Slicing Broadcast Disks

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

Abstract

Because a multicast data server or broadcast disk can have clients of very different network capacities, the server needs to offer its repository of data items at a variety of transmission speeds to service clients' varied requests. In this paper, we study how to slice a server's available outgoing network capacity into data channels, how to assign the server's data to those channels, and how to assign clients to the channels given clients' varied requests and download speeds. We find that good choices in this area can improve performance for clients by three-fold or more, and surprisingly, finding the good choices do not require advance knowledge of clients' exact download speeds, once we have chosen the slowest client speed we want to support.

1 Introduction

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

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

The idea of such a multicast facility is well-established, having been envisioned and studied for Teletext [3], Datacycle [6], and broadcast disks [1], among others. As a local case in point, we are creating a multicast facility for our WebBase project, which crawls the Web to create a repository of Web pages for research. A multicast facility will offer our repository to other researchers so that they can request subsets of our repository in a simple and efficient way. (There are other benefits to a multicast-distributed crawled Web repository, including less load on the individual Web servers that get crawled for multicast distribution. For additional details, see [7].)

In such a multicast service, the server can expect some available outgoing network capacity at its disposal, which it can use as it sees fit. In particular, the server can decide to use the entire capacity as one multicast channel, or it can decide to split its outgoing capacity into multiple multicast channels. Using multiple channels can be desirable because it allows a server administrator to split the data to distribute across multiple servers if necessary, with each server providing its own subset of the data. Similarly, the servers may share all the data but split the client load, so that each server maintains the requests for a smaller subset of clients, smaller sets of requested data, or both, compared to what a single large server would have to manage.

Further, a multicast service can expect its clients to vary widely in capacity. Unlike the telephone network, where every node is given the same fixed bandwidth for transmission, data networks such as the Internet have widely-different-throughput links between nodes and networks. As a result, multicast-service clients from a heterogenenous network such as the Internet may be connecting to the server with very different network throughputs at their disposal. Multiple channels helps a server satisfy such heterogeneous clients, by making it likely that clients will have at least one channel whose throughput fits within the client's network capacity.

On the other hand, splitting the server's network capacity into smaller channels has 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 this paper, we consider different ways to use a server's available network capacity to serve heterogeneous clients, and consider their impact on system performance. As far as we know, this issue of how and when to split a multicast repository server's data channel into multiple smaller data channels, and how to assign the server's data items to those channels, has not been considered in detail in prior multicast data dissemination work. 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.

To illustrate how our multicast facility might be designed, let us consider an example in which four clients request data as shown in Table 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. With this example, we illustrate how three classes of multicast-server design—the single-channel, multiple-channel, and split-channel designs—would behave, in the next three sections below.

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

Table 1: Example Clients

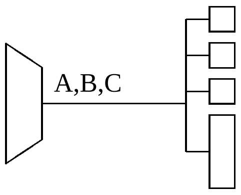

Figure 1: Single-Channel Multicast

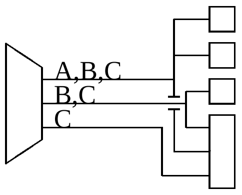

Figure 2: Multiple-Channel Multicast

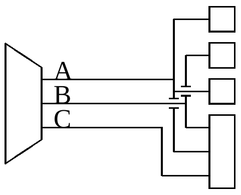

Figure 3: Split-Channel Multicast

1.1 Single-Channel Multicast

In a single-channel multicast facility as shown in Figure 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 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 Figure 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.

1.2 Multiple-Channel Multicast

In a multiple-channel multicast facility, as shown in Figure 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 Figure 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 channel as well as its own.

1.3 Split-Channel Multicast

In another version of the multiple-channel multicast facility, shown in Figure 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. Figure 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.

1.4 Outline

We will explore in this paper 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.

In particular, we have some pressing design questions to address:

- How should a multicast server offer its data on multiple channels? A server could offer all its data on all channels, or divide its data into partitions (as in Figure 3, requiring clients to connect to different channels in turn.
- How many multicast channels should the server run concurrently? Is data delivery faster using fewer channels of more throughput each, serving numerous clients, or using more channels of lower throughput, each more closely matching its clients' immediate requests?
- How should the throughput of each channel be chosen to optimize performance, given a total throughput at the server's disposal?
- Given a server's offering of channels, how should a client decide which channels to use to optimize its performance? (Alternatively, we can restate the question from the server's persective: Given a client's data request and maximum throughput, how many channels, up to the client's throughput limit, should the server instruct the client to receive for its data? How should the server choose those channels from all the ones it offers?)

