears, in that order. The author then subjectively chose topics to enclose these searches: "college sports," "anime" (Japanese animation), "MP3" (a popular form of compressed sound/music), "holidays," and "pop music," respectively.

Now, we try to guess the size of these topics on the Web, in Web sites. We might turn to Yahoo!'s hand-organized directory and count the number of Web sites to which it points for each topic, but this creates the risk that we severely underestimate a topic's size because of Yahoo!'s finite resources. (For example, one could argue that perhaps Open Directory is already a larger directory, but it too is manually maintained, and so may suffer the same underestimation problem.) To assess this risk, we try to find a well-PageRanked portal or link directory for our example topics, to see if such a link directory has recorded many more sites than Yahoo! has for the same topic. If so, we resort to well-ranked topical link directories as the largest and most "authoritative" census of Web sites on each topic. If not, we can use Yahoo!'s outgoing links to determine the number of sites on each topic.

Checking what Google considered the most "authoritative" link directories for our example topics, it turned out only "anime" (number two) had a link directory site as a top-ranked page (http://www.anipike.com). In other cases, no topical directories were to be found, as top-ranked content sites pushed out any directories there might have been (as was the case for "sports," number one). Or, relatively low-ranking Yahoo! directory subtrees matched or beat topical link directories, suggesting that the corresponding topical directories were not particularly popular or well-referenced (for example, this applied to "holidays," number four). (The intuition in this latter case is, if a Web author who links to a directory about holidays would as soon point to Yahoo! as to any specialized directory on holidays, then perhaps the specialized directory is not particularly authoritative or comprehensive.)

With only one seed data point left, we attempt to estimate the number of Web sites for a topic with a well-PageRanked directory. Comparing the number of sites Yahoo! had for the topic (about 919 sites across 1705 outgoing links) and the number the specialized directory had (about 8273 sites across 46228 links), it becomes clear that Yahoo!'s broader, but less deep, directory does not sufficiently capture the size of a topic on the Web. As a result, we resort instead to the topical directory we found, and estimate that a client seeking the topic could call for 8000–9000 Web sites.

Now, we are left with estimating the size of other topics, for which no such topical directory is available. Ideally, we would like to adjust the 8000–9000 Web site number above to the relative sizes of other topics; this lets us estimate how many sites might have been cited had there been an authoritative directory.

To estimate the relative size of each topic, we turn to Google, which estimates that the five topics, as searches, generate around 2 million, 2 million, 12 million, 3 million, and 2 million pages, respectively. While it is not clear whether the same fraction of pages are on-topic for each of the five searches, the relative consistency of the numbers (hovering on average just over 2 million pages) suggest that for a number of specialized topics, we can reuse our estimate of 8000–9000 Web sites for the size of a topic. (The relative consistency of the numbers may also suggest that the author chose topics poorly to contain each of the top-five search terms. Certainly, one can choose broader topics and get correspondingly larger estimates. MP3, for example, generated a much larger page count perhaps in part because MP3 can match both pages about the audio compression format and about music so compressed.)

So, we adopt 9000 Web sites as our estimate for the size of a topic that a client may wish to receive or track. To contain the complexity of the simulation, we say a client requests these Web sites uniformly across the Web. That is, for our base case, we assume that a Web site about "college sports" (for example) is generally not a Web site about "MP3," and as such, it would not be requested more often than usual, by many clients of different interests. We do, however, consider the performance effect of client-request skew in our results.

Bibliography

[1] Swarup Acharya, Rafael Alonso, Michael J. Franklin, and Stanley B. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 199–210. ACM Press, 1995.

[2] Swarup Acharya and S. Muthukrishnan. Scheduling on-demand broadcasts: New metrics and algorithms. In *Proceedings of MobiCom'98, Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 43–54, 1998.

[3] Mukesh Agrawal, Amit Manjhi, Nikhil Bansal, and Srinivasan Seshan. Improving web performance in broadcast-unicast networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 1, pages 229–239. IEEE Computer Society, 2003.

[4] 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.

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

[6] Claudio Basile, Long Wang, Zbigniew Kalbarczyk, and Ravi Iyer. Group communication protocols under errors. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS 2003)*, pages 35–44. IEEE Computer Society, October 2003.

