Transparent Information Dissemination

Amol Nayate and Mike Dahlin — University of Texas at Austin
Arun Iyengar — IBM TJ Watson Research

Abstract

This paper explores integrating self-tuning updates and sequential 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 our data replication system 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. We meet these aggressive consistency and self-tuning goals using a novel architecture that (1) pushes invalidations on reliable FIFO network channels, (2) pushes updates on unreliable, priority-ordered, low-priority network channels, and (3) carefully schedules the application of invalidations and updates at each 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. For example, in simulations our system's performance is a factor of three to four faster than a demand-based system for a wide range of configurations.

1 Introduction

This paper explores integrating self-tuning updates and sequential consistency to enable transparent replication of large-scale information dissemination services. Researchers are working to develop programming environments [2, 12, 52, 23] and scalable servers [3, 56] for distributing service code to replicas across a network in order to improve service availability [18, 35, 63] 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.

We pursue the aggressive goal of developing a data replication toolkit that supports *transparent* service replication by providing two key properties.

1. The toolkit 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 resources, and more prone to catastrophic overload if they are mis-tuned or during periods of high system load [33].
2. The toolkit provides *sequential consistency* [37] with a tunable maximum-staleness parameter to reduce application complexity. Weaker consistency guarantees can introduce subtle bugs [25], and as Internet-scale applications become more widespread, ambitious, and complex, simplifying the programming model becomes increasingly desirable [29]. 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. Strong 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 near the time they are used, so a prefetching system must rely heavily on its consistency protocol for correct operation.

Providing strong consistency guarantees in a large scale system while providing good availability [10] and performance [39] is fundamentally difficult. We therefore restrict our attention to 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 [13] both focus on improving the performance, availability, and scale of dissemination services. Furthermore, we believe that this case represents an important building block for more general services with per-object-customized consistency [26, 53].

In this paper, we describe the TRIP (Transparent Replication through Invalidation and Prefetching) system that integrates self tuning updates with sequential consistency in order to provide transparent replication for dissemination services.

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 [54]. 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 local hit rate, minimize response time, and maximize availability.

This paper evaluates TRIP using both trace-based simulation and evaluation of an implementation. Our simulations use traces of access to the 2000 Summer Olympics web site, a large-scale information dissemination service that was served from several geographically distributed replicas. Our prototype provides a file system interface at each replica via a local NFS server. 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 simulations our system's performance is a factor of three to four faster than a demand-based system for a wide range of assumptions about available bandwidth. Third, whereas strong consistency might be expected to hurt availability [63], prefetching updates and carefully scheduling the application of invalidations allows replicas to maintain local copies of large fractions of system state and thereby mask server failures and network partitions. For example, if a network failure disconnects a replica from the origin server, the replica can continue to provide sequentially-consistent service for an average of over two hours when the bandwidth before the failure was 50% of the trace's average update bandwidth.

This paper makes three contributions. First, it provides evidence that systems can maintain sequential consistency for some key WAN distributed service despite

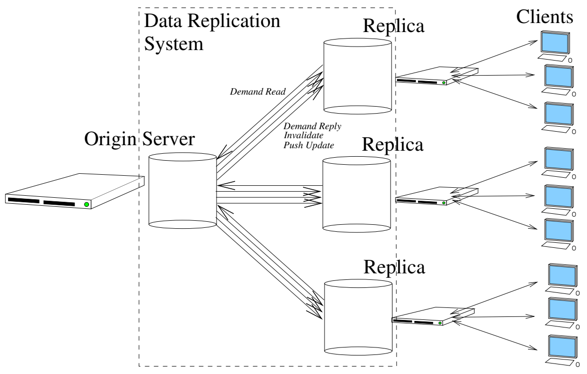

Fig. 1: High level system architecture.

the CAP dilemma, which states that systems cannot get strong **C**onsistency and high **A**vailability for systems vulnerable to **P**artitions [10]. 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 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, 12, 23, 52, 56]. A redirection infrastructure [13, 32, 62] 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, 12, 23, 52, 56] 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. The rest of this subsection defines these concepts more precisely.

