Multicasting a Web Repository

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

Abstract

Web crawlers generate significant loads on Web servers, and are difficult to operate. Instead of running crawlers at many "client" sites, we propose a central crawler and Web repository that then multicasts appropriate subsets of the central repository to clients. Loads at Web servers are reduced because a single crawler visits the servers, as opposed to all the client crawlers. In this paper we model and evaluate such a central Web multicast facility. We develop multicast algorithms for the facility, comparing them with ones for "broadcasts disks." We also evaluate performance as several factors, such as object granularity and client batching, are varied.

1 Introduction

A crawler (or spider) is a program that visits Web pages, using links on seen pages to identify more pages. The crawled pages are typically used to build indexes for search engines such as AltaVista and Google. However, crawlers are also used to gather pages for data mining, or to produce Web caches than can be more conveniently accessed by users. For example, there are products on the market today that do "personalized crawling" so a user can download in advance pages he may want to browse later.

As crawlers proliferate, more and more crawlers visit each Web server on the net, repeatedly requesting the same pages over and over. Popular servers are often visited by hundreds of crawlers in a week [10]. The extra load is especially problematic to small companies that pay their ISPs by page accessed: A large fraction of their networking budget goes to pay for pages fetched by crawlers and not 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 `robots.txt` conventions and does not overload Web sites. (The file `robots.txt` at each Web site indicates what parts of a Web site may be visited.) A bug in the crawler code can cause it to visit a Web site too frequently, triggering a so-called "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.

In this paper we propose 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 on 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.

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.

We are in the process of building such a Web repository and multicast facility. Our second-generation crawler is currently used to build a cache we call WebBase. WebBase currently holds over 40 million pages, distributed over several computers. WebBase provides an API through which local programs can request streams of pages [8]. The programs, developed by various students in our group, data mine the Web data, analyzing for example the link structure or the way words are used. Our goal is to extend the WebBase interface to support requests from remote clients, and to efficiently combine and multicast the requested data.

As we design and build WebBase, a number of interesting and challenging questions have arisen, some of which we address in this paper:

- 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 there is at least one key difference: in our case, a client request is not fulfilled until *all* requested Web pages are

delivered. In traditional scenarios, on the other hand, each page 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 7.)

- What is the best way to schedule client requests? That is, how does WebBase 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" clients should subscribe to? Should they request individual pages? Should clients request pages be by URL or by content (e.g., a query gives keywords that must appear on the page)? Or should a client request full sites? Or 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 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 crawl on their own? Are the savings significant enough to merit the expense of a central WebBase repository? Again, we will present results that shed light on this question.

- How are incremental updates handled? The WebBase crawler periodically revisits Web pages to get more up-to-date versions. This fresher data can also be multicast to clients, if they request it. Due to space limitations, we do not address these questions in this paper.

- What are the legal implications of re-broadcasting Web pages? Some lawyers claim that it is illegal to redistribute Web pages, but several Web sites (e.g., Google) do this today with no apparent problem. Chances are that as long as the redistribution does not affect the revenue of Web sites (e.g., if Web pages are only used for indexing or data mining), then multicasting would be acceptable. In this paper we do not address these legal issues further.

In summary, this paper makes the following contributions:

1. We define the Web multicast problem, and present a model and metrics for studying it (Section 2). Although our model may be applicable in other instances (e.g., distributing software), our focus here is on Web pages.

2. We define and evaluate several heuristics for scheduling transmissions (Section 3).

3. We address some of the questions outlined above (Sections 5 and 6), illustrating the tradeoffs among unit size, unit popularity, client delay, number of clients, and so on.

2 Metrics

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

We define a *server schedule* S to be a sequence of elements $S = \langle s_1, s_2, s_3, \ldots \rangle$ where $\forall 1 \leq i \leq n, s_i \in D$ or $s_i =$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 paper we assume that all data items d_i 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 subscribe to whole Web sites, then items will be roughly of the same size. Because we expect Web sites to be a common and convenient unit to subscribe to, we will think of data items as Web sites in this paper, and use the terms interchangeably.

