Data Dissemination on the Web: Speculative and Unobtrusive

Institute for Advanced Computer Studies

Vincenzo Liberatore[1] Brian D. Davison[2]

May 10, 1999

[1]UMIACS, A. V. Williams Building, University of Maryland, College Park, MD 20742. E-mail: vliberatore@acm.org. URL: http://www.umiacs.umd.edu/users/liberato/.

[2]Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019. E-mail: davison@cs.rutgers.edu. URL: http://www.cs.rutgers.edu/~davison/.

Abstract

The rapid growth of the Web results in heavier loads on server/network and in increased latency experienced while retrieving Web documents. Internet traffic is further aggravated by its burstiness, which complicates the design and allocation of network components. Bursty traffic alternates peak periods with lulls. This paper presents a framework that exploits idle periods to satisfy future HTTP requests speculatively and opportunistically. Our proposal differs from previous schemes in that it is explicitly aware of current HTTP traffic loads so as to be unobtrusive. This paper highlights several design trade-offs and details the problem of server arbitration among several candidate documents. We present a theoretical analysis of arbitration and validate it by extensive simulation on server logs, in which we calculate latency experienced by clients. Perfect traffic shaping during peak periods is observed and substantial latency improvements for non-dynamic documents are reported over pure on-demand strategies.

1 Introduction

Internet traffic has experienced exponential and sustained growth during the past few years. Increased traffic results in heavier load on servers and on intermediate routers and links. Because of heavy traffic, users perceive significant latency while retrieving Web documents. Moreover, data demand at a server is essentially bursty [30]. Traffic burstiness complicates the design and allocation of effective network components and contributes to the perception that the Internet is an unreliable system. Bursty traffic alternates lulls with spikes of extreme activity. On the negative side, high traffic spikes lead to significant delays and to difficult network design problems. On the positive side, traffic lulls give the chance of speculatively executing alternative or background activities. In this paper, we describe a protocol that exploits traffic lulls unobtrusively and speculatively in order to anticipate future HTTP requests. We simulated our architecture on actual Web traces and calculated the stretch [3] and latency experienced by clients. Our architecture provided up to a factor of 1.6 speed-up over the traditional (on-demand) strategy. It also led to perfect traffic shaping during peak periods, when the server transmits a constant amount of data per unit of time.

Much previous research has examined load-aware protocols at the transport layer. For example, TCP adapts its transmission rate to estimates of network load. At the application layer, a client can exploit periods of local inactivity to speculatively load documents that will be likely needed in the near future (*prefetching*). If prefetching is not aware of network load, it can lead to substantial increases in traffic burstiness and network delays [13]. Prefetching can be effective when restricted to documents that are very likely to be accessed [12, 20] or by using a transport-layer mechanism to limit datagram transmission rate [13]. However, all prefetching schemes thus far can issue prefetching requests even when the server/network is overloaded and, conversely, can refrain from prefetching even when the server/network is idle. In this paper, we explore a solution that is based on an intuitive *principle of unobtrusiveness*: speculative data dissemination should occur if and only if it can be supported by underutilized network components. In practice, complete unobtrusiveness cannot be achieved because additional traffic always imposes some load on servers, network, and clients. However, our architecture carefully prioritizes resources against speculative data dissemination and is (almost) unobtrusive from the viewpoint of on-demand traffic. In practice, an UDiD (Unobtrusive Dissemination of Data) server proactively pushes documents to selected clients via low-priority datagrams whenever the server is idle. Our framework exploits priorities in the network-layer protocol (IPv6), and shifts prefetching decisions from clients to servers. Server lulls are fully exploited, so that UDiD is unobtrusive with respect to the server. Moreover, low-priority load can be the first to be shed so that intermediate network components are not overloaded by UDiD packets, and so UDiD is unobtrusive with respect to the intermediate network components. As an alternative to network-layer priorities, distributed systems could use either control packets, per-packet processing, or coordination among clients, servers, and intermediate components to determine and exploit global network lulls.

UDiD design and implementation poses a number of serious practical and theoretical issues at the server and at the client sites. We will specify a system architecture, and identify several issues and trade-off in its design. This paper will then turn to the fundamental problem of server arbitration (defined below). Several other issues are sketched, but deserve more work.

A server's first problem is to decide which documents to push to which clients. If the server has more than one candidate document, server's bandwidth is exposed to contention among candidate documents. A server should push documents that are likely to be used by clients in the future, and so it should predict future access patterns. Although estimates of future usage are crucial, there are also other factors that servers must take into account when they decide which document to push.

For example, some files could be marked as unsuitable for push because of security or policy reasons. In that case, the server is forbidden from pushing an otherwise valid document. However, servers could face a situation where a popular document has no restrictions from push, but should not be sent to clients solely for performance reasons. The following elementary example illustrates such a situation: the set of candidate documents could consist of a long and hot document along with several shorter documents that are not quite as popular. If the server pushes the long document, such choice results in a hot document being prepushed, but it consumes server time and bandwidth and results in a heavy load on the receiving client and on the network. If the server opts in favor of short documents, the server could be able to push several documents, but those documents would not be very valuable. The example demonstrates a general trade-off: server's optimal decision depends both on document (estimated) usefulness as well as on the (estimated) load that documents would impose on the server and on the network. Fan *et al.* propose server-initiated prefetching in the context of a proxy server handling several dial-up, low-bandwidth connections, and emphasis is given to high accuracy prediction [15]. In UDiD, contention among documents is crucial and the focus shifts from accurate prediction to the quest for a balance between prediction and resource consumption. Such problem will be henceforth referred as to *arbitration*. In this paper, we will give a theoretical treatment of the arbitration problem under the simplifying assumption that clients generate requests according to a Poisson process, and we will validate our theoretical results against algorithm performance on actual Web traces. Our theoretical analysis is of independent interest and can be applied to systems other than UDiD (e.g. [15, 29]).

The paper is organized as follows. In section §2, the architecture of the UDiD system is described, and basic design trade-offs are discussed. In §3, we give a discussion and theoretical analysis of the *arbitration* object that decides at the server site which documents are pushed to which clients. In §4, simulation results are reported. In §5 related work is summarized and §6 concludes the paper.

