Multicasting a Changing 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 repeatedly running crawlers at many "client" sites, we propose a central crawler and Web repository that multicasts appropriate subsets of the central repository, and their subsequent changes, to subscribing 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 for subscriber clients, and for mixes of subscriber and one-time downloader clients. We consider different performance metrics and multicast algorithms for such a multicast facility, and develop guidelines for its design under various conditions.

1 Introduction

As the size and amount of information on the World Wide Web grows, an increasing number of users will want to download and keep a local copy of large amounts of Web data. These users may include, for example, analysts tracking Web sites of competitors, product review, and service rating information; software developers seeking to maintain Web-based FAQs and documentation; and individuals who wish to maintain their own "portal" Web pages summarizing new changes to Web resources they regularly follow.

Currently, the only way to automatically retrieve such volumes of Web data is through the use of a Web crawler, or spider, a program that visits Web pages and scans their hyperlinks for more Web pages to visit. Crawlers, however, are problematic for both Web servers and users.

As crawlers proliferate, more and more crawlers visit each Web server, fetching the same pages over and over: Popular servers often see hundreds of crawlers every week [14]. This load can easily drain the networking budget of small companies paying by the bit for their Web service.

On the other hand, writing and running Web crawlers is nontrivial. Besides following `robots.txt` conventions, Web crawlers also must make sure not to overload Web sites by requesting pages from them too frequently, or overload the crawler's local network with Web page traffic and draw the wrath of the network administrators. As a result, Web crawlers must be free of bugs that may cause such overload, and typically require expert staff to be on call and field complaints whenever the crawler runs.

In this paper, we consider an alternative to large numbers of individual Web 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, as illustrated in Figure 1. In this alternative, clients do not run individual crawlers as they otherwise would have done; they simply subscribe to the Web data they need. For example, clients may be interested in a handful of particular

Web sites, or in all pages hosted in the `.edu` domain. Web servers, on the other hand, receive requests only from the central crawler (the multicast server) representing the requests of all its clients. Lastly, the network is used more efficiently, since it is better to multicast a single copy of a page P than to have every client fetch a separate copy of P individually.

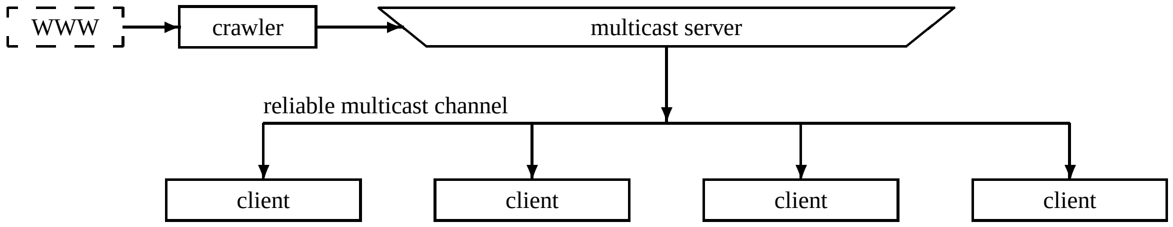

Figure 1: A Multicast Facility

In the real world, we would expect more than one such "central" crawler and multicast facility in the world. Having multiple facilities allows clients to cope with the loss of one facility by resubmitting their requests to a different one. Also, clients could choose a "nearby" facility for improved service.

We are in the process of building such a Web repository and multicast facility, which we call WebBase. WebBase currently holds over 120 million Web pages, and provides a way for local clients to access and analyze the data [10]. We would like to extend it to support requests from remote clients, and to efficiently combine and multicast the required data.

Clients may desire to keep their requested Web images up-to-date (matching the live World Wide Web) by maintaining a subscription to a WebBase multicast facility. This leads to several questions we need to address in the design of the WebBase multicast facility:

- How should we measure the performance of our WebBase multicast facility? We have a number of metrics at our disposal. We might measure the "lag" between when a change happens on the live Web and when clients are next synchronized with the Web. Or, we could measure the lag between a change and when the change is next broadcast. We can also use metrics described in [5], freshness and age, which measure how up-to-date a client's image is at each point in time. We will see how the choice of metric affects the design decisions we make for the facility.

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

While there is existing work in multicast systems, such as that of broadcast disks, our multicast scenario diverges from prior work significantly. Because our data is updated over time, our metrics—and our design—must take the freshness of clients' data into account. Also, our scenario can accommodate a mix of not only subscribers keeping up with changing data, but also downloaders seeking a single copy of their requests in minimum time, introducing new complexity to our system. We survey related work more fully in Section 7.