Since the data items (crawled 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 iS_T, where S_T is the time to transmit one site. (For simplicity, we assume that network idle periods are also a multiple of S_T.) A particular client $c_i \in C$ will have its request for Web site $r \in R_i$ *satisfied* the first time after t_i that r is transmitted. That is, the request will be satisfied at time

$$t_{ir} = S_T \times \min\{t \in \mathbb{Z} | S_T t > t_i \wedge s_t = r\}$$

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,

$$T_D = \frac{1}{k} \sum_{i=1}^{k} (\max \{t_{ir} | r \in R_i\} - t_i)$$

- 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

$$S = \frac{\max \{t_{ir} | c_i \in C \wedge r \in R_i\}}{S_T}.$$

  We need to subtract from these S slots, the slots that were unused, and then multiply the number of resulting utilized slots by the size of each slot, S_D. Thus,

$$N = \frac{1}{k} S_D [S - |\{t \in \mathbb{Z} | 1 \leq t \leq S \wedge s_t = \text{ null }\}|]$$

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:

$$T_R = \frac{1}{k} \sum_{i=1}^{k} \sum_{r \in R_i} (t_{ir} - t_i)$$

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 giving 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 [5] 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 $S_T = 1$ 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	

The minimum total response time is, from the table, 3+6+6+4 = 19. The total delay, similarly, is 2+2+2+3+3+4 = 16. 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 paper, we only focus on the delay and network cost metrics.

3 Basic Scheduling Heuristics

The server scheduling heuristic determines what data item (in our case, Web site crawl) 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.

3.1 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 $S_T = 1$. 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. □

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.

3.2 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 1.

```
Input: A set of clients and their still-pendingrequests
Output: The data item to transmit
   MinRequestSize ← infinity;
      MinRequestClient ← null;
      foreach client c
         if size of remaining request of c < MinRequestSize
            MinRequestSize ← size of remaining request of c;
            MinRequestClient ← c;
      choose a data item requestedby MinRequestClient.
```

Figure 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 MinRequestClient, 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.

3.3 RxW

The *RxW* heuristic is shown in Figure 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: RxW Heuristic

