Integrating Prefetching and Invalidation for Transparent Replication of Dissemination Services

Abstract

This paper explores integrating data prefetching and data consistency to enable transparent replication of large-scale information dissemination services. We focus our attention on *information dissemination* services, a class of service where updates occur at an origin server and reads occur at a number of replicas, and we seek to provide *transparent* replication by providing sequential consistency guarantees across the replicated data. To meet these goals our system integrates prefetching and consistency by (1) providing self-tuning push-based prefetch from the server and (2) buffering and carefully scheduling the application of invalidations and updates at replicas to maximize the amount of valid data—and therefore maximize the hit rate, minimize the response time, and maximize availability—at a replica. Our analysis of simulations and our evaluation of a prototype implementation support the hypothesis that it is feasible to provide transparent replication for information dissemination applications by carefully integrating consistency and prefetching. For example, in our simulations when the network 40% of the bandwidth it needs to push all updates, our system's performance is more than a factor of three faster than a demand-based system.

1 Introduction

This paper explores integrating data prefetching and data consistency to enable transparent replication of large-scale information dissemination services. Researchers are working to develop programming environments [11, 46, 2, 20] and scalable servers [3, 50] for distributing service code to replicas across a network in order to improve service availability [15, 30, 55] and performance [6]. But for this approach to be useful, this distributed code must operate on a common set of shared data. Thus, a fundamental challenge to large-scale service replication is replication of the underlying data.

Our goal is to develop a data replication toolkit that supports *transparent* service replication where service threads designed to run on a single machine or LAN cluster by accessing shared state in a file system or database can be distributed across WAN edge servers without rewriting the service and without introducing new bugs. Thus, we focus on providing the strong guarantee of sequential consistency [32] because weaker semantics can introduce new bugs, require subtle reasoning from the programmer, or both [21].

Providing strong consistency guarantees in a large scale system while providing good availability [9] and performance [34] is fundamentally difficult in the general case. We therefore focus our attention on the key subproblem of replicated *dissemination services* where all updates occur at one origin server and where multiple edge server replicas treat the underlying data as read only and perform services such as data caching, fragment assembly, per-user customization, and advertising insertion. Although this case is restrictive, it represents an important class of services; for example, Akamai's Edge Side Include [2] and IBM's Sport and Event replication system [12] both focus on improving the performance, availability, and scale of dissemination services. Furthermore, we believe that this case may represent an important building block for more general services where one subset of the data is read-only at the replicas, where another subset is read/write at the replicas, and where different subsets use different consistency algorithms [47].

This paper explores integrating prefetching and consistency to provide this transparent replication. We believe this combination is vital. On one hand strong consistency increases the need for prefetching because strong consistency prevents the use of stale data, which could hurt performance and availability, but prefetching replaces stale data with valid data. On the other hand, prefetching means that data are no longer fetched near the time they are used, so a prefetching system must rely heavily on its consistency protocol for correct operation.

A simple solution would be to replace an invalidation-based protocol with an update-based protocol that sends all updates to all replicas. Such an approach could provide sequential consistency if all updates are applied in the order they occur and could provide excellent response time and availability because a replica would service all requests from its local storage. Unfortunately, this *FIFO push-all* algorithm sends all updates over the wire whether they are eventually read or not, which may require arbitrarily large amounts of network bandwidth. This approach may be appropriate in some environments, since falling bandwidth costs may make it attractive to trade bandwidth for improved latency and availability. At the same time, the approach is potentially fragile: if the update rate exceeds the available bandwidth, the updates overload the network, interfering with other applications and causing unbounded delays between when the update occurs at the origin server and when it is seen at the replicas.

In this paper, we describe the TRIP (Transparent Replication through Invalidation and Prefetching) system that integrates sequential consistency with self tuning prefetch in a way that approximates FIFO update

when network bandwidth is plentiful but that approximates FIFO invalidate when bandwidth is scarce, and that gracefully/incrementally tracks between these two extremes based on available bandwidth.

The TRIP algorithm has two parts. First, the server implements the system's self-tuning, push-based prefetch by sending a replica's invalidation messages and responses to demand reads on one channel and by pushing the replica's updates on another channel. In particular, the server sends invalidations and demand responses over FIFO channels at normal network priority, but it buffers updates in a priority queue that drains through a low priority network connection to avoid interfering with other network traffic [48]. Thus, when bandwidth is high, the priority queue is empty and the approach approximates FIFO push-all, but when bandwidth is low, only the most valuable updates are sent. The replicas implement the second important part of the algorithm by buffering the messages they receive and both (1) applying them in a careful order to maintain sequential consistency and (2) delaying application of some messages to minimize the amount of invalid data and thereby maximize the local hit rate, minimize response time, and maximize availability.

This paper evaluates this strategy using both trace-based simulation and evaluation of an implementation. Our simulations use traces of access to the 2000 Summer Olympics Games web site, a large-scale information dissemination service that was served from several geographically distributed service replicas. Our prototype provides a file system interface at each replica via a local NFS server [36]. This implementation allows us to run unmodified edge servers that provide both static HTML files and dynamic responses generated by programs (e.g., CGI, Servelets, Server Side Include, or Edge Side Include), and that share data through the file system. A similar approach could be used to support a database interface to the shared state.

Our evaluation supports the hypothesis that it is feasible to provide transparent replication for information dissemination applications by carefully integrating consistency and prefetching. In particular, this combination yields three good properties. First, it simplifies application development by providing sequential consistency and supporting transparency. Second, whereas strong consistency might be expected to hurt system performance, by combining it with self-tuning prefetching the system often gets better performance than demand-based replication systems that provide weaker consistency guarantees. For example, in our simulations when the network 40% of the bandwidth it needs to push all updates while meeting timeliness guarantees, our system's performance is more than a factor of three faster than a demand-based system. Third, whereas strong consistency might be expected to hurt availability, prefetching updates and attempting to delay the application of invalidations until the corresponding update is available can allow replicas to maintain local copies of large fractions of system state and thereby mask server failures and network partitions. For example, in our simulations when the network has enough bandwidth to allow the server to push 70% of all updates, a replica is able to completely mask server disconnections by serving local copies of sequentially consistent state for varying periods of time with the maximum masking duration exceeding 9000 seconds for more than half of the randomly arriving disconnections.

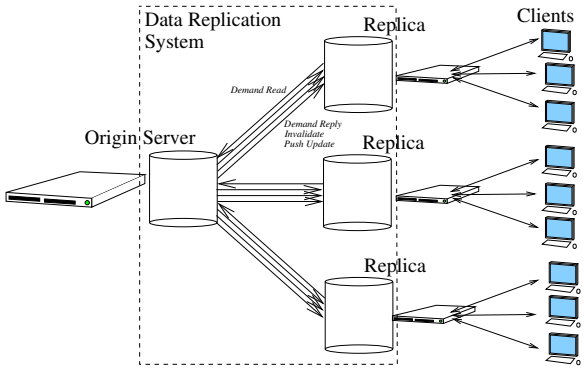

Fig. 1: High level system architecture.

This paper makes three contributions. First, it provides evidence that systems can maintain sequential consistency for some key WAN distributed service despite CAP dilemma, which states that systems cannot get strong **C**onsistency and high **A**vailability for systems vulnerable to **P**artitions [9]. The replication system circumvents this dilemma by (a) restricting the workload it considers and (b) integrating consistency with prefetching. Second, it presents a novel system that integrates prefetching and consistency by (a) using a new self-tuning push-based prefetching algorithm and (b) carefully ordering and delaying the application of messages at replicas. Third, it provides a systematic evaluation and a working prototype of such a system to provide evidence for the effectiveness and practicality of the approach.

The rest of the paper proceeds as follows. Section 2 provides background on prefetching and consistency and more precisely defines the environments in which our framework can be used. Then, Section 3 details the algorithms at the core of our approach. Section 4 describes our prototype implementation, and Section 5 discusses both our simulation and prototype evaluation. Finally, Section 6 provides an overview of related work and Section 7 highlights our conclusions and discusses some potential future directions.