In this paper, we will present our model of a multicast facility, and enumerate several ways it could be designed using multiple channels. We develop an function to describe its performance, and show how this function approximates a multicast facility's simulated behavior. Finally, we will use this model of multicast behavior to evaluate the performance of our various designs.

2 The Multicast Facility

In our multicast facility, the multicast server has a fixed outgoing network throughput (which we will informally call "bandwidth") available for its use, and a number of data items (intuitively, files) ready for dissemination, from which each client will request some (not necessarily proper) subset. A new client requests a subset of the server's data items using a reliable unicast connection to the server. The multicast facility may split its bandwidth into multiple multicast data channels. Because each such channel behaves similarly, we will first focus on 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.

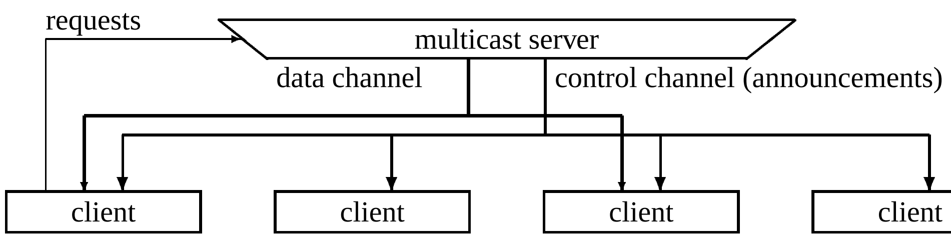

Figure 4: Information Flow in the Multicast Facility

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 remaining allocated bandwidth, to carry the data items themselves. The resulting separation of network traffic is shown in Figure 4.

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 we will use this as our performance metric.

2.1 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 [7] 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.

2.2 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. 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 can simply partition its throughput into a fixed number s of equal-bandwidth channels, each offering a partition of distinct N/s 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 N 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 s of linearly-increasing-bandwidth channels, starting from a base throughput t_0 (no higher than the minimum client bandwidth) and incrementing by some constant factor c, i.e., t_0, $(1+c)t_0$, $(1+2c)t_0$, $(1+3c)t_0$, $...(1+(s-1)c)t_0$. 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** A server can partition its throughput into a number s of exponentially-increasing-bandwidth channels, starting from a base throughput t_0, i.e., $2^0t_0, 2^1t_0, 2^2t_0, ...2^{s-1}t_0$. 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 bw-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.

3 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 4 and 5, 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.

Our model assumes that the expected average client delay (D) is a function of three principal factors: the number of clients making requests on a multicast channel (c); the number of data items each client requests on the multicast channel (r); and the throughput of the multicast channel (b).

$$delay = D(c, r, b)$$

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 $D(c, r)$ 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 c. This information will allow us to approximate the client delay function $D(c, r)$ 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 $D(c, r)$, and apply a factor from b to determine $D(c, r, b)$.

3.1 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 $N = \frac{cs}{1-cs}$ for s the service time of a single client and c 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; $cs < 1$.) Also, by Little's result, the corresponding expected client delay is $D = \frac{N}{c} = \frac{s}{1-cs}$.

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 M, 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) $(1 - \frac{1}{M})^{Nr}$ 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 $r(1 - \frac{1}{M})^{Nr}$ 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 $s = r(1 - \frac{1}{M})^{Nr}$. With our substitution for s, we get $D(c, r) = \frac{N}{c} = \frac{r(1-\frac{1}{M})^{Nr}}{1-cr(1-\frac{1}{M})^{Nr}}$.

We cannot get a closed form for $D(c, r)$ 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 $D(c, r)$.

This approximation of $D(c, r)$, however, is restricted to regions of relatively low load ("queue-like" behavior), so we must next approximate the client delay function at higher load.

3.2 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 $\frac{r}{r+1}$ [1]; therefore, for M data items total, this is $M\frac{r}{r+1}$ 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 $M/2$, which is what we would expect as the client

[1]Extreme Value Distribution, MathWorld; derivation in Feller, 2/e

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 M, 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.

