Transparent Information Dissemination *

Amol Nayate[1], Mike Dahlin[1] and Arun Iyengar[2]

[1] University of Texas at Austin, Austin TX 78712, USA
{nayate,dahlin}@cs.utexas.edu
[2] IBM TJ Watson Research Center, Yorktown Heights, NY 10598, USA
aruni@us.ibm.com

Abstract. This paper describes Transparent Replication through Invalidation and Prefetching (*TRIP*), a self tuning data replication middleware system that enables transparent replication of large-scale information dissemination services. The TRIP middleware is a key building block for constructing *information dissemination* services, a class of services where updates occur at an origin server and reads occur at a number of replicas; examples information dissemination services include content distribution networks such as Akamai [1] and IBM's Sport and Event replication system [2]. Furthermore, the TRIP middleware can be used to build key parts of general applications that distribute content such as file systems, distributed databases, and publish-subscribe systems.
Our data replication middleware supports *transparent* replication by providing two crucial properties: (1) sequential consistency to avoid introducing anomalous behavior to increasingly complex services and (2) self-tuning transmission of updates to maximize performance and availability given available system resources. Our analysis of simulations and our evaluation of a prototype support the hypothesis that it is feasible to provide transparent replication for dissemination services. For example, in simulations, our system's performance is a factor of three to four faster than a demand-based middleware system for a wide range of configurations.

1 Introduction

This paper explores integrating self-tuning updates and sequential consistency to provide middleware support for replication of large-scale information dissemination services. We pursue the aggressive goal of supporting *transparent* service replication by providing two key properties.

1. The middleware provides *self-tuning updates* to maximize performance and availability given the system resources available at any moment. Self-tuning updates are crucial for transparent replication because static replication policies are more complex to maintain, less able to benefit from spare system

* This work was supported in part by the Texas Advanced Technology Program, the National Science Foundation (CNS-0411026), and an IBM Faculty Partnership Award.

resources, and more prone to catastrophic overload if they are mis-tuned or during periods of high system load [3].

2. The middleware provides *sequential consistency* [4] with a tunable maximum-staleness parameter to reduce application complexity. Weaker consistency guarantees can introduce subtle bugs [5], and as Internet-scale applications become more widespread, ambitious, and complex, simplifying the programming model becomes increasingly desirable [6]. If we can provide sequential consistency, then we can take a single machine's or LAN cluster's service threads that access shared state via a file system or database and distribute these threads across WAN edge servers without re-writing the service and without introducing new bugs.

Not only is each of these properties important, but their combination is vital. Sequential consistency prevents the use of stale data, which could hurt performance and availability, but prefetching replaces stale data with valid data. Conversely, prefetching means that data are no longer fetched when they are used, so a prefetching system must rely on its consistency protocol for correct operation.

Providing sequential consistency in a large scale system while providing good availability [7] and performance [8] is fundamentally difficult. We therefore restrict our attention to replicated *dissemination services*, in which updates occur at one origin server and multiple edge server replicas treat the underlying replicated data as read-only and perform 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 [1] and IBM's Sport and Event replication system [2] both focus on improving the performance, availability, and scale of dissemination services.

In this paper, we describe the TRIP (Transparent Replication through Invalidation and Prefetching) middleware that integrates self tuning updates with sequential consistency to enable transparent replication for dissemination services. We define the node where updates originate to be the *origin server* and the receiving nodes the *replicas* of the system. Although we focus on dissemination services, more general services can make use of dissemination for subsets of their workloads. We therefore believe that TRIP can be used as a building block for services such as file systems, distributed databases, publisher/subscriber systems, and applications that use per-object-customized consistency [9].

This paper evaluates the TRIP middleware using both trace-based simulation and evaluation of an implementation. Our simulations use access/update traces obtained for a highly accessed sporting and event web site [10]. We build an NFS loopback interface [11] to emulate a smaller version of this web service on our TRIP middleware. Our configuration 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. Although our implementation exports a file system interface, a similar approach could be used to support a database or publisher/subscriber interface to the shared state.

This paper makes three contributions. First, it provides evidence that systems can maintain sequential consistency for some key WAN distributed services despite the CAP dilemma, which states that systems cannot get strong **C**onsistency and high **A**vailability for systems vulnerable to **P**artitions [7]. The replication middleware circumvents this dilemma by (a) restricting the workload it considers and (b) integrating consistency with prefetching. Second, it presents a novel middleware component 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 middleware component to provide evidence for the effectiveness and practicality of the approach.