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* [25] 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 [1].

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." [37] 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 [11, 39] or availability [10]. However, we restrict our study to *dissemination services* that have one writer and many readers, and we enforce *FIFO consistency* [39] 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 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, 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 too-stale data.

Second, some redirection infrastructures [13, 32, 62] 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 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* [36] B_1, by switching between replicas the client can observe such an inconsistent state. In Section 3.3 we discuss how to adapt Bayou's session consistency protocol [51] 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

TRIP is based on a novel replication algorithm that revolves around two simple parts: (1) the server's self-

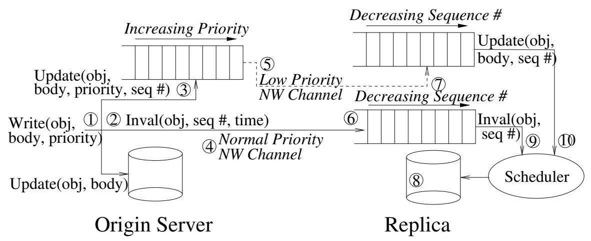

Fig. 2: Overview of replication algorithm. The circled numbers are discussed in the text.

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.

Figure 2 provides a high-level view of the algorithm for synchronizing a replica's data store with the origin server's. 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.

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.

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 several limitations of the basic algorithm and possible optimizations available within this framework.

3.1 Origin server

The core of the origin server is a novel and generally-applicable architecture for push-based prefetching where each update channel to a replica consists of a priority queue of updates that drains via a low-priority network connection to a replica. By combining a priority queue and a low-priority network protocol, the updates' channel provides for self-tuning prefetching for each replica.

Algorithm 1 Origin server

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

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 [20, 27, 28, 46, 55], there is no pre-set threshold that determines whether a given object is valuable enough to send; instead, TRIP relies on the low-priority network protocol to ensure that objects are only sent when the value of doing so exceeds the cost [33].

In order to integrate sequential consistency and Δ-coherence with self-tuning updates, the origin server separates each replica's invalidation channel from its update channel. When an update occurs, the origin server immediately sends the invalidation to each replica, but it enqueues the update bodies in the per-replica priority queues. Unfortunately, separating these channels prevents replicas from depending on message arrival order for consistency, so the origin server associates a sequence number with each update and each stored object, and it includes an object's sequence number in all invalidation, update, and demand-reply messages.

Algorithm details. As the pseudocode in Algorithm 1 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 *invDemChnl[]* for sending invalidations and demand replies, and a set of per-replica channels *updtChnl[]* 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 *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. 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 [20, 27, 28, 46, 55] 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, 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 *updtChnl*.

Communication channels. The system design depends on the distinct properties of the *invDemChnl*s and the *updtChnl*s.

Each *invDemChnl* for invalidations and demand replies is a lossless FIFO channel that operates at normal network priority. Our protocol uses a persistent message queue [31] to ensure that this channel is lossless even across crashes and network partitions, which dramatically simplifies crash recovery.

Each *updtChnl* provides an abstraction suited for self-tuning push-based prefetch by (1) buffering updates in a priority queue and (2) 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, 45, 54].

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.[1]

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

Here, 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.

Each scheduler therefore applies invalidations in sequence number order and maximizes the amount of valid data in its local storage by trying to delay applying an invalidation with sequence number N until it has an update with the same sequence number. But, a scheduler is forced to apply an invalidation earlier than that in two circumstances: (1) the staleness deadline for an invalidation expires or (2) a demand read reply that reflects state M ($M > N$) arrives at the replica, forcing the scheduler to immediately apply pending invalidations with sequence numbers up to M to avoid stalling the demand read.

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 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

[1]We show that enforcing condition C1 yields sequential consistency in the Appendix.