2 System Architecture

The UDiD system is composed of objects that are located at the server and client sites and that exchange Web documents and control information. The client has a *control* object that handles the interaction with the server and manages the local *cache*. The server *accepts requests* that are originated by clients and collects statistics on document usage. Moreover, the server maintains *client agents*, which provide on demand estimates of future client access patterns, cache utilization, and consistency state of the corresponding client. If a server receives a pull request, it will queue it. When the queue is underutilized, the *arbitration* object is activated. The arbitration objects maintain a set of candidate documents, one for each agent. The arbitration iteratively chooses to push a candidate document to the corresponding client and then solicits a new recommendation from the agent of that client. A UDiD client starts the UDiD protocol by requesting that the server create a client agent. The initial agent activation request is accompanied by parameters that describe the client behavior, such as client cache size and consistency protocol description. In practice, a client starts an agent by invoking a server script with the parameters that describe the client. Agents are killed by the server after the client has been inactive for a certain period of time. Pull requests are ordinary HTTP requests. Pushes can be aborted by servers and partially pushed documents are discarded by the client. If an agent is live, the server will insert a special field in the cache control extension header of ordinary pull requests. If a client detects that its agent is dead or if it desires to change agent parameters, it can piggyback new agent descriptors on the next pull request. Finally, the protocol can be implemented with no change to browsers if there is a proxy

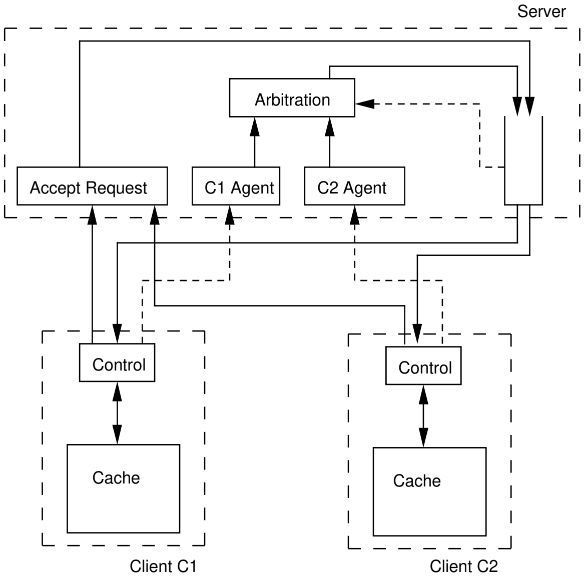


Figure 1: Architecture of the UDiD system. The picture demonstrates an example with two active clients C1 and C2.

that complies with the protocol on behalf of clients.

A thorny issue is how much interaction there should be between clients and their agents. Obviously, a strict interaction between clients and agents allows the server to estimate more precisely the state and needs of each client, but it also implies additional network load. Currently, in UDiD, agent-client interaction is limited to agent creation or modification. As a result, agent-client interaction is almost completely unobtrusive as it can be piggybacked on pull requests. Performance improvements could stem out of tighter agent-client coupling, as for example clients can attempt to communicate cache hits to their agents, but such interaction strategies are left to future development. A related issue is the complexity of the agents. If agents are complex, the server has access to more accurate predictions, but it also suffers the corresponding computational load. There is also a trade-off between complexity of the agent and amount of interaction, as a complex agent can reconstruct most of the client state without explicit interaction. Such issues deserve more work and are beyond the scope of this paper.

The principle of unobtrusiveness demands that pushes should occur only when the server, as well as the client and the network, are idle. A possible way to consider client lull is to have additional information exchanges between server and clients. However, such a solution brings in additional network load, ignores the load of intermediate network components, and client information could be out-of-date by the time it reaches the server. An alternative is to have every network component execute per-packet processing (active networks). The current UDiD architecture is based on the choice of a protocol for pushed documents, and, specifically, it uses UDP over IPv6. Pushed datagrams should abide by the principle of unobtrusiveness, and so resources should be prioritized in favor of pulled datagrams. Hence, the choice of pushing documents with very low priority (0 or 1) by means of IPv6. A low priority connection between the server and the client could take a large amount of time. On the other hand, a pushed document is not transmitted on demand and its usefulness is speculative, so that it is not critical that the pushed document reach the client under all circumstances. A connectionless protocol delivers pushed documents on a best-effort basis and frees the server to attend to other pull or push requests as soon as the document has been sent. Moreover, the server can achieve constant transmission rate while pushing documents. Another issue is the transmission rate of the UDP protocol that is sustainable for the client and for the network. UDiD estimates push transmission rate to a certain client from the length of time required to satisfy the last pull request from that specific client.

3 Server Push

Access Pattern Prediction. Client agents propose a set of candidate documents to the arbitration object upon demand. This agent candidate set is based on agent predictions of future accesses from the corresponding client. First, servers must decide if a client will generate any future requests. Currently, servers use a simple time-out mechanism to phase out client that have not been active for a certain period of time. If the server estimates that a client is indeed active, it must predict future access patterns for that client. We implemented a prediction scheme based on maximum likelihood first-order Markov chains. Similar schemes have been used in much previous work on prediction for prefetching [5, 6, 12, 13, 20, 24]. The server maintains a counter c_{pq} of the number of transitions from document p to document q and increments it whenever a client request for document q was preceded by a request for document p. The counter c_{pq} contains a value that at all steps is proportional to the maximum likelihood estimate of the probability that a client accesses q after p. More complex prediction schemes are possible [10, 15], but since they are not unique to UDiD, they will not be discussed further in this paper. In the UDiD prototype, agents are simple because they do not keep complete information on client cache state and do not perform a complicated traversal of server documents. A client agent is fundamentally an iterator that determines the last document p pulled by the corresponding client and returns to the arbitration object candidate documents in decreasing order of next access probability.

