Wide Area Tentative Update Propagation

Jeffrey Pang and Gruia Pitigoi-Aron

Abstract

In this paper we discuss a base architecture for tentative update propagation between replica servers in wide area storage systems. Our system provides eventual consistency, update service during network partition, and improved client response time. We describe an initial implementation on OceanStore and evaluate its performance and penalties for a range of operating points.

1 Introduction

A well known dilemma for distributed systems is the theoretical impossibility of providing 100% availability while maintaining both complete consistency and tolerance to partition. This problem is particularly acute for extremely wide area storage systems, which must balance the three without alienating applications that expect what they have received from traditional file systems and databases.

Although many wide area systems have recognized the value of *caching* to provide readers with dramatic improvements in response time, few have embraced similar ideas to improve performance for writers. Indeed, with the exception of peer-to-peer sharing systems, nearly all highly deployed wide area systems that allow write sharing are not distributed at all but rely on traditional client-server architectures to ensure consistency guarantees.

This model is unrealistic for scalable, extremely wide area storage systems where servers are mostly likely *federated*, *insecure*, and *fault-prone*. When write serialization is centralized, compromise means data loss, downtime ensures unavailability, and response time grows with geographical distance.

In this paper we address the later two problems (without sacrificing durability) motivated by two observations.

First, many applications do not need single copy serializability. Message boards or calendar systems, for example, do not require ACID transactions and users would likely endure limited inconsistency. In addition, many traditional Internet systems such as email have little or no shared data and their users would much rather have tolerable write latencies than strong consistency guarantees.

Second, most extremely wide area systems already maintain fixed or floating replicas to improve client read service. The servers that hold these replicas form a prime substrate for servicing *tentative updates*, or writes that the infrastructure commits tentatively to improve response time and write availability between replicas.

We present Tsunami, a base architecture for tentative update service and propagation between loosely consistent replicas in the wide area. Tsunami maintains eventual consistency using a primary tier of strongly consistent replicas, ensures read and write service availability even while replicas are partitioned by *decoupling* updates from commit notifications, and dramatically improves response time for clients that choose to utilize tentative updates.

Tsunami is implemented as part of OceanStore, which is described in Section 2. The Tsunami architecture is discussed in Section 3, and we evaluate an initial implementation in Section 4. Section 5 describes related work and we conclude in Section 6.

2 OceanStore

In this section we briefly describe the components of OceanStore, "a utility infrastructure that is designed to span the globe and provide continuous access to persistent information." We implement Tsunami as a component of OceanStore to demonstrate its viability in a *extremely wide-area, highly available, federated,* and *untrusted* storage infrastructure. For more details, the interested reader is directed to [12].

2.1 Tapestry

Tapestry [24] is the Distributed Object Location and Routing (DOLR) substrate upon which OceanStore is built. Tapestry allows servers to *publish* objects in the infrastructure that clients can later *locate* without centralized management. Objects and nodes in Tapestry are identified by globally unique identifiers (GUIDs), such as secure hashes of public keys.

Each node GUID in Tapestry is represented as a hexadecimal string. Nodes keep a routing table of the GUIDs of their neighbors. A message routes toward particular GUID; each node along the path forwards it closer to the target by matching successively longer sequences of the hexadecimal string (e.g., to route a message toward `4234`, a node with GUID `**34` routes the message to a node matching `*234`).

Objects are published by routing a publish message toward the object's GUID. Each node along the publish path stores a back pointer to the originating node associating requests for the GUID to the originating server which is called a *storage server*.

Clients locate objects by routing a locate message toward the object's GUID. The message will eventually reach a node which contains a back pointer to the storage server. The node redirects the message to the server, which can then service the client's request.

More than one storage server can publish the availability of an object and locate messages will likely converge on back-pointers to close replicas.

2.2 Update Model

Each object in OceanStore is managed by a primary tier of replicas called the *inner ring*. The inner ring commits changes to the object in a strict serial order using a Byzantine agreement protocol [14] to ensure fault tolerance. Because Byzantine commit is a fairly expensive, the primary tier is relatively small and is optimally located in high connectivity regions of the network.

OceanStore objects are composed of data blocks, organized in b-trees. Blocks are keyed by block GUIDs so they can individually be dispersed and located in Tapestry. To ensure confidentiality, clients encrypt data in objects.

Objects are modified through *updates* which clients send to the inner ring. Because there are no read locks in OceanStore, updates consist of a set of dependency checks called *predicates* which the inner ring ensures are valid before applying update actions. More specifically, each update is composed of one or more *update tuples*, each of which is a predicate/action pair. Each tuple's predicate is checked in turn; the first one that passes has its actions executed.