Algorithm 2 Replica

```
State
    $storage$; // $Validity,\ sequence\, number,\ and\, body\, of\, each\, object$
    $pendingInval$; // $Received\, but\, unprocessed\, invalidation$
    $pendingUpdate$; // $Received\, but\, unprocessed\, updates$
    $delta$; // $Max\ staleness\ between\ server\ and\ replica$
    $maxSkew$; // $Max\ clock\ skew\ between\ server\ and\ replica$
receive (INVAL, objId, seqNo, timestamp) on invDemChnl
    $pendingInval.put(objId, seqNo, timestamp)$;
receive (UPDATE, objId, body, seqNo) on updtChnl
    $pendingUpdate.put(objId, body, seqNo)$;

pendingUpdate.head.seqNo $\leq$ pendingInval.nextSeqToProcess()
    // $Scheduler\ applies\ an\ update$
    $(objId,\ body,\ seqNo) = pendingUpdate.removeHead()$;
    if $(seqNo \geq storage.getSeqNo(objId))$ then
        $storage.update(objId,\ VALID,\ seqNo,\ body)$;
    if $(seqNo == pendingInval.nextSeqToProcess())$ then
        $pendingInval.doneProcessing(seqNo)$;
currentTime() $\leq$ pendingInval.head.timestamp + delta - maxSkew
    $Scheduler\ applies\ an\ invalidate$
    $applyNextInval()$; // $See\, below$
local call to read(objId)
    if $(VALID == storage.getState(objId))$ then
        $return\ storage.getBody(objId)$;
    $send(READ,\ objId)\ to\ origin\ server$;
    $storage.waitUntilValid(objId)$;
    $return\ storage.getBody(objId)$;
receive (REPLY, objId, body, seqNo) on invDemChnl
    while $(pendingInval.nextSeqToProcess() \leq seqNo)$ do
        $applyNextInval()$; // $See\, below$
    $storage.update(objId,\ VALID,\ seqNo,\ body)$; // $Unblock\, rd$
applyNextInval() // Internal private method called from above
    $(objId,\ seqNo,\ timestamp) = pendingInval.readHead()$;
    if $(seqNo \geq storage.getSeqNo(objId))$ // $'At\, least\, once'\ chnl$
    then
        $storage.update(objId,\ INVALID,\ seqNo)$;
    $pendingInval.doneProcessing(seqNo)$;
```

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. 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 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 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 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] if program order specifies that r_1 *happens before* r_2. On the other hand, 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 [25, 37].

TRIP therefore provides two options. *Conservative* mode preserves transparency 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. 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. But, if a read miss occurs during a disconnection, it logically blocks until the connection is reestablished and the server satisfies the demand miss.

In a web service environment, blocking a client indefinitely is an undesirable behavior. Therefore, TRIP provides three ways for services to give up some transparency 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 can return an error code to the calling edge server program. 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.)

Second, rather than require applications to deal with time-outs internally, TRIP can be configured to take two actions when a demand read times out: (1) signal the redirection layer [13, 32, 62] 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 [22] to a different replica. The approach then relies on client-initiated request retransmission for end-to-end recovery [10]. This option provides less precise control to the application, but it also requires less invasive modifications of the service-specific code.

Third, given the choice between reducing availability and increasing staleness during disconnections, some services may choose the latter. Such services may configure TRIP to increase Δ when it detects a disconnection from the server. This increase 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. For example, if a replica sets $\Delta = \infty$ during disconnections, it will apply no invalidations while disconnected, but it may serve arbitrarily stale data.

3.3 Limitations and optimizations

Our current protocol is limited in at least two ways. These limitations 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. 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 [51] 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 [30]. 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 the features described above except that (1) it does not implement the *aggressive* optimization (Section 3.2), so it is always *conservative* and blocks reads to the cache when a read miss is outstanding and (2) of the three interfaces for handling read misses during disconnection described in Section 3.2, it only implements read time-outs;

it does not provide the “close all connections and change redirection” option or the “increase Δ” option described above. 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 [42] and a policy for prioritizing which documents to push to which replicas [28, 55], which may, in turn, require some facility for gathering read frequency information from replicas [48, 59].