3.3 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 M. At slightly lower load, by contrast, the clients in the system may collectively request fewer than M 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 [5]. In the collector's problem, a collector repeatedly purchases items, each chosen at random from an infinite bag of M distinct items so that each of the M items is equally likely to be acquired from any purchase. For a large number of purchases p and not-as-large M (such that $Me^{-\frac{p}{M}}$ remains bounded), the collector can expect to acquire approximately $M(1 - e^{-\frac{p}{M}})$ of the M 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 M 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 $M(1 - e^{-\frac{Nr}{M}})$ distinct items in total.

Because the total data requested by N clients is $M(1 - e^{-\frac{Nr}{M}})$, 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 $D(N, r) = M(1 - e^{-\frac{Nr}{M}})(\frac{r}{r+1})$ for N clients requesting r data items each.

To obtain our high-load approximation $D(c, r)$, we have to determine what client arrival rate c 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, $c = N/D(N, r)$.

In short, given some N and r, we can determine $D(N, r)$, then c. These values determine a domain-range pair of the delay function $(c, r) \mapsto D(c, r) = D(N, r)$.

3.4 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 (c, r) are represented for integer N, so we use the delay of the nearest c and r for which a value is available. In the lowest loads (ones for which there is no computable client delay at any lower c), 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.

3.5 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 $D(c, r, b) = \frac{1}{b} D(c, r)$. 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 $(\frac{1}{b})^\alpha$ for α near 1. If $0 < \alpha < 1$, a linear increase in network cost buys a less-than-linear reduction on client delay (i.e., bandwidth is expensive). If $\alpha > 1$, 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 $D(c, r, b) = (\frac{1}{b})^\alpha D(c, r)$.

3.6 Comparison to Data

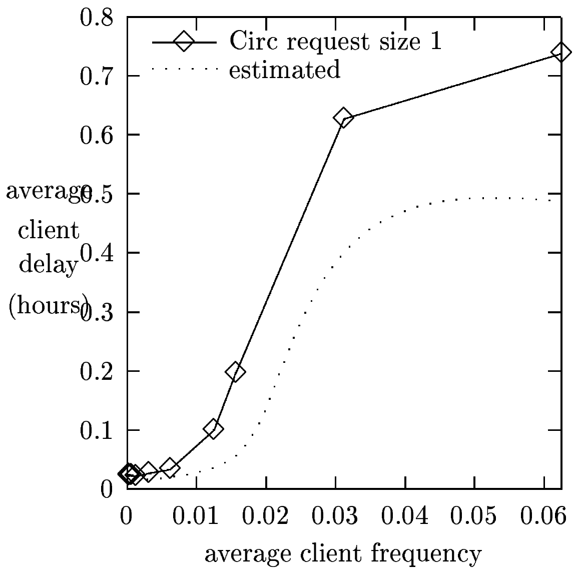

Figure 5: Delay versus Client Frequency

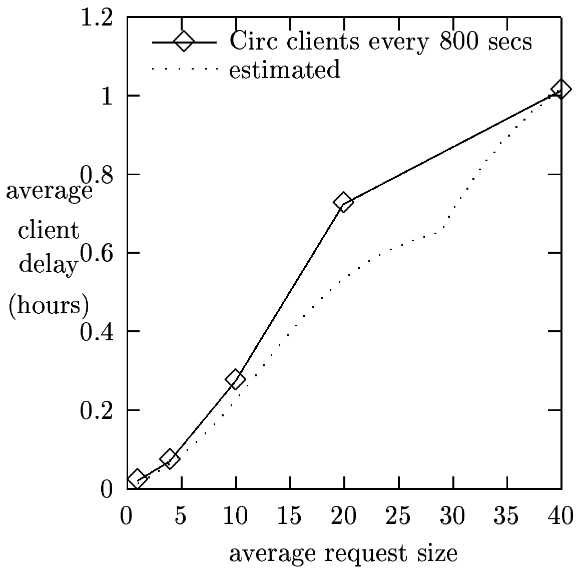

Figure 6: Delay versus Request Size

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 ananlytic model. Also, both our analytic models, based on the M/M/1 queue and the collector's problem, use a simplication 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 and 6, 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, 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 6, 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 6 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.

4 Results for Varying Numbers of Channels

Parameter	Description	Range
M	Number of data items available	100
r	Number of data items requested per client	1–50
	Number of channels	1–5
	Base throughput (minimum supported bandwidth)	1 (20 kbps)
α	Bandwidth split factor	1.0
	Server bandwidth	50 (1 Mbps)
	Client bandwidth	50 (1 Mbps)

Table 2: Some Study Parameters

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 data the server can send.