2 System model

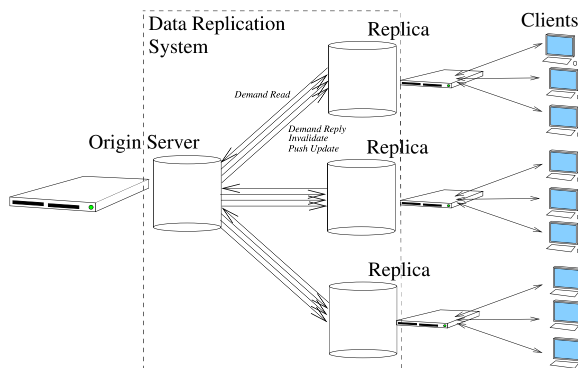

Fig. 1. High level system architecture.

Figure 1 provides a high level view of the environment we assume. An *origin server* and several *replicas* (also called content distribution nodes or edge servers) share data, and logical *clients*—either on the same machine or another—access the service via the replicas, which run service-specific code to dynamically generate responses to requests [1, 12–14]. The system typically uses some application-specific mechanism [2] to direct client requests to a good (e.g., nearby, lightly loaded, or available) replica. The design of such a redirection infrastructure is outside the scope of the paper; instead, we focus on the *data replication middleware* that provides shared state across the origin server and replicas. We focus on supporting on the order of 10 to 100 long-lived replicas that each have sufficient local storage to maintain a local copy of the full set of their service's shared data. Although our protocol remains correct under other assumptions about the number of replicas, replica lifetimes, and whether replicas replicate

all shared data or only a subset, optimizing performance in other environments may require different trade-offs.

2.1 Consistency and timeliness

Evaluating the semantic guarantees of large-scale replication systems requires careful distinctions between *consistency*, which constrains the order that updates across multiple memory locations become *observable* [5] to nodes in the system, *coherence*, which constrains the order that updates to a single location become observable but does not additionally constrain the ordering of updates across different locations, and *staleness*, which constrains the real-time delay between when an update completes and when it becomes observable. Adve discusses the distinction between consistency and coherence in more detail [15].

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." [4] 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 a distributed storage system.

Typically, providing sequential consistency is expensive in terms of latency [8, 16] or availability [7]. However, we restrict our study to *dissemination services* that have one writer and many readers, and we enforce *FIFO consistency* [8] under which 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 or the weaker causal consistency.

Although sequential consistency provides strong semantic guarantees at replicas, clients of those 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 stale (but consistent) data. For example, a network partition between the origin server and replica could cause the client of a stock ticker service to observe the anomalous behavior of a stock price not changing for several minutes. We note that in this scenario, physical time acts as a communications channel outside of the control of the data replication middleware that allows a user to observe anomalous behavior from the replication system. Hence, we allow systems to enforce timeliness constraints on data updates by providing *Δ-coherence*, which requires that any read reflect at least all writes that occurred before the current time minus Δ. By combining Δ-coherence with sequential consistency, TRIP enforces a tunable staleness limit on the sequentially consistent view. The Δ parameter reflects a per-service trade-off between

availability and worst case staleness: reducing Δ improves timeliness guarantees but may hurt availability because disconnected edge servers may need to refuse a request rather than serve overly stale data.

Second, some redirection infrastructures [2] may cause a client to switch between replicas, allowing it to observe inconsistent state. For example, consider two replicas r_1 and r_2 where r_2 processes messages more slowly than r_1, and updates u_1 and u_2 such that u_1 *happens before* [17] u_2. If a client of r_1 sees update u_2, switches to r_2 (which has not seen u_1 yet) and sees data that should have been modified by u_1 but is not, it observes an inconsistency. In [18] we discuss how to adapt Bayou's session consistency protocol [19] to our system to ensure that each client observes a sequentially consistent view regardless of how often the redirection infrastructure switches the client among replicas.

3 Algorithm

TRIP is based on a novel replication algorithm that revolves around two simple parts: (1) the origin'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. We describe the algorithm in the rest of the section.

3.1 Origin server

Algorithm 1 Origin server

```
State
  seqNo; // Global sequence number
  storage; // Seq number + body of each object
  nReplicas; // Number of replicas
  updtChnl[]; // Lossy, prior. order, low prior. link
  invDemChnl[]; // Lossless, FIFO channels
On local call to write(objID, body, priority, timestamp):
  seqNo++;
  storage.update(objId, body, seqNo);
  for (i = 0; i < nReplicas; i++) do
    invDemChnl[i].send(INVAL, objId, seqNo, timestamp);
    updtChnl[i].insert(UPDATE, objId, body, seqNo, priority);
On receiving (READ, objId) from replica:
  (body, objSeqNo) = storage.get(objId);
  invDemChnl[replica].send(REPLY, objId, body, objSeqNo);
  updtChnl[replica].cancel(objId);
```

As we show in the pseudocode in Algorithm 1, the origin server maintains a global monotonically increasing sequence number *seqNo*, local *storage* with the body and sequence number of each object, per-replica channels *invDemChnl[]* for sending invalidations and demand replies, and per-replica channels *updtChnl[]* for

pushing updates. Each *invDemChnl* is a FIFO ordered, lossless network channel, and each *updtChnl* is a priority ordered, low-priority channel.