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

2 Model and Metrics

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

Let us assume a client requests to track, or subscribes to, N data items, which we can number 1 to N. (Intuitively, these data items are Web pages for our WebBase facility.) This means the client is connected to the multicast facility, and strives to keep up-to-date copies of its N data items. If a client does not already have a copy of a particular data item, the client effectively requests a copy of the data item for the first time. To the multicast facility, a client not having a data item is similar to the client having an out-of-date data item; in both cases the up-to-date item needs to be sent to the client.

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

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

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

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

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

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

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

$F = \frac{1}{T} \sum_{t \geq 1}^{T} \frac{1}{N} \sum_{i=1}^{N} \delta(W_{i,t} - C_{i,t})$

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

For example, if a client is tracking four data items over time 1 to $T = 10$, starts with all of them up-to-date, and has one of them fall out of date halfway in time, the client’s time-averaged freshness is 7/8, or 87.5% (half the time it was 100% fresh, the other half it was 75% fresh).

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

$F_T = \frac{1}{N} \sum_{i=1}^{N} \delta(W_{i,T} - C_{i,T})$

We can use this notion of freshness to “sample” the freshness of a client after its transient initial behavior. For example, in the time-averaged freshness example above, the freshness at time T is 75%.

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

$A = \frac{1}{T} \sum_{t \geq 1}^{T} \frac{1}{N} \sum_{i=1}^{N} (1 - \delta(W_{i,t} - C_{i,t}))(t - \max\{u | W_{i,u} = C_{i,u}\})$

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

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

These definitions of time-averaged freshness and time-averaged age are designed to be equivalent to the notions of freshness and age in [5].

age the time since last modification of each of the client’s items, averaged over all items at the end of an interval of time: $A_T = \frac{1}{N} \sum_{i=1}^{N} (1 - \delta(W_{i,T} - C_{i,T}))(T - \max\{u | W_{i,u} = C_{i,u}\})$

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

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

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

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

To generalize our one-client metrics to metrics for clients of a multicast facility as a whole, we can compute the metric for each client, then combine the results in two intuitive ways. We assume that, as before, we are considering each metric over a fixed discrete time from 1 to T. For simplicity, let us say further that there are a fixed number of clients, c, that are subscribed during the entire time. (In practice, we consider a client's connection or disconnection an instantaneous event, so we can split up the time during which a multicast facility runs into blocks of time during which the number of clients subscribed is constant.) Each client i requests N_i data items, and has score M_i for a one-client metric. We can then define an aggregate metric M in one of two ways:

over clients One way to define the global version of each metric is simply to average (or sum) each one-client metric M_i over all clients: $M = \frac{1}{c} \sum_{i=1}^{c} M_i$

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

over data There is an alternative corresponding global version of each metric, defined as the *weighted* average (or sum) of the one-client metric over all clients, using the number of items in each client’s request as its weight: $M = \frac{1}{c}\sum_{i=1}^{c} M_i N_i$

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

We can also measure the network usage of the multicast facility, and use it as a metric; unfortunately, this is not particularly enlightening because a multicast facility striving to serve its clients will always be using the network whenever there is new, requested data to distribute. As such, the network usage of the facility depends primarily on how much data clients request and how frequently such data changes, rather than on any particular server’s design.

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

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

Example. Here is an example showing why unpopular items should be abandoned to maximize freshness. ([5] demonstrates why too-frequently changing items should be abandoned, so we will omit such an example here.)

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

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

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

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

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

3 Scheduling

Now, we turn to considerations on the server side of the multicast facility. How should the multicast facility schedule which data item to disseminate next?

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

Let us begin by adapting a number of heuristics designed for a related (simpler) multicast facility. In the simpler multicast facility, clients request a subset of a multicast server's static data items, and then connect to a shared multicast channel until they receive a copy of their data from the multicast server on that channel. The simpler facility is similar to the multicast facility we describe here, except that the data never changes in the simpler facility, so clients only stay connected long enough to get a single copy of the static data they need. We call such clients "downloaders" here, in contrast to the "subscriber" clients who stay connected to keep their data up-to-date.

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

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

R/Q In the R/Q heuristic, for each data item i, the server determines R_i, how many clients are requesting the data item i, and Q_i, the size of the smallest outstanding request for a client requesting that data item i. The heuristic is shown in Figure 2; the ***foreach*** *client* c loop determines the correct values for R_i and Q_i.

The server then computes the R_i/Q_i ratio (denoted RQ_i in Figure 2) for each data item i, and sends an item with a maximal ratio. In practice, computing each R_i/Q_i ratio for every data item at every time slot proved computationally intensive, so in our study we do not implement R/Q this way. In our study, we maintain a heap of R_i/Q_i values and update those values only when it needs to be changed (for example, because a new client requests the item, or because the client with the smallest pending request that includes this item has just had its pending request changed by new data changes or updates). This makes R/Q fast enough to use, because it needs only remove the top of the heap to make its decision.