2 System model

Figure 1 provides a high level view of the replication environment we assume. An *origin server* and several *replicas* (also called content distribution nodes or edge servers) share data, and *clients* access the service via the replicas, which can not only provide static HTML files but can also run service-specific code to dynamically generate responses to requests [2, 3, 11, 20, 46, 50]. A redirection infrastructure [12, 28, 54] directs client requests to a good (e.g., nearby, lightly loaded, or available) replica. In such an environment, the focus of this paper is on the *data replication system* that provides shared state across the origin server and the replicas.

Proposed service replication architectures [2, 3, 11, 20, 46, 50] vary in their assumptions about the number of replicas (e.g., 10 replicas to thousands), whether a given replica is typically installed for long periods of time on the same machine(s) or whether replicas are dynamically created, destroyed, or moved over fine time scales to respond to changing demand, and whether a replica caches a small subset of hot pages or replicates most or all of a service. We focus on supporting modest numbers (e.g., 10-100) of long-lived replicas that each have sufficient local storage to maintain a local copy of the full set of their service's shared data. Our protocol remains correct under other assumptions, but optimizing performance in other environments may require different trade-offs.

2.1 Consistency and timeliness

This study focuses on protocols that simultaneously enforce both sequential consistency, which restricts the permitted ordering among reads and writes across all objects, and Δ-coherence, which limits the real-time duration between when a write of an object occurs and when the write becomes visible to subsequent reads of that object. The rest of this subsection defines these concepts more precisely.

To support transparency, we focus on providing sequential consistency. As defined by Lamport, "The result of any execution is the same as if the [read and write] operations by all processes were executed in some sequential order and the operations of each individual processor appear in this sequence in the order specified by its program." [32] Sequential consistency is attractive for transparent replication because the results of all read and write operations are consistent with an order that could legally occur in a centralized system, so—absent time or other communication channels outside of the shared state—a program that is correct for all executions under a local model with a centralized storage system is also correct for the distributed storage system.

Typically, providing sequential consistency is expensive in terms of latency [10, 34] or availability [9]. However, we restrict our study to *dissemination services* that have one writer (the origin server) and many readers (the replicas), and we enforce *FIFO consistency* [34] guarantees. Under FIFO consistency (a.k.a., PRAM consistency) writes by a process appear to all other processes in the order they were issued, but different processes can observe different interleavings between the writes issued by one process and the writes issued by another. Note that for applications that include only a single writer, FIFO consistency is identical to sequential consistency.

Although ensuring sequential consistency at each replica provides strong semantic guarantees, clients accessing a service through the replicas may observe unexpected behaviors in at least two ways due to communication channels outside of the shared state.

First, because sequential consistency does not specify any real-time requirement, a client may observe a stale version of the service. For example, if a network partition separates a replica from the origin server, the view of the service provided by the replica will not reflect recent updates even if the view continues to obey sequential consistency. A user could observe, for example, the anomalous behavior of a stock price not changing for several minutes during a disconnection. In this case, physical time acts as a communications channel outside of the control of the data replication system that could allow a user to detect anomalous behavior introduced by the replication system.

Therefore, replicated services may want to enforce timeliness constraints on data updates to ensure that, for example, replicas transmit views of the service that are within Δ seconds of the view at the central server and of other replicas. An extreme version of such guarantees is *linearizability* [24], which enforces the same constraints as sequential consistency but which also requires that the total ordering of events follow timestamps assigned to read and write operations so that if $ts_{OP1(x)} < ts_{OP2(y)}$, then operation $OP1(x)$ should precede $OP2(y)$ in the total order. In web service environments, however, a more flexible definition may be useful because different services will depend on timeliness to different degrees. Δ-*coherence* requires that any read reflect at least all writes that occurred before the current global time minus Δ. By combining Δ-coherence with sequential consistency, a protocol enforces a tunable staleness limit on the sequentially consistent view. Note that reducing Δ improves timeliness guarantees but may hurt availability because disconnected edge servers may be obligated to refuse a request rather than serve too-stale data.

Second, some redirection infrastructures [12, 28, 54] may cause a client to switch between replicas. Even if each replica provides a sequentially consistent view of the data, a client switching between replicas may see inconsistencies. For example, consider two replicas r_1 and r_2 where r_2 processes messages somewhat more slowly

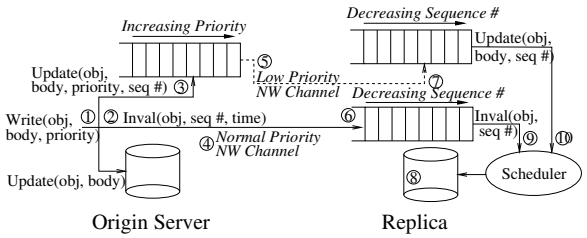

Fig. 2: Overview of replication algorithm. The circled numbers are discussed in the text.

than r_1. If objects A and B are initially in states A_0 and B_0, then A is written to state A_1, and finally B is written to state B_1, a client could read object B and observe state B_1 from replica r_1 and then switch to replica r_2 and read object A and observe state A_0. Even though neither r_1 nor r_2 observes any state inconsistent with A_1 *happens before* [31] B_1, by switching between replicas the client can observe such an inconsistent state. In Section 3.4 we discuss how to adapt Bayou's session consistency protocol [44] to our replication environment to ensure that each client observes a sequentially consistent view regardless of how often the redirection infrastructure switches the client among replicas.

3 Algorithm

Figure 2 provides a high-level view of the algorithm for synchronizing a replica's data store with the origin server's. The strategy revolves around two simple parts (1) the server's self-tuning efforts to send updates in priority order without interfering with other network users and (2) each replica's efforts to buffer messages it receives, to apply them in an order that meets consistency constraints, and to delay applying some of these messages to improve availability and performance. In particular,

1. When the origin server writes an object (number 1 in the figure), it immediately sends an invalidation to each replica (2) and it enqueues the body of the update in a priority queue for each replica (3). In contrast with the immediate transmission of invalidations on a normal-priority lossless network connection (4), each priority queue drains by sending its highest-priority update to its replica via a low-priority network channel when the network path between the origin server and replica has spare capacity (5).
2. At the replica *both* invalidation (6) and update (7) messages that arrive are buffered rather than being immediately applied to the replica's local data store (8). A scheduler at each replica applies invalidations in strict sequence-number order (9), delaying the application of each successive invalidation until its corresponding update appears in the update buffer or until its deadline (under Δ-coherence) arrives. Similarly, when the scheduler at a replica applies a buffered update (10), it always applies the one with the lowest available sequence number and it only applies an update if all invalidations with lower sequence numbers have already been applied.

```
State
  seqNo                  // Global sequence number
  storage                // Sequence number and body of each object
  nReplicas              // Number of replicas
  updateChannel[]        // Lossy, priority-order, low-priority channels
  invalDemandChannel[]   // Lossless, FIFO channels
local call to write(objId, body, priority, timestamp)
  seqNo++;
  storage.update(objId, body, seqNo);
  for(i = 0; i < nReplicas; i++)
    invalDemandChannel[i].send(INVAL, objId, seqNo, timestamp);
    updateChannel[i].insert(UPDATE, objId, body, seqNo, priority);
receive (READ, objId) from replicaId
  (body, objSeqNo) = storage.get(objId);
  invalDemandChannel[replicaId].send(REPLY, objId, body, objSeqNo);
  updateChannel[replicaId].cancel(objId);
```

Fig. 3: Origin server algorithm. **Bold** labels indicate the events that trigger the actions.

The full algorithm must also handle demand reads, network disconnections, and machine failures. We therefore detail the server and replica algorithms in the next two subsections. Then Section 3.3 discusses key properties of this algorithm, and Section 3.4 discusses several limitations of the basic algorithm and possible optimizations available within this framework.