The algorithm proceeds as follows. 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 *invDemChnl*, and enqueues updates on each replica's *updtChnl*. Each enqueued update includes a *priority* that specifies the update's relative ranking to other pending updates. These priorities can be calculated using existing mechanisms [20–23], or using application-specific knowledge. By adding a *replicaID* parameter to our *write* method, our algorithm can be extended to accommodate per-replica priorities as well.

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

Each *updtChnl* provides an abstraction suited for self-tuning push-based prefetching by (1) buffering updates in a priority queue and (2) sending them across the network using a lossless, blocking, low priority network protocol. Three actions manipulate each per-replica priority queue. First, an *insert* adds an update with a specified priority, replacing any other update to the same *objId*. 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 [24]. Thus, when sufficient bandwidth is available, an *updtChnl* behaves like a lossless FIFO channel and delivers all updates to its replica. When less bandwidth is available, however, (1) it only allows valuable updates to be sent, and (2) it allows unsent updates to the same object to be merged and sent later either when requested by a replica or during a lull in network traffic.

3.2 Replica

The core of each replica is a novel *scheduler* that coordinates the application of invalidations, updates, and demand read replies to the replica's local state. The scheduler has two conflicting goals. On one hand, it would like to delay applying invalidations for as long as possible to minimize the amount of invalid data and thereby maximize local hit rate, maximize availability, and minimize response time. On the other hand, it must enforce sequential consistency and Δ-coherence, so it must enforce two constraints:

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.[3]

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

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.