In this initial scenario, we run one server for each of the multi-channel techniques we consider in Section 2.2. 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 [7].) 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 2 summarizes our space of parameters. For simplicity and clarity, we have chosen to have 100 data items (M) 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.

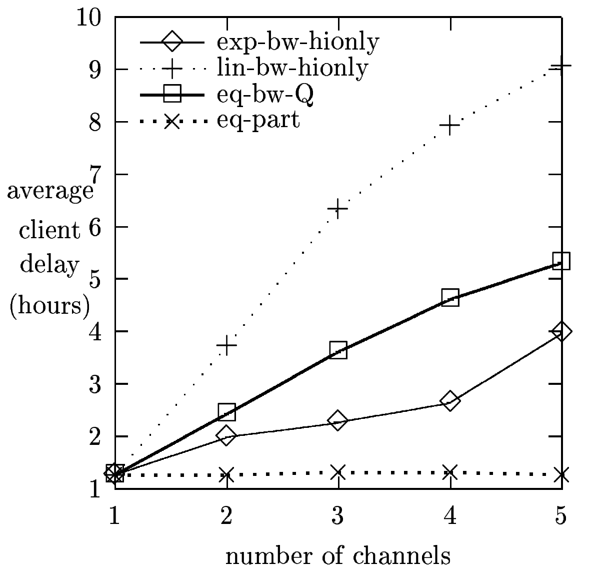

Figure 7: Varying Number of Channels for Request Size 9

In Figure 7, 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 Results for Two-Tiered Clients

Parameter	Description	Range
M	Number of data items available	100
r	Number of data items requested per client	1–50
	Number of channels	varies by technique
	Speed of slow clients	1
	Speed of fast clients	1–50
α	Bandwidth split factor	1.0

Fast and slow clients appear in equal numbers.

Table 3: 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, 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 3 summarizes the parameter space described above, in which we evaluate our various multicast schemes.

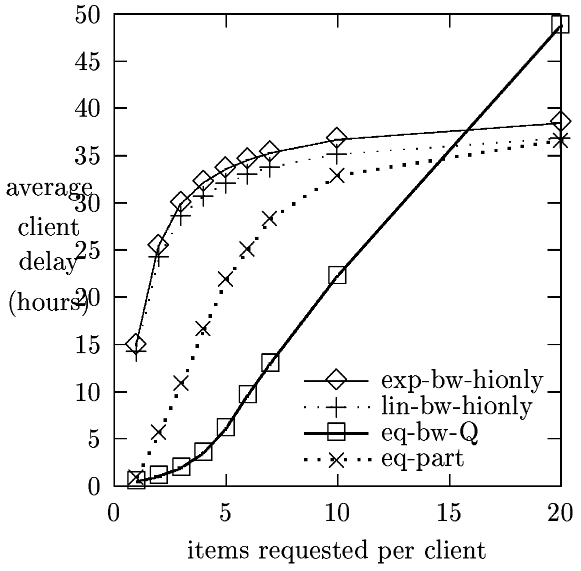

Figure 8: Various Request Sizes for Clients Having 5:1 Bandwidth

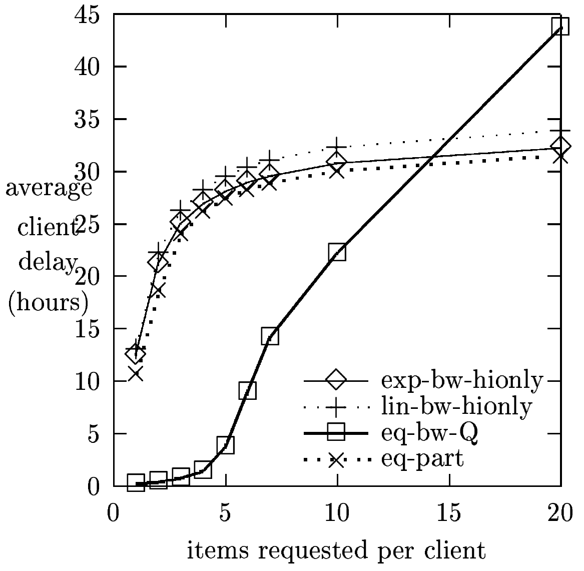

Figure 9: Various Request Sizes for Clients Having 30:1 Bandwidth

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 8 and 9 we plot the client delay, averaged for all clients, as a function of the clients' request size in data items. In Figure 8, the fast clients have five times the bandwidth of the slow ones. In Figure 9, 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.