3.1 Origin server

As the pseudocode in Figure 3 shows, the origin server maintains a global monotonically increasing sequence number *seqNo*, local *storage* with the body and sequence number of each object, a set of per-replica channels *invalDemandChannel[]* for sending invalidations and demand replies, and a set of per-replica channels *updateChannel[]* for pushing updates.

To write an object, an origin server increments *seqNo*, updates *storage* with *seqNo* and the object's new body, sends invalidations on each replica's *invalDemandChannel*, and enqueues updates on each replica's *updateChannel*.

Each enqueued update includes a *priority* that specifies the update's relative ranking to other pending updates. Our interface allows a server to use any algorithm for choosing the priority of an update, and this paper does not attempt to extend the state of the art in prefetch prediction policies. A number of standard prefetching prediction algorithms exist [17, 22, 23, 40, 49] or the server may make use of application-specific knowledge to prioritize an item (e.g., a news editor may know that the day's headline article will be widely read before the system has measured the story's read frequency). Note that some implementations may extend this interface to specify different priorities for propagating a given update to

different replicas to, for example, account for different access patterns at different replicas.

When the server receives a demand *read(objId)* from a replica, it retrieves from its local store the object's body and per-object sequence number, it sends on the replica's *invalDemandChannel* a demand reply message. Notice that this reply includes the sequence number stored with the object when it was last updated, which may be much smaller than the current global *seqNo*. Upon sending a demand reply to a client, the origin server also cancels any push of the object's update to that client still pending in *updateChannel*.

Communication channels. The system design depends on the distinct properties of the *invalDemandChannel*s and the *updateChannel*s.

Each *invalDemandChannel* for invalidations and demand replies is a lossless FIFO channel that operates at normal network priority. Our protocol uses a persistent message queue [27] to ensure that this channel is lossless even across crashes and network partitions, which dramatically simplifies crash recovery.

Each *updateChannel* provides an abstraction suited for self-tuning push-based prefetch by buffering updates in a priority queue and then sending them across the network using a low priority network protocol. Three actions manipulate each per-replica priority queue.

First, an *insert* adds an update with a specified priority. If another update to the same *objId* occupies the priority queue, the older update is discarded. An implementation may bound the upper size of the priority queue buffer and discard low priority items to maintain this size bound. Second a *cancel(objId)* call removes any pending update for *objId*. Third, a worker thread loops, removing the highest priority update from the queue and then doing a low-priority network send of a push-update message containing the *objId*, *body*, and *seqNo* of the item. The low priority network protocol should ensure that low priority traffic does not delay, inflict losses on, or take bandwidth from normal-priority traffic; a number of such protocols have been proposed [7, 8, 39, 48].

3.2 Replica

The pseudocode in Figure 4 describes the behavior of a replica. Each replica maintains five main data structures. First, a replica maintains a local data store that maps each object ID for the shared state to either the tuple *(INVALID, seqNo)* if the local copy of the object is in the invalid state or the tuple *(VALID, seqNo, body)* if the local copy of the object is in the valid state. Second, a replica maintains *pendingInval*, a list of pending invalidation messages that have been received over the network but not yet applied to the local data store; these invalidation messages are sorted by sequence number. Third, a replica maintains *pendingUpdate*, a list of pending pushed updates that have been received over the network but not yet applied to the local data store; notice that although the origin server sorts and sends these update messages by priority, each replica sorts its list of pending updates by *sequence number*. Finally, Δ specifies the maximum staleness allowed between when an update is applied at the origin server and when the update affects subsequent reads, and *maxSkew* bounds the clock skew between the origin server and the replica.

```
State
    storage        // Validity, sequence number, and body of each object
    pendingInval   // Received but unprocessed invalidation
    pendingUpdate  // Received but unprocessed updates
    delta          // Max staleness between server and replica
    maxSkew        // Max clock skew between server and replica
receive (INVAL, objId, seqNo, timestamp) on invalDemand channel
    pendingInval.put(objId, seqNo, timestamp);
receive (UPDATE, objId, body, seqNo) on updateChannel
    pendingUpdate.put(objId, body, seqNo);
pendingUpdate.head.seqNo ≤ pendingInval.nextSeqToProcess()
    // Scheduler applies an update
    (objId, body, seqNo) = pendingUpdate.removeHead();
    if(seqNo ≥ storage.getSeqNo(objId))
        storage.update(objId, VALID, seqNo, body);
    if(seqNo == pendingInval.nextSeqToProcess())
        pendingInval.doneProcessing(seqNo);
currentTime() ≤ pendingInval.head.timestamp + delta - maxSkew
    // Scheduler applies an invalidate
    applyNextInval(); // See below
local call to read(objId)
    if(VALID == storage.getState(objId))
        return storage.getBody(objId);
    send (READ, objId) to origin server;
    storage.waitUntilValid(objId);
    return storage.getBody(objId);
receive (REPLY, objId, body, seqNo) on invalDemandChannel
    while(pendingInval.nextSeqToProcess() ≤ seqNo)
        applyNextInval();
    storage.update(objId, VALID, seqNo, body); // Unblocks waiting read
applyNextInval() // Internal private method called from above
    (objId, seqNo, timestamp) = pendingInval.readHead();
    if(seqNo ≥ storage.getSeqNo(objId)) // 'At least once' channel
        storage.update(objId, INVALID, seqNo);
    pendingInval.doneProcessing(seqNo);
```

Fig. 4: Replica algorithm. **Bold** labels indicate the conditions and events that trigger the actions.

The system's correctness depends on following two constraints on its actions:

C1 A replica must apply all invalidations with sequence numbers less than N to its storage before it can apply an invalidation, update, or demand reply with sequence number N.

C2 A replica must apply an invalidation with timestamp t to its storage no later than $t + \Delta - maxSkew$.

Scheduler actions. After *INVAL* and *UPDATE* messages arrive and are enqueued in *pendingInval* and

pendingUpdate, a scheduler applies these buffered messages in a careful order to meet the two constraints above and to minimize the amount of invalid data and thereby maximize local hit rate, maximize availability, and minimize response time.

The scheduler removes the update message with the lowest sequence number from its *pendingUpdates* and applies it to its *storage* as soon as it knows it has applied all invalidations with lower sequence numbers. Applying a prefetched update normally entails updating the local sequence number and body for the object, but if the locally stored sequence number already exceeds the update's sequence number, the replica must discard the update because a newer demand reply or invalidation has already been processed. Also note that in the case where update N arrives before invalidation N is applied, update N can be applied as soon as invalidation $N-1$ has been applied and then invalidation N need never be applied. In this case, the procedure informs the *pendingInval* queue that *seqNo* has been processed, which allows *pendingInval* to garbage collect the message and to acknowledge processing of invalidation *seqNo* to the origin server.

The scheduler removes the invalidation message with the lowest sequence number from *pendingInval* and applies it to its *storage* when the invalidation's deadline arrives at $timestamp + \Delta - maxSkew$. The *pendingInval* queue and network channel normally provide FIFO message delivery, and they guarantee at least once delivery of each invalidation when crashes occur. To support end-to-end at-least-once semantics, before applying an invalidation, a replica verifies that it is a new one, and after applying an invalidation a replica calls *pendingInval.doneProcessing(seqNo)* to allow garbage collection of the message and to acknowlege processing of invalidation *seqNo* to the origin server.

Processing requests from clients. When a servicing a client request that reads object $objId$ (either as input to a dynamic content-generation program or as the reply to a request for a static data file), a replica uses the locally stored body if *objId* is in the *VALID* state. But, if the object is in the *INVALID* state, the replica sends a demand request message to the server and then waits for the demand reply message. Note that by sending demand replies and invalidations on the same FIFO network channel, the origin server guarantees that when a demand reply with sequence number N arrives at a replica, the replica has already received all invalidations with sequence numbers less than N, though some of these invalidations may still be buffered in *pendingInval*. So when a demand reply arrives, the replica enforces condition C1 by simply applying all invalidation messages whose sequence numbers are at most the reply's sequenceNumber before applying the reply's update to the local state and returning the reply's value to the read request.