Algorithm 2 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
On receiving (INVAL, objId, seqNo, timestamp) on invDemChnl:
  pendingInval.put(objId, seqNo, timestamp);
On receiving (UPDATE, objId, body, seqNo) on updtChnl:
  pendingUpdate.put(objId, body, seqNo);
If pendingUpdate.head.seqNo≤pendingInval.nextSeqToProcess():
  // Scheduler applies an update
  (objId, body, seqNo) = pendingUpdate.removeHead();
  if (seqNo ≥ storage.getSeqNo(objId)) then
    storage.update(objId, VALID, seqNo, body);
  if (seqNo == pendingInval.nextSeqToProcess()) then
    pendingInval.doneProcessing(seqNo);
If currentTime() ≤ pendingInval.head.timestamp+delta-maxSkew:
  Scheduler applies an invalidate
  applyNextInval(); // See below
On local call to read(objId):
  if (VALID == storage.getState(objId)) then
    return storage.getBody(objId);
  send(READ, objId) to origin server;
  storage.waitUntilValid(objId);
  return storage.getBody(objId);
On receiving (REPLY, objId, body, seqNo) on invDemChnl:
  while (pendingInval.nextSeqToProcess() ≤ seqNo) do
    applyNextInval(); // See below
  storage.update(objId, VALID, seqNo, body); // Unblock read
applyNextInval() // Internal private method called from above
  (objId, seqNo, timestamp) = pendingInval.readHead();
  if (seqNo ≥ storage.getSeqNo(objId)) // 'At least once' chnl then
    storage.update(objId, INVALID, seqNo);
  pendingInval.doneProcessing(seqNo);
```

Algorithm details. The pseudocode in Algorithm 2 describes the behavior of a replica. Each replica maintains five main data structures. First, a replica maintains a local data store *storage* 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

[3] We show in [18] that enforcing condition C1 yields sequential consistency

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 *storage*; 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.

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.

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 from *pendingInvals*. 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.

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 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 pseudocode 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 this 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 request 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 [15] if program order specifies that r_1 *happens before* r_2. On the other hand, if r_1 and r_2 are issued by independent threads of computation that are not 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 [4, 5]. TRIP therefore provides two options. *Conservative* mode preserves transparancy but requires a read issued while an earlier read is blocking on a miss to block. *Aggressive* mode compromises transparency because it requires knowledge of application internals, but it allows a cached read to pass a pending read miss. Our experiments examine this trade-off in more detail.

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 as long as possible. However, to enforce Δ-coherence, a replica must block all reads if it has not communicated with the origin server for Δ seconds. In a web service environment, blocking a client indefinitely is an undesirable behavior. Therefore, TRIP provides three ways for services to give up some transparancy in order to gain control of recovery in the case where a replica blocks because it is disconnected from the origin server. First, TRIP can reply to read requests from the calling edge server program by returning an error code. Because this approach requires that the edge server program be designed to expect such an error code, it prevents the replication layer from being fully transparent. Second, TRIP can (1) signal the redirection layer [2] to stop sending requests to this replica and (2) signal the local web server infrastructure to close all existing client connections and to respond to subsequent client requests with HTTP redirects [25] to different replicas. Although this approach requires web servers to be augmented with the ability to handle signals from the replication layer, we do not expect these changes to be invasive. Third, TRIP can increase Δ (and thus increase observable data staleness) when it detects a disconnection from the server. Increasing Δ allows the system to further delay applying pending invalidations and thus maximize the amount of valid local data and maximize the amount of time the replica can operate before suffering a miss.

3.3 Limitations and optimizations

Our current protocol faces two limitations that could be addressed with future optimizations. First, as described in Section 2.1 our current protocol can allow a client that switches between replicas to observe violations of sequential consistency. We speculate in [18] that a system could shield a client from inconsistency by adapting Bayou's session guarantees protocol [19]. 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, although our protocols could be extended to more traditional caching environments where replicas maintain small subsets of data by adding callback state [26].

4 Prototype

We have developed a prototype that implements the *conservative* version (Section 3.2) of the algorithm described in Section 3. Deployment depends on two subsystems that are outside the scope of this project: a protocol for limiting the clock skew between each replica and the origin server [27] and a policy for prioritizing which documents to push to which replicas [21, 23], which may, in turn, require some facility for gathering read frequency information from replicas [28]. We discuss limitations of our prototype in more detail in [18].

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 code is available for download from http://www.cs.utexas.edu/users/nayate/TRIP.