5.1 Results Comparing More Techniques

We can also use the scenario of Table 3 to study the other variants that have not been considered. In Figure 10, for example, we chart all the exp-bw techniques in the scenario corresponding to Figure 8. (That is, the exp-bw-hionly points in the two figures are identical because they represent the same scenario.) In Figure 11, we chart all the lin-bw techniques in the scenario corresponding to Figure 9.

We observe that the -hionly servers perform well relative to other client-channel assignment techniques. For example, in Figure 10 exp-bw-hionly outperforms the other exp-bw servers, and in Figure 11 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. Conse-

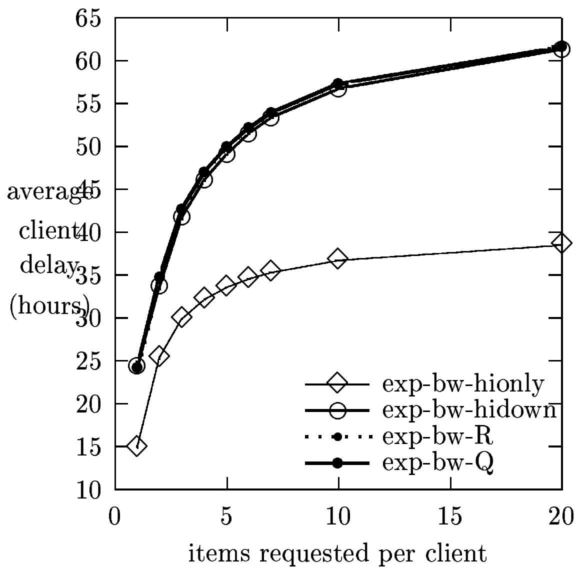

Figure 10: Various Request Sizes for Clients Having 5:1 Bandwidth (exp-bw only)

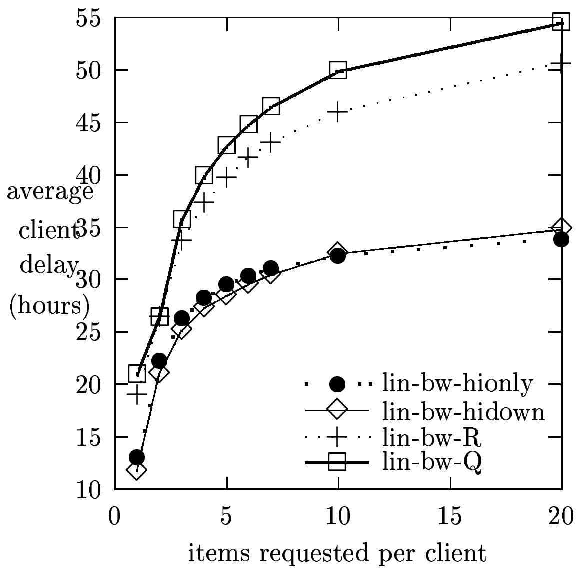

Figure 11: Various Request Sizes for Clients Having 30:1 Bandwidth (lin-bw only)

quently, exp-bw-hionly and lin-bw-hionly servers are useful choices to represent the exp-bw and lin-bw 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 11, let us examine how the -hionly and -hidown systems actually behave given the scenarios plotted in Figures 10 and 11. 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

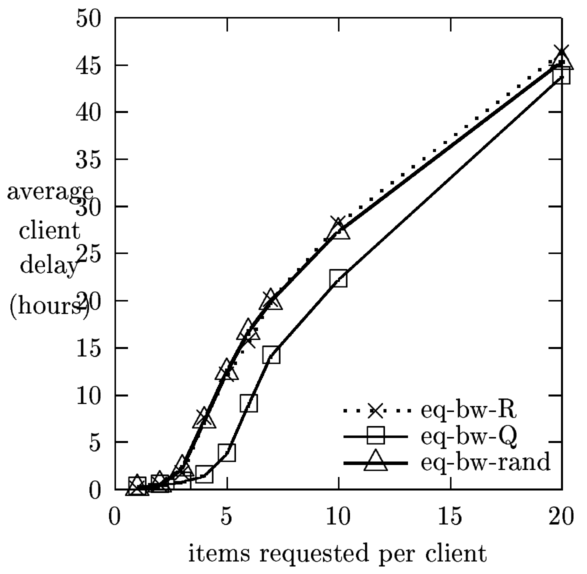

Figure 12: Various Request Sizes for Clients Having 30:1 Bandwidth (eq-bw only)

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 8, 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 11. 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 12 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.