Our protocol implements an additional optimization (not shown in the pseudo-code for simplicity) by maintaining an index of pending updates searchable by object ID. Then, when a read request encounters an invalid object, before sending a demand request to the origin server, the replica checks the pending update list. If a pending update for the requested object is in the pending update list, the system applies all invalidations whose sequence numbers are no larger than the pending update's sequence number, applies that pending update, and returns the value to the read request.

A remaining design choice is how to handle a second read requests r_2 for object o_2 that arrives when a first read request r_1 for object o_1 is blocked and waiting to receive a demand reply from the origin server. Allowing r_2 to proceed and potentially access a cached copy of o_2 risks violating sequential consistency [1]: for example, if the replica has processed all invalidations with sequence numbers up to $n_{current}$, if o_2 has a pending invalidation with sequence nuber $n_{w2} > n_{current}$, and if o_1 has a pending invalidation with sequence number $n_{w1} > n_{w2}$, then read r_1 will ultimately return the version of o_1 associated with sequence number n_{w1}. But, for sequential consistency to hold, any read of o_2 that happens after read r_1 must return a version of o_2 with a sequence number at least n_{w2}. Thus, if program order specifies that r_1 *happens before* r_2, then request r_2 must block until request r_1 has been processed. The program order relationship between r_1 and r_2 depends on the programming model: if r_1 and r_2 are asynchronous reads issued by a single thread, it is clear that read r_2 must block in this case. Similarly, if r_1 and r_2 are issued by threads that coordinate their actions through local shared memory and if these threads synchronize through local shared memory in a way that reveals that read r_1 was issued before read r_2, then read r_2 must block [21]. But, if r_1 and r_2 are issued by independent threads of computation that have not so synchronized, then the threads are logically concurrent and it would be legal to allow read r_2 to "pass" read r_1 in the cache [21, 32].

Absent information about the causal relationship among different requests, our system makes the conservative choice and blocks all read requests when a replica is waiting for a demand read miss to be satisifed by the origin server. Note that many web server runtime systems (1) assume that concurrent client requests are causally independent, (2) process each client request with a single thread, and (3) use blocking read requests when accessing file system or database state and could therefore tolerate allowing cached reads to "pass" concurrent read misses. It would be straightforward to change our implementation to process multiple read requests concurrently rather than block when the system is used in enviroments where

such assumptions are known to be valid. Therefore, our performance and availability results should be regarded as conservative. [1]

Operating during disconnection. When a replica becomes disconnected from the server due to a network partition or server failure, the replica attempts to service requests from its local store. If the local copies of most objects are valid, a replica may be able to mask the disconnection for an extended period. Note that to enforce Δ-coherence, a replica must block all reads if it has not communicated with the origin server for Δ seconds. We use a heartbeat protocol to ensure liveness when the network is available.

If a read miss occurs during a disconnection, it logically blocks until the connection is reestablished and the server satisfies the demand miss. Note that in our conservative design described above, reads that arrive after such a blocking read must also block to ensure sequential consistency.

Note that in a web service environment, blocking a client indefinitely is an undesirable behavior. Therefore, the protocol provides two ways for services to give up some transparancy of distributing their application in order to gain control of recovery in the case where a replica blocks because it is disconnected from the origin server.

First, after a time-out a read returns an error code. Although a correct program should always check for error codes on file or database reads, in practice this interface is not fully transparent because (a) many applications fail to check for error codes on IO operations and (b) the actions an application should take on a read error may differ in this distributed case (where, say, redirecting the request to a different replica may work) versus the centralized case (where probably little can be done.)

A second option is for the replication layer to take two actions when a demand read times out: (1) signal the redirection layer [12, 28, 54] to stop sending requests to this replica and (2) signal the local web server infrastructure to close all existing connections to all clients and to respond to subsequent client requests with an HTTP redirect [19] to a different replica. The approach then relies on client-initiated request retransmission for end-to-end recovery [9]. This option provides less precise control to the application, but it also requires less invasive modifications of the service application code.

Finally, given the choice between reducing availability and increasing staleness during disconnections, some services may choose the latter. Such services may configure the replication layer to increase Δ when a disconnection from a server is suspected.

[1] The reader may notice that if one assumes that different clients are completely independent, it is in principle possible to go even further and have a replica to maintain different sequentially consistent "views" for different clients and serve data to client A that client B's view regards as invalid. We are not certain that such optimizations warrent the additional complexity, but this may be an avenue for future work.

3.3 Properties and rationale

Sequential consistency. Although invalidation messages arrive at a replica in strict sequence number order, updates and demand read replies can arrive out of order: updates arrive out of sequence number order because they are sent in priority order, and demand replies arrive out of sequence number order because they are sent at the time of a demand read but they include the sequence number of their earlier write.

The replica ensures sequential consistency [32] by enforcing condition C1 across invalidations, updates, and demand read replies.

Lemma 1 *If condition C1 holds, then the system provides sequential consistency.*

Proof sketch. Assemble a sequential order for all write and read operations as follows. Assign w_i, the ith write operation at the origin server s, an *ordering number* $n_{(s,i)}$ which equals the global sequence number the origin server uses for w_i's invalidations. At each replica server copy c, assign r_j, the jth read operation at that replica, the ordering number $n_{(c,j)}$ equal to the highest sequence number of any invalidation processed by replica c when r_j executes its return. Now, sort all read and write operations by their ordering numbers with writes coming before reads with the same ordering number and with ties among reads broken by lexical ordering of replica IDs. Such an ordering is sequentially consistent because (1) the result of each read in the system is the same as its result if executed in this sequential order and (2) read and writes from any program that executes at a node in the system appear in this sequence in program order.

The system enforces C1 by the program constraints described above and by relying on per-replica reliable FIFO channels both to deliver invalidations in sequence number order and to ensure that when a demand read reply arrives, all earlier invalidations have arrived as well.

Self-tuning prefetching. By combining a priority queue and a low-priority network protocol, the updates' channel provides for self-tuning prefetching for each replica. When the network between the origin server and a replica provides a large amount of spare bandwidth, the priority queue drains quickly and the channel approximates a lossless, FIFO, push-all channel. But, when network bandwidth is scarce, only valuable items are sent and the buffering delay allows multiple updates of the same data to collapse into a single update and save network bandwidth [5]. Note that unlike many traditional prefetching protocols [17, 22, 23, 40, 49], there is no preset threshold that determines whether a given object is

valuable enough to send; instead, we rely on the low-priority network protocol to ensure that objects are only sent when the value of doing so exceeds the cost .

A replica maximizes the benefit of this prefetching by maximizing the amount of valid data in its local storage. It does this by delaying application of invalidation messages as long as it can (subject to consistency and staleness constraints). In particular, it tries to delay applying an invalidation with sequence number N until it has an update with the same sequence number. But, it is forced to apply an invalidation earlier than that in two circumstances: (1) the staleness deadline for an invalidation forces it to be applied or (2) a demand read reply that reflects state M ($M > N$) arrives at the replica. Note that in the latter case, a replica applies pending invalidations "early" rather than force a demand request to wait for staleness deadlines to expire.

3.4 Limitations and optimizations

Our current protocol is limited in at least two ways. These limitations could be addressed with future optimizations.

First, our current protocol can allow a client that switches between replicas to observe violations of sequential consistency. In particular, if replica A has processed an invalidation with sequence number $seqNum_A$ and replica B has processed an invalidation with sequence number $seqNum_B$ ($seqNum_A > seqNum_B$) and a client first access replica A and then accesses replica B, it could observe values at B that are inconsistent with sequential consistency. Therefore, for best results the redirection algorithm should direct a client to the same replica for long periods of time.