Arbitration. When the arbitration object detects that the queue is empty, it obtains one guess from each client agent. It then decides which document to push. Servers should limit the total size of pushed documents depending on client cache size: if between two client requests the server pushed more data than the client cache can contain, the client would not be able to keep all pushed documents in its own cache, and bandwidth would be wasted. UDiD terminates push operations according to a simple eager strategy: push operations are aborted as soon as either (1) a pull request is received by the server from the client to which the page is pushed, or (2) it is detected that on-demand connections require more bandwidth. In case (2), if more than one push is concurrently being pursued, the server aborts the one(s) with longest remaining time to completion. Variations of the eager scheme could be beneficial, but are beyond the scope of this paper.

In general, several client agents are live and so there are several documents that are candidates for pushing. Each alternative takes a certain (estimated) amount of time and resources to be pushed. A document could be very likely, but also too long to be pushed before the end of the current lull. Therefore, the arbitration choice is affected by document probabilities as well as an estimate of the duration of idle periods. Obviously, if the actual lull duration exceeds server estimates, the server can exploit the additional time to push other pages. Although in general servers could explicitly estimate idle period duration, the probabilistic analysis below will show optimality for a strategy that does not explicitly use any such estimate.

A formalization of the problem is obtained by assigning to each document a *value*, corresponding to the expected performance improvement obtained by pushing that document, and a *length*, corresponding to the expected length of time necessary to push that document. In order to obtain a theoretically analyzable model, we assume that the set of candidate documents is not updated during the lull with new agent recommendations and that at most one push is pursued at each point in time. We remark that such assumptions are made only for the purpose of analysis, and that we simulated and implemented a complete system where more than one push can occur at one time and where candidate pages are replaced by new recommendations, as described in §2. In the theoretical model, the problem is to find a set of documents that maximizes the sum of the values of the chosen documents while their total length is no more than a certain deadline D. If values, lengths, and deadlines are known at the beginning of the lull, such problem is exactly the *knapsack problem*, which is well-known to be NP-hard [17]. In actual arbitration, values, lengths, and deadlines are not known exactly at the beginning of the lull, and estimates can be obtained as follows. A length is the amount of time needed to transmit a document via UDP, and so lengths can be estimated from document size and available bandwidth. The value of pushing document q to client c is the expected performance improvement obtained by pushing q to c. If q is pushed, received by c, and subsequently requested by c, then the server saves a pull of q. Hence, document value depends on three factors: (1) the probability $\rho_c(q)$ that q reaches c, (2) the probability that c subsequently requests q, and (3) the length $l_c(q)$ of q with respect to c. Under the prediction scheme in the previous paragraph, the quantity $\pi_{pq} = c_{pq}/\sum_r c_{pr}$ is the maximum likelihood estimate of the probability that a client c that requested p will request q. Different estimates of π_{pq} can be obtained with other prediction algorithms and would not change the following theoretical analysis. An estimate of the expected value of pushing document q to client c is then the product $\rho_c(q)\pi_{pq}l_c(q)$. An important quantity is the value per unit of length, which will be referred to as *value density*. The value density is exactly equal to the the probability $\rho_c(q)\pi_{pq}$. A special case of the arbitration problem arises when all value densities (probabilities) are equal. Such case is interesting in theory, because it corresponds to the *subset sum problem*, which is also NP-hard [17], and in practice, if the difference between probability estimates are statistically insignificant. Finally, the deadline D is the amount of time before a pull request arrives and forces the server to abort the push. Therefore, D follows the same distribution as request interarrivals. For example, if pull requests were generated by a Poisson process with rate λ, then D would follow an exponential distribution with mean $1/\lambda$, that is, $Pr[D \geq t] = e^{-\lambda t}$. Given estimates of document lengths $l_c(q)$, values $v_c(q)$, and a distribution on D, the arbitration object chooses a document q to push to a client c. The *value-density* strategy (VD) pushes documents in decreasing order of value density. VD is a natural algorithm that aims at maximizing the value transmitted per unit of time. However, VD could fail when valuable documents are longer than lull duration D. The *shortest-first* strategy (SF) pushes documents with positive value in increasing order of length. Obviously, SF maximizes the number of pushed documents, but it fails when short documents have little value. Actually, SF could fail even when all document densities (probabilities) are equal, because SF could waste time at the beginning of the lull to push short documents and be left with no time to push a longer and more valuable document later on. In fact, if SF were optimal when all densities are equal (subset sum), SF would be a polynomial-time algorithm for an NP-hard problem. However, it can be shown that

Theorem 1 *If value densities are equal (subset sum) and D is exponentially distributed, then SF maximizes the expected sum of pushed document values.*

The proof is postponed to appendix A. It is important to notice the strength of the theorem. First, SF maximizes not only the number of pushed documents, but also their total expected value. Second, SF is optimal for all rates λ although SF works independently of the value of λ. An interesting intuition stems out of the proof: when the deadline D is exponentially distributed, lengths are more important than values. Ultimately, this observation can be traced back to the

Trace name	**Requests**	**Hosts**	**Documents**	**AvgBW** (B/s)	**WkBW** (B/s)	**rTime** (s)	**No-Push**
epa-http	47721	2333	4828	3609.54	6428	24	14.5%
nasa (Aug 4)	59557	4191	2369	12844.3	20082.1	25	0.1%
nasa (Aug 14)	59874	4454	2257	12541	18452.4	23	0.1%
cs.edu (Dec 18)	16548	1407	3644	1638.72	2904.1	21	2.0%

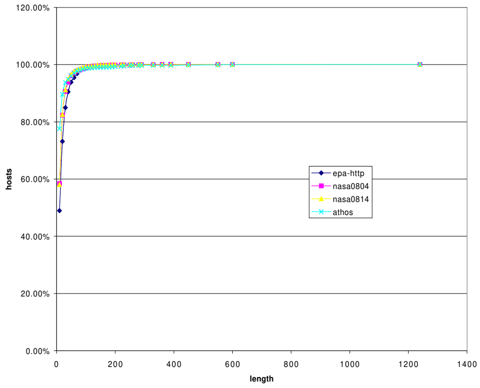
Table 1: Characteristics of the Web traces. Requests is the trace total number of requests, Hosts is the total number of distinct hosts making requests, Documents is the number of distinct Web documents in the trace, AvgBW is the average number of bytes requested per second, WkBW is the average number of bytes requested per second from 9AM to 5PM (server local time), rTime is the average time between two requests from the same host (intervals longer than 5 minutes have been disregarded), and No-Push is the percentage of bytes in files that are marked as unsuitable for push over total traffic.

fact that the probability distribution for $D \geq t$ drops exponentially, i.e. extremely rapidly. SF is a simple algorithm that does not require any value estimate beyond $v_c(q) > 0$. Finally, the theorem holds also when value densities are only approximately equal (further details and proof are omitted). In reality, value densities could differ and request arrivals could be non-Poisson [30]. An empirical analysis of VD and SF on actual Web traces is presented in the next section.