3.4 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 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
            ChunkToSend ← i;
            MaxRQ ← RQ_i;
    return ChunkToSend.
```

Figure 3: R/Q Heuristic

Variable	Description	Base value
N	Total number of data items (Web sites)	10000
h	Fraction of data (Web sites) deemed hot	0.20
p	Fraction of requests that are of hot sites (approximate)	0.80
T	Average time between clients	30 minutes
R	Average size of client requests (Web sites)	20
S_D	Size of each item (Web site) (kilobytes)	7800
S_T	Time to transmit one item (Web site)	1 minute

Table 1: Simulation Parameters and their Base Values

4 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.

4.1 Description

In our simulation, a server is presumed to have all its N 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 units of time.[1] Each client, when it appears, listens to a (unique) multicast tree with root 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 mutlicast 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 $0 \leq h \leq 1$ of the items are designated *hot*, the remainder *cold*. A fraction $p \geq 0.5$ of all requests, made by any client on any item, are of hot items. The remainder of all requests $1-p$ 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 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 introducted in Section 1. We guess that analysts may track on the order of twenty competitor 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 N, 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 S_D to this value. The unit of time (S_T), one minute, comes from an assumption that we

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

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 4.2 we study the impact of varying N, and show that the *relative* performance of the heuristics is unaffected by this scale.

4.2 Initial Results

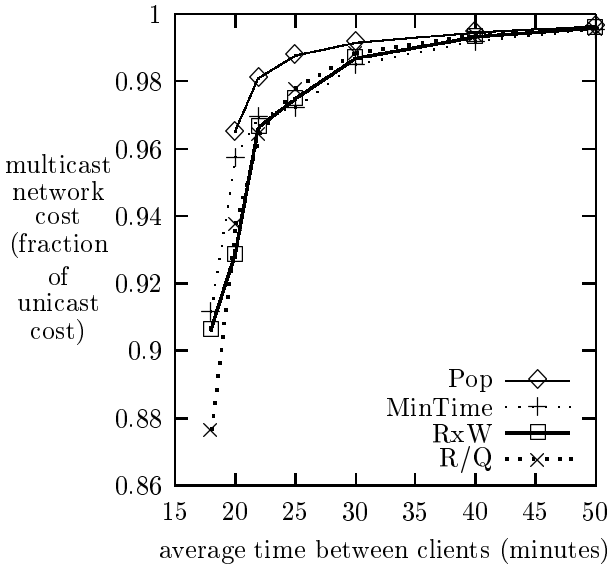

Figure 4: Benefit in amortized network cost

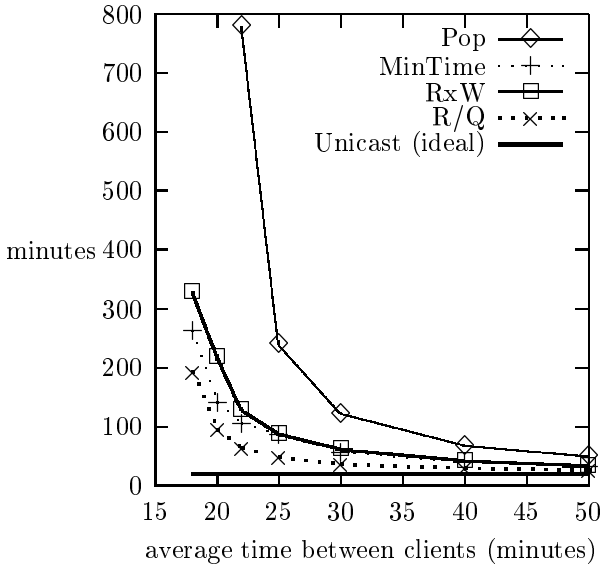

Figure 5: Client delay

First, let us compare the scheduling heuristics for a multicast facility. In our plots for this paper, "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 RS_T. All simulations are run at least until the 95% confidence interval for average client delay is smaller than 10% of the value itself.

A typical network cost plot appears in Figure 4. In this plot, we consider the network cost of multicast distribution as a function of the average client interarrival time (T). 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 $T = 20$ (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 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 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 5 is the average client delay graph that corresponds to the same simulations plotted in Figure 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). 