We speculate that a system could adapt Bayou's session guarantees protocol [44] to maintain sequential consistency semantics when a client switches replicas. In particular, a replica's web server could insert an HTTP cookie reflecting the highest sequence number observed by a client in responses to a client and inspect this cookie on all requests from a client. If the sequence number in a request exceeds the replica's sequence number, the replica web server signals the replication infrastructure to process pending invalidations to bring the sequence number to a point where the request can be processed. This optimization compromises transparency, but we speculate that the necessary modifications to the server would generally not be too invasive.

Second, our protocol sends each invalidation to all replicas even if a replica does not currently have a valid copy of the object being invalidated. We take this approach for simplicity and because we primarily target environments that trade cheap bandwidth and storage for improved availability and responsiveness and where replicas are therefore able to maintain valid copies of most data. Our protocols could be extended to more traditional caching environments where replicas maintain small subsets of data by adding callback state [26]. Given our target environment, we have no current plans to pursue this optimization.

4 Prototype

We have developed a prototype that implements the algorithm described in Section 3. The implementation includes essentially all of the features described above: the prototype accepts updates at servers and reads at replicas, it prioritizes and orders messages and requests at the server and replica using the policies described above, it includes custom implementations of persistent message queues [27] to ensure end-to-end reliable communication, it implements low-priority network send using a server-side congestion control protocol [48], and the prototype includes operation across and recovery from replica, server, and network failures. Deployment does depend on two additional subsystems that are outside the scope of this project: a protocol for limiting the clock skew between each replica and the origin server [37] and a policy for prioritizing which documents to push to which replicas [23, 49], which may, in turn, require some facility for gathering read frequency information from replicas [41].

Our prototype is implemented in Java, C, and C++ on a Linux platform, but we expect the server code to be readily portable to any standard operating system and the replica code to be portable to any system that supports mounting an NFS server.

The rest of this section discusses internal details and design decisions in the server and replica implementations.

4.1 Server

The server is implemented as user-level daemon that provides an interface for local write insertions and remote reads. It uses the local file system for file storage. Note that rather than store per-file sequence numbers, which the protocol sends with demand read replies, our prototype only maintains a global sequence number. The algorithm operates as described in Section 3 except the server includes the current global sequence number when sending a demand reply rather than the sequence number of the object's most recent update. This simplification can force a replica to process more invalidation messages before processing a demand reply; the resulting protocol thus continues to provide sequential consistency, but its performance and availability may be reduced compared to the full protocol.

The server uses a custom-built persistent message queue for sending updates and invalidations to each replica. The implementation buffers invalidation messages on the server's disk, manages TCP connections be-

tween the server and replicas, and buffers pending invalidations and updates in the replica's memory sorted by sequence number. The implementation ensures end-to-end, at-least-once message delivery by allowing a replica to wait to process a pending message until the message's deadline, read the message and apply it to its local persistent state, and finally explicitly tell the replica's message layer to acknowledge the message to the server's message layer.

The use of a persistent message queue for delivering invalidation messages simplifies our implementation by avoiding the need for a separate resynchronization protocol to handle failures [4].

Each update channel between the server and a replica is similar to a persistent message queue except (a) the server buffer is in memory because it is permissible to lose an update if the server crashes and (b) messages are queued in priority rather than FIFO order at the server. A key optimization in our implementation of the update queue is to enqueue an updated file's name rather than the updated file's body. As described in Section 3, our update protocol only ever sends the most recent version of a file, so there is no need for each queue to maintain its own copies of files. A potential future optimization would be to send diffs rather than the entire new file [38].

To provide a low-priority network channel for updates that does not interfere with other network traffic, we reimplement TCP-Nice [48] as a user-level protocol that makes use of libpcap for packet monitoring to measure round-trip times. This implementation retains TCP-Nice's non-interference properties, but because of the additional measurement overheads at user-level, the implementation may be too conservative and may therefore realize somewhat lower network utilization than an in-kernel implementation.

4.2 Replica

Our replica exports the system's shared state via a local user-level NFS file server [36]. The replica mounts this local file server as if it were a normal NFS server, allowing local processes to access shared data as if they were stored in a standard file system. The replica's in-kernel NFS client sends all requests to the local user-level NFS server, which implements our replication algorithm.

Our implementation uses the local file system for storage. Each shared file is represented by two local files: a *shadow file* for metadata (whether the file is valid and the version number of the local copy) and a *data file* for the body of valid files.

4.3 Limitations to transparency

Our goal is to provide transparent replication to existing applications, but the system does expose a few aspects of replication. Some of these issues are implementation choices and some are more fundamental.

In our current implementation, an application at the server inserts updates into the system using a special write call that includes the object ID, the updated data, and the replication priority. We provide this interface to allow applications to control the replication policy. An alternative would be to intercept write calls at the origin server as we now intercept read calls at the replicas. In such an implementation, the system would have to implement a default policy for prioritizing updates by, for example, tracking the write rate of each object at the server, tracking the read rate of each object at each client, propagating read frequency information to the server, and estimating the priorities of an update as the read rate divided by the write rate and scaled by the object size [49].

A more fundamental issue is that the correct configuration of a replicated service may depend on the internal structure of a service. For example, we currently set a single Δ value to limit the staleness of a replica, but it might be desirable to allow different updates to specify different Δ values. Similarly, our current interface applies each update individually, but some applications may wish group a set of updates into a single atomic group. Finally, although we focus on dissemination services, it may be desirable restructure a more complex services into different pieces with different replication strategies for each piece. Some services, for instance, may not replicate some critical pieces for security when replicas are less trusted than the origin server. Or, some services may wish to make use of different consistency protocols for different subsets of data [47].

5 Evaluation

We evaluate our traces using two approaches: by employing a trace-driven simulator and constructing a prototype.

5.1 Simulation methodology

Our trace-driven simulator models an origin server and twenty replicas, and it assumes that the primary bottleneck in the system is the network bandwidth from the origin server. To simplify analysis and comparisons among algorithms, we assume that the bandwidth available to the system does not change throughout a simulation run. As described below, we take the size of objects from a trace, and we assume that the size of control messages is insignificant compared to the size of objects, so we consider them to be of 0 size. Transferring an object over the network thus consumes a link for $objectsize/bandwidth$ seconds, the delay from when a message is sent to when it is received is $nwLatency + messageSize/bandwidth$, and by default we assume that the *nwLatency* of the network between the server and a replica is 200ms +/- 90%.

We compare TRIP's *FIFO-Delayed-Invalidation/Priority-Delayed-Update* algorithm to (a) *FIFO-Inval/Demand-Update* which delivers updates eagerly in FIFO order and which does no prefetching, (b) *FIFO-Inval/Priority-Update* which delivers both invalidations and updates eagerly while still enforcing sequential consistency, and (c) *FIFO-Push-All*, which pushes all updates to all replicas. All of the algorithms enforce sequential consistency, and we assume that the system also requires a Δ-coherence guarantee of $\Delta = 60$ seconds, which the FIFO-Inval algorithms naturally meet, which the Delayed algorithm consciously enforces, and which the FIFO-Update-All algorithm may or may not meet depending on available bandwidth.

5.1.1 Workload

We evaluate the algorithms using a trace-based workload of the Web site of a major sporting event [2] hosted at several geographically distributed locations. The logs contain a total of 22.8 million requests from clients and 281 thousand writes occurring at the origin server, and span one day. We use logs in two formats: standard web traces and update traces. Web traces are traces of requests by clients at each replica, whereas update traces contain information about which object was modified and when.

In order to simplify simulations we ignore certain entries in our trace file. In particular, we remove from the trace files (1) all dynamic requests, (13.9%) (2) all requests that do not contain 200 or 304 as server return codes, (36.7%) (3) entries that appear out of order in the trace files (0.58%), and (4) requests that our parser fails to parse (0.17%).