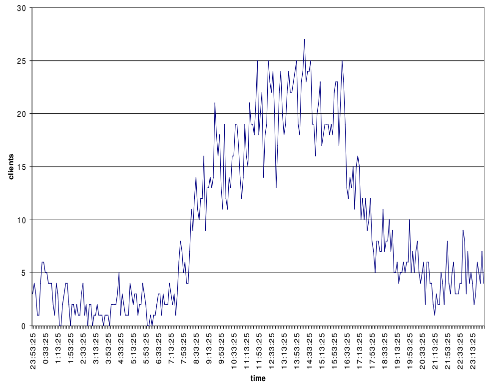
4 Simulation

Set-up and Parameters Simulation is based on Web server traces. Trace characteristics are reported in table 1. The traces are relative to exactly one day of activity and were collected during work-days. The first three traces were collected in 1995 and the last one in late 1998. The cs.edu trace has been logged by the Web server of a university Computer Science department. Our traces report only the HTTP request, the host making the request, the time of the request, the reply code, and the reply size. The table gives total number of requests, the total number of distinct hosts making requests, the number of distinct Web documents in the trace, the average number of bytes requested per second, the average number of bytes requested per second from 9AM to 5PM (server local time), and the average time between two requests from the same host (intervals longer than 5 minutes have been disregarded). It also reports the percentage of bytes in files that are marked as unsuitable for push (see below) over total traffic. The NASA server is more busy than the epa-http and cs.edu servers, as, in 24 hours, it had more requests from more hosts, and the amount of traffic per second was higher both during the 24 hours and in the 9AM to 5PM interval. However, the NASA server handled fewer distinct documents than epa-http or cs.edu. The cs.edu server handles almost as many documents as epa-http, but it is by far the least busy in all other categories. The average rTime is comparable across all traces. Figure 2(a) shows the cumulative percentage of clients that made a certain number of requests. The length distribution is similar in the four traces, and, although there are occasionally some very long client traces, almost 70% of the clients issued less than 20 requests.

In our simulations, clients always create their agents upon their first document request to a server. A document is marked as suitable for push unless either (1) its request URI contains the substring "cgi", or (2) it is ever invoked through a POST, or (3) it is ever invoked with a question mark, or (4) it provokes a non-200, non-304 reply. Almost all documents are suitable for push, as reported in table 1. No-Push resources have *not* been deleted from the original trace and *do*



(a) Client length



(b) Live clients

Figure 2: Cumulative percentage of hosts that generated request traces of up to the given length and number of live clients over time in the epa-http trace (server local time).

Parameter	**BaseValue**	**Description**
CacheSize	128KB	size of client's extension cache
CacheTimeOut	1hr	time before an extension cache entry is considered invalid
Replacement Strategy	LRU	
Arbitration Strategy	SF	see §3
Prediction Strategy	Markov	see §3
Bandwidth	server dependent	bandwidth available at the server site
ClientBw	*Bandwidth*	bandwidth available at the proxy or client site
Latency	500ms	latency imposed by the network, server, and clients
PacketDropProbability	5%	probability a low-priority datagram is dropped
PushTimeOut	90s	time before a server stops pushing documents to idle clients
InitEffectiveProbability	0%	minimum initial access probability for a candidate document

Table 2: Parameters used in simulations.

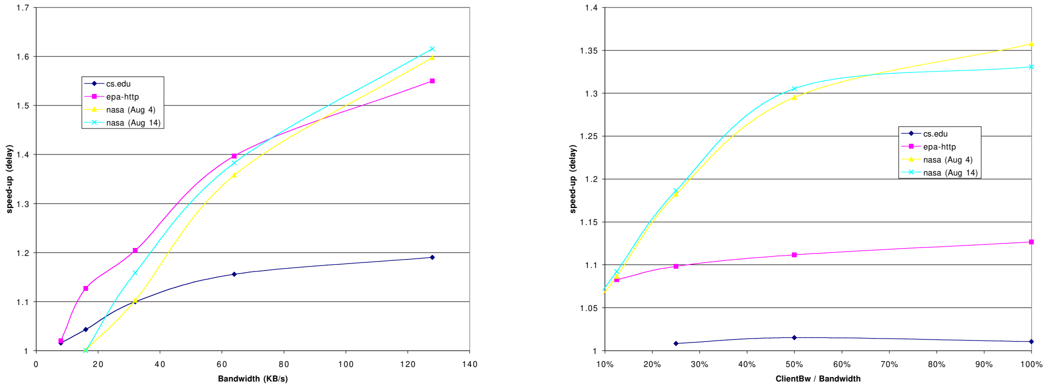
participate in the prediction model. Our traces do not allow us to determine whether a client has a cache and of which size. Hence, our logs could be the trace of client cache misses. We assume that each client has an extension cache [15] of moderate size (see below) to keep pushed documents.