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

Origin Server The origin server 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.

To simplify handing failures, the origin server uses a custom persistent message queue [29] for sending updates and invalidations to each replica. Because our protocol only uses the update channel to push update data, the origin server does not store out-bound updates to persistent storage and considers it permissible to lose these updates across crashes. To provide a low-priority network channel for updates that does not interfere with other network traffic, we use an implementation of TCP-Nice [24].

Replica The replica implements a single *read* method to access shared data. The simplicity of this interface allows us to use TRIP as a building block for a variety of replicated applications that require sophisticated interfaces. For example, publish/subscribe systems can be implemented by having the publisher perform write calls to publish data to the matching service, and the matching service can later make read calls to request data to serve to clients. Chen et al. [30] shows an approach that can be adopted to compute priorities for pages in a publisher/subscriber model. For our prototype, however, we build TRIP to export a subset of the interface used by the NFS file system via a local user-level NFS file server [11], allowing the replica to mount this local file server as if it were a normal NFS server. Shared objects are accessed as if they are stored in a standard file system. For simplicity, we respond to reads to invalidated data during disconnections by returning an NFS IO error code to the calling program.

5 Evaluation

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

5.1 Simulation methodology

Our trace-driven simulator models an origin server and twenty replicas and 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. We also assume that bandwidth consumed by control information (invalidate messages, message queue acknowledgments, meta data, etc.) is insignificant compared to the bandwidth consumed transferring objects; we confirm using our prototype that control messages account for less than 1% of the data transferred by the system. Transferring an object over the network thus consumes a link for $objectsize/bandwidth$ seconds, and the delay from when a message is sent to when it is received is given by $nwLatency+messageSize/bandwidth$. By default we simulate a round-trip time (or 2 * *nwLatency*) of 200ms +/- 90% between the origin server and a replica.

We compare TRIP's *FIFO-Delayed-Invalidation/Priority-Delayed-Update* algorithm with two algorithms: *Demand Only*, which delivers invalidates eagerly in FIFO order but does no prefetching, and *Push All* which eagerly pushes all updates to all replicas in FIFO order. We initially assume that the system requires (1) sequential consistency, which all of these algorithms provide, and (2) a Δ-coherence guarantee of $\Delta = 60$ seconds, which Demand Only naturally meets, TRIP consciously enforces, and Push All may or may not meet depending on available bandwidth. We will later modify these assumptions.

We evaluate our algorithms using a trace-based workload of the Web site of a major sporting event [10] hosted at several geographically distributed locations. In order to simplify simulations we ignore those entries in our trace files that

contain dynamic/malformed requests, result in invalid server return codes, or that appear out of order.

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 [20–23] or the server may make use of application-specific knowledge to prioritize an item. 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 [23]. 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.

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) 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, (3) when prefetching is used, delaying application of invalidation messages by up to 60 seconds provides a modest additional improvement in response times, 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.

Response times and staleness In Figure 2 we quantify the effects of different replication strategies on client-perceived response times as we vary available bandwidth. 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 an average round-trip latency of 200ms as noted above. To put these results in perspective, Figure 3 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 simulated 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 under each algorithm that receives stale data.

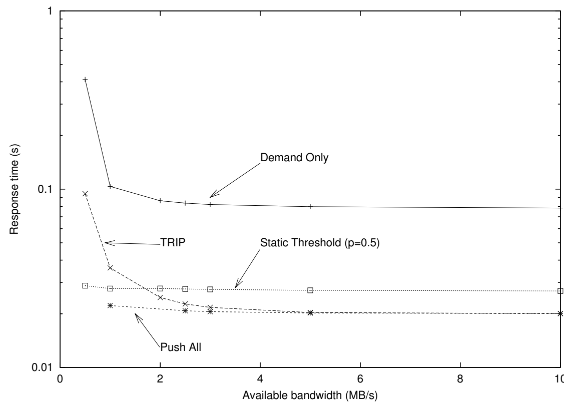

Fig. 2. The effect of bandwidth availability on response times

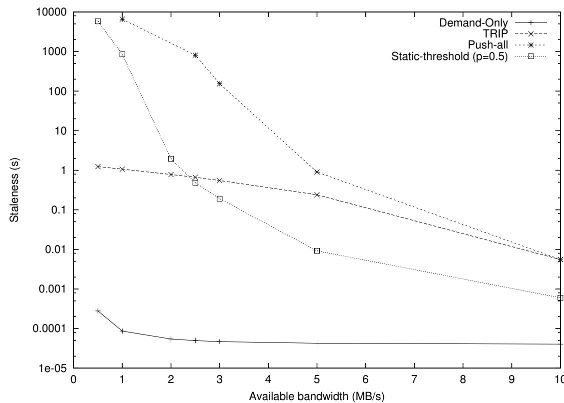

Fig. 3. Average staleness of data served by replicas.

We also show in figures 2 and 3 the latency and staleness yielded when using the *static-threshold-prefetching* algorithm, which prefetches objects when the predicted likelihood of their being accessed exceeds a statically chosen threshold. We plot the behavior of this algorithm when it is tuned to prefetch objects that have a greater than 50% estimated chance of being accessed (denoted *Static Threhold* ($p = 0.5$) on the graph). We note that Push All and Demand Only represent extreme cases of this algorithm with thresholds of 0 (push an update regardless of its likelihood of being accessed) and 1 (only push an update if it is certain to be accessed), respectively.