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 5, Pop's delay (4075 minutes) at 20 minutes inter-arrival time is not shown becasue 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 6, we plot the average client delay of our heuristics as a function of the number of data items the server maintains (N). The request skew—the number of hot data items—is scaled with N, meaning that h 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 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 N) 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

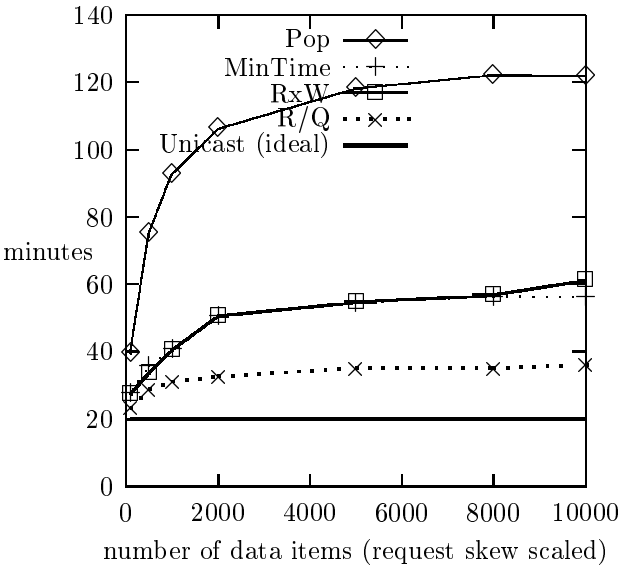

Figure 6: Heuristics compare similarly with more data

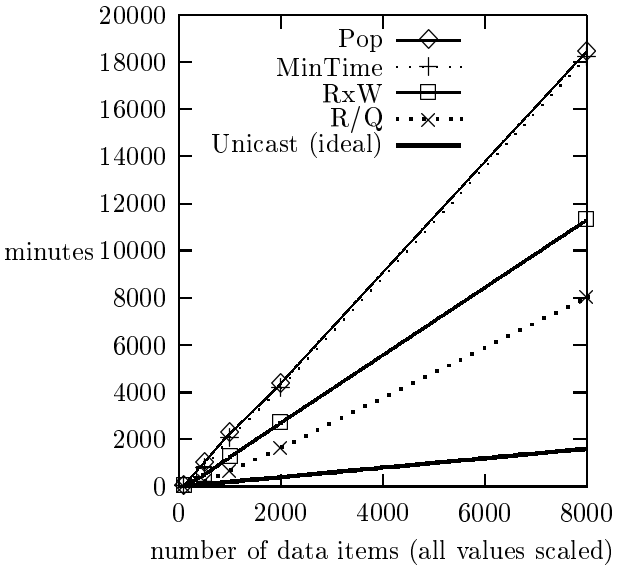

Figure 7: Heuristics compare similarly at larger scale

widely varying N. For example, for $N = 1000$ 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 ($N = 10000$, our base value).

In Figure 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 N, number of hot items hN, and average client request size R) concurrently. Again, we plot average client delay for our heuristics as we vary N, the total number of data items, over a large range. For this latter case, unlike Figure 6, we had clients request a significant fraction ($R = 0.2N$) 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 paramters in each simulation remain proportional to our base parameter values for the $N = 10000$ case. We notice that in Figure 7, unlike Figure 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 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 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 ($N = 100$

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 ($T \geq 50$, meaning in this case that $T \geq N/2$). 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 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 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 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 8 and 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

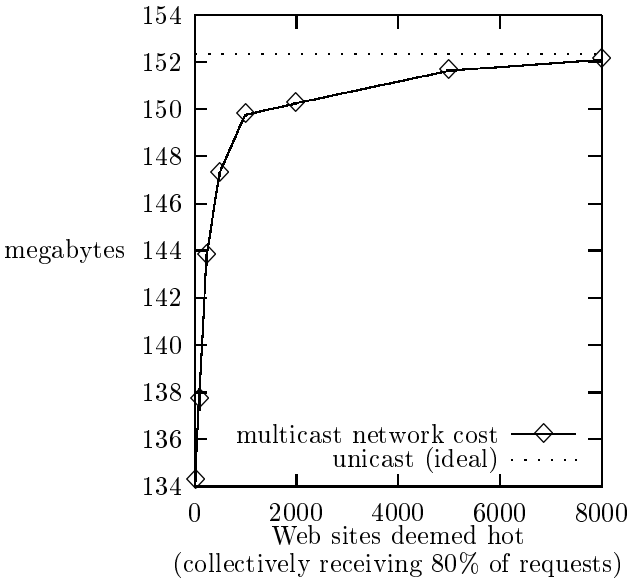

Figure 8: Network cost varies by number of hot items

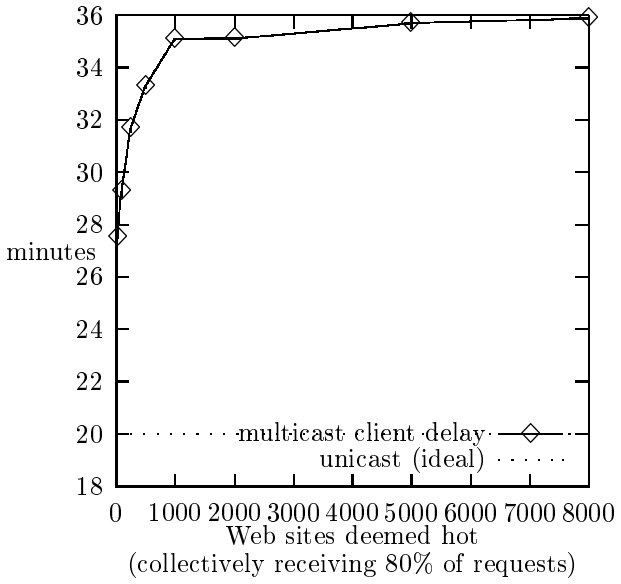

Figure 9: Delay varies by number of hot items

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 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 consumption and lower client delay.

5 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

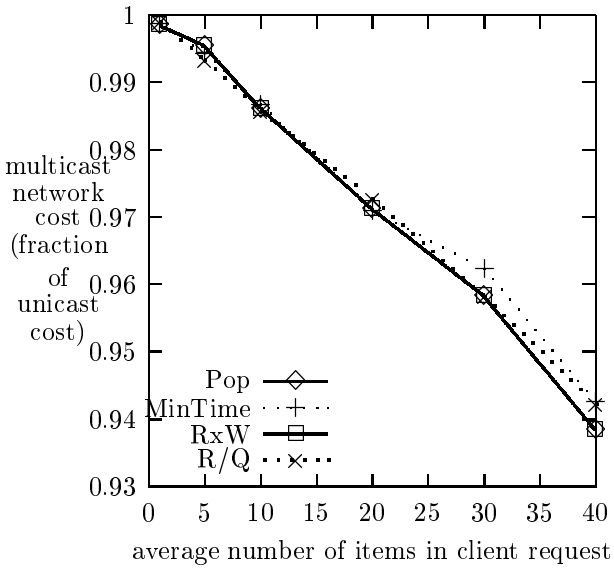

Figure 10: Benefit in network cost for batched clients

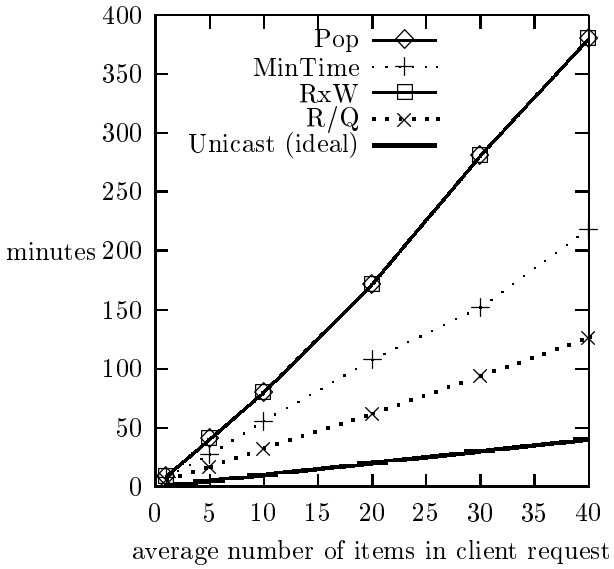

Figure 11: Client delay for batched clients

of whether clients arrive at once, or arrive as scheduling decisions are made. Figures 10 and 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.

Now, we turn to consider how we might be able to batch clients into intervals. In Figures 12

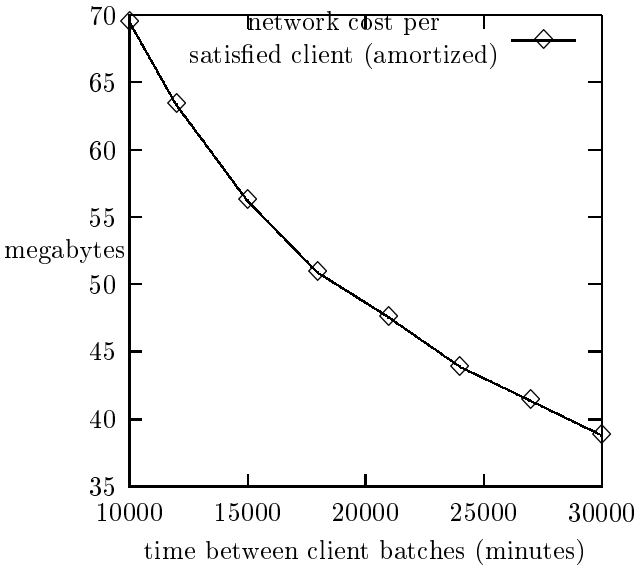

Figure 12: Amortized network cost for batched clients

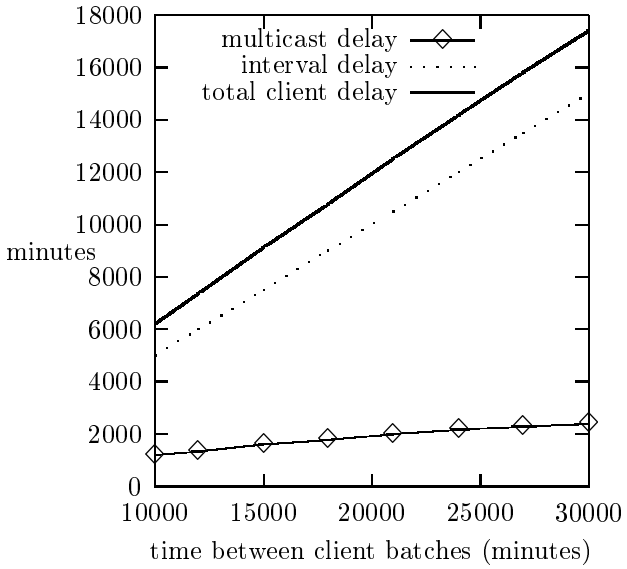

Figure 13: Client delay for batched clients

