The Cost of Increased Transactional Correctness and Durability in Distributed Databases

Aspen Olmsted
olmsted@email.sc.edu

Csilla Farkas
farkas@cec.sc.edu

Center for Information Assurance Engineering
Department of Computer Science and Engineering
University of South Carolina, Columbia, SC 29208

Abstract

In this paper we investigate the problem of providing durability for Web Service transactions in the presence of system failures. We show that the popular lazy replica update propagation method is vulnerable to loss of transactional updates in the presence of hardware failures. We propose an extension to the lazy update propagation approach to reduce the risk of data loss. Our approach is based on the "buddy" system, requiring that updates are preserved synchronously in two replicas, called buddies. The rest of the replicas are updated using lazy update propagation protocols. Our method provides a balance between durability (i.e., effects of the transaction are preserved even if the server, executing the transaction, crashes before the update can be propagated to the other replicas) and efficiency (i.e., our approach requires a synchronous update between two replicas only, adding a minimal overhead to the lazy replication protocol). Moreover, we show that our method of selecting the buddies ensures correct execution and can be easily extended to balance workload, and reduce latency observable by the client.

1 Introduction

Service Oriented Architecture (SOA) supports the composition of multiple Web Services (WSs) to perform complex business processes. One of the important aspects for SOA applications is to provide a high-level of service availability. Replication of the services and their corresponding resources, such as databases, is frequently used to increase availability. In this paper we address issues related to durability and consistency of replicated databases in the context of SOA.

A Web Service (WS) farm may host multiple replicas of the services and their resources. Service requests are distributed among the replicas within a WS farm to ensure efficient response capabilities. Usually, a WS farm is placed behind a dispatcher. Clients send service requests to the dispatcher, and the dispatcher distributes the requests to one of the redundant services. In a simple architecture, the redundant web servers will share a single database, so all replicas will have access to the same data. It is often required to replicate the database to support high availability, geographic distribution for low latency response time and efficient performance. Figure 1 shows an example of a WS farm with replicated databases in two WS clusters.

A WS farm also needs to ensure that the replicated databases are consistent. Several approaches have been proposed to support mutual consistency of replicated databases (see Oszu and Valduriez [14] for

Figure 1: Web Service farm with two clusters. Each cluster contains a load balancer, a single database, and replicated services.
an overview). However, the risk of lost updates in the presence of hardware failures has not been addressed sufficiently. Consequences of lost updates range from inconvenience, e.g., consumer needs to reissue a service request, to major losses, e.g., legal liability of lost financial data. A related event was recently reported by CBC News[5]. According to CBC, Google’s Gmail service had a replication issue that affected 38,000 user accounts by removing emails stored on replicated servers. Google recovered lost replicas by restoring from tape and manually refreshing the replicas. This slow and manual process caused an outage from February 27th, 2011 at 9pm till February 28th, 2011 at 5pm. During this manual operation any inbound emails to the 38,000 user accounts were not written to a persistence store and were lost. Google has not released enough information for us to fully understand the cause of this particular problem but it highlights the issues with durability in today’s SOA architectures.

In this paper we focus on the effect of server failures in the presence of lazy update propagation strategies. We show that, using the popular WS farm architecture, it is possible that updates are lost and cannot be recovered by current update propagation protocols. For example, assume that the database server crashes after the transaction commits but before the updates are propagated to any other replicas. Current WS farms are often constructed with share-nothing [13] architectures to enable assembly and replacement using commodity hardware. Service applications are implemented on the multiple copies of the commodity hardware. Similarly, replicated databases are implemented on commodity servers. If a database server crashes, technicians often replace a crashed server with a commodity replacement and never recover the original server. However, if the server crashes, before it has propagated the update of a transaction, then its update will be lost.