The data indicate that the simple Push All algorithm provides much better response time than the Demand Only 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 3 indicates: for bandwidth budgets below 2.1MB/s, 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, 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 static-threshold line illustrates precisely the problem with static thresholds. When the system has less than 2MB/s available bandwidth, the static-threshold algorithm yields lower response times than the TRIP algorithm. However, we note that for this bandwidth range the static-threshold algorithm also violates staleness guarantees. Similarly, when the system has more than 2MB/s bandwidth available, the static-threshold algorithm fails to utilize it to reduce response times.

Even at low bandwidths, TRIP gets significantly better response times than the Demand Only algorithm because (a) the self-tuning network scheduler allows prefetching to occur during lulls in demand traffic even for a heavily loaded system [3] and (b) the priority queue at the origin server ensures that the prefetching that occurs is of high benefit/cost items. TRIP's ability to exploit lulls in demand bandwidth also constitutes the reason that when the system has 2MB/s available bandwidth TRIP can outperform static-threshold while still retaining its timeliness guarantees.

Variations of TRIP Due to space constraints, we omit a graph that plots response times for two variations of TRIP. In the first variation, we reduce the Δ parameter to 0 to evaluate the behavior of TRIP when we require replicas to apply all invalidate messages immediately. Under this scenario we find that values of Δ below 60s inflict a modest cost on response times, but this cost falls as available bandwidth increases. For example, at 1MB/s of available bandwidth, the $\Delta = 60s$ case yields 12.6% lower response times than the $\Delta = 0s$ case. However, our second variation of TRIP, *TRIP-aggressive*, which sacrifices some transparency and assumes that parallel read requests are independent, can result in substantial benefits. For example, for a system with 500KB/s of available bandwidth, this optimization improves response time by a factor of 2.5. But, this benefit falls as available bandwidth increases, suggesting that this optimization may become less valuable as network costs fall relative to the cost of requiring programmers to carefully analyze applications to rule out the possibility of unexpected interactions [6].

5.3 Availability

We measure the replication policies' effect on availability as follows. For each of 50 runs of our simulator for a given set of parameters, we randomly choose a point in time when we assume that the origin server becomes unreachable to replicas. We simulate a failure at that moment and measure the length of time 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

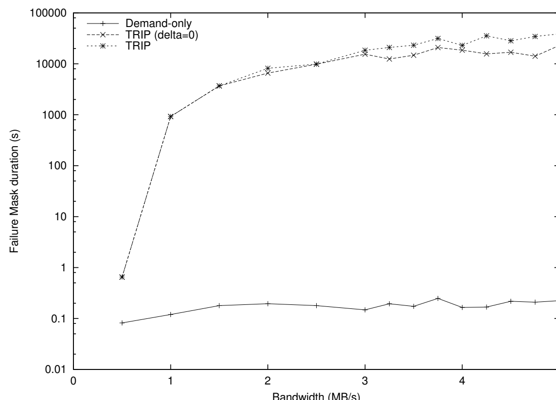

Fig. 4. Dependence of mask duration on bandwidth.

updates during disconnections and extending Δ as long as they can continue to service requests. Thus, during periods of disconnectivity, our system chooses to provide stale data rather than failing to satisfy client requests. Note that given these 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 4 shows how the average mask duration varies with bandwidth for the TRIP, TRIP ($\Delta = 0$), and Demand Only algorithms. Because mask duration is highly sensitive to the timing of a failure, different trials show high variability. We quantify this variability in more detail in an extended technical report [18].

Note that the traditional Demand Only algorithm performs poorly. In Figure 4, the line closely follow $y = 0$, indicating virtually no ability to mask failures. This poor behavior arises because the first request for an object after that object is modified causes a disconnected replica to experience an unmaskable failure. On the other hand, the Push 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 TRIP algorithm outperforms the Demand Only algorithm in the graph by maximizing the amount of local valid data. We note that both TRIP variations 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 even for relatively large Δ' limits.

5.4 Prototype measurements

We evaluate our prototype on the Emulab testbed [31]. We configure the network to consist of an origin server and 4 replicas that receive 5MBps of bandwidth and

200ms round-trip times. We mount the local user-level file server using NFS with attribute caching disabled. For simplicity, we do not monitor object replication priorities in real time but instead pre-calculate them using each object's average read rate, write rate, and size [23].

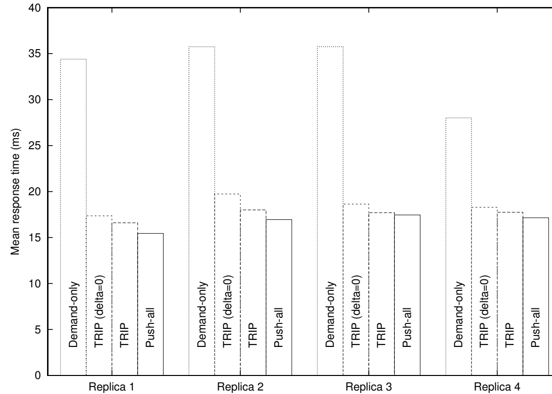

Fig. 5. Replica-perceived response times yielded by the Demand-fetch-only, FIFO-push-all, and the TRIP algorithms

Since the goal of the prototype is to evaluate how our system performs in practice, we use a more realistic evaluation methodology from the one we use for our simulator. In particular, when evaluating our prototype we do not remove any entries from our traces and make no simplifying assumptions about the size of invalidate messages or the behavior of network links. However, due to the lack of data on which resources or objects get accessed to handle dynamic requests, our system incorrectly treats dynamic requests as accesses to static objects.

Figure 5 shows the response times as seen at each of the 4 replicas. We collect these data by replaying at the origin and at each replica the first hour of our update trace and web traces in real time. The response time for a given request is calculated as the difference between when the request arrives at a replica and when its reply is generated. Note that these response times do not represent the end-to-end delay experienced by clients because they do not include the network delays between clients and replicas. However, one can easily compute total end-to-end delays by adding client-replica network delays to this data.