Other systems such as Bayou [21] use merge procedures to resolve unmet dependencies. Update tuples approximate merge procedures, which are not viable in OceanStore because they require execution of arbitrary code at servers and operations on encrypted data are not tractable.

Each update creates a new *version* of the object (a new top block in the b-tree with pointers changed to refer to the modified blocks). The new blocks created by updates are erasure coded and distributed widely to *deep archival storage* to make versions extremely durable. OceanStore uses this *copy-on-write*, versioned update model to simplify recovery.

2.3 Secondary Tier Replicas

Because the set of inner ring servers must be small and geographically close for Byzantine commit to be efficient, servers maintaining a second class of floating replicas are widely distributed to improve performance and availability. This secondary tier forms an application level multicast tree rooted at the inner ring. When updates at the inner ring are committed, they are disseminated to the secondary tier through this tree along with *commit certificates* which certify the results. Hence, secondary tier replicas serve as weakly consistent read caches for clients close to them.

Many applications that utilize secondary tier replica servers (which we will just call *replicas* in this paper) do not require strict ACID semantics. We are thus motivated to have replicas exchange updates tentatively with each other while the inner ring performs Byzantine commit.

3 System Architecture

Secondary tier replicas are only responsible for servicing reads in the basic OceanStore replication model. In this section we describe the modifications we make to the replica architecture to allow the secondary tier to accept updates as well. Because updates accepted at a replica can not be committed without communication with the inner ring, we con-

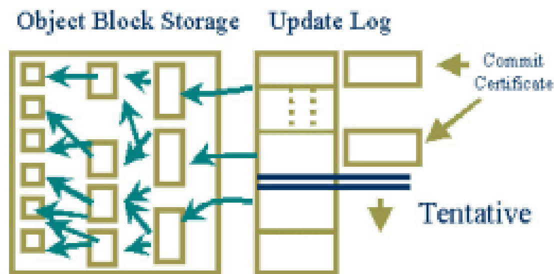

Figure 1: The *update log* and corresponding versions of objects in OceanStore's storage system.

sider them to be *tentative* until such communication takes place. We describe a method for propagating tentative updates that is tolerant of message loss and ensures that this communication eventually occurs.

3.1 Managing Tentative Updates

Allowing servers to accept and apply tentative updates means that they must maintain tentative versions of their replica objects. To facilitate this, we have each replica server maintain a *pool* of the updates it has accepted from clients and received from other replicas. We provide a formal order by placing some pooled updates in an *update log*, shown in Figure 1.

Usually, new updates are appended to the end of the log and possibly reordered when we receive commit notification from the inner ring. Each update in the log corresponds to a *version* of the object that resulted from the application of that update in the particular log ordering. When updates in the pool are placed in the log, their effects become visible to clients.

We maintain two properties to guide our consistency decisions:

1. Replicated objects eventually converge to the official version created by the inner ring. This ensures that the divergence between different replica versions is bounded.
2. The application of updates should maintain the dependencies in each update. The predicates in each update should always be true for any tentative version of the object. This allows applications to maintain invariants even as tentative versions may differ in their contents.

Achieving the first is easy since we already trust the inner ring to provide a total ordering of updates. The inner ring communicates this ordering to replicas which then rearrange their update logs. Hence, we partition the update log into a committed section and a tentative section. The committed section does not change and is identical at all replicas (up to the newest committed update). The tentative section contains updates from clients and other replicas that are not yet committed.

To achieve the second property, we use the same update application model as the inner ring. Before applying a particular update tuple's actions, we check that its predicates return true when evaluated on the last tentative version of the target object. To provide an even stronger consistency guarantee, we maintain that an update accepted at a replica at time t_0 will always be reflected in the tentative version when we apply an update accepted at time $t > t_0$; i.e., all updates accepted at a particular replica are always applied in order. This is achieved by tagging each update with an increasing sequence number. We use a simple counter so other replicas can easily tell if there are "gaps" in the series of updates they receive. This method of ordering updates (which maintains a "happens before" order) was used in Bayou [16] and they found that it was consistent enough for a range of group-ware applications. Like Bayou, replicas maintain a version vector as a lightweight mechanism for determining the latest updates it has logged from other servers.

Hence, updates in the pool can not enter the log until a server has received all updates from the same replica before it. We do not place constraints on updates entering the pool (except that they not be duplicates) because they may be useful to us in the future as more updates arrive. We periodically check the pool of non-logged updates to see if they can be added to the log due to new arrivals.

3.2 Updates Entries and Commit Certificates

Although the inner ring is composed of several replicas, the update model described in Section 2.2 does not treat it very different from a single server system. The inner ring is tolerant to faults, but remains a single point of serialization. Though commit certificates can be placed anywhere in the system allowing reads to be distributed, updates cannot flow around as easily.