We eliminate those requests with return codes other than 304 and 200 because we assume that the expensive operations at a replica are those that potentially lead to communication with the origin server. Although requests that result in error codes of 302 (server redirection) are valid requests, we remove them from our traces because those requests reappear in our trace files as requests with 304 or 200 as return codes. We remove out-of-order requests because they pose a problem for the event queue in our trace-driven simulator. Given that they are infrequent we do not believe they influence our results. Finally, we remove requests that have valid return codes but that our conservative trace parser fails to parse. Since the number of requests we remove because of our parser is small, we do not believe removing them influences our results.

5.1.2 Prediction policy

Our interface allows a server to use any algorithm to choose the priority of an update, and this paper does not attempt to extend the state of the art in prefetch prediction. A number of standard prefetching prediction algorithms exist [17, 22, 23, 40, 49] or the server may make use of application-specific knowledge to prioritize an item (e.g., a news editor may know that the day's headline article will be widely read before the system has measured the story's read frequency). Our simple default heuristic for estimating the benefit/cost ratio of one update compared to another is to first approximate the probability that the new version of an object will be read before it is written as the observed read frequency of the object divided by the observed write frequency of the object and then to set the relative priority of the object to be this probability divided by the object's size. This algorithm appears to be a reasonable heuristic for server push-update protocols: it favors read-often objects over write-often objects and it favors small objects over large ones.

[2] The 2000 Summer Olympic games

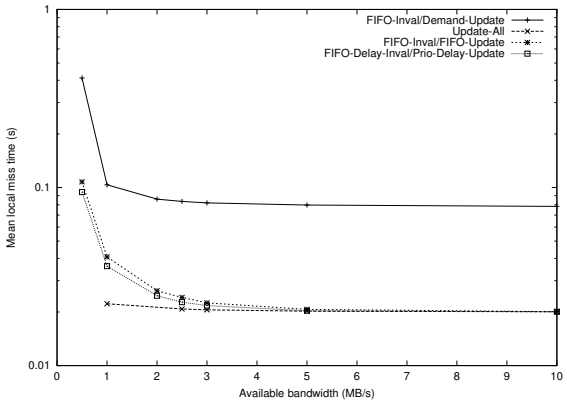

Fig. 5: The effect of bandwidth availability on response times

5.2 Simulation results

Our primary simulation results are that (1) self-tuning prefetching can dramatically improve the response time of serving requests at replicas compared to demand-based strategies, (2) when prefetching is used, delaying application of invalidation messages by up to 60 seconds provides a modest additional improvement in response times, (3) although a push-all strategy enjoys excellent response times by serving all requests directly from replicas' local storage, this strategy is fragile in that if update rates exceed available bandwidth for an extended period of time, the service must either violate its Δ-consistency guarantee or become unavailable, and (4) by maximizing the amount of valid data at replicas, prefetching can improve availability by masking disconnections between a replica and the origin server.

5.2.1 Response times and staleness

In Figure 5 we quantify the effect of available bandwidth on client-perceived response times. We assume that client requests for valid objects at the replica are satisfied in 20ms, whereas requests for invalidated objects are forwarded from the replica to the origin over a network with

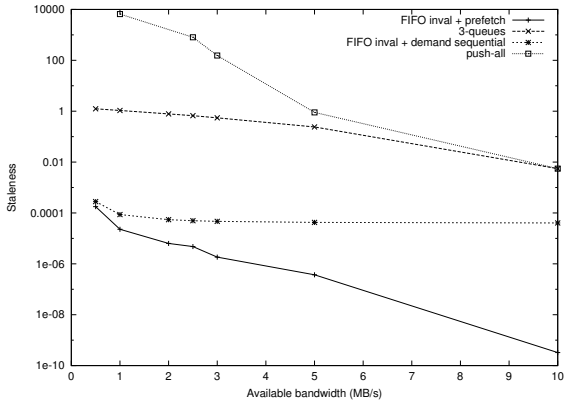

Fig. 6: Average staleness of data served by replicas.

an average round-trip latency of 200ms as noted above. Additionally, Figure 6 plots the average *staleness* observed by a request. We define staleness as follows. If a replica serves version k of an object after the origin site has already (in real time) written version j ($j > k$), we define the staleness of a request to be the difference between when the request arrived at the replica and when version $k+1$ was written. To facilitate comparison across algorithms, this average staleness figure includes non-stale requests in the calculations. We omit due to space constraints a second graph that shows the (higher) average staleness observed by the subset of reads that receive stale data. In both figures, we vary the available bandwidth on the x-axis.

The data indicate that the simple FIFO-Push-All algorithm provides much better response time than the FIFO-Inval/Demand-Update strategy, speeding up responses by a factor of at least four for all bandwidth budgets examined. However, this comparison is a bit misleading as Figure 6 indicates: for bandwidth budgets below 2.1MB/s, FIFO-Push-All fails to deliver all of the updates and serves data that becomes increasingly stale as the simulation progresses. If the system enforces Δ-coherence with $\Delta = 60$ seconds, FIFO-Push-All replicas would be forced to either violate this freshness guarantee or become unavailable when the available bandwidth falls below about 5MB/s.

The systems that use self-tuning prefetch have significant advantages over both FIFO-Push-All and FIFO-Inval/Demand-Update even when they enforce both sequential consistency and a Δ-coherence guarantee with $\Delta = 60$ seconds. When available bandwidth exceeds 5MB/s they match FIFO-Push-All's excellent response time and provide 4x speedups compared to the Demand-Update system. At lower bandwidths, these algorithms meet the timeliness bound of 60 seconds, but they still significantly outperform the Demand-Update strategy. For example, when 2MB/s of bandwidth is available (about 40% of the bandwidth required for FIFO-Push-All to meet the $\Delta = 60s$ constraint), FIFO-Inval/Priority-Update and FIFO-Delayed-Inval/Priority-Delayed-Update provide speedups of 3.3 and 3.5 respectively compared to FIFO-Inval/Demand-Update; and FIFO-Update-All provides only additional speedups of 1.3 and 1.2 despite the latter's liberties with the system's freshness requirements. Even at low bandwidths, the self-tuning prefetching algorithms get significantly better response time than the demand algorithm because (a) the self-tuning network scheduler allows prefetching to occur during lulls in demand traffic even for a heavily loaded system and (b) the priority queue at the origin server ensures that the prefetching that occurs is of high benefit/cost items.

5.3 Bandwidth v. availability

We measure the replication policies' effect on availability as follows. For each run of our simulator, we randomly choose a point in time when we assume that the origin server becomes unreachable to replicas. We simulate a failure at that point in time and measure the length of time that the system can continue to function before any replica receives a request that it cannot mask due to disconnection. We refer to this duration as the *mask duration*. We assume that systems enforce Δ-coherence with $\Delta = 60$ seconds before the disconnection but that disconnected replicas maximize their mask duration by stopping their processing of invalidations and updates during disconnections and extending Δ as long as they can continue to service requests. Note that given this data, the impact of enforcing shorter Δs during disconnections can be estimated as the minimum of the time reported here and the Δ limit enforced.

Figure 7(a) shows the distribution of mask durations when we configure our system with 3.5MBps of available bandwidth available. We show on the x-axis the mask time and use the y-axis the fraction of samples in the cumulative distribution. Figure 7(b) shows how the average mask duration varies with bandwidth for the FIFO-Inval/Demand-Update, FIFO-Inval/Priority-Update, and FIFO-Delayed-Inval/Priority-Delayed-Update algorithms (note that the Demand-Update algorithm's line is near the x axis.) We plot along with the lines, bars representing confidence intervals between 5% and 95% for all samples. Different trials show high variability due to impact of the timing of failures.