The parameters that describe our simulator are in table 2. Each client has an extension cache of size *CacheSize*. The extension cache keeps pulled and pushed documents in LRU order. Henceforth, such extension cache will be simply called client cache. If a client has kept a version of a Web document in its cache longer than the *CacheTimeOut* period, the client considers that version invalid. Server are connected through links with a certain available *Bandwidth* and we performed experiments for several bandwidth values with *Bandwidth* > AvgBW. The base *Bandwidth* value depends on the server load and is 8KB/s for cs.edu, 16KB/s for epa-http and 64KB/s for NASA (these values are roughly three times WkBW). Clients and proxies have *ClientBw* available bandwidth. The base case $ClientBw = Bandwidth$ corresponds to a client or proxy with the same available bandwidth as the server. Experiments will be executed for values of $ClientBw \leq Bandwidth$. If client bandwidth is less than that of the server, the server can communicate with several clients simultaneously, and, in the simplified simulation set-up, the server handles $\lceil Bandwidth/ClientBw \rceil$ clients. The server handles its queue in FIFO order and has a cache of 128MB, which, for our traces, resulted in no server cache replacement. Our traces report no other server load, and we assume that the only server operation is to satisfy HTTP requests. Network latency is simulated by the parameter *Latency* that represents the amount of time spent to establish and release a connection and to account for the delay with which data are received from the server and processed by the client. The parameter *PacketDropProbability* is the probability that a low-priority pushed datagram is dropped by an intermediate network router. In our simulations, pushed documents are fragmented into 1KB datagrams transparently of the server and of the client, and the document is received only if all its datagram components are received. The SF strategy is independent of the probability $\rho_c(q)$ that document q reaches client c. However, VD depends on $\rho_c(q)$, and experiments were performed for $\rho_c(q) = 1$ (no information on *PacketDropProbability* is available) and $\rho_c(q) = (1 - PacketDropProbability)^{\text{size of } q \text{ in KB}}$ (perfect information on *PacketDropProbability* is available). The server kills agents *PushTimeOut* seconds after the last time a request has been received from those clients. While there are 2312 distinct hosts that contacted the epa-http server, no more than 32 were simultaneously live. Figure 2(b) reports the number of live clients during the epa-http trace. The number of live clients varied, but was never very large. Similar observations hold also for the other traces. Finally, *InitEffectiveProbability* is a threshold parameter whose purpose is to avoid pushing very infrequent documents. The server considers a document q to be a candidate for push only when $\pi_{pq} > EffectiveProbability$, where *EffectiveProbability* is a parameter that is defined as follows. At the beginning of a lull, $EffectiveProbability \leftarrow InitEffectiveProbability$. The server could run out of candidate documents simply because the effective probability is too high. In that case, the effective probability is squared $EffectiveProbability \leftarrow EffectiveProbability^2$. At the beginning of the next lull, the effective probability value is reset to *InitEffectiveProbability* and the process is repeated.

The two major performance measures are the average *delay* and the average *stretch*. The delay is the time for a client to receive the requested document, and it includes the *Latency*, the transmission time (document size over *ClientBw*), and the time spent in the server queue. The stretch is a relative measure of delay, which is equal to the time a request spends at the server (queue + transmission) over the transmission time only [3]. Stretch is a server-only measure of performance and so it is independent of *Latency*. Moreover, stretch is normalized to the bandwidth, while delay is not. If a document is cached, clients can access it with no delay and no stretch. Finally, we also measured *hit rate* and *byte hit rate* in client caches.

Results. Speculative push improved performance by up to a factor of 1.6 over pure pull, and the speed-up increased with the *Bandwidth*, as shown in figure 3(a). The speed-up was an almost perfect match for a logarithmic interpolant, viz a function of the form $a \log Bandwidth + b$ for some a and b (chart omitted for lack of space). The increase of the speed-up with the bandwidth is due to a larger number of documents that can be pushed by the server. Eventually, increasing bandwidth has a more limited effect when the bandwidth is already large, as the server runs out of guesses for documents to push. The speed-up in the stretch metric showed similar behavior, and figures are omitted. Figure 3(b) shows the observed speed-up as a function of the ratio *ClientBw* / *Bandwidth*. The NASA traces were more sensitive to a changes in *ClientBw*: the speed-up increased rapidly when $ClientBw < Bandwidth/2$, and then it flattens. The other two traces were by comparison fairly insensitive to changes in *ClientBw*.

Although the speed-up generally decreases as more datagrams are dropped, UDiD can outperform a pure pull strategy even when $PacketDropProbability = 50\%$, as demonstrated in figure 4(a).



(a) latency speed-up: *Bandwidth*

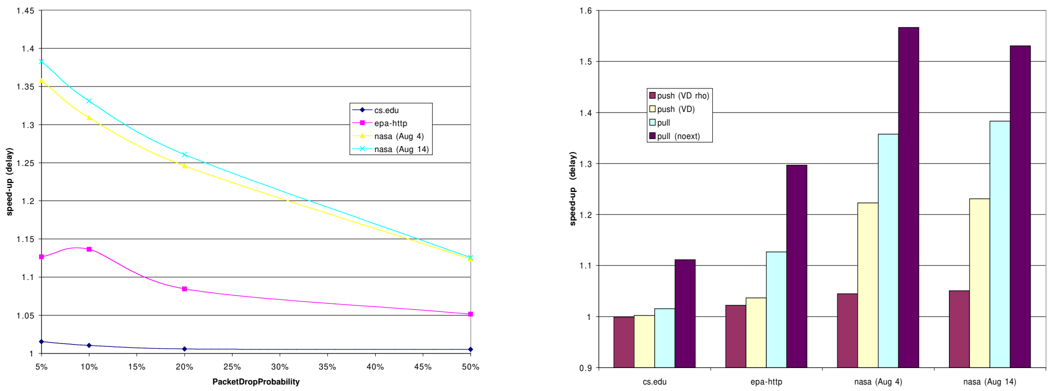
(b) latency speed-up: *ClientBw*

Figure 3: Speed-up of push over pull in the stretch metric for increasing values of *Bandwidth* and *ClientBw*. Graph scales differ.

