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 “broadcast 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. Such users might include (for example) an analyst 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. All of these users can benefit from having Web data available to them offline.

As crawlers proliferate, more and more crawlers visit each Web server on the Internet, repeatedly requesting the same pages over and over. Popular servers are often visited by hundreds of crawlers in a week [8]. 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. Everyday users may not have the expertise, and certainly not the resources, to craft and manage a well-behaved Web crawler.

Here, 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.
Similarly, we would not require, or expect, profit-driven commercial organizations to sustain all such servers. Instead, such servers could be run by trusted organizations as a clearinghouse for general research. Alternatively, an individual organization (such as a research group or an information-gathering agency) can run such a server internally, for the convenience of its own members.

At Stanford, we are in the process of building such a Web repository and multicast facility. Our initial crawler was used exclusively for the Google search engine (before it went commercial). Our second-generation crawler is currently used to build a cache we call WebBase. WebBase currently holds over 130 million pages, distributed over several computers. WebBase provides an API through which local programs can request streams of pages [6]. 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 describe and address in this paper. To do so, we will first 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.

Next, we will define and evaluate several heuristics for scheduling transmissions (Section 3). Lastly, we will address some of the questions we face in designing and building such a Web multicast facility, illustrating the tradeoffs among unit size, unit popularity, client delay, number of clients, and so on. This abstract is intended as a summary of the extended paper [7]. The extended paper explores all these issues in further detail than we will consider here.

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 next.

For a multicast server with a fixed number of data items and a predetermined number of clients, each client \(c_i\) is characterized by the subset of the server’s data items that it requests and by the time at which the client makes its request, \(t_i\).

We define a server schedule to be a sequence of elements, each corresponding to a time slot, where each element is either a particular data item to be sent in that time slot, or null to indicate that the server sends nothing at all for that time slot.

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 time slot \(i\) ends at time \(i S_T\), where \(S_T\) is the time to transmit one site. (For simplicity, we assume that network idle periods are also a multiple of \(S_T\).) We say a particular client \(c_i\) will have its request for a Web site satisfied the first time after \(t_i\) that the Web site is transmitted.

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, for each client, we measure the time between when the client makes its request and when the client’s requests are first all satisfied. We average this measure, the delay of an individual client, across all clients.

- 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.

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, to
compute the average client response time or response time, we must take each client’s request, and add up the time between request and transmission of each data item separately. Then, we can take the average over all clients.

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 [4], 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 response time metric defined earlier, as opposed to the client response time of broadcast disks.

It turns out that the two notions of client delay and response time are quite different, as we study in detail in the extended paper [7], so it is worth noting that we will only focus on the client delay and network cost metrics here. In the extended paper, we show that server schedules (and schedulers) that optimize one metric do not optimize the other metric. Hence, we need to explore new scheduling heuristics.

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. 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.

**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. 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, contributing a (needlessly) high client delay to the overall average.

**Shortest Time to Completion** The above problem with the popularity heuristic suggests a scheduling approach borrowed from operating systems task scheduling: have the server service first the clients that have the shortest time to completion.

It turns out this heuristic will not guarantee a minimal client delay either. We find instead the inverse of the problem that plagued the popularity heuristic: There are example scenarios in which transmitting more popular items is a better approach, because doing so can satisfy a larger number of clients sooner.

**RxW** The RxW heuristic is designed for the scenario in which every client requests a single data item (for us, a single Web site) [2]. 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. This heuristic always chooses to send a data item with maximal RxW product.

**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. In this heuristic, we assign to each data item \( d_i \) values for \( R_i \), the number of clients requesting the data item, and \( Q_i \), the minimum number of data items that must be transmitted to satisfy some client requesting the data item \( d_i \). We choose to send a data item with a maximal \( R \) divided by \( Q \).
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 briefly highlight some of our results using this simulation.

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. 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 multicast tree, and of issuing data requests, are dismissed because they are small relative to the data transfer costs.

To refine the model further, each data item has the property of being either “hot” or “cold,” indicating how much it is requested relative to other items. A fraction $h$ of the items are designated hot, the remainder cold. A fraction $p$ of all requests, made by any client on any item, are of hot items. The remainder of all requests are made on cold items. The clients make requests of an exponentially-distributed random number of items, with 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.

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

### Table 1: Simulation Parameters and their Base Values

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

### 5 Some Results

We consider a number of interesting questions in the design of a Web multicast facility, and refer readers to the extended paper [7] for details. Here we will simply present two “representative” figures to illustrate the types of experiments we performed and the types of results we obtained. Then we summarize some of our overall findings.

In Figures 1 and 2, we compare the scheduling heuristics for a multicast facility, in order to evaluate the expected savings from multicast, and in order to compare scheduling policies. For our plots here, “MinTime” refers to the

---

1. Given our simplifying assumption of fixed data item size, we model time as fixed-size slots. Thus, clients may appear only at the beginning of a slot. This simplification does not have significant effect on our results.
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 a corresponding unicast distribution, and is represented as a constant cost of one client getting all its data.

![Figure 1: Benefit in amortized network cost](image1)

Figure 1: Benefit in amortized network cost

A typical network cost plot appears in Figure 1. In this plot, we consider the network cost of multicast distribution as a function of the average client interarrival time ($T$). First, we note that multicast distribution does save network cost over unicast distribution, especially as multicast gains more clients listening at the same time from higher load. For example, at $T = 20$ (clients arriving on average twenty minutes apart), 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, Popularity (Pop) and Shortest Time to Completion (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.

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. While this measurement omits the cost of the initial crawl that creates the server’s copies of its Web sites, that is a fixed cost whose impact quickly becomes negligible over time. 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.

We also find in our results 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.

To evaluate client delay for the schedulers, we consider the average client delay as a function of varying average client interarrival time ($T$). Figure 2 is the average client delay graph that corresponds to the same simulations plotted in Figure 1. For example, we find that 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. We see that RxW and MinTime have significant gains over Pop, and that an even lower average client delay results 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 interarrival times we consider, 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. We see, then, that given the choices here, we should choose R/Q as most appropriate for our Web multicast facility.

We should point out that these results apply to scenarios of widely varying scale (widely varying $N$) in the same way. 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).
Some of the other findings in our extended paper include the following:

- While we have a number of similarities with broadcast disks, 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. Schedules that work for the traditional broadcast disk may not be the best schedules for our case.

- We can “batch” clients by requiring them to wait until a predetermined time, in order to avoid multicasting pages repeatedly. When we batch clients this way, the time between client batches dominates the client delay, but the network cost per satisfied client falls dramatically. The extended paper quantifies this tradeoff, suggesting how to design a multicast facility to a target network cost per client or a target client delay.

- The choice of the data “units” to which clients subscribe must be carefully chosen to match their needs, because client delay grows rapidly as the number of multicast units get smaller. The extended paper quantifies this effect, detailing how the choice of data “units” affects the average client delay of the resulting system.

6 Related Work

Our work is related to work in Web caching ([3], [9]), but strive for different primary goals. We build on existing work in “broadcast disks” ([1], [5], [10], [2], and others) by removing its implicit single-data-item-request assumption. Outside the context of data dissemination, our work on scheduling data for multicast may remind readers of process scheduling in operating systems and of job scheduling in operations research, but neither has the key property of our multicast facility: data being scheduled for multicast can benefit multiple clients simultaneously. Also, our multicast distribution may show passing resemblance to research for Video on Demand (VoD), but VoD research can exploit properties of video that do not apply to Web data in general.

The extended paper [7] surveys some of this work and explains in more detail their relation to our work.

Acknowledgments

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

References