5.2 Fast and Slow Clients

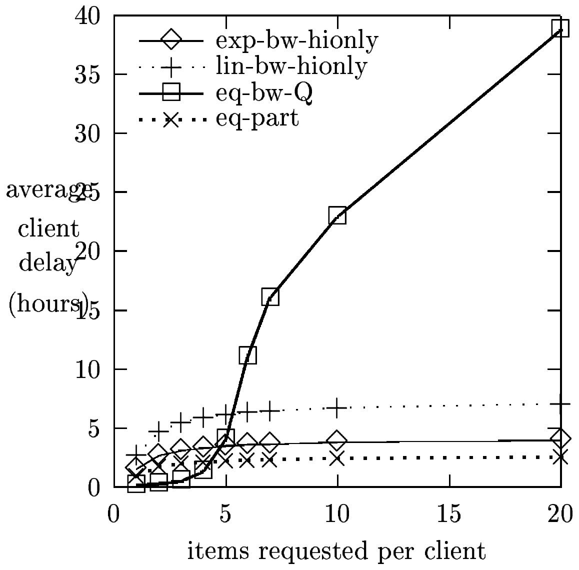

Figure 13: Various Request Sizes for Fast Clients Having 40:1 Bandwidth

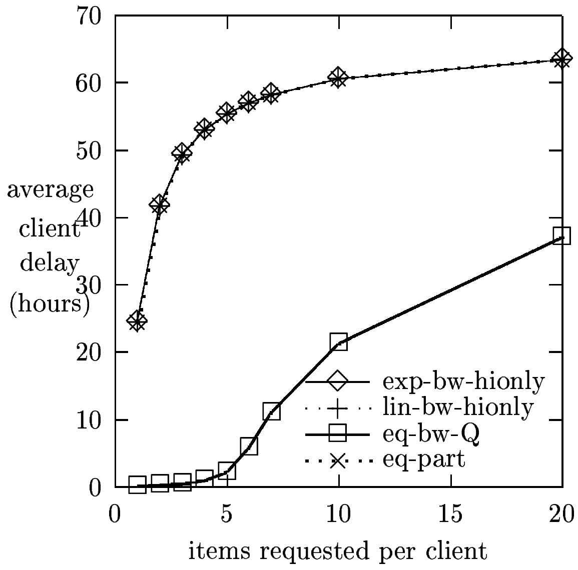

Figure 14: Various Request Sizes for Slow Clients Having 40:1 Bandwidth

With two tiers of clients, we can further examine how our various techniques treat fast and slow clients separately. In Figures 13 and 14, we consider a scenario in 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 13, we show the average client delay of the fast clients. In Figure 14, 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.

6 Bandwidth Factor

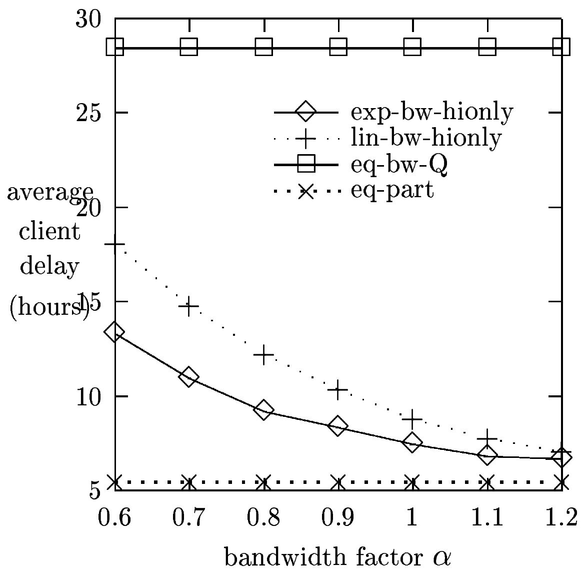

Figure 15: Effect of Nonlinear Bandwidth Factors on Exponentially-Distributed-Bandwidth Clients