and 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 I must be chosen so that $I \geq NS_T$. 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 $I/2$ waiting for the next interval. This $I/2$ delay is plotted in Figure 13 as "interval delay" (the dotted line).

Also, notice the dramatic fall in network cost as the interval spacing I grows. This is because with clients appearing on average every T time, we expect the server to accumulate I/T 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 $NS_D/(I/T) = NS_DT/I$ units of network cost per client, an upper bound that falls rapidly as I grows. In contrast, if we did not batch clients in our multicast service, the upper-bound network cost per client is a larger $S_DT \geq NS_DT/I$. (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 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 NS_T. 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.

6 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 (N/u) large chunks, rather than more (N) 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.

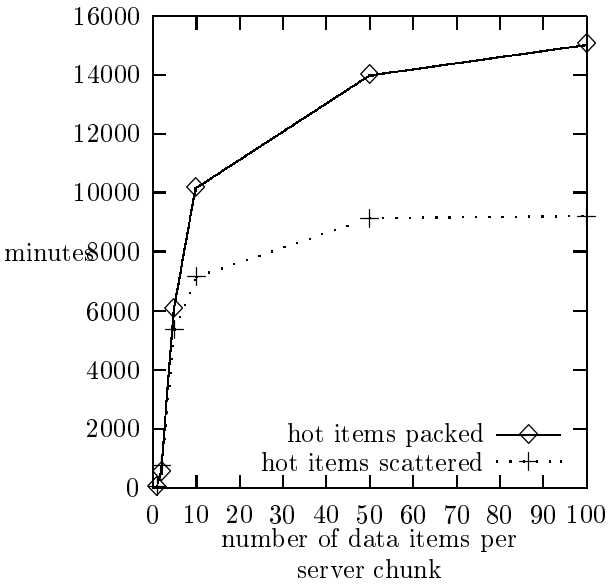

Figure 14: Client delay grows as server chunks grow larger

In Figure 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 (N/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 NS_T.

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.

7 Related Work

Using specialized intermediaries to disseminate data to large numbers of clients is not new. One technique is for the intermediaries to acts as "caches," reducing access latency. For example, in [4], the authors argue that Web page access follows a Zipfian distribution, which is modeled in this paper as hot and cold data items. Then, to get reduced network consumption and server load, they 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 would not alleviate the need for client crawlers to fetch large bodies of data.

In [11], the authors do consider multicast distribution (for example, using a satellite over a transatlantic link) together with caching. Unlike our work, however, which allows for noticeable client delay to reduce network consumption, the authors focus on low-latency online Web browsing.

Multicast distribution becomes a secondary means of filling Web caching proxies, complementing organized hierarchical caching.

In contrast to caching work is the work related to "broadcast disks," in which the intermediary disseminating data is more prominent than a cache, and is instead the primary source of data. In [1] and [6], "broadcast disks" are described as a shared-distribution-channel data dissemination technique driven entirely by a server (repository), without input from client requests. This means that a 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, [15] determines scheduling algorithms for broadcast disks that minimize average "access time" (average response time) for a given data-access probability distribution. [3] develops a scheduling heuristic (RxW, described briefly in Section 2) 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. In effect, 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 paper, represented instead by RxW) and can have its results and performance approximated when run-time for the scheduler is limited.