We propose an extension to the lazy replica update propagation method, to reduce the risk of data loss. Our approach is based on a “buddy” system. The transaction executes on a primary replica. However, the transaction cannot commit until a secondary replica, called, buddy, also preserves the effects of the transaction. The rest of the replicas are updated using one of the standard lazy update propagation protocols. Our method provides a balance between a guarantee of transactional durability (i.e., effects of the transaction are preserved even if the server, hosting the primary replica crashes before the update can be propagated to the other replicas) and efficient update propagation (i.e., our approach requires the synchronized update between two replicas only, therefore adding minimal overhead to the lazy replication protocol).

Our solution uses an application-layer dispatcher [1] to select the buddies based on the data items and the operations of the transactions, the data versions available, and the network characteristics of the WS farm. The main criteria is that the selection of the buddies ensures freshness of the read operations, that is read-requests must access the latest committed data version, and serializability of the update operations. We show that our approach reduces the window of vulnerability, that is the time period after a transaction is committed but before the updates are propagated to the other replicas. Although our solution can be viewed as a special case of a hybrid update propagation method, it provides several advantages not addressed in traditional distributed database replica update protocols. First, our approach provides the scalability required by modern applications, such as web services, and is suitable for the architectures and technologies implementing these applications. Second, the buddy-selection algorithm supports dynamic master-slave site selection for data items and ensures correct transaction execution. Third, we show that our method can be easily extended to incorporate network specific characteristics, such as distance and bandwidth, that further reduce the latency observed by the client and to provide load-balancing among the replicas. Our empirical results support our hypothesis that in the presence of large data sets, the efficiency of our approach is comparable to the efficiency of the lazy update propagation method.

The organization of the paper is as follows. Section 2 describes the related work and the limitations of current methods. In Section 3 we give an overview of the proposed Buddy System and present our algorithms. Section 4 contains the performance analysis of our method and compare it with the eager and lazy update propagation methods. In Section 5 we present our empirical results. We conclude and discuss future work in Section 6.

2 Related Work

Several methods have been developed to ensure mutual consistency in replicated databases. The aim of these methods is to eventually provide one-copy serializability (1SR). Transactions on traditional replicated databases are based on reading any copy and writing (updating) all copies of data items. Based on the time of the update propagation, two main approaches have been proposed. Approaches that update all replicas before the transaction can commit are called eager update propagation protocols; approaches that allow the
propagation of the update after the transaction is committed are called lazy update propagation. While eager update propagation guarantees mutual consistency among the replicas, this approach is not scalable. Lazy update propagation is efficient but it may result in violation of mutual consistency. During the last decade, several methods have been proposed to ensure mutual consistency in the presence of lazy update propagation (see [14] for an overview.) More recently, Snapshot Isolation (SI) [10, 8] has been proposed to provide concurrency control in replicated databases. The aim of this approach is to provide global one-copy serializability using SI at each replica. The advantage is, that SI provides scalability, and is supported by most database management systems.

One of the current application areas for replicated databases is Web Services applications. Lou and Yang[12] study the two primary replica update protocols in the context of web services. The authors state that eager replication has a problem of increasing latency as the number of replicas increases. This increasing latency diminishes the availability gains from introducing replicas. Most commercial implementations use lazy replication because its efficiency and scalability. Lazy replication methods are also more partition tolerant then eager replication methods. However, lazy replication protocols require additional considerations to ensure consistency.

Breitbart and Korth [2], and Daudjee et al. [3] propose frameworks for master-slave lazy replication updates with consistency guarantee. These approaches are based on requiring all writes to be performed on the master replica. Updates are propagated to the other sites after the updating transaction is committed. Their framework provides a distributed serializable schedule where the ordering of updates is not guaranteed. The approach proposed by Daudjee et al. provides multiversion serializability where different versions of data can be returned for read requests during the period that replication has not completed.