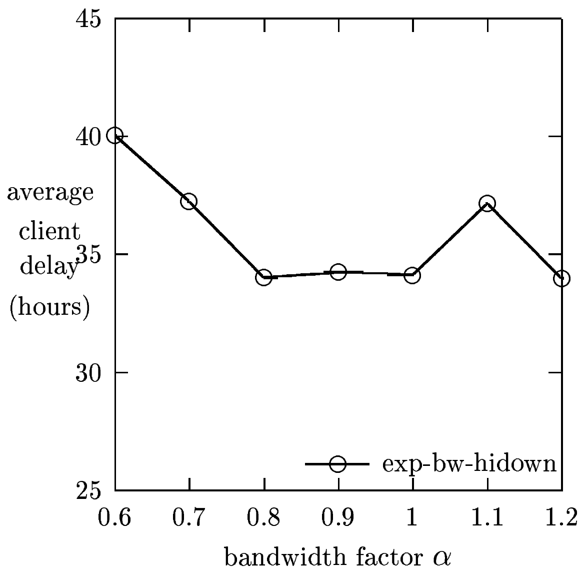

Figure 16: Effect of Nonlinear Bandwidth Factors on Exponentially-Distributed-Bandwidth Clients

In Figure 15, we consider the effect of nonlinear bandwidth factors α on the client delay of our techniques. Recall from Section 3 that a low factor ($\alpha < 1$) 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 ($\alpha > 1$) 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.

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 α 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 15, 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 α, 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 16, in which we chart the performance of exp-bw-hidown under the same scenario. Though the increase in α from 1 to 1.1 increases the bandwidth of two channels, from 8 to 9 and 16 to 21, performance actually deteriorated. At α 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 15, so we need not worry about their more chaotic behavior.

7 Client Bandwidth Information

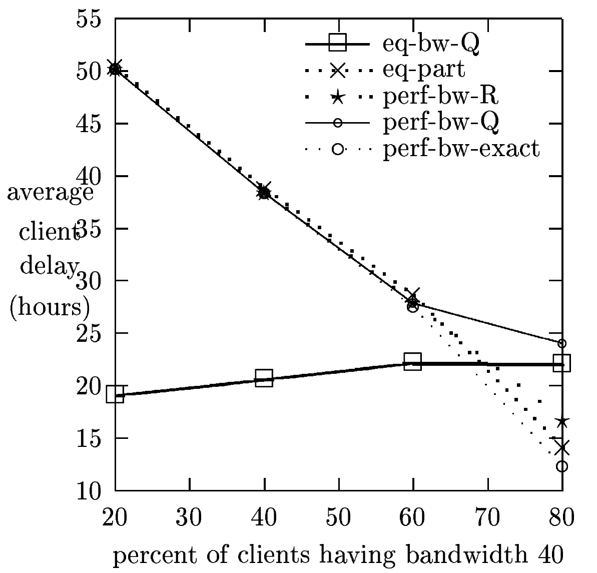

Figure 17: Various Numbers of Fast Clients Having 40:1 Bandwidth

Lastly, we consider whether a server benefits from knowing its clients' exact bandwidth. In

Figure 17, 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 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 2.2. 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 17.

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 17.)

8 Related Work

There is a large body of research into multicast protocols and systems, including data dissemination systems such as ours [8] and broadcast disks [1, 13], as a component of Web caching and service [12, 2], and as a component of publish/subscribe systems [11] such as Gryphon [10].

Though there is work on multiple-channel systems for data, such as [4, 14], it 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 paper studies how to allocate different data items to different channels, and how to assign clients making varying requests to such channels.

There is also a body of work on layered multicast of a single audio or video stream [9, 15]. This work often assumes a scenario that, unlike ours, allows sending only partial data to lower-bandwidth clients. Such 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. This paper, on the other hand, considers the distribution of many data items across different channels without degrading the data for slower clients.

9 Conclusion

In this paper 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.

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 five-fold. By correctly determining how clients are assigned to the channels that fit within the clients' download limits, we can improve performance three-fold or more, suggesting the importance of careful design choices. These results provide insights that are guiding the design of our own multicast facility for Web data.

References

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

[2] Kevin C. Almeroth, Mostafa H. Ammar, and Zongming Fei. Scalable delivery of Web pages using cyclic best-effort multicast. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, volume 3, pages 1214–1221. IEEE, March 1998.

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

[4] 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.

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

[6] 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.

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

[8] Wang Lam and Hector Garcia-Molina. Multicasting a changing repository. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pages 215–226. IEEE Computer Society, March 2003. Available at http://dbpubs.stanford.edu/pub/2003-48.

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.