We extend the update by encapsulating it in an *update entry* created by the sever that accepts the

update. Update entries contain the signed client update, sequence number the replica assigns to the update, and GUID of the replica itself. This allows any server or client to determine the origin of the update and its rank in the accept-ordering. The entry is signed by the replica to prevent tampering. Update entries can be exchanged between replicas at will.

In addition, we modify commit certificates created by the inner ring to include a secure digest of the committed update entries so third parties can match the two. The digest can also be used as an identifier if the update needs to be located. The sequence number in a commit certificate that identifies object versions also defines the order in which that particular update was applied. This is the *commit sequence number* and it determines the update's place in the total ordering which all replicas eventually converge to.

Tentative versions act as local unofficial commit certificates that reside on a single server. Hence there is nothing that prevents replicas from utilizing OceanStore's distributed storage mechanisms to maintain the blocks that are part of its tentative versions. The only difference between a secondary replica and the inner ring is that the secondary replica cannot certify its tentative versions as official; this should not matter to clients if they trust the replica enough to use its tentative versions (e.g., if they are serving as their own replica). Hence, replicas need only maintain the update log locally. The actual data can be stored elsewhere in the infrastructure and located later through Tapestry. This allows continued operation even if the inner ring is offline for extended periods of time.

3.3 Update Application and Log Rollback

When operating normally, replicas can expect commit certificates to arrive soon after tentative updates are received so tentative logs should not grow too long. Thus, it makes sense to cache tentative blocks locally most of the time. This also makes it easier to remove blocks produced by tentative versions that are no longer pertinent.

When a commit certificate moves a tentative update to the committed log, the reordering could change the the effects of the tentative updates. Thus, we usually roll back the log and reapply the tentative updates. Our copy-on-write storage scheme simplifies this task since we can use a previous version and roll the log forward based on that version.

Sometimes updates may be commutative and rollback may not be necessary. Because encrypted block-based storage presents a narrow scope for update operations, it is relatively straight forward to determine whether two updates are commutative based on their predicates (dependencies or read-sets) and actions (write-sets).

For example, consider an update that predicates on block $i = X$ and replaces block j with value Y. Another similar update that does not examine i or j can be applied before or after the first with the same effect. We can avoid reapplication and just move the update entry in the log.

3.4 Servicing Client Requests

There are several considerations when replicas service clients. Because OceanStore is just an object store, replicas must be flexible enough to conform to consistency models required by a range of applications. Our design is flexible enough to allow both tentative and non-tentative updates on any object.

The inner ring serves as a primary tier of replicas and acts identically to the secondary tier except it can commit updates. Hence, a client can submit an update directly to the inner ring and receive a non-tentative response. At the opposite extreme, clients can act as their own servers and maintain their own tentative versions. Because they can distribute the tentative versions to several other replicas which will continue to propagate the updates, the updates remain durable even if the client goes offline or is destroyed.

The existence of tentative versions also changes the nature of reads at replicas. We extend the interface to allow clients to specify whether they wish to see tentative updates in their reads. In addition, our system contains the necessary components to provide various *session guarantees* to clients as proposed in [20]. Finally, because OceanStore's storage capacity is potentially unbounded and object versions are uniquely defined by a version GUID that can be used to locate a copy of it, it is possible to make tentative versions *permanent*. This results in branches from the official version series established by the inner ring and may be useful for application level conflict resolution and recovery.

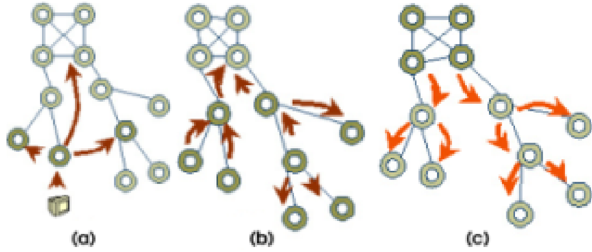

Figure 2: Stages of tentative update propagation. (See text)

3.5 Tentative Update Propagation

A key feature of our design is that we place very few limits on how updates are exchanged between replicas. Any replica can accept any update at any time; the only constraints are on the way those updates affect a client's view of data. We chose a update propagation model that exploits this by optimistically pushing updates to other replicas as a mechanism for improving their durability while they are tentative.

Usually, the secondary tier of replicas are connected together in a spanning tree that is used to disseminate updates from the inner ring. This tree exploits locality aspects of Tapestry and neighboring nodes in the tree are likely to be "close" to each other in Tapestry distance so we reuse it as a update dissemination network for secondary tier replicas. The stages of propagation are shown in Figure 2.