Jajodia and Mutchler[7] and Long et al.[11] both define forms of hybrid replication that reduce the requirement that all replicas participate in eager update propagation. The proposed methods aim to increase availability in the presence of network isolations or hardware failures. Both approaches have limited scalability because they require a majority of replicas to participate in eager update propagation. Most recently, Garcia-Munos et al. [6] proposed a hybrid replication protocol that can be configured to behave as eager or lazy update propagation protocol. The authors provide empirical data and show that their protocol provides scalability and reduced communication cost over other hybrid update protocols.

In addition to academic research, several database management systems have been developed that support some form of replicated data management. For example, Lakshman and Malik [9] describe a hybrid system, called Cassandra, that was built by Facebook to handle their inbox search. Cassandra allows a configuration parameter that controls the number of nodes that must be updated synchronously. The Cassandra system can be configured so nodes chosen for synchronous inclusion cross data center boundaries to increase durability and availability.

3 Proposed System

Our proposed system addresses three problems: decrease the risk of losing committed transactional data in case of a site failure, increase consistency of transactions, and increase availability of read requests. The three main components of our proposed system are: 1) Synchronous Transactional Buddy System, 2) Version Master-Slave Lazy Replication, and 3) Serializable Snapshot Isolation Schedule.

To support the above components, the dispatcher will operate at the OSI TCP/IP level 7. This will allow the dispatcher to use application specific data for transaction distribution and buddy selection. The dispatcher receives the requests from clients and distributes them to the WS clusters. Each WS cluster contains a load balancer, a single database, and replicated services. The load balancer receives the service requests from the dispatcher and distributes them among the service replicas. Within a WS cluster, each service shares the same database. Database updates among the clusters are propagated using lazy replication propagation.

3.1 Buddy System

As we have shown in Section 2, lazy update propagation is vulnerable for loss of updates in the presence of a database server failure. This is a particularly serious problem in the context of WS farms, where efficiency and availability are often prioritized over consistency. The window of vulnerability for this loss is after the transaction has committed but before the replica updates are initiated. To guarantee data persistence even in the presence of hardware failures we propose to form strict replication between pairs of replica clusters, ”buddies.” Our aim is to ensure that at least one of the replicas in addition to the primary replica is updated and, therefore, preserves the updates.
The dispatcher will maintain a version table for every object modified by web services. Each service request may include modification and read requests for several objects. When a service request is received, the dispatcher ensures that the request is delivered to the appropriate cluster. If the request is read-only, the primary buddy must have the latest version of all committed objects in the request. If the request includes writes, the dispatcher needs to determine if there is any uncommitted transaction accessing the requested data items. If it finds such active transactions then the request is sent to the web service cluster where the active transaction is being executed. If the dispatcher can not find a cluster with the latest version due to the distribution of the requested object, then the request is queued until the currently active transactions complete or the updates are propagated.

The dispatcher must also ensure snapshot isolation anomalies can be avoided. For this we address blind writes and analyse the read log to determine if an anomaly could take place. Operationally blind writes are writes that follow an earlier read operation where the write updates a value that was read earlier. Fekete et al. [4] documented anomalies that can be avoided to turn a snap shot isolation schedule into a serialized schedule. We incorporate these results to support serializability.

The dispatcher will maintain the following data structures for processing the algorithms: a.) Cluster List - contains the names of the clusters and their IP addresses. b.) Object Version Table - contains the name of the data items and their version numbers, corresponding to the completed and in-progress transactions. c.) Cluster Object Table - contains the cluster names, stored objects, and the version number of the objects at that cluster. For example, the example data structure tables (Table 1a, Table 1b, Table 1c) show that clusters 1 and 2 have two update operations on object A sent to them that are still in-progress.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Object</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>1012</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>1014</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>1014</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Object</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>954</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>954</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>954</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cluster</th>
<th>Object</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>2014</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>2014</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>2014</td>
</tr>
</tbody>
</table>

3.3 Dispatcher Service Request Algorithm

The dispatcher service request algorithm (Algorithm 1) is executed by the dispatcher for every incoming request containing write operations. The goal of the service request algorithm is to find a pair of buddies that have the correct version for the incoming request. If a pair can not be found then the request is added to a queue for later processing.