We note that the FIFO-Inval/Demand-Update algorithm performs poorly. In Figure 7(a), the line closely follow $y = 0$, indicating virtually no ability to mask failures. This poor behavior arises from the fact that a replica waits until the arrival of a request for an object to refresh that object in its cache. As a result, the first request for an object after that object is modified at the server causes the replica to experience an unmaskable failure. On the

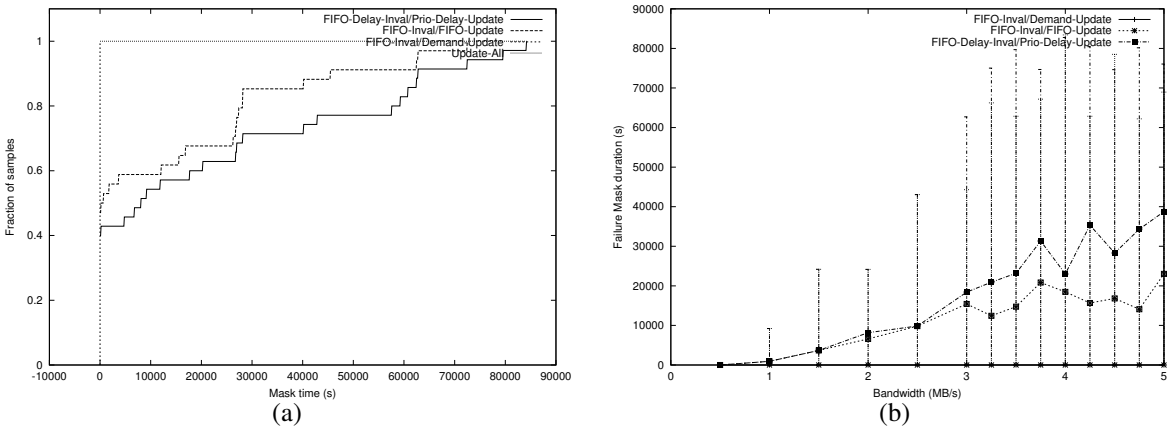

Fig. 7: (a) Distribution of mask durations and (b) dependence of mask duration on bandwidth.

other hand, as Figure 7(a) indicates, the FIFO-Update-All algorithm can mask all failures due to the fact that at any point in time, the entries in a replica's cache form a sequentially consistent (though potentially stale) view of data.

The FIFO-Delayed-Inval/Priority-Delayed-Update algorithm outperforms the eager FIFO-Inval/Demand-Update algorithm in both graphs by maximizing the amount of local valid data. We note that both algorithms provide average masking times of thousands of seconds for bandwidth of 1.5MB/s and above and that providing additional bandwidth allows these systems to prefetch more data and hence mask a failure for a longer duration. As noted in Section 3, systems may choose to relax their Δ-coherence time bound to some longer Δ' value during periods of disconnection to improve availability. These data suggest that systems may often be able to completely mask failures that last the maximum maskable duration Δ' even for relatively large Δ' limits during disconnections. Finally note the wide experimental variability in measured masking durations. Although replication often allows systems to mask failures for long durations, they can occasionally get unlucky and block on a demand read soon after a failure.

5.4 Prototype measurements

We run the system over the Emulab testbed [51]. The network between our origin servers and replicas is configured to have 3Mbit/s of bandwidth and 100ms of latency. We mount the local user-level file server using NFS with attribute caching disabled.

Figure 8 shows the values read from two variables stored in the shared file system provided by the TRIP prototype. In this experiment, the origin server increments the value in a file, alternating between the files every five seconds, and each replica reads each file every second. In this system, we allow prefetching to one replica but disable it for the other. First, note that both replicas observe

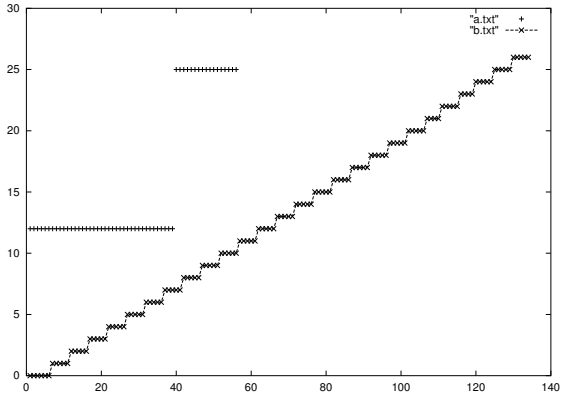

Fig. 8: Consistency and timeliness observed by a TRIP replica when one replica is able to prefetch and the other is not.

sequentially consistent views of the data—once a replica observes that the second file has been incremented, it never observes an older value of the first file. Second note that the first replica applies invalidations as soon as it receives updates to mask them, but the second replica receives no prefetched updates and therefore delays applying invalidations as long as it can. Therefore, during the first 60 seconds of the experiment, the second replica sees the original value of the data; after that, it sees values that ware current 60 seconds earlier.

6 Related work

Yu and Vahdat [55] show that minimizing the time between when an update occurs and when it propagates maximizes system availability for any given consistency constraint. Our protocol demonstrates how to exploit this observation for dissemination workloads by integrating consistency and self-tuning prefetch.

Our argument is similar in spirit to Hill's position that multiprocessors should support simple memory consistency models like sequential consistency rather than weaker models [25]. Hill argues that speculative execution reduces the performance benefit that weaker models

provide to the point that their additional complexity is not worth it. We similarly argue that for dissemination workloads, as technology trends reduce the cost of bandwidth, prefetching can reduce the cost of sequential consistency so that little additional benefit is gained by using a weaker model and exposing more complexity to the programmer.

A number of efforts address providing coherence for individual web objects [53, 35, 52], but little that addresses providing consistency across objects or integrating consistency and prefetching in this environment.

Several studies have examined ways to cache pages that are dynamically generated based on some underlying data [43]. Challenger et al.'s [12, 13] Data Update Propagation allows replicas to cache pages or page fragments that are dynamically generated at an origin server by tracking dependencies between pages and the underlying data used to generate them and by sending invalidations or updates to cached pages when the underlying data change.

Several strategies for mixing updates and invalidates have been explored for multicast networks. Fei [18] simulates a threshold policy that uses an object's read rate, its write rate, and the network topology to choose between multicasting updates on one hand or multicasting invalidates and unicasting demand read replies. SPREAD [42] dynamically builds application-level invalidation and multicast hierarchies for each volume of objects, with each proxy cache using a threshold scheme to choose between polling, joining the invalidation tree, and joining the update tree for each volume. Li and Cheriton [33] propose a push-all multicast strategy that separates data into volumes so that a replica only receives updates for volumes it has referenced. All of these multicast proposals provide a best-effort approximation of linearizability by immediately applying all messages to reduce the risk that reordering compromises consistency and to minimize real-time staleness. A potential avenue for future work is to develop a way for our algorithm with delayed message delivery, sequential consistency constraints on message order, and self-tuning prefetch to make use of multicast or application-level multicast to scale to larger numbers of replicas.

In replicated databases, several systems have explored ways to allow different updates to specify different consistency requirements. Lazy Replication [30] allows an update to enforce causal, sequential, or linearizable consistency. Bayou [45] allows each update to specify its consistency dependencies and inconsistency-resolution techniques. These systems both focus on multi-writer environments and eventually propagate all updates to all replicas.

A number of web prefetching systems have been proposed, but most rely on statically tuned thresholds to decide which objects are valuable enough to push in order to reduce network and server interference [17, 22, 23, 40]. Davison et al. [16] propose using a connectionless transport protocol and using low priority datagrams (the infrastructure for which is assumed) to reduce network interference. Crovella et al. [14] show that a window-based rate controlling strategy for sending prefetched data leads to less bursty traffic and network smaller queue lengths. Kokku et al. [29] describe a threshold-free prefetching system that like our system also makes use of TCP-Nice [48] to avoid network interference. They focus on supporting prefetching of soon-to-be-accessed objects by client browsers rather than pushing of updates by origin servers to replicas, and they do not consider the problem of maintaining consistency for data that may be prefetched long before it is used.