When a new update is accepted at a replica, it first uses Tapestry to send the update to several others in the tree, including the inner ring (a). The update is propagated throughout the tree beginning at these points (b). By starting the propagation at multiple points in the tree, we reduce the likelihood that a dropped message will stop propagation. Though we could use Tapestry to multicast the update to all the replicas, this would require routing the message all the way to the object root and then along separate paths to each replica.[1] We choose to use direct paths between replicas to conserve bandwidth.

To prevent excessive numbers of duplicate messages, a replica stops propagating an update if it has already seen it. The version vector provides a lightweight way of detecting this; if the sequence number for a particular node is v, then we can stop propagating any update with a sequence number $\leq v$.

[1] A more efficient Tapestry multicast is discussed in [25] but has not been implemented.

Finally, when the inner ring commits the update, a commit certificate is multicast to the tree (c).

If no replicas join the tree and update messages are never lost, this propagation method will ensure that all updates reach all replicas. However, these are not realistic assumptions, so we require a method to bring replicas up to date if they are missing updates.

We use Bayou's anti-entropy algorithm because it is simple and allows any two replicas to bring each other up to date. Furthermore, we make it the responsibility of the receiver to initiate the anti-entropy session for two reasons. First, the sender must obtain the receiver's version vector before it can know which updates the receiver requires. If the receiver initiates the session by sending its version vector to the sender, all communication can take place in one round trip. Second, we believe receivers are better equipped to know when they want or need to "pull" updates from other replicas. Also, because we allow replicas to pool unloggable updates (where there is a gap in the sequence number series), the receiver can be more specific about which updates they are missing.

We use two simple heuristics for choosing anti-entropy partners. First, a server expect updates from a particular replica to arrive in order. If a server receives an update with a sequence number that is greater than 1 plus the latest sequence number in its version vector, it knows it is missing updates from the replica that originated the update, so the server performs anti-entropy with it next. Second, our parent in the dissemination tree is close to us in Tapestry distance to us, so we periodically perform anti-entropy with it. We were motivated by Golding's study [7] which demonstrated that a distance-based scheme for choosing partners works as well as a random scheme.

Finally, we note that by having the originating server initiate the propagation of the update, we relieve the inner ring of that responsibility. When update is committed, the inner ring only needs to multicast the commit certificate to the secondary tier. This decouples update propagation with commit notification and allows updates to propagate whether or not the inner ring is accessible.

3.6 Practical Considerations

There are several practical matters that would have to be dealt with in a real world implementation of

this architecture.

Even though OceanStore allows data objects to be offloaded from replica servers, there is still limited space for pooling updates and keeping update logs. We can limit pooled, but non-logged updates by simply limiting the size of the pool; there is no constraint forcing replicas to accept updates. To limit the log size, we allow replicas to truncate their committed logs; only the latest committed entry is necessary since the committed log does not change.

However, this presents a problem when we perform anti-entropy. If the replica is missing an update which we have truncated from our log, it will not be able to know whether it can move tentative updates to its committed log or not. Bayou solved this problem by requiring a full database transfer when it occurs, recognizing the trade-off between storage space and bandwidth. OceanStore permits another solution when replicas are on-line: because commit certificates are *tombstoned* in the archive and can be located in Tapestry via its update's secure digest, it is possible to query the infrastructure to determine if updates are committed. This relieves the sender in an anti-entropy session from transferring anything more than updates.

Another consideration is the size of version vectors. Although they do not take up much storage locally, they must be transfered to other replicas to initiate anti-entropy, and are proportional in size to the number of replicas in the system. Other systems prune version vectors when all replicas have received updates or by sending "retirement writes," but neither of these are suitable solutions in a federated, untrusted system like OceanStore. Instead, we place an explicit time limit on updates, perhaps set by clients, on how long they are allowed to remain tentative.[2] If they are not committed within the allotted time, then they are dropped from the system. Hence, replicas can unilaterally prune replicas from their version vectors when the latest version expires. This algorithm, originally described in [18], provably captures the "happens-before" relationship described in Section 3.1.

Our initial implementation did not include these features, but we do not believe they would be difficult to add.

3.7 Implementation Details

We implemented our system architecture on top of Pond [13], the latest OceanStore prototype. Pond is implemented by a number of asynchronous modules called *stages* using Sandstorm, an implementation of SEDA [22], an architecture for scalable, highly-concurrent Internet systems. All code was written in Java.

Although we reused several OceanStore components including its cache/buffer manager, update and read stages, and Tapestry and network stages, we reimplemented stages to handle construction of the dissemination tree and replica management to better suit our needs. Our dissemination tree module implements the load balancing join algorithm described in [4], using the number of neighbors in the tree as a load metric (our workloads evenly distributed work on all replicas). However, we do not enforce client-to-replica distance constraints since we are testing dissemination between replicas.

To improve responsiveness, we allow client requests to be processed in parallel with on-going anti-entropy sessions (and anti-entropy sessions to be performed in parallel with each other).