As we see in the graph, the Push All algorithm yields the best response time. For example, it outperforms the Demand Only algorithm by a factor of 2 for 3 of the 4 replicas. We note that at 5MBps bandwidth available to the system, TRIP incurs only minor increases in response times over Push All: 7.5%, 6.2%, 1.4%, and 3.4% overhead for each replica respectively. We also note that by delaying the application of invalidate messages, TRIP with $\Delta = 60s$ reduces response times compared to $\Delta = 0$ by 4.4%, 8.7%, 5.0%, and 3.0% respectively. Because

we use real traces instead of simulated workloads, we notice that our response times vary greatly between replicas. However, our TRIP algorithms consistently outperform the Demand Only algorithm on each replica.

6 Related work

In contrast with TRIP, most existing and proposed replication systems provide neither self-tuning replication nor sequential consistency with tunable staleness.

In particular, most replication systems use static replication policies such as always-conservative demand fetching [1, 32], always-aggressive push-all [2, 33], or hand-tuned threshold-based prefetching [20–22]. Davison et al. [34] propose using a connectionless transport protocol and using low priority datagrams (the infrastructure for which is assumed) to reduce network interference. Chen et al. [30] study content delivery and caching in publish/subscribe systems and discuss methods to estimate the benefit of caching pages that are directly applicable in computing update priorities in our system. In earlier work, we describe a threshold-free prefetching system called NPS [3] that like TRIP makes use of TCP-Nice [24] to avoid network interference. The rest of NPS's design is quite different than TRIP's: NPS focuses on supporting prefetching of soon-to-be-accessed objects by client browsers rather than pushing of updates by origin servers to replicas, and it does not consider the problem of maintaining consistency for data that may be prefetched long before being used.

Most proposed Internet-scale data replication systems focus on ensuring various levels of coherence or staleness or both [35–38], but few provide explicit consistency guarantees. Bradley and Bestavros [39] argue that increasingly complex Internet-scale services will demand sequential consistency and propose a vector-clock-based algorithm for achieving it. They focus on developing a backwards-compatible browser-server protocol and do not explore prefetching. The IBM Sporting and Event CDN system uses a push-all replication strategy and enforces delta coherence via invalidations [40]. Akamai's EdgeSuite [1] primarily relies on demand reads and enforces delta coherence via polling with stronger consistency available via object renaming. Most of these systems use demand reads, but several strategies for mixing updates and invalidates have been explored for multicast networks [41, 35] or broadcast media [42]. These proposals all use static thresholds to control prefetching and provide best-effort consistency, coherence, and timeliness semantics by sending and applying all messages eagerly. A potential avenue for future work is to develop a way for TRIP to make use of multicast or hierarchies 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 [43] allows an update to enforce causal, sequential, or linearizable consistency. Bayou [33] maintains causal consistency at all times and asynchronously reorders operations to eventually reach a global sequentially-consistent ordering of updates. These systems both focus on multi-writer environments and eventually propagate all updates to all replicas. Yu and Vahdat [44] show that in

such systems minimizing the time between when an update occurs and when it propagates maximizes system availability for any given consistency constraint. Our protocol exploits this observation for dissemination workloads by integrating consistency and self-tuning prefetch.

Our argument for sequential consistency is similar in spirit to Hill's position that multiprocessors should support simple memory consistency models like sequential consistency rather than weaker models [6]. 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.

7 Conclusion

This paper explores integrating self-tuning updates and sequential consistency to enable transparent replication of large-scale dissemination services. Our novel architecture succeeds in this goal 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 thus maximize the hit rate, minimize the response time, and maximize availability—at a replica. Our analysis of simulations and our evaluation of a prototype support the hypothesis that it is feasible to provide transparent replication for dissemination applications by 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 [9].

Acknowledgments

We thank Paul Dantzig for his help in obtaining access and update logs, Arun Venkataramani for his crucial help in the design of the algorithm, and Jian Yin for his helpful comments on the presentation.

References

1. Turbo-charging dynamic web data with akamai edgesuite. Akamai White Paper (2001)
2. Challenger, J., Dantzig, P., Iyengar, A.: A scalable and highly available system for serving dynamic data at frequently accessed web sites. In: ACM/IEEE, Supercomputing '98. (1998)

3. Kokku, R., Yalagandula, P., Venkatramani, A., Dahlin, M.: NPS: A non-interfering deployable web prefetching system. In: 4th USENIX Symposium on Internet Technologies and Systems. (2003)
4. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **C-28** (1979) 690–691
5. Frigo, M., Luchangco, V.: Computation-Centric Memory Models. In: Tenth Annual ACM Symposium on Parallel Algorithms and Architectures. (1998)