7 Conclusion

This paper explores integrating data prefetching and data consistency to enable transparent replication of large-scale information dissemination services. Our system succeeds in integrating prefetching and consistency by (1) providing self-tuning push-based prefetch from the server and (2) buffering and carefully scheduling the application of invalidations and updates at replicas to maximize the amount of valid data—and therefore maximize the hit rate, minimize the response time, and maximize availability—at a replica. Our analysis of simulations and our evaluation of a prototype implementation support the hypothesis that it is feasible to provide transparent replication for information dissemination applications by carefully integrating consistency and prefetching.

A limitation of this work is its focus on information dissemination applications. This class of applications is important, but in the future we hope to apply our protocol as one part of a more general system where one subset of the data is read-only at the replicas, where another subset is read/write at the replicas, and where different subsets use different consistency algorithms [47].

References

[1] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Turbo-charging dynamic web data with akamai edgesuite. Akamai White Paper, 2001.

[3] A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Internat. Workshop on Web Caching and Content Distribution*. Aug. 2002.

[4] M. Baker. *Fast Crash Recovery in Distributed File Systems*. Ph.D. thesis, University of California at Berkeley, 1994.

[5] M. Baker, S. Asami, et al. Non-Volatile Memory for Fast, Reliable File Systems. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 10–22. Sep. 1992.

[6] Bent and Voelker. Whole page performance. In *Internat. Workshop on Web Caching and Content Distribution*. Sep. 2002.

[7] S. Blake, D. Black, et al. An architecture for differentiated services. Tech. Rep. RFC 2475, IETF, Dec. 1998.

[8] R. Braden, D. Clark, et al. Integrated services in the internet architecture: an overview. Tech. Rep. RFC 1633, IETF, Jun. 1994.

[9] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, July/August 2001.

[10] R. Burns, R. Rees, et al. Consistency and locking for distributing updates to web servers using a file system. In *Workshop on Performance and Architecture of Web Servers*. Jun. 2000.

[11] P. Cao, J. Zhang, et al. Active Cache: Caching Dynamic Contents on the Web. In *Proc. Middleware 98*. 1998.

[12] J. Challenger, P. Dantzig, et al. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *ACM/IEEE, Supercomputing '98*. Nov. 1998.

[13] J. Challenger, A. Iyengar, et al. A scalable system for consistently caching dynamic web data. In *IEEE INFOCOM*. Mar. 1999.

[14] M. Crovella and P. Barford. The network effects of prefetching. In *Proc. of IEEE Infocom*. 1998.

[15] M. Dahlin, B. Chandra, et al. End-to-end wan service availability. *ACM/IEEE Transactions on Networking*, 11(2), Apr. 2003.

[16] B. D. Davison and V. Liberatore. Pushing politely: Improving Web responsiveness one packet at a time (extended abstract). *Performance Evaluation Review*, 28(2):43–49, Sep. 2000.

[17] D. Duchamp. Prefetching Hyperlinks. In *2nd USENIX Symposium on Internet Technologies and Systems*. Oct. 1999.

[18] Z. Fei. A novel approach to managing consistency in content distribution networks. In *Internat. Workshop on Web Caching and Content Distribution*. Jun. 2001.

[19] R. Fielding, J. Gettys, et al. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, Jun. 1999.

[20] I. Foster, C. Kesselman, et al. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Global Grid Forum*. Jun. 2002.

[21] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*. Jun. 1998.

[22] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*. Oct. 1993.

[23] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *HOTOS95*, pp. 51–55. May 1995.

[24] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, Jul. 1990.

[25] M. Hill. Multiprocessors should support simple memory consistency models,. In *IEEE Computer*. Aug. 1998.

[26] J. Howard, M. Kazar, et al. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, Feb. 1988.

[27] MQSeries: An introduction to messaging and queueing. IBM Corporation GC33-0805-01, Jul. 1995.

[28] D. Karger, E. Lehman, et al. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Twenty-ninth ACM Symposium on Theory of Computing*. 1997.

[29] R. Kokku, P. Yalagandula, et al. Nps: A non-interfering deployable web prefetching system. In *4th USENIX Symposium on Internet Technologies and Systems*. Mar. 2003.

[30] R. Ladin, B. Liskov, et al. Providing high availability using lazy replication. *ACM Trans. on Computer Systems*, 10(4):360–361, Nov. 1992.

[31] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.

[32] —. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sep. 1979.

[33] D. Li and D. R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In *Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99)*. Oct. 1999.

[34] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Tech. Rep. CS-TR-180-88, Princeton, 1988.

[35] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proc. of the Seventeenth International Conference on Distributed Computing Systems*. May 1997.

[36] D. Mazires. A toolkit for user-level file systems. In *2001 USENIX Technical Conference*, pp. 261–274. Jun. 2001.

[37] D. Mills. Network time protocol (version 3) specification, implementation and analysis. Tech. rep., IETF, 1992.

[38] A. Muthitacharoen, B. Chen, et al. A low-bandwidth network file system. In *SOSP01*. Oct. 2001.

[39] K. Nichols, V. Jacobson, et al. A two-bit differentiated services architecture for the internet. Tech. Rep. RFC 2638, IETF, Jul. 1999.

[40] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *ACM SIGCOMM Conference*, pp. 22–36. Jul. 1996.

[41] R. V. Renesse, K. Birman, et al. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. on Computer Systems*, 21(2):164–206, May 2003.

[42] P. Rodriguez and S. Sibal. SPREAD: Scalable platform for reliable and efficient automated distribution. In *Proceedings of the 9th International WWW Conference*. May 2000.

[43] B. Smith, A. Acharya, et al. Exploiting result equivalence in caching dynamic web data. In *2nd USENIX Symposium on Internet Technologies and Systems*. Oct. 1999.

[44] B. Terry, A. Demers, et al. Session guarantees for weakly consistent replicated data. In *International Conference on Parallel and Distributed Information Systems*, pp. 140–149. Sep. 1994.

[45] D. Terry, M. Theimer, et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15th ACM-Symposium on Operating Systems Principles*, pp. 172–183. Dec. 1995.

[46] A. Vahdat, M. Dahlin, et al. Active Naming: Flexible Location and Transport of Wide-Area Resources. In *2nd USENIX Symposium on Internet Technologies and Systems*. Oct. 1999.

[47] M. van Steen, P. Homburg, et al. The Architectural Design of Globe: A Wide-Area Distributed System. Tech. Rep. IR-422, Vrije Universiteit, Amsterdam, Mar. 1997.

[48] A. Venkataramani, R. Kokku, et al. TCP-Nice: A mechanism for background transfers. In *OSDI02*. Dec. 2002.

[49] A. Venkataramani, P. Yalagandula, et al. Potential costs and benefits of long-term prefetching for content-distribution. In *Web Caching and Content Distribution Workshop*. Jun. 2001.

[50] A. Whitaker, M. Shaw, et al. Denali: Lightweight virtual machines for distributed and networked applications. In *2002 USENIX Technical Conference*. Jun. 2002.

[51] B. White, J. Lepreau, et al. An integrated experimental environment for distributed systems and networks. In *5th Symp on Operating Systems Design and Impl.* Dec. 2002.

[52] K. Worrell. *Invalidation in Large Scale Network Object Caches*. Master's thesis, University of Colorado, Boulder, 1994.

[53] J. Yin, L. Alvisi, et al. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proc. of the Eighteenth International Conference on Distributed Computing Systems*. May 1998.

[54] C. Yoshikawa, B. Chun, et al. Using Smart Clients to Build Scalable Services. In *1997 USENIX Technical Conference*. Jan. 1997.

[55] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *18th ACM Symposium on Operating Systems Principles*. 2001.