[7] Azer Bestavros and Carlos Cunha. Server-initated document dissemination for the WWW. *Data Engineering Bulletin*, 19(3):3–11, 1996.

[8] Supratik Bhattacharyya. RFC 3569: An overview of source-specific multicast (SSM), July 2003.

[9] Supratik Bhattacharyya, James F. Kurose, Don Towsley, and Ramesh Nagarajan. Efficient rate-controlled bulk data transfer using multiple multicast groups. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, volume 3, pages 1172–1179. IEEE, March 1998.

[10] Yitzhak Birk and Diego Crupnicoff. A multicast transmission schedule for scalable multi-rate distribution of bulk data using non-scalable erasure-correcting codes. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 2, pages 1033–1043. IEEE Computer Society, 2003.

[11] Robert Braden. RFC 1122: Requirements for Internet hosts — communication layers, October 1989.

[12] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, 1998.

[13] John Byers, Michael Luby, and Michael Mitzenmacher. Fine-grained layered multicast. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 2, pages 1143–1151. IEEE Computer Society, April 2001.

[14] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 2, pages 1510–1520. IEEE Computer Society, 2003.

[15] Ranveer Chandra, Venugopalan Ramasubramanian, and Kenneth P. Birman. Anonymous gossip: Improving multicast reliability in mobile ad-hoc networks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 275–283. IEEE Computer Society, April 2001.

[16] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, May 14–19, 2000*, pages 117–128. ACM Press, 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-1999-0116.

[17] S. E. Deering and D. R. Cheriton. RFC 966: Host groups: A multicast extension to the Internet Protocol, December 1985.

[18] Stephen E. Deering and Robert M. Hinden. RFC 1883: Internet Protocol, version 6 (IPv6) specification, December 1995.

[19] Stephen E. Deering and Robert M. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) specification, December 1998.

[20] Deborah Estrin, Dino Farinacci, Ahmed Helmy, David Thaler, Stephen Deering, Mark Handley, Van Jacobson, Ching gung Liu, Puneet Sharma, and Liming Wei. RFC 2362: Protocol independent multicast-sparse mode (PIM-SM): Protocol specification, June 1998.

[21] Pascal Felber and Fernando Pedone. Probabilistic atomic broadcast. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 170–179. IEEE Computer Society, October 2002.

[22] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley and Sons, Inc., New York, second edition, 1957.

[23] 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.

[24] Michael J. Franklin and Stanley B. Zdonik. Dissemination-based information systems. *Data Engineering Bulletin*, 19(3):20–30, 1996.

[25] Simon French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood Limited, Chichster, England, 1982.

[26] Chris Gonsalves. Akamai DDoS attack whacks Web traffic, sites. *eWEEK*, June 2004. Available at http://www.eweek.com/article2/0,1759,1612739,00.asp.

[27] Google, 2001. http://www.google.com/.

[28] Sergey Gorinsky, Sugat Jain, Harrick Vin, and Yongguang Zhang. Robustness to inflated subscription in multicast congestion control. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–98. ACM Press, August 2003.

[29] Indranil Gupta, Anne-Marie Kermarrec, and Ayalvadi J. Ganesh. Probabilistic atomic broadcast. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002)*, pages 170–179. IEEE Computer Society, October 2002.

[30] 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.

[31] Hobbes' Internet timeline. http://info.isoc.org/guest/zakon/Internet/History/HIT.html.

[32] E. G. Coffman Jr., editor. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, Inc., 1976.

[33] Koushik Kar, Saswati Sarkar, and Leandros Tassiulas. Optimization based rate control for multirate multicast sessions. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 1, pages 123–132. IEEE Computer Society, April 2001.

[34] Gu-In Kwon and John W. Byers. Smooth multirate multicast congestion control. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 2, pages 1022–1032. IEEE Computer Society, 2003.

[35] Mike Lesk. Personal communication to Hector Garcia-Molina.

[36] Kurt Lidl, Josh Osborne, and Joseph Malcolm. Drinking from the firehose: Multicast USENET news. In *USENIX Winter 1994 Technical Conference Proceedings*, pages 33–45, January 1994.