For read only requests the dispatcher will execute the read only version of the algorithm (Algorithm 2). This version only requires a single cluster to respond to the request. The cluster must have completed versions for each object in the request.
Theorem 1: The Dispatcher Service Request Algorithms (Algorithms 1 and 2) guarantee one-copy serializability.

Proof sketch: Our proof is based on the following claim: Let $H$ be a history over a set of transactions $T$, such that each transaction $T_i$, $i = 1, \ldots, n$ is made up of a set of read $R_i(A)$ and write $W_i(A)$ operations on elements from a data set. $H$ is one-copy serializable if the following three conditions hold:
1. Each request (transaction) is an atomic transaction
2. Concurrent writes on the same data item are sent to the same cluster, and
3. Each cluster guarantees serializable transaction history on their local database.

To show that the claim holds, assume, by contradiction that $H$ is not one-copy serializable. Then, there must exist a cycle among the committed transactions in the serialization graph of $H$. Let $T_1$ and $T_2$ be the two transactions responsible for the cycle. We show that the serialization graph cannot contain a cycle for the three potential scenarios or these transactions. The three scenarios are: Read Set/Write Set overlap, Write Set/Write Set overlap, and Read Set/Read Set overlap.

Load Balancing

Algorithms 1 and 2 choose the first available cluster for read only requests, and the first available pair of clusters for requests containing write operations. The selection can be improved by decorating the Cluster List table (Table1a) with properties to represent system properties (e.g., processing power, available applications, process wait-time, etc.) and network-related information (e.g., link properties, hop-distances, etc.) that can influence buddy selection. For example, buddies may be selected based on their geographical location and the reliability of the communication network. Our current work extends Algorithm 1 with the capability of incorporating these semantics.

3.4 Dispatcher Version Update Algorithm

The Dispatcher Version Update algorithm (Algorithm 3) is executed by the dispatcher when a data item is updated. When a primary buddy or any lazy update cluster completes a transaction, it will send a version update request to the dispatcher. The dispatcher will update the latest completed version value for these clusters. After the version is updated any requests in the queue will be reprocessed in hopes that the dispatcher can now find a pair of buddies with the correct versions.
Algorithm 3 Dispatcher Version Update Algorithm
INPUT: versionUpdates (Triple of cluster, object, version)
OUTPUT: buddyList (Pair of buddies or empty list if no pair available), objectVersions (Table1b)

1 for each object, version in versionUpdates
2 update completed = version in objectVersions
3 process requests from queue

Algorithm 4 Primary Buddy Service Algorithm
INPUT: requestObjects (Request containing objects to be read and written)
OUTPUT: dataSet (data requested in read operations), objectVersions (Table1b)

1 initialize writelist to an empty list
2 for each object, action in requestObjects
3 update the latest completed version
4 if action == WRITE
5 Add object to writelist
6 Lock object
7 Log write operation
8 Write undo log for write operation
9 else if action == READ
10 add data to dataSet
11 if there are items in writelist
12 send writelist to secondary buddy
13 if secondary buddy committed properly
14 for each object in requestObjects
15 complete write on object
16 release lock on object
17 else if secondary buddy aborted
18 for each object in requestObjects
19 undo write on object
20 release lock on object
21 send response to client
22 send version update to dispatcher

3.5 Primary Buddy Service Algorithm

This section describes the interaction between the primary and secondary buddies. The primary buddy service algorithm (Algorithm 4) is executed on the primary buddy for every incoming request from the dispatcher. The goal of the primary buddy algorithm is to prepare the request on its cluster by locking resources. If the request includes write operations then the request is sent to the secondary buddy. If the secondary buddy can execute the transaction then the primary will finish the transaction and send a response to the client and a version update to the dispatcher.