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 a 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 persistent message queue [31] for sending updates and invalidations to each replica. The implementation buffers invalidation messages on the server’s disk, manages TCP connections between the server and replicas, and buffers pending messages 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 acknowledge message processing 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 [43].

To provide a low-priority network channel for updates that does not interfere with other network traffic, we reimplement TCP-Nice [54] 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 [40]. 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 [55].

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 atomically-applied 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 [26, 53].

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 this assumption using our prototype—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.

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 client requests and 281 thousand writes by the origin server, and they span one day.

[2]The 2000 Summer Olympic games

In order to simplify simulations we ignore certain entries in our trace file. In particular, we remove from the trace files (1) all requests that do not contain 200 or 304 as server return codes, (36.7%), (2) all dynamic requests, (13.9%), (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 dynamic requests because we do not have data of which underlying objects they access. We remove out-of-order requests because they pose a problem for the event queue in our trace-driven simulator. Finally, we remove requests that have valid return codes but that our trace parser fails to parse because it is conservative. Since the number of requests in the traces that are either unparseable or appear out-of-order is small, we do not believe that removing them significantly 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 [20, 27, 28, 46, 55] 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 [55]. 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

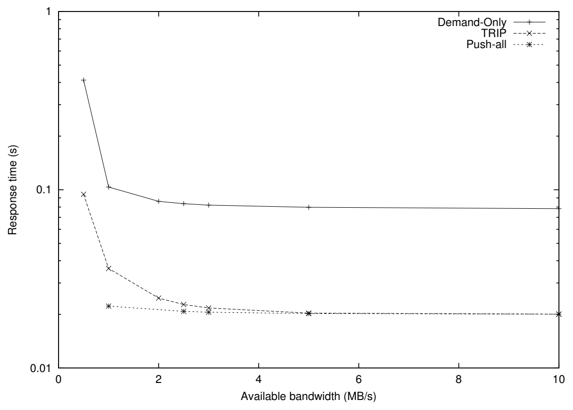

Fig. 3: The effect of bandwidth availability on response times

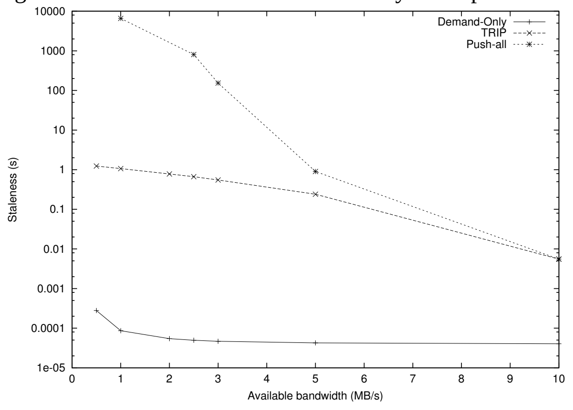

Fig. 4: Average staleness of data served by replicas.

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.

5.2.1 Response times and staleness

In Figure 3 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 4 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 receive stale data.

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 4 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 TRIP algorithm has significant advantages over both Push All and Demand Only. When available bandwidth exceeds 5MB/s, TRIP matches Push All's excellent response time and provides 4x speedups compared to the Demand Only system. At lower bandwidths, this algorithm meets the timeliness bound of 60 seconds, but it still significantly outperforms the Demand Only strategy. For example, when 2MB/s of bandwidth is available, TRIP provides a speedup of 3.5 compared to Demand Only, and Push All provides only an additional speedup of 1.2 despite the latter's liberties with the system's freshness requirements. 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 [33] and (b) the priority queue at the origin server ensures that the prefetching that occurs is of high benefit/cost items. For example, at 500KB/s of available bandwidth, which causes significant congestion for even the Demand Only case, TRIP has more than a 3x speedup over Demand Only.

Push All and Demand Only represent extreme cases of static-threshold-based prefetching 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. Intermediate static thresholds would land between these cases—increasing the threshold above 0 would reduce the system's best-case performance by increasing the high-bandwidth response time plateau towards the Demand Only line, but higher prefetch thresholds would also shift to the left the overload point where staleness guarantees are violated.