In work on the simpler downloaders-only multicast facility [13], R/Q proved an effective way to minimize the average delay of downloader clients.

```
Input: A set of clients and their still-pending requests; Current time $t$
Output: The data item to transmit
    $ChunkToSend \leftarrow$ null;
    $MaxRQ \leftarrow -\infty$;
    foreach data item $i$ // Compute $R_i$ and $Q_i$
        $R_i \leftarrow 0$;
        $Q_i \leftarrow 0$;
        foreach client $c$
            if $c$ requests $i$
                $R_i \leftarrow R_i + 1$;
                $Q \leftarrow$ number of items $c$ still has pending;
                if $Q < Q_i$
                    $Q_i \leftarrow Q$;
        $RQ_i \leftarrow R_i/Q_i$;
        if $RQ_i > MaxRQ$ // Save item with largest $R_i/Q_i$
            $ChunkToSend \leftarrow i$;
            $MaxRQ \leftarrow RQ_i$;
    Choose $ChunkToSend$.
```

Figure 2: R/Q Heuristic

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

This heuristic requires the computation of many R_iW_i products, because the product changes for every requested data item at every slot of time. This made the heuristic very computationally and memory-access intensive given the scale of our study, much more so than the other heuristics described here. The published way [3] to most substantially increase the speed of RxW is to approximate the heuristic (computing a few, but not all, of the R_iW_i products to find a possible, not definite, maximum), but doing so changes the heuristic and affects its performance. Because of the time it takes to run this heuristic, we did not include it in our study.

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

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

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

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

This leaves us with the heuristic shown in Figure 3, which we call RxC (“C” stands for the (estimated) time of next change). For each data item, this heuristic tracks R, the number of clients requesting the data item, and estimates C, the time between updates to the data item.

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

In Figure 3’s **Initialize** function, we initialize the array of C_i, which holds the server’s estimate for how long the data item i will remain unchanged. Before a data item changes for the first time, the server cannot guess how long the data item will remain unchanged, so we declare it to be infinity. (If the data item never changes, this guess is correct. If the data item changes frequently, this guess will be quickly corrected by the first change in the data item.)

Once a data item changes, we can create an estimate: the first time the data item changes, the time until that change is recorded as the estimate. For subsequent changes to the data item, we take the weighted average of the current estimate and the time interval until the latest change, recording the average as the new estimate. This occurs in Figure 3’s **UpdateC** function. The time interval until the latest change is computed and stored in $ThisChangeTime$, and used as necessary to update our estimate C_i. The average is weighted with a fixed (predetermined) weight w for the old estimate and $1 - w$ for the new data point. A larger w would yield an estimate less vulnerable to normal variations in the intervals between changes to a data item. A smaller w allows the estimate to more rapidly track changes in the change rate of the data item.

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

Finally, **RxC** actually computes the R_i times C_i product for each data item i, to send a data item with a maximum product. There is a special case to consider in this process: If any C_i are infinity, then item i’s corresponding RxC

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

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

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

Figure 3: RxC Heuristic

product would be undefined (infinity), making further comparison to other values (in particular, other infinities) impossible. The server therefore considers data items that have never changed specially: For data items that have never changed, a more popular data item is defined to have a "higher" RxC product than a less popular one.

As for R/Q, computing RxC for every data item at every time slot was prohibitively time-consuming, so RxC is also slightly optimized for our study as R/Q was. In our study, we maintain two heaps, one of finite RxC products, and one of the popularity (R value) of the still-unchanged data items. We update values in the heaps only as necessary, moving items from the unchanged-data-item heap to the finite-RxC heap as appropriate. This makes RxC fast enough to run, because RxC now needs only examine, at most, the top of two heaps to make its decision.

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

4 Simulating the Heuristics

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

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

Table 1: Simulation Parameters and their Base Values

how we determine their values for the Web, and how we scale those values for our simulation. In Sections 5 and 6, we present our results.

4.1 About the Web

To begin, we determine a few statistics about the Web. Because it is difficult to capture the structure and nature of the large and growing Web, we will complement the measured numbers with some estimates to create our model for the Web. The values we use for the Web appear as “Web value” in Table 1.

We find from [5] that a large fraction of pages (nearly a quarter) change at least every day, pages that we will not attempt to track in this system. (It is not practical for us to constantly crawl a large number of popular pages, much less repeatedly disseminate them over multicast as they change throughout a day. A client that wishes to track such pages will have to crawl them independently, as it needs them.) We will estimate the change frequency of the remaining pages, then—the ones we can attempt to track. We can observe from [5] that the mean change interval for the remaining Web pages is roughly 75 days.