4 Evaluation

We developed several synthetic workloads to gage the performance of our system in a hybrid simulation and real-time environment. Our main goal was to determine what benefits applications could gain at various operating points. In addition, we performed experiments to determine the costs that applications utilizing tentative updates would have to endure from seeing tentative data.

4.1 Experimental Setup

We used a hybrid simulation and real-time test bed to evaluate our system. We used 11 machines that are part of a 42 machine cluster at Berkeley. Each machine in the cluster is a IBM xSeries 330 1U rackmount PC with two 1.0 GHz Pentium III CPUs, 1.5 GB ECC PC133 SDRAM, and two 36 GB IBM UltraStar 36LZX hard drives. Each node is running Linux 2.4.18 SMP. Average ping times between machines are < 0.5ms.

On top of these machines, we simulate an artificial network using a transit-stub topology generated by the GT-ITM topology generator from Georgia Tech [23]. The simulator does not simulate bandwidth and queuing delays but we do not believe these would be substantial because our workloads generated small messages. All our experiments were

[2]We assume replicas have loosely synchronized clocks.

conducted with 50 replica servers plus a inner ring server placed on nodes chosen uniformly at random over a transit domain and its stubs from this topology, and were spread evenly over our 11 machine testbed. The average link latency was 26.49 ms. The same placement was used for all experiment.

Times were measured using wall-clock time.

4.2 Simulation Parameters

We used the following system parameters during our evaluation.

Replicas initiate anti-entropy with their parent once every minute. When updates are accepted from clients, they are multicast using Tapestry to 5 other replicas, including the inner ring. Replicas are initialized knowing about all other replicas in their version vector, though they could gather this information dynamically during warm up periods.

Our "inner ring" is a specialized replica which commits the updates it receives from other replicas. We artificially delay each commit by 40ms, which we believe is reasonable for Byzantine commit (if not overly optimistic in the wide area). In practice, we found that this delay is trivial compared to the transit latencies in our wide area simulation. The inner ring can still exchange updates in the process of commit as tentative.

The dissemination tree is constructed by starting up nodes in a random order and having them synchronously join the tree. We use the same join order for all plots in an experiment.

Due to limited resources, we simulate clients by running workload stages on the replica servers. This assumes there is minimal latency between clients and replicas, but we believe it is justifiable since most clients that trust a replica to perform tentative updates on their behalf will have it on the same machine or in the local area. Algorithms for placing replicas close to clients are discussed in other papers [4, 17].

Unless otherwise noted, all clients perform reads and updates on a single fixed sized object, each update/read consisting of a 1KB block of the object. This is to approximate changes to small shared meta-data which is of primary interest to applications with shared data. Update and read requests arrive at exponentially distributed intervals with the indicated mean.

In the cases where we run our experiments with *No Tentative*, replicas' exchange of tentative updates is disabled and updates are not applied until the inner ring sends back commit certificates. In this case the commit certificate also contains the update so other replicas can apply it.

	Mean Latency (ms)	Std. Dev.
No Tentative	1340.65	334.56
Tentative	5.75	1.75

Table 1: Client perceived update latency with and without tentative updates.

4.3 Results

Here we present the results of our experiments.

4.3.1 Update Latency

Our first experiment examined the differences in client perceived update time with and without tentative updates. We measure the average response time for 30 updates made at each of our 50 replicas after a period of warm up. Updates arrive at each replica at 10 second intervals.

The results, shown in Table 1, are fairly obvious. Without tentative updates, the time to make updates is tied to the latency between clients and the inner ring. When enabling tentative updates, the processing time for performing the update dominates because clients talk to replicas for "free."

There are several things to note in these results. First, the standard deviation of the *No Tentative* results is fairly substantial, demonstrating that clients receive different levels of quality of service depending on their location in the network. Allowing tentative updates decouples update time and commit time.

Second, cost of performing a local update is surprising since Pond's buffer manager performs most updates in memory. The majority of the time comes from the verification of client signatures on updates and copying and secure hashing of new blocks created by the update. This is a trade-off we make when using using a copy-on-write update model.

4.3.2 Propagation Delay

Next, we evaluated the time it takes to propagate updates from the replica they originated from. Again, we performed 50 updates at each replica at 10 second intervals. In the *Tentative* cases, the de-

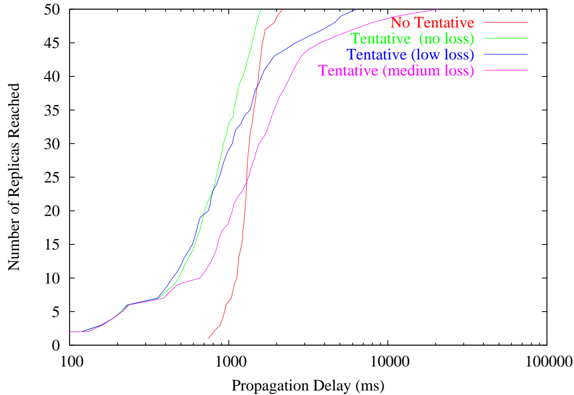

Figure 3: Average time for updates to reach other replicas.