5.2.2 Variations of TRIP

Figure 5 shows the behavior of response times under variations of TRIP. We modify the TRIP algorithm in two ways. We first vary the coherence parameter Δ on response times by evaluating setting $\Delta = 0$, which

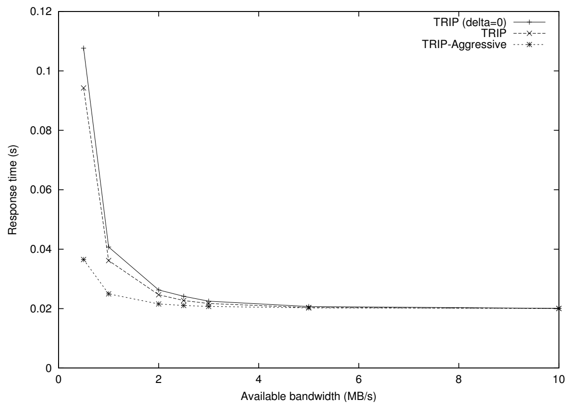

Fig. 5: Latencies yielded by variations of TRIP

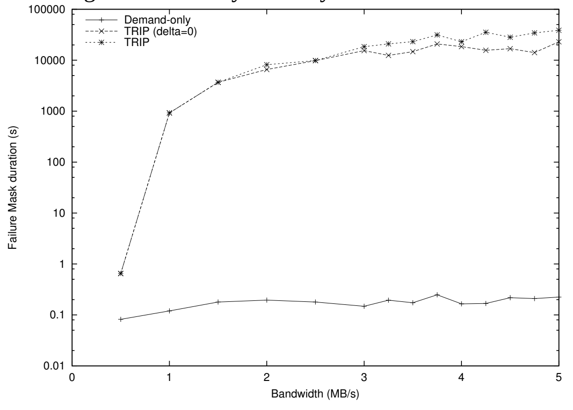

Fig. 6: Dependence of mask duration on bandwidth.

forces a replica's scheduler to apply all invalidations immediately. We then investigate the potential benefit of the *TRIP-aggressive* optimization, which sacrifices some transparency by assuming that when the application issues concurrent read requests, the requests are logically independent and which allows reads of cached objects to pass reads that have blocked (as described in Section 3.2).

Reducing average staleness by reducing Δ below 60s inflicts a modest cost on response time, and this cost declines as available bandwidth increases. The benefit of allowing some applications to exploit independence across read requests can be substantial. 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 [29].

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 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 6 shows how the average mask duration varies with bandwidth for the TRIP, TRIP-aggressive, 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 [44].

Note that the traditional Demand Only algorithm performs poorly. In Figure 6, 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 [57]. 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 [55].

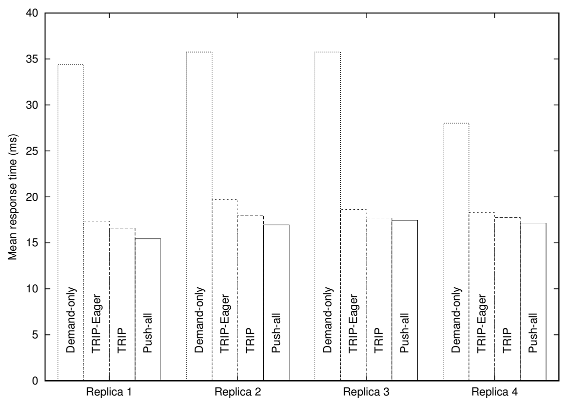

Fig. 7: Replica-perceived response times yielded by the Demand-fetch-only, FIFO-push-all, and the TRIP algorithms

Figure 7 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 increase 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.