Estimates of the number of pages on the Web varies depending the source of the statistic, but a fair number is around 1.3 billion (thousand million) hypertext pages, as of the turn of the century. Google [8], in particular, claims to have seen and indexed around that many pages for its search engine according to its root page. Meanwhile, Hobbes’ Internet Timeline [11] claims there were 21 million Web sites around that time, where a Web site is defined by domain name and port number (that is, a Web site is defined by shared elements in the URL, rather than any judgments about the content or authorship of the available Web pages themselves).

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

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

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

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

4.2 About the Multicast Server

Our experience with WebBase's crawler suggests we can fetch about 6000 pages per minute without taking down the local network, a speed that seems reasonable given other published crawler speeds (such as DEC Mercator [9]). This means the crawler will take just over 150 days to crawl the 1.3 billion pages of the Web once.

Assuming that we have 10BaseT Ethernet as a bottleneck connection speed for our wide-area multicast, we might safely consume 1 megabit per second of network usage to disseminate our data. Given this number, we are able to send about 3000 compressed Web pages per minute. (This is calculated in [13] as follows: 2.6 kilobytes per page compressed, times 3000 pages, yields 7800 kilobytes of data to send over 1 megabit per second. Such a transmission takes about 61 seconds.)

In summary, the multicast server takes about 150 days to refresh its image of the Web, and can disseminate a compressed version of those Web pages in about twice that time.

4.3 About the Clients

How much data does a client request? Because such a multicast service does not exist in wide deployment, it is difficult to know for sure. Instead, we must make an educated guess to get an initial value. We consider a hypothetical scenario in which clients are interested in maintaining topical subsets of the Web for search or data mining, and so would request all the Web *sites* on a particular "topic," such as "college sports" or "MP3." (Because in hypertext, select Web pages in isolation are probably less useful than the full Web sites on a topic, we assume clients to request entire Web sites.)

We find some popularly requested topics on the Web, estimate their sizes in number of sites, and determine a size of about 9000 Web sites for a few typical topics. (The determination of this value is described in detail in the Appendix.) So, we will use this number as our base value for the average client-request size. To contain the complexity of the simulation, we say a client requests these Web sites uniformly across the Web. That is, for our base case, we assume that a Web site about "college sports" (for example) is generally not a Web site about "MP3," and as such, it would not be requested more often than usual by many clients of different interests. We do, however, consider the performance effect of client-request skew later in our results.

4.4 About Chunks

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

The numbers we have gathered so far ("Web value" in Table 1), however, are all (not surprisingly) large. As a result, a multicast facility on current equipment may be unable to run efficiently when scheduling the multicast of each Web page individually. This is because of two factors in unison: One, it often takes more time to schedule more data objects. Two, the more data objects the server schedules, the smaller each data object is, and the less time it takes to transmit, so a multicast server has less time to schedule each of the data objects.

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

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

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

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

4.5 About the Simulation

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

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

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

Table 2: Optimizing for Subscribers' Freshness Over Clients

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

To preserve the time scale from distortion, we rescale the network resources available to the multicast system by the same factor. As a consequence, the Web still takes 150 days to crawl, and a chunk now takes five minutes to transmit.

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

Table 1 summarizes all these simulation parameters in the "Simulation value" column.

5 Scheduling Subscribers

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

To illustrate, let us first focus on optimizing our facility for the freshness over clients metric. This metric represents the fraction of a client's subscribed data that is expected to be fresh, on average. Based on our experiments, we present in Table 2 approximate guidelines for the choice of scheduler in a multicast facility optimizing freshness over clients. The table shows the recommended heuristic depending on the number of clients in the system (columns) and the clients' subscription size (rows), as determined by our experiments. In this and subsequent tables, "Circ" refers to the circular broadcast heuristic; "Pop" refers to the popularity heuristic, and "R/Q" refers to the heuristic of same name.