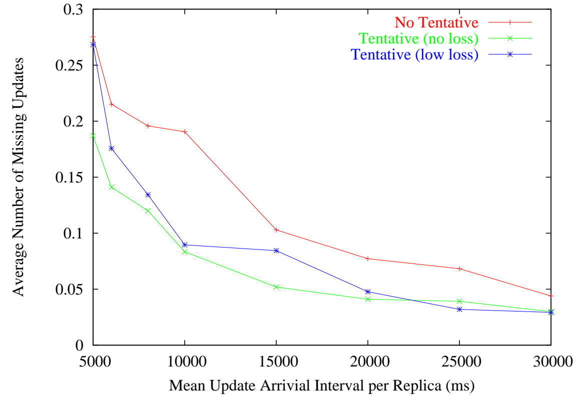

Figure 4: Average number of updates missing from data reflected in client reads.

lay is until updates enter a replica's log and can be made visible to clients. The results are in Figure 3.

With the *No Tentative* case, updates take a substantial time to reach the first replica because all updates must first be transmitted to the inner ring, committed, and then propagated back down the dissemination tree. However, we observe that once it has been committed at the inner ring, it reaches all replicas quickly because it is at an efficient dissemination point in the tree.

With tentative updates enabled, the accepting replica can quickly propagate the update to its neighbors and have it reach a significant portion (about 50%) of the total replicas before the inner ring commits. When no messages are lost in the network, updates are able to saturate the network faster than the *No Tentative* case.

We considered how message loss would affect propagation time. In the low loss case, we have replicas drop update messages each with probability 0.001. Although this seems excessively low, each replica can drop messages independently so there is an actual 0.05 probability that a message will be lost somewhere in the tree. The medium loss case has independent drop probabilities of 0.01 (or 0.4 probability of loss somewhere in the tree).

These results are somewhat disappointing and demonstrate two inherent flaws in our backup heuristics. First, our "notice when updates are missing" heuristic for determining when to perform anti-entropy depends on further updates arriving for updates from the accepting server. Thus, a replica missing an update may wait up to the update arrival interval of that server before noticing anything wrong. Furthermore, if subsequent updates or commit certificates are lost, replicas may have to wait until their periodic poll timer goes off so they can initiate anti-entropy with their parent to come up to date. Although replicas can choose to poll more often, it must trade-off bandwidth for freshness.

Nevertheless, comparing these last two plots with the *No Tentative* case is not really fair since we assume no updates are lost in that case. We are comforted by the fact that updates still reach a fair number of the replicas quickly, making it semi-durable, even if it takes a while to reach the fringes of the network.

However, these plots do show that there is a cost to decoupling updates and commit notifications. It is more likely that one of the two will be lost making the other less useful.

4.3.3 Read Staleness

Because tentative update propagation allows updates to reach replicas faster, it should also make client reads at those replicas less stale. To test this, we counted the average number of updates that have been submitted by some client in the network, but have not yet been reflected in the data for read performed by another client elsewhere. Each replica has 300 client reads arrive at 1 second intervals. Each read chooses a random 1KB block to read out of a 50KB object, and we count the number of updates to that particular block which are missing. This approximates a certain degree of data sharing. We vary the arrival rate of writes to test against several possible application operating points.

Figure 4 shows our results. Our results verified our hypothesis and demonstrate that even with a

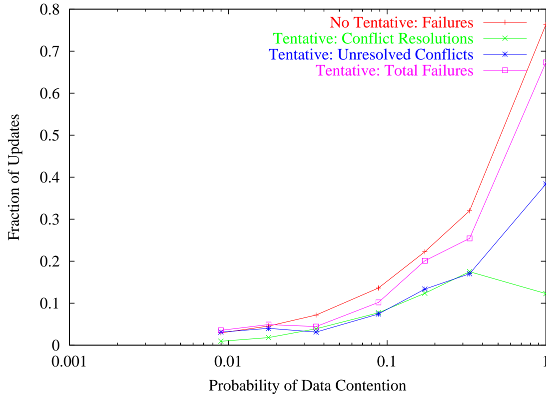

Figure 5: Fraction of updates that underwent "conflict resolution" or failed.

small message loss rate, enough updates reach most replicas to improve their data freshness. For applications with very frequent update rates, tentative updates help even more because there are a larger number of outstanding updates waiting for commit in the *No Tentative* case, and the inner ring may be backed up since it is a single point of serialization.