6. Hill, M.: Multiprocessors should support simple memory consistency models,. In: IEEE Computer. (1998)
7. Brewer, E.: Lessons from giant-scale services. IEEE Internet Computing (2001)
8. Lipton, R., Sandberg, J.: PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton (1988)
9. Gao, L., Dahlin, M., Nayate, A., Zheng, J., Iyengar, A.: Application specific data replication for edge services. In: International World Wide Web Conference. (2003)
10. Sydney Olympic Games Web Site. http://www.olympic.com—site is no longer available (2000)
11. Mazi`eres, D.: A toolkit for user-level file systems. In: 2001 USENIX Technical Conference. (2001) 261–274
12. Awadallah, A., Rosenblum, M.: The vMatrix: A network of virtual machine monitors for dynamic content distribution. In: Internat. Workshop on Web Caching and Content Distribution. (2002)
13. Vahdat, A., Dahlin, M., Anderson, T., Aggarwal, A.: Active Naming: Flexible Location and Transport of Wide-Area Resources. In: 2nd USENIX Symposium on Internet Technologies and Systems. (1999)
14. Whitaker, A., Shaw, M., Gribble, S.: Denali: Lightweight virtual machines for distributed and networked applications. In: 2002 USENIX Technical Conference. (2002)
15. Adve, S., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29** (1996) 66–76
16. Burns, R., Rees, R., Long, D.: Consistency and locking for distributing updates to web servers using a file system. In: Workshop on Performance and Architecture of Web Servers. (2000)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Comm. of the ACM **21** (1978)
18. Nayate, A., Dahlin, M., Iyengar, A.: Integrating Prefetching and Invalidation for Transparent Replication of Dissemination Services. Technical Report TR-03-44, University of Texas at Austin (2003)
19. Terry, B., Demers, A., Petersen, K., Spreitzer, M.J., Theimer, M., Welch, B.: Session guarantees for weakly consistent replicated data. In: International Conference on Parallel and Distributed Information Systems. (1994) 140–149
20. Duchamp, D.: Prefetching Hyperlinks. In: 2nd USENIX Symposium on Internet Technologies and Systems. (1999)
21. Gwertzman, J., Seltzer, M.: The case for geographical pushcaching. In: HOTOS95. (1995) 51–55
22. Padmanabhan, V., Mogul, J.: Using Predictive Prefetching to Improve World Wide Web Latency. In: ACM SIGCOMM Conference. (1996) 22–36
23. Venkataramani, A., Yalagandula, P., Kokku, R., Sharif, S., Dahlin, M.: Potential costs and benefits of long-term prefetching for content-distribution. In: Web Caching and Content Distribution Workshop. (2001)
24. Venkataramani, A., Kokku, R., Dahlin, M.: TCP-Nice: A mechanism for background transfers. In: OSDI02. (2002)

25. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Misinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF (1999)
26. Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M.: Scale and Performance in a Distributed File System. ACM Trans. on Computer Systems **6** (1988) 51–81
27. Mills, D.: Network time protocol (version 3) specification, implementation and analysis. Technical report, IETF (1992)
28. Yalagandula, P., Dahlin, M.: SDIMS: A scalable distributed information management system. Technical Report TR-03-47, University of Texas Dept. of CS (2003)
29. MQSeries: An introduction to messaging and queueing. IBM Corporation GC33-0805-01 (1995)
30. Chen, M., LaPaugh, A., Singh, J.P.: Content distribution for publish/subscribe services. In: Proceedings of the International Middleware Conference. (2003)
31. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: 5th Symp on Operating Systems Design and Impl. (2002)
32. Chankhunthod, A., Danzig, P., Neerdaels, C., Schwartz, M., Worrell, K.: A Hierarchical Internet Object Cache. In: 1996 USENIX Technical Conference. (1996)
33. Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Demers, A.: Flexible Update Propagation for Weakly Consistent Replication. In: 16th ACM Symposium on Operating Systems Principles. (1997)
34. Davison, B.D., Liberatore, V.: Pushing politely: Improving Web responsiveness one packet at a time (extended abstract). Performance Evaluation Review **28** (2000) 43–49
35. Li, D., Cheriton, D.R.: Scalable web caching of frequently updated objects using reliable multicast. In: Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99). (1999)
36. Worrell, K.: Invalidation in Large Scale Network Object Caches. Master's thesis, University of Colorado, Boulder (1994)
37. Yin, J., Alvisi, L., Dahlin, M., Lin, C.: Volume Leases to Support Consistency in Large-Scale Systems. IEEE Transactions on Knowledge and Data Engineering **11** (1999) 563–576
38. Yin, J., Alvisi, L., Dahlin, M., Iyengar, A.: Engineering web cache consistency. ACM Transactions on Internet Technologies **2** (2002)
39. Bradley, A., Bestavros, A.: Basis token consistency: Supporting strong web cache consistency. In: GLOBECOMM. (2003)
40. Challenger, J., Dantzig, P., Iyengar, A., Squillante, M., Zhang, L.: Efficiently serving dynamic data at highly accessed web sites. IEEE/ACM Transactions on Networking **12** (2004) 233–246
41. Fei, Z.: A novel approach to managing consistency in content distribution networks. In: Internat. Workshop on Web Caching and Content Distribution. (2001)
42. Acharya, S., Franklin, M., Zdonik, S.: Balancing push and pull for data broadcast. In: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, ACM Press (1997) 183–194
43. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S.: Providing high availability using lazy replication. ACM Trans. on Computer Systems **10** (1992) 360–361
44. Yu, H., Vahdat, A.: The costs and limits of availability for replicated services. In: 18th ACM Symposium on Operating Systems Principles. (2001)