[37] Jiangchuan Liu, Bo Li, Y. Thomas Hou, and Imrich Chlamtac. Dynamic layering and bandwidth allocation for multi-session video broadcasting with general utility functions. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 1, pages 630–640. IEEE Computer Society, 2003.

[38] Jiangchuan Liu, Bo Li, and Ya-Qin Zhang. A hybrid adaptation protocol for TCP-friendly layered multicast and its optimal rate allocation. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 3, pages 1520–1529. IEEE Computer Society, June 2002.

[39] Michael Luby, Vivek K. Goyal, Simon Skaria, and Gavin B. Horn. Wave and equation based rate control using multicast round trip time. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 191–204. ACM Press, August 2002.

[40] Jun Luo, Patrick Th. Eugster, and Jean-Pierre Hubaux. Route driven gossip: Probabilistic reliable multicast in ad hoc networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, volume 3, pages 2229–2239. IEEE Computer Society, 2003.

[41] Lycos 50, 2001. http://50.lycos.com/.

[42] Steve McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 117–130. ACM Press, August 1996.

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

[44] 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.

[45] Inc. Multicast Technologies. Multicast tech faqs, 2001. Available at http://www.multicasttech.com/faq/.

[46] 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*, volume 1795 of *Lecture Notes in Computer Science*, pages 185–207. Springer-Verlag, 2000.

[47] J. Pereira, L. Rodrigues, M. J. Monteiro, R. Oliveira, and A.-M. Kermarrec. NEEM: network-friendly epidemic multicast. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS 2003)*, pages 15–24. IEEE Computer Society, October 2003.

[48] Christine Perey and Marshall Eubanks. Debunking multicast urban legends, January 2004. Available at http://www.streamingmedia.com/article.asp?id=8557.

[49] Roberto Di Pietro, Antonio Durante, and Luigi V. Mancini. A reliable key authentication schema for secure multicast communications. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS 2003)*, pages 231–240. IEEE Computer Society, October 2003.

[50] J. Postel. RFC 768: User datagram protocol, August 1980.

[51] Jon Postel. RFC 791: Internet Protocol, September 1981.

[52] Anton Riabov, Zhen Liu, Joel L. Wolf, Philip S. Yu, and Li Zhang. New algorithms for content-based publication-subscription systems. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 678–686. IEEE Computer Society, May 2003.

[53] Luigi Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, April 1997. source code at http://info.iet.unipi.it/ luigi/vdm.tgz.

[54] 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.

[55] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison Wesley Longman, Reading, Massachusetts, fifth edition, 1998.

[56] Jack Snoeyink, Subhash Suri, and George Varghese. A lower bound for multicast key distribution. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, volume 1, pages 422–431. IEEE Computer Society, April 2001.

[57] Chi-Jiun Su and Leandros Tassiulas. Broadcast scheduling for information distribution. In *Proceedings of the INFOCOM '97, Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, 1997.

[58] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

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

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

[61] Lorenzo Vicisano, Luigi Rizzo, and Jon Crowcroft. TCP-like congestion control for layered multicast data transfer. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, volume 3, pages 996–1003. IEEE, March 1998.

[62] Brett J. Vickers, Célio Albuquerque, and Tatsuya Suda. Adaptive multicast of multi-layered video: Rate-based and credit-based approaches. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, volume 3, pages 1073–1083. IEEE, March 1998.

[63] D. Waitzman, C. Partridge, and S. E. Deering. RFC 1075: Distance vector multicast routing protocol, November 1988.

[64] Ping Xuan, Subhabrata Sen, Oscar González, Jesus Fernandez, and Krithi Ramamritham. Broadcast on demand: Efficient and timely dissemination of data in mobile environments. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, pages 38–48, 1997.

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

[66] Tak W. Yan and Hector Garcia-Molina. SIFT—A tool for wide-area information dissemination. In *Proceedings of the 1995 USENIX Technical Conference*, pages 177–186, New Orleans, 1995.

[67] Beichaun Zhang, Sugih Jamin, and Lixia Zhang. Host multicast: A framework for delivering multicast to end users. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 3, pages 1366–1375. IEEE Computer Society, June 2002.