Nonetheless, we should note that reads performed in the tentative case are different from reads performed in the *No Tentative* case. Tentative reads may have more updates reflected in their data, but they can also be more inaccurate because the effects of those updates eventually change when they are committed.

4.3.4 Update Conflicts

To study the cost of inaccurate reads, we developed a synthetic workload that uses OceanStore's update model to detect and resolve conflicts. In this workload, clients perform work units consisting of 3 reads and 1 update with 0.5 seconds of processing time in between. Each block read and updated includes a version number in its first byte, which is incremented each time it is updated. The update consists of two update tuples: the first predicates on the version numbers of the 3 read blocks being the same as when they were read and writes one of the 3 blocks; the second predicates only on the version of the block being written. The idea is to first attempt to apply the update "safely" by ensuring read dependencies are met and then, if that fails, to try to perform "conflict resolution" with a slightly looser constraint. If neither predicates pass then the update fails. Real applications would use specific semantics to resolve conflicts more intelligently, and would probably never have a failure case (instead they could log the failure and resolve it in the application on the next read). However this strict approximation can provide us with an upperbound on the number of conflict resolution sessions and failures.

200 workload units arrive at 10 second intervals at each replica and we vary the probability that two requests will contend for the same data (i.e., reference the same blocks in their read/write sets) to approximate degrees of read/write sharing.[3] The results of this test are shown in Figure 5.

The *No Tentative: Failures* line shows the percentage of updates which fail when no tentative updates are enabled. The *Tentative: Conflict Resolutions* line shows the percentage of updates which originally succeeded with the first update tuple when tentatively applied but finally executed with the second update tuple when committed by the inner ring. The *Tentative: Unresolved Conflicts* line shows the number of updates that tentatively succeeded by executing tuple 1 or 2, but eventually failed when executed by the inner ring. The *Tentative: Total Failures* line shows the total number of tentative failures, including both updates which failed tentatively and those depicted by the *Tentative: Unresolved Conflicts* line.

The first observation is that the total number of failures is improved in the tentative case. Reads are fresher, so clients are more likely to see the actual "current" state of the data. However, a fraction of these failures succeeded erroneously when applied tentatively and a similar fraction tentatively succeeded with tuple 1 but was committed executing tuple 2 (except when the probability of data contention is 1; in this case there are too few chances for conflict resolution to occur).

Nonetheless, the majority of tentative results are correct even when there is 100% sharing of read dependencies and as the the degree contention decreases, the number of false positives falls to arguably acceptable levels. We have observed the same effect when the update rate is reduced. Finally, we are optimistic we can improve this result even further by enforcing a tentative serialization order based on client timestamps on updates. By having the inner ring batch and delay updates before committal, it will be more likely to serialize

[3] Each experiment was run with an object of n blocks; clients pick blocks to process from this set uniformly at random. Since clients pick different 3 blocks per request, the probability that two requests' read/write sets will have a nonempty intersection is $1 - \frac{n-3}{n}\frac{n-3}{n-1}\frac{n-3}{n-2}$.

updates in the same order as they occur tentatively. Due to limitations on time and of the implementors, our prototype did not have this option at the time of this writing.[4]

During this evaluation we realized that client caching of data significantly affects the conflict rate. When a client read blocks and then holds them during their process/think time, the values it holds may become stale. One way to notify a client of this is to have replicas send it a callback when new updates that affect their data arrive (in the same way as AFS [19]). As processing time decreases, the total number of failures in the tentative case goes down, but the number of false positive successes goes up because the replica server has less chance to correct updates that were applied in the wrong order. As processing time increases, it is more likely that "official" results have time to come back from the inner ring, so the number of false positives goes down, but total failures goes up since more updates are likely to fail tentatively. Nonetheless, we have observed that the total number of failures in the tentative case is never greater than the number of failures in the no tentative case, and *tentative failures* (updates that fail at the replica) are better than non-tentative failures because they come back to clients quickly, allowing them to retry without impacting response time severely.

Applications which operate on human time scales or have moderately low rates of sharing can trade-off a little inaccuracy to gain the benefits of improved response time and data freshness.

5 Related Work

Several other projects use weakly consistent update protocols to improve data availability and scalability.

Tsunami uses Bayou's anti-entropy protocol [16] to bring replicas up to date when they are missing updates. While Bayou was designed to work well with disconnected networks, Tsunami is designed for mostly online replica servers in the extreme wide area. In addition, our update model, although based on the idea of dependencies and merge procedures, differs from Bayou's.