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

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

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

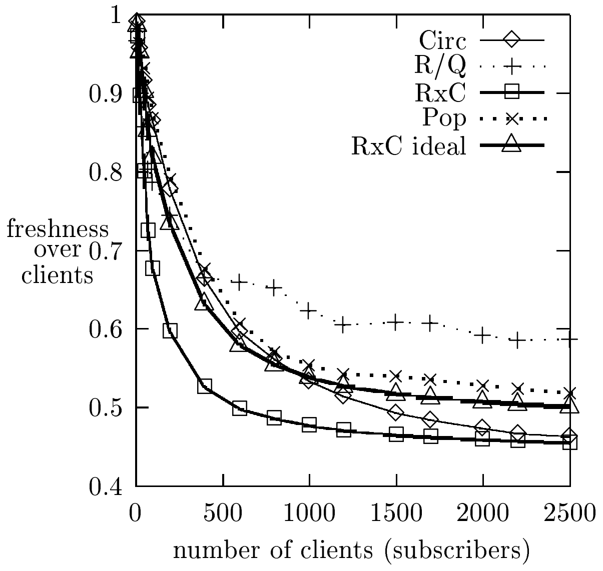

Figure 4: Freshness over clients for subscribers

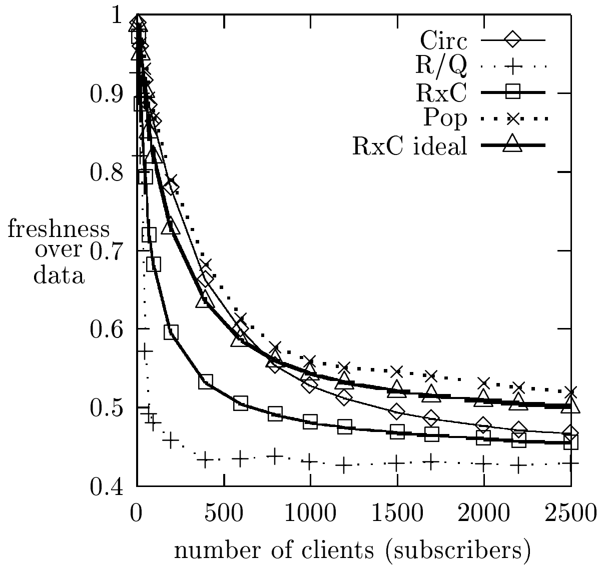

Figure 5: Freshness over data for subscribers

For lack of space, we are unable to present all the experiments whose results are summarized in the table. So, for illustration, we focus on the parts of the table marked (1), which are plotted in Figure 4. In Figure 4, we see the performance of our multicast facility using our various scheduling heuristics, for various numbers of subscriber clients. In this and subsequent figures, "Circ" refers to the circular broadcast heuristic; "Pop" refers to the popularity heuristic; "R/Q" refers to the heuristic of same name; and "RxC" refers to the RxC heuristic, with its weight factor arbitrarily chosen as $w = 2/3$. (The effect of w and the plot "RxC ideal" are explained later.) Recall from our simulation values that in our base case, a client requests on average 9 out of 85680 available (about 0.01% of available) Web sites.

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

Clients start the simulation with ideal freshness (all fresh data), so this plot measures the freshness clients *lost* during their subscription. (Clearly, with more data on the Web changing than the multicast facility can disseminate, clients' freshness cannot remain perfect over the course of their subscription.) Starting with fresh data allows us to differentiate the performance of the schedulers more clearly, as mentioned in Section 2.

For example, if our multicast facility has a thousand subscribers, a client starting with fully fresh data would end the simulation with an expected 48% of its data fresh if the server used an RxC scheduler, and 53% if the server used a circular broadcast scheduler. If the server used the popularity scheduler instead, such a client would have an expected 55% of its data fresh. Lastly, if the server used R/Q, a client would have an expected 62% of its data fresh at the end of 75 days.

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

From this figure, we can draw several conclusions. One is that, for freshness over clients, R/Q is a good heuristic to use when the multicast facility is providing data for over 400 clients. (This portion of the figure corresponds to the upper-right cell of Table 2.) This is because R/Q uses as a weighing factor the amount of data (effectively, the number of Web pages) a client needs to return to ideal freshness. It turns out this property of R/Q allows R/Q to favor smaller clients and boost their freshness, as we now see.

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

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

After running many more simulations like the above, varying a number of other parameters, we determine the following guidelines to optimize for freshness over clients, as summarized Table 2:

- For our base parameter values and well under 400 clients, circular broadcast or popularity suffice as well-performing schedulers. This is the part of the Table 2 marked (2), and shown in Figure 4. At 100 clients, for example, circular broadcast and popularity are near 86.5% freshness over clients with circular broadcast doing slightly better, while the next-best performer R/Q is at 78.7% freshness.

- As detailed in the text and figures above, if the multicast facility is providing data for a large number of clients (starting at over 400), R/Q becomes a good choice of heuristic. That is, the results favoring R/Q in the upper-right cell of Table 2, as shown in Figure 4, extends for the entire the right column of Table 2.