4 Analysis of the Buddy System

In this section we study a specific aspect of our proposed system. First, we evaluate the performance of our system in high-volume scenarios. Next, we compare our approach with eager and lazy replica update propagation in the presence of hardware failures.

4.1 Performance Analysis in High Volume Scenarios

Some Web Service transactions involve large volumes of data items of the same type. For example, if a client is purchasing a concert ticket, multiple tickets have the same characteristics but different row and seat numbers. If we study a high volume scenario where there are large number of tickets being purchased, then there are three types of consumption patterns that are exposed in this scenario:

- Anonymous Item Consumption - In this pattern each ticket is interchangeable, for example all seats are general admission. The buddy system would not improve latency over simple master-slave replication since all concurrent resources requests would need to be sent to the same buddy pair.

- Attribute Item Consumption - In this pattern each client’s request has attribute filters, such as main-floor or balcony. The buddy system would improve latency over simple master-slave replication because each set of attributes could be sent to a different buddy pair.

- Serialized Item Consumption - In this pattern each client’s consumption request is for a specific seat. The buddy system would greatly improve latency over simple master-slave replication because each seat request could be sent to a different buddy pair.

4.2 Analysis of Lost Updates in the Presence of Failures

Lazy Replication Durability: In each proposed lazy replication scenario, there is one master for a particular data item. After a transaction has committed there is a period of time where there is a vulnerability that a lost update can occur if hardware hosting the master replica fails before the lazy update propagation is initiated.

Eager Replication Durability: In eager replication the window of vulnerability of lost updates is removed because the updating transaction cannot commit until all other replicas are also updated. Generally, the two phase commit (2PC) protocol is used across replicas to achieve this goal. However, the update cost of eager update propagation is high, and, therefore, it is not used
frequently.

*Buddy System Durability:* Using the buddy system, we can guarantee durability. The weakest point of the buddy system is the durability of the dispatcher. If the dispatcher fails, the data structures may get lost and recovery activities must be performed.

Figure 3 shows the workflow of the hybrid eager and lazy solution we proposed. This solution has higher durability than the lazy propagation because two replicas will get the original transaction so a hardware error on one replica will not result in the loss of update.

Table 2 presents our analysis of the hardware failures at the different stages of the transaction execution. The first column represents the failed hardware, the following columns detail the stages: before the transaction started, during execution, and after the transaction committed but before the update is propagated.

### Table 2: Windows of Vulnerability

<table>
<thead>
<tr>
<th>Hardware</th>
<th>InActive</th>
<th>During Transaction</th>
<th>After Transaction Committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Server</td>
<td>No issue because of redundancy</td>
<td>2PC will roll back trans, Client will get error from primary</td>
<td>No issue because of redundancy</td>
</tr>
<tr>
<td>Housing Web Service of primary buddy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Web Server</td>
<td>Normal</td>
<td>2PC will roll back trans, Client will get error from primary</td>
<td>No issue because of redundancy</td>
</tr>
<tr>
<td>Housing Web Service of secondary buddy</td>
<td>No issue because of redundancy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Database Server</td>
<td>Normal</td>
<td></td>
<td>No issue because of redundancy</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>New dispatcher, Each clusters will update dispatcher versions</td>
<td>After dispatch to primary, no issue because primary responds directly to client</td>
<td>New dispatcher, Each clusters will update dispatcher versions</td>
</tr>
</tbody>
</table>

5 Implementation

We tested the performance of our Buddy-system against the lazy and eager replica update protocols. We also considered two possible communication architectures: synchronous and asynchronous communication. Using asynchronous communication, the client sends a request and waits for a response to be sent asynchronously. In synchronous communication the client waits until the response is received. The major difference in these two methods is how the enqueue process is handled when the dispatcher can not fulfill the request with the current state of the clusters.

Figure 4 shows the empirical data from an implementation using synchronous requests from a Java desktop application. The dispatcher is written