Still, in this scheduling work a lingering assumption remains: clients request a single piece of data at a time. This is true in much scheduling work for broadcast delivery in general, such as in [13] and even when the authors are considering less conventional measures of client waiting, such as [2] ("stretch" as client response time divided by the size of the data item the client is requesting) and [16] (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 ([14], [12]), and of job scheduling in operations research ([7], [9]). 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 process on a CPU, for example, benefits only that process, and the general job-scheduling problem [7] does not allow one machine doing one operation to benefit multiple jobs requiring it.

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 Web 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.

8 Conclusion

As the Web gains importance, we believe that gathering, analyzing, and indexing large amounts of Web information will be critical. Having clients independently gather (crawl) their information is inherently expensive. Web mulitcast, as proposed here, is a promising technology that can dramatically reduce loads at source web sites, and can significantly cut network traffic, without introducing inordinate client delays. In this paper we have modeled such 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 have introduced here, R/Q, is able to substantially outperform existing heuristics. We have also studied various Web multicast isues, such as granule size and batching delay. The results provide insights that are guiding the design of our own WebBase multicast facility.

Acknowledgments

We thank Sriram Raghavan for his helpful input during many of our early discussions.

References

[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] 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.

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

[5] 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.

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

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

[8] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: A repository of web pages. Technical report, Stanford University, 1999.

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

[10] Mike Lesk. Personal communication.

[11] 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.

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

[13] 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.

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

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

[16] 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.