- If clients request, on average, larger numbers of Web sites (greater than fifteen, versus the nine-site default), popularity is a good choice. For smaller request averages, circular broadcast does a little better (at base parameter values, for example, circular broadcast edges out popularity by just under one percent freshness), and for very

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

Table 3: Optimizing for Subscribers' Freshness Over Data

large numbers of sites (above 75 per client) R/Q does just as well, but popularity is competitive at both ends of the spectrum and is the best performer for the values in-between (15-75). At an average of forty Web sites per client, for example, popularity holds about 85% freshness over 100 clients after 75 days, beating R/Q's second-best value of 81%. At an average of a hundred Web sites per client, R/Q begins to take a small lead, at 74% freshness over popularity's second-ranked 73%. Considering the relatively small difference by which popularity loses at such extreme values, it is not much loss to approximate the left column of Table 2 as "use popularity."

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

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

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

Let us now turn briefly to freshness over data. Our results for this metric are summarized in Table 3. As we can see, the popularity heuristic is a good choice for this heuristic. This is because, as we observed while describing and evaluating the metrics, a freshness metric averaged over data favors popular items, just as the popularity heuristic does. Consequently, the popularity heuristic is a good match for this metric, as we have observed in simulations over a variety of conditions.

Also, we observe that R/Q's favor toward small-request clients is completely unhelpful if the metric to optimize is freshness over data. In fact, if we turn to Figure 5, we see that this is exactly the case. In this figure, the vertical axis is now freshness (F_{75days}) over data, so that clients with large requests get more weight in the average. The simulated scenario is the same as in Figure 4. This means, for example, that for a system with a thousand clients using the R/Q heuristic, just under 43% of all client data is fresh at the end of 75 simulated days. More notably, R/Q is the worst performer for freshness over data, because of the differences in the two freshness metrics. This underscores the importance of choosing the appropriate metric for each application before making design decisions around it.

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

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

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

- RxC does not work. Surprisingly, though RxC was designed to optimize for freshness, it does not do particularly well. In none of the experiments we ran was RxC the preferred heuristic. To determine whether our choice of the heuristic's weight parameter w was hurting its performance, we create a variant of RxC in which an oracle provides the actual average rate at which each Web page chunk changes. (This was implemented by running a simulation, computing the actual average rate at which each Web page chunk changed, then feeding those averages back into a new run of the simulation using the same random seed.) We call this variant heuristic "RxC ideal" in Figures 4 and 5. As we can see in the figures, the oracle-assisted RxC performs better than RxC, suggesting that there is improvement to be gained in tuning w, but the improved result is still not enough to make an ideal RxC the preferred scheduler for this subscriber scenario.

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

  Another reason may come from grouping pages into chunks; RxC schedules chunks, whose change rate is affected primarily by its fastest changing pages. Thus, a chunk may appear to change rapidly, but actually have little change

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

Table 4: Optimizing for Subscribers' Age

over time because only one page of the chunk is changing frequently, and RxC would misjudge its value.

For the age metrics (age over clients and age over data), R/Q becomes a poor performer. From our simulations, we conclude the guidelines shown in Table 4 instead.