Figure 4(b) compares SF push, VD push, and pure pull in the base configuration. SF outperformed VD in all traces, except cs.edu, where the two algorithms have nearly equal performance. The VD algorithm performed better if it uses the correct value of $\rho_c(q)$ (VD(rho)), but even so, it was almost always outperformed by SF. Both SF and VD consistently outperform pure pull. A small cache extension (*CacheSize* = 128KB) considerably improved the performance of pull, which we interpret as a sign that in our traces clients did not use their caches effectively, if at all. Figure 5 report the hit rate and byte hit rate for SF, VD(rho), VD, and pure pull. Byte hit rates are between 4% and 14%. In general, the byte hit rate of SF and VD(rho) is comparable and bigger than that of other strategies. However, SF typically outperforms VD(rho) in hit rate. SF's hit rate was more than 45% on the NASA traces, where it triples the hit rate of pure pull. The speed-up of SF over VD(rho) should be attributed mostly to the hit rate gap. Changes in the *Latency* parameter did not significantly affect the speed-up of SF over VD(rho) (chart omitted).

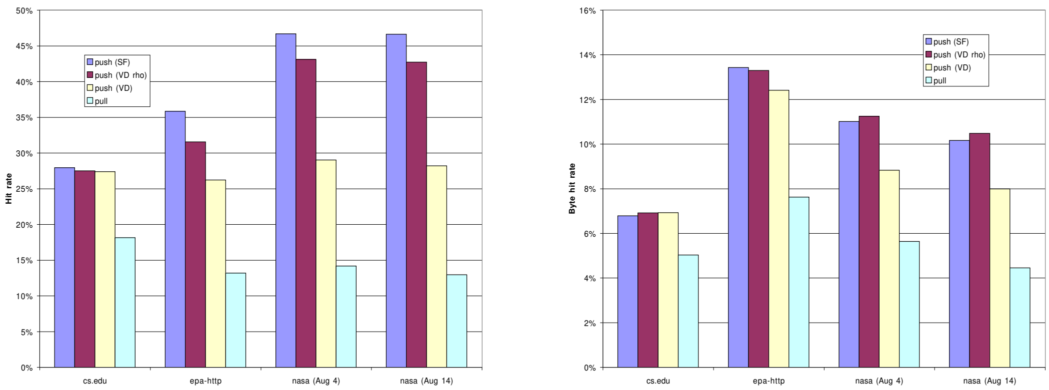
Figure 6 reports bandwidth utilization for epa-http when push is performed compared to the case when only pull is done. Other traces show similar behavior. In general, we found that pull utilization curves was very variable, which is in accordance to previous observations [30]. In the period between 12:30 and 2:30PM, the push strategy transmitted a constant amount of data, whereas during the 2 to 4AM period, push is at least as bursty as a pull strategy. Such observation suggests that push performs perfect traffic shaping during peak periods, but it is actually more bursty than plain pull during low-activity periods. Figure 6 also points out that the amount of data transmitted by a server is in fact much larger for a push strategy than for pure pull, but the principle of unobtrusiveness implies that added traffic is immaterial and can be pursued by exploiting only underutilized resources.

In general, most parameters did not considerably affect push performance. The most surprising is *InitEffectiveProbability*. In our experiments, delay and stretch actually increased when *InitEffectiveProbability* > 0. The effects of changes in *CacheTimeOut*, *Latency*, and *PushTimeOut* are small and are not reported. We also tried the "prefetch buffer" caching strategy by [19]. Such strategy was especially effective when *CacheSize* and *PacketDropProbability* were small, but it is comparable with LRU for larger caches or less reliable connections (charts omitted).



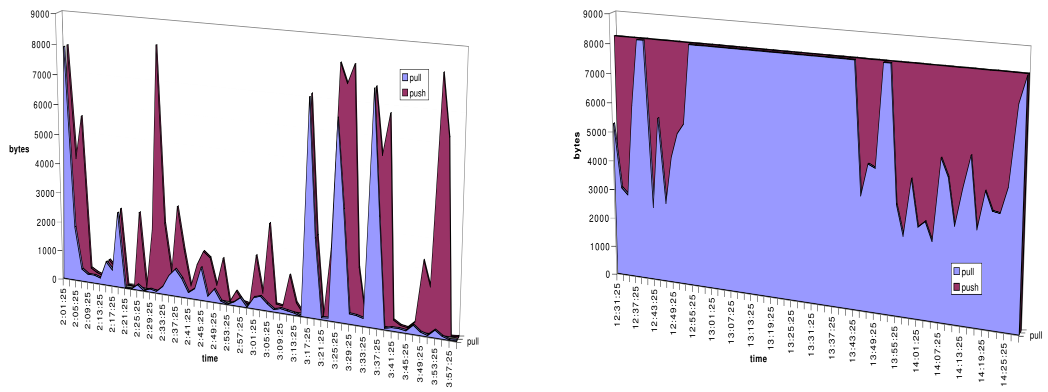
(a) *PacketDropProbability* (b) SF, VD, pull

Figure 4: Speed-up over a pure pull strategy for increasing values of *PacketDropProbability*. Speed-up of SF over VD with perfect knowledge of $\rho_c(q)$ (VD(rho)), VD with $\rho_c(q) = 1$ (VD), pure pull, pure pull with no cache extension. Graph scales differ.



(a) Hit rate (b) Byte hit rate

Figure 5: Hit rates and byte hit rates for SF, VD with perfect knowledge of $\rho_c(q)$, VD with no $\rho_c(q)$ estimate, and pure pull. Graph scales differ.



(a) 2AM-4AM

(b) 12:30PM-2:30PM

Figure 6: Bandwidth utilization during the epa-http trace when *Bandwidth* = 8K. Values are averaged on 30s time intervals. Comparison among SF push, VD push, and pure pull, and between pull/push and LRU.

Prototype. A UDiD prototype has been implemented and run over the Internet. It consists of a Web server and a command-line client browser. The major remaining difficulty is the lack of an application program interface for IPv6 (eg. through sockets [27]). The prototype is temporarily implemented with traditional BSD sockets that we plan to replace as soon as we obtain an IPv6 API. As a result, measurements are not yet available in the IPv6 framework.