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 [2, 15], always-aggressive push-all [13, 47], or hand-tuned threshold-based prefetching [20, 27, 28, 46]. Davison et al. [19] propose using a connectionless transport protocol and using low priority datagrams (the infrastructure for which is assumed) to reduce network interference. Crovella et al. [17] show that a window-based rate controlling strategy for sending prefetched data leads to less bursty traffic and smaller network queue lengths. In earlier work, we describe a threshold-free prefetching system called NPS [33] that like TRIP makes use of TCP-Nice [54] 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 [16, 34, 38, 41, 58, 60, 61], but few provide explicit consistency guarantees. Unfortunately, Frigo notes that even strong coherence is considerably weaker than sequential consistency [24]. Bradley and Bestavros [9] 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 [14]. Akamai's EdgeSuite [2] primarily relies on demand reads and enforces delta coherence via polling with stronger consistency available via object renaming. Burns et al. [11] discuss a *publish consistency* model of consistency that is useful for web workloads and show that consistency implemented by file systems has inefficiencies that prevents easily scaling them to many clients. Most of these systems use demand reads, but several strategies for mixing updates and invalidates have been explored for multicast networks [21, 49, 38]. These multicast-based 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 hierarchy 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 [35] allows an update to enforce causal, sequential, or linearizable consistency. Bayou [47] maintains causal consistency at all times and it asynchronously reorders operations to eventually reach a global sequentially-consistent ordering of updates that may differ considerably from the causal order seen when a node first applies a given update. These systems both focus on multi-writer environments and eventually propagate all updates to all replicas. Yu and Vahdat [63] 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 demonstrates how to exploit this observation for dissemination workloads by integrating consistency and self-tuning prefetch.

Several studies have examined ways to cache pages that are dynamically generated based on some underlying data [50]. Challenger et al.'s [13, 14] 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.

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 [29]. 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 information 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 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 [26, 53].

Acknowlegements

We thank Arun Venkataramani for making a user-level version of TCP-Nice available to us and for his crucial suggestion that the algorithm could be made simpler by coordinating the update and invalidation streams at the receiver rather than at the sender. We thank Jian Yin for his helpful comments on the presentation.

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] A. Bradley and A. Bestavros. Basis token consistency: Supporting strong web cache consistency. In *GLOBECOMM*. 2003.

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

[11] 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.

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

[13] 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.

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

[15] A. Chankhunthod, P. Danzig, et al. A Hierarchical Internet Object Cache. In *1996 USENIX Technical Conference*. Jan. 1996.

[16] E. Cohen, B. Krishnamurthy, et al. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *ACM SIGCOMM Conference*. 1998.

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

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

[19] 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.

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

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

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

[23] 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.

[24] M. Frigo. *The Weakest Reasonable Memory*. Master's thesis, MIT, 1998.

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

[26] L. Gao, M. Dahlin, et al. Application specific data replication for edge services. In *International World Wide Web Conference*. May 2003.

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

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

[29] M. Hill. Multiprocessors should support simple memory consis-

tency models,. In *IEEE Computer*. Aug. 1998.

[30] 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.

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

[32] 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.

[33] 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.

[34] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. In *Proc. of the Seventh International World Wide Web Conference*. 1998.

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

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

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

[38] 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.

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

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

[41] M. Mikhailov and C. Wills. Evaluating a new approach to strong web cache consistency with snapshots of collected content. In *International World Wide Web Conference*. May 2003.

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

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

[44] A. Nayate, M. Dahlin, et al. Integrating Prefetching and Invalidation for Transparent Replication of Dissemination Services. Technical Report TR-03-44, University of Texas at Austin, Dec. 2003.

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

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

[47] K. Petersen, M. Spreitzer, et al. Flexible Update Propagation for Weakly Consistent Replication. In *16th ACM Symposium on Operating Systems Principles*. Oct. 1997.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

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

[57] 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.

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

[59] P. Yalagandula and M. Dahlin. SDIMS: A scalable distributed information management system. Tech. Rep. TR-03-47, University of Texas Dept. of CS, 2003.

[60] J. Yin, L. Alvisi, et al. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, Feb. 1999.

[61] —. Engineering scalable consistency. *ACM Transactions on Internet Technologies*, 2(3), Aug. 2002.

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

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

Appendix: Sequential consistency proof sketch

The replica ensures sequential consistency [37] 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.