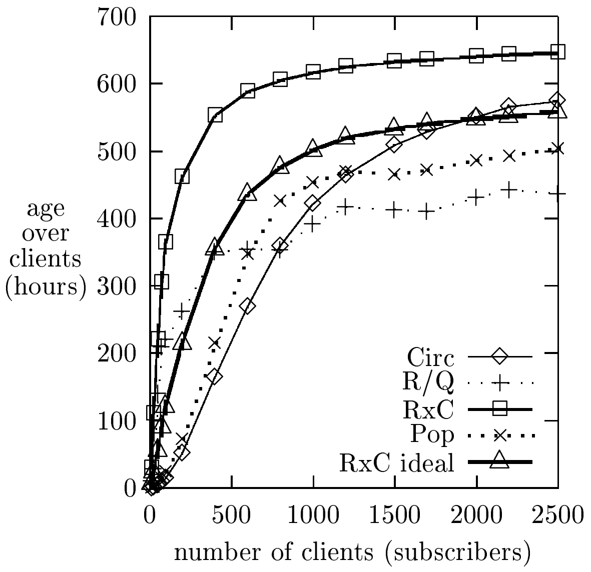

Figure 6: Age over clients for subscribers

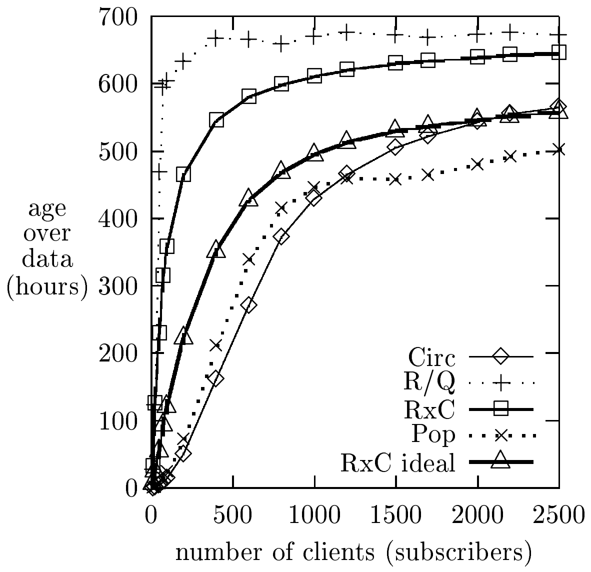

Figure 7: Age over data for subscribers

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

This means that the cell marked (3) in Table 4 represents R/Q if the desired metric is age over clients, and Pop if the metric is age over data. We see this in Figure 6, which plots the age over clients for varying numbers of clients, and Figure 7, which plots the age over data for varying numbers of clients. These plots span the cells marked (4) in Table 4.

Because the age metrics accumulate higher age for neglected data items, the circular broadcast approach, which does not neglect any data, is effective for a larger variety of conditions than it was under the freshness metrics. Still, it is hard to characterize in general the regions where each scheme does best under the age metrics, due to the complex interactions of factors such as update rate and distribution, number of items, number of clients, and size of subscriptions. Consequently, a designer whose scenario diverges radically from the ones we study here may need to empirically determine which cell in Table 4 best corresponds to the new situation.

6 Scheduling Downloaders and Subscribers

It is likely that in our multicast facility not every client will be a subscriber, able or wanting to stay connected to the multicast facility for updates. As we considered in [13], clients to a multicast facility may be *downloaders*, clients that connect to a multicast facility, request a subset of the facility's repository, then disconnect as soon as it has received all of its requests (at least once). Here, we would like to consider how the two types of clients—downloaders and subscribers—would interact in a multicast facility, and how we should adjust our scheduler to balance their different interests.

For this section, we define as in [13] the metric *client delay* or *delay* for downloader clients: the delay of a downloader client is the amount of time between the client's connection to the multicast facility (neglecting the time for the client to issue its request) and the time the client first has a copy (any copy) of *all* the data it requested. This definition is useful for a system like WebBase, in which our clients will typically need to have all the data available before doing batch indexing, mining, or analysis of the data. By definition, a downloader has no data when it first connects to the multicast facility.

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

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

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

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

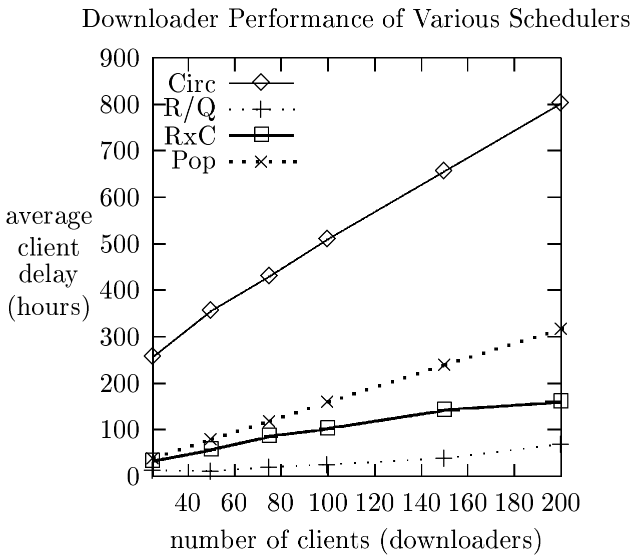

Figure 8: Client delay for downloaders in a mixed downloader/subscriber system

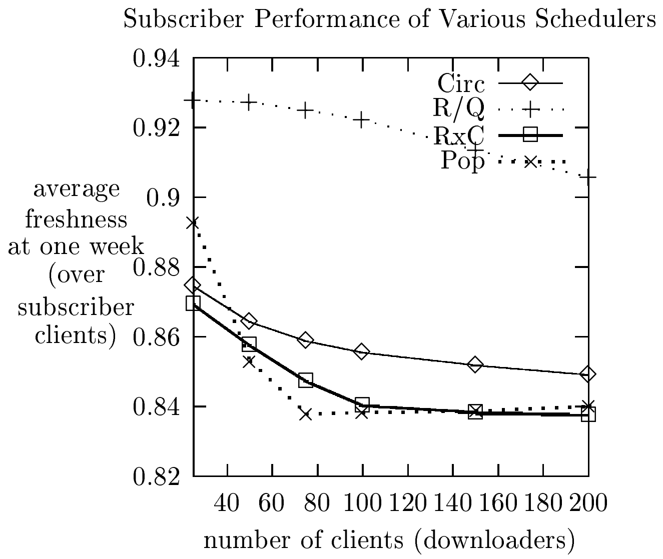

Figure 9: Client freshness for subscribers in a mixed downloader/subscriber system

In Figures 8 and 9, we see the performance of the multicast facility charted for downloaders and subscribers, respectively, given one hundred subscribers and, along the horizontal axis, a varying number of downloaders at simulation start. On the vertical axis in Figure 8 is the downloaders' delay, averaged over clients. On the vertical axis in Figure 9 is the corresponding subscribers' freshness at one simulated week.