5 Related Work

Much research has been devoted to the problem of caching and replication of Web documents, see for example [8, 28]. Long-term data dissemination is provided by *mirrors* [23], which differ from UDiD because UDiD operates on a short time-scale, takes advantage of relatively short lulls in the transmission time, and does not necessarily require the presence of an intermediate information repository where data is mirrored. Data dissemination of Web documents has been considered in [4, 5, 18], but those techniques differ from UDiD because UDiD follows peak and lulls in pulled data traffic, and either does not assume the availability of proxies that can mirror data for long periods of time, or can delay push decisions, or does not need topology information. Problems in disseminating large files has been noted before [5, 6, 15], where an upper bound is imposed on the maximum size of pushed files. Data dissemination has been considered in the context of broadcast disks [16], hybrid networks [26], and cyclic Web multicast [2], where periodic broadcast is used.

The burstiness of server data traffic is well-documented in the literature [30], is substantially different from that in traditional Poisson models, and requires the use of self-similar statistics. Speculative data dissemination is analogous to *prefetching*, except that, in the case of prefetching, data transfers are initiated by clients. Prefetching has been studied in virtually every area of Computer Systems, and a sample recent paper is [7]. Several Web prefetching systems assume that client prefetching is helped by the server, which piggybacks hints on top of regular HTTP replies [12, 13, 20, 21, 24]. If prefetching is not aware of network performance, it can aggravate the load on the network and the burstiness of the data traffic [13]. Two schemes have been proposed to address the issue. The first scheme allows a client to prefetch a document only if estimated access probabilities are above a certain threshold but, once a prefetch request is issued, it has the same priority as pull requests [12, 20]. The second scheme is *rate-controlled prefetching* [13], whereby no bound is imposed on probabilities, but the transmission rate of prefetched documents is less than that of requested documents. Finally, server-initiated prefetching has been proposed in [15] in the context of a proxy servicing dial-up customers.

UDiD exploits idle bandwidth available in a network. As a related topic, much research has been devoted to exploiting idle CPUs (see [25] for a recent paper) and memory [1] in distributed systems. Prediction has been often based on a maximum likelihood first-order Markov reference model (MRM) [5, 6, 12, 13, 20, 24]. It has been observed that MRM prediction stems mostly from *traversal dependencies* due to links embedded in the Web document, and from *embedding dependencies* due to documents embedded in a document, such as a picture [6]. MRM construction can be modified to be fast at the expense of precision [10]. While MRM strikes a trade-off between prediction accuracy, recall, and computational load, it is not a statistically valid model [22]. In probabilistic volume construction, an effective probability of 20% considerably improves performance [12], but such finding does not extend to the UDiD adaptive prediction scheme, as noticed in section §4. Cache consistency and time-out schemes are discussed in [14]. Examples of caching in a push environment are described in [16, 19].

6 Conclusions

In this paper, the UDiD (Unobtrusive Dissemination of Data) architecture was described. Servers exploit local lulls to proactively push documents in low-priority datagrams to clients or proxies when the server predicts that clients/proxies will need those documents in the future. UDiD is a departure from prefetching in that additional traffic is generated if and only if there is an opportunity to transmit documents during idle periods. The architecture has been simulated on Web traces, where speed-ups up to a factor of 1.6 are reported over pure pull, and a preliminary prototype is running. The arbitration problem was analyzed, and the intuitive VD strategy was outperformed theoretically and empirically by SF. SF is also simpler as it does not need precise $\rho_c(q)$ or $v_c(q)$ estimates.

There is virtually no component of UDiD that cannot conceivably be improved. Agent interaction and complexity pose difficult trade-offs that deserve further investigation. The type of pushed documents can be expanded to include ranged pushes. Access prediction could be done according to several different strategies. The arbitration SF algorithm was proved optimal under certain assumptions and was effective in practice, but it is conceivable that other schemes could have better performance and have stronger theoretical properties, especially if coupled with server file caching strategies. Our current arbitration algorithms considers all clients as equal. It has been shown that performance improvement in the context of server-aided caching can be obtained by considering a small number of different client classes [11]. Different and more efficient cache replacement strategies can stem from a tighter interaction between servers and clients. We have only considered dissemination of non-dynamic documents, which form the bulk of our traces, but several issues would arise if dynamic contents were a more significant presence [9]. The current prototype should be finished with IPv6 support. Although load-aware data dissemination poses a number of challenging theoretical and applied problems, we have shown that it can significantly improve user-perceived latency.

References

[1] Anurag Acharya and Sanjeev Setia. The utility of exploiting idle memory for data-intensive computations. In *ACM SIGMETRICS '98*, 1998.

[2] K. C. Almeroth, M. H. Ammar, and Z. Fei. Scalable delivery of Web pages using cyclic best-effort (UDP) multicast. In *Proceedings of Infocom 98*, 1998.

[3] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and strecth metrics for scheduling continuous job streams. In *Proceedings of the Ninth ACM-SIAM Symposium on Discrete Algorithms*, pages 270–279, 1998.

[4] A. Bestavros and C. Cunha. Server-initiated document dissemination for the WWW. *IEEE Data Engineering Bulletin*, 19(3), September 1996.

[5] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of CIKM'95: The 4th ACM International Conference on Information and Knowledge Management*, 1995.

[6] Azer Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems. In *Proceedings of ICDE'96: The 1996 International Conference on Data Engineering*, 1996.

[7] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 188–196, 1995.

[8] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 193–206, 1997.

[9] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching dynamic contents on the Web. In *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, pages 373–388, 1998.

[10] E. Cohen, B. Krishnamurthy, and J. Rexford. Efficient algorithms for predicting requests to Web servers. In *Proceedings of the IEEE INFOCOM'99 Conference*, 1999.

[11] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Evaluating server-assisted cache replacement in the Web. In *Proceedings of the European Symposium on Algorithms*, 1998.

[12] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *Proceedings of SIGCOMM 98*, 1998.

[13] Mark Crovella and Paul Barford. The network effects of prefetching. In *Proceedings of Infocom '98*, 1998.

[14] A. Dingle and T. Partl. Web cache coherence. In *Proceedings of World Wide Web Conference*, 1996.

[15] Li Fan, Pei Cao, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1999.

[16] Michael Franklin and Stanley Zdonik. A framework for scalable dissemination-based systems. In *Proceedings of the International Conference Object Oriented Programming Languages Systems*, pages 94–105, 1997.