The Coda file system [11] demonstrated that weakly consistent update protocols can make disconnected operation feasible. In Coda, only parents and their children in the replica hierarchy can exchange updates, while a replica in our system may exchange updates with any other. Moreover, Coda was designed specifically as a filesystem; OceanStore acts as a block-based object store.

Similary, in the Ficus file system [9], updates are propagated to accessible replicas. Conflicts to directories are automatically repaired while conflicts to files are reported to the user. Such a scheme could be implemented on top of our system.

Some other systems that use lazy replication base their information exchange on data objects that use timestamps and version vectors. Grapevine [2] and Clearinghouse [15] are among the earliest examples. Oracle7 performs replication among multiple master sites [1] and uses an anti-entropy scheme that was proposed by Golding [7], which is similar to Bayou's but assumes clients have loosely synchronized clocks.

OceanStore's two tiered replica architecture is similar to the lazy-master model proposed by Gray [8] and maintains its scalability and safety from system dellusion.

Finally, our update propogation model bares simularties to many multicast protocols [5, 6] and content distribution networks [3]. Future research could utilize ideas from these areas to improve the performance of update dissemination.

6 Conclusion

We have presented Tsunami, a base architecture for tentative update propagation between secondary tier replica servers in OceanStore. Our update management strategy is flexible and amendable to any form of peer communication, though we optimize it for mostly online replica servers in the wide area. We demonstrate that decoupling updates from commit notifications is a useful idea.

We evaluated our initial implementation on a simulated network in real time by measuring several metrics that are of interest to applications that might benefit from tentative updates and compared our system against one using a non-tentative update model.

Preliminary results indicate that a utilizing tentative updates can drastically improve client response time and fast propagation can improve data freshness. At certain operating ranges, the cost of inaccurate tentative values can be manageable.

[4] Because we made our system highly concurrent, we found log management was rather painful to get correct.

We believe our basic system validates the usefulness of tentative update propagation in wide area systems.

There are several areas that hold future research potential. The update propagation network can be greatly improved if the inner ring is no longer the locus of update activity. For example, different application level dissemination trees could be constructed rooted at secondary tier servers with high update rates. In addition, propagation methods could be developed to better suit heterogeneous replica quality of service constraints. Non-version vector schemes are being developed to improve scalability and managability of divergent replica versions [10]. Finally, there is much that could be learned from building actual applications which utilize tentative updates, especially using a narrow encrypted block-based update model like OceanStore's.

7 Acknowledgments

We would like to thank Steve Czerwinski, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, and the rest of the OceanStore team for their help in understanding the OceanStore prototype and putting up with our questions. We would also like to thank Professor John Kubiatowicz for some initial motivations.

References

[1] Oracle Corporation. *Oracle7 Server Distributed Systems: Replicated Data, Release 7.1*. Part No. A21903-2, 1995.

[2] A. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, April 1982.

[3] Y. Chawathe, S. McCanne, and E. A. Brewer. An architecture for internet content distribution as an infrastructure service. `http://www.cs.berkeley.edu/~yatin`, 1999.

[4] Y. Chen, R. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Proc. of International Workshop on Peer-to-Peer Systems*, 2002.

[5] Y. H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of SIGMETRICS*, June 2000.

[6] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast - sparse mode (pim-sm): Protocol spec ification. Internet Request for Comments RFC 2117, June 1997.

[7] R. Golding. The performance of weak-consistency replication protocols. Technical report, July 1992.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of ACM SIGMOD Conf.*, June 1996.

[9] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71, Anaheim, CA, June 1990. USENIX.

[10] B. H. Kang. Personal correspondence.

[11] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[12] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*. ACM, 2000.

[13] J. Kubiatowicz et al. Pond: the first Oceanstore prototype. March 2003. To appear in 2nd USENIX Conference on File and Storage Technologies.

[14] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, 1982.

[15] D. C. Oppen and Y. K. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1(3):230–253, July 1983.

[16] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propogation for weakly consistent replication. In *Proc. of ACM SOSP*, pages 288 – 301, Oct. 1997.

[17] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *Proc. of IEEE ICDCS*, June 1999.

[18] Y. Saito. Unilateral version vector pruning using loosely synchronized clocks. Technical Report HPL-2002-51, HP Laboratories, Mar. 2002.

[19] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5), May 1990.

[20] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of Intl. Conf. on PDIS*, pages 140–149, Sept. 1994.

[21] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of ACM SOSP*, pages 172–183, Dec. 1995.

[22] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM SOSP*, October 2001.

[23] E. W. Zegura, K. Calvert, and M. J. Donahoo. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Transactions on Networking*, Dec. 1997.

[24] B. Zhao, A. Joseph, and J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California, Berkeley Computer Science Division, April 2001.

[25] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of the 11th International Workshop on Network and Operating System Support for Digital Audio and Video*. ACM, June 2001.