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

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

Table 5: Optimizing for Client Delay and Metrics Over Clients When Downloaders Are Present

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

Table 6: Optimizing for Freshness Metrics Over Data When Downloaders Are Present

84% for popularity and RxC. The R/Q heuristic, in this case, achieves the lowest client delay for downloaders as well as the highest maintained freshness for subscribers.

Perhaps the most striking observation in this and other simulation runs not plotted here is that R/Q performs well for downloader client delay as well as both freshness and age metrics over subscriber clients, as suggested by the numbers above. The results for these three metrics, as summarized from a number of simulations, appear in Table 5.

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

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

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

- Popularity or circular broadcast usually optimize freshness or age over data, as shown in Table 6. It is difficult to characterize in general the regions where each heuristic does best, due to the complex interactions of factors from update rate and distribution to client request sizes to the proportion between subscriber and downloader clients. Consequently, a designer who needs to optimize for metrics over data, under a mixed subscriber and downloader scenario that diverges radically from the ones we study here, may need to empirically determine which cell in the above table best corresponds to the new situation.

We also found the following observations about this multicast scenario:

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

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

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

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 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 [16], 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 or freshness loss 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, [20] 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 3) 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 most of the heuristics we consider, without specific request information. In effect, we, too, assume that specific client-request information is available to our multicast facility.)

Still, in this scheduling work two lingering assumptions remain: clients request a single piece of data at a time, and clients request data only once (rather than requiring updates). This is true in much scheduling work for broadcast delivery in general, such as in [18], 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 [21] (which supposes client requests to have deadlines that may or may not be met). We remove the single-data-item-request assumption, and consider subscriber clients. 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 ([19], [17]), and of job scheduling in operations research ([7], [12]). 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.

Also, 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.

Lastly, subscriptions are studied in projects such as SIFT [22], where a server provides subsets of Usenet to subscriber clients who specify their interests, but in SIFT, the focus is on efficiently filtering data for clients' interests, on the assumption that the system can keep up with and broadcast the amount of information being requested and made available. In practice, we see that a Web multicast facility cannot keep up with all relevant updates on the Web as they happen, and must make quick choices to maintain the best performance that it can.

Also, [5] considers the problem of keeping Web data up-to-date, but does so for only a single client (a crawler) and assuming no knowledge of when updates actually occur in the source data. As such, this work could help keep our source Web repository up-to-date with respect to a live Web.

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. 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 either receive all the data they requested or subscribe to updates of the data to maintain its freshness. We consider a number of metrics by which we could measure the performance of the facility, and determine from numerous simulations the recommendations for best performance under each metric and under a variety of conditions. The results provide insights that are guiding the design of our own WebBase multicast facility.

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

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

[9] Allan Heydon and Marc Najork. Mercator: A scalable, extensible Web crawler. *World Wide Web*, pages 219–229, December 1999.

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

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

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

[13] Wang Lam and Hector Garcia-Molina. Multicasting a Web repository (extended version). Technical report, Stanford University, 2000. Available at http://www-diglib.stanford.edu/cgi-bin/get/SIDL-WP-2001-0151.

[14] Mike Lesk. Personal communication.

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

[16] 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.

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

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

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

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

[21] 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.

[22] Tak W. Yan and Hector Garcia-Molina. SIFT—A tool for wide-area information dissemination. In *Proceedings of the 1995 USENIX Technical Conference*, pages 177–186, New Orleans, 1995.

Appendix: About the Client Requests

In Section 4.3, we asked, how much data does a client request? Not having such a multicast service in wide deployment, we must take an educated guess as our initial value. We detail how we determined our estimate here.

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

For a randomly chosen day (17 Mar 2001), Lycos 50 [15] reported that the top five searches on the Lycos search engine were NCAA basketball, Dragonball, Napster, St. Patrick’s Day, and Britney Spears, in that order. The author then subjectively chose topics to enclose these searches: “college sports,” “anime” (Japanese animation), “MP3” (a popular form of compressed sound/music), “holidays,” and “pop music,” respectively.

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

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

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

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

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

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