[17] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[18] James Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, 1995.

[19] Quinn Jacobson and Pei Cao. Potential and limits of Web prefetching between low-bandwidth clients and proxies. Technical Report CS-TR-98-1372, University of Wisconsin, April 1998.

[20] Z. Jiang and L. Kleinrock. Prefetching links on the WWW. In *Proc. IEEE Inter. Conf. on Communications*, pages 483–489, 1997.

[21] T. M. Kroeger, D. E. Long, and J. C. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proc. USENIX Symposium on Internet Technologies and Systems*, pages 13–22, 1997.

[22] Vincenzo Liberatore. Empirical investigation of the Markov Reference Model. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[23] Katia Obraczka. *Massively Replicating Services in Wide-Area Internetworks.* PhD thesis, University of Southern California, 1994.

[24] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World Wide Web latency. *Computer Communications Review*, 26(3):22–36, 1996.

[25] K. D. Ryu and J. K. Hollingsworth. Linger longer: Fine-grain cycle stealing for networks of workstations. In *SC'98*, 1998.

[26] K. Stathatos, N. Roussopoulos, and J. S. Baras. Adaptive data broadcast in hybrid networks. In *Proc. 23rd International Conference on Very Large DataBases*, 1997.

[27] W. Stevens and M. Thomas. Advanced sockets API for IPv6. RFC 2292, February 1998.

[28] Renu Tewari, Michael Dahlin, Harrick M. Vin, and Jonathan S. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report TR98-04, Department of Computer Sciences, University of Texas at Austin, February 1998.

[29] Tolga Urhan, Michael Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. In *ACM SIGMOD Intl. Conference on Management of Data (SIGMOD)*, 1998.

[30] Walter Willinger and Vern Paxson. Where mathematics meets the Internet. *Notices of the AMS*, 45(8):961–970, September 1998.

A Poisson Arrivals

In this section, we study the case when the deadline D is extracted according to an exponential distribution with rate λ. The exponential distribution models the case when document requests arrive according to a Poisson process, and each request terminates the push phase. Let $P = \{1, 2, \ldots, n\}$ be the set of all documents, $v(i) > 0$ be the value of document i and $l(i)$ the length of document i. Analogously, $v(J) = \sum_{i \in P} v(i)$ and $l(J) = \sum_{i \in P} v(i)$ for all $J \subseteq P$. First, recall that, if D is exponentially distributed, we have

$$Pr[D \geq t] = e^{-\lambda t} \; .$$

Observe that any arbitration algorithm arranges the documents according to a permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ of $P = \{1, 2, \ldots, n\}$ and pushes them in the order given by π. Let $J_i = \cup_{j=1}^{i} \{\pi_j\}$. Then, the expected value of an arbitration algorithm is

$$\sum_{i=1}^{n} v(J_i) Pr[l(J_i) \leq D < l(J_{i+1})] = \sum_{i=1}^{n} v(\pi_i) Pr[D \geq l(J_i)] = \sum_{i=1}^{n} v(\pi_i) e^{-\lambda l(J_i)} \; . \tag{1}$$

Our objective is to find a permutation π that minimizes (1). Theorem 1 follows immediately from the following lemma.

Lemma 1 *SF's permutation maximizes (1) when $l(i) = v(i)$ for all $i \in P$ (subset sum problem).*

Proof. Assume without loss of generality that SF's permutation is the identity, that is, for all $i \in \{1, 2, \ldots, n-1\}$, $l(i) \leq l(i+1)$. Suppose by contradiction that a different permutation $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ minimizes (1). Let h be the smallest index where $\pi_h > h$ and k the index where $\pi_k = h$. We will show that we can exchange h with π_h without increasing (1), and, by induction on h, the lemma will follow. We rewrite the expected value of π in (1) as $\sum_{i=1}^{n} T(i)$ where $T(i) = v(\pi_i) e^{-\lambda l(J_i)}$ is its ith term. If we exchange h with π_h, we change the value only of the terms $T(i)$ for $h \leq i \leq k$. Moreover, if $h < i < k$, the value of $T(i)$ increases. We will show that the change of $T(h) + T(k)$ is non-negative. Indeed, let $J_0 = \emptyset$, $H = \cup_{j=h+1}^{k-1} \{\pi_j\}$, and observe that the value of $T(h) + T(k)$ before the exchange is equal to

$$e^{-\lambda l(J_{h-1})} \left(l(\pi_h) e^{-\lambda l(\pi_h)} + l(h) e^{-\lambda l(H \cup \{h, \pi_h\})} \right) = e^{-\lambda l(J_k)} \left(l(\pi_h) e^{\lambda l(H \cup \{h\})} + l(h) \right) \; ,$$

and after the exchange becomes

$$e^{-\lambda l(J_{h-1})} \left(l(h) e^{-\lambda l(h)} + l(\pi_h) e^{-\lambda l(H \cup \{h, \pi_h\})} \right) = e^{-\lambda l(J_k)} \left(l(h) e^{\lambda l(H \cup \{\pi_h\})} + l(\pi_h) \right) \; .$$

Suppose by contradiction that difference is negative. Then, we would have

$$l(h) e^{\lambda l(H \cup \{\pi_h\})} + l(\pi_h) < l(\pi_h) e^{\lambda l(H \cup \{h\})} + l(h) \; ,$$

or

$$\frac{l(h)}{e^{\lambda l(H \cup \{h\})} - 1} < \frac{l(\pi_h)}{e^{\lambda l(H \cup \{\pi_h\})} - 1} \; .$$

Moreover, $l(H \cup \{i\}) = l(H) + l(i)$ for any $i \notin H$. However, $\pi_h > h$, and so $l(\pi_h) \geq l(h)$ and the function $x/(e^{x+a} - 1)$ is decreasing for all $x > 0$ and all a's, so that the difference is non-negative. In conclusion, when we exchange h and π_h, the expected value does not decrease, and so the lemma is proved. $\diamondsuit$

Theorem 1 follows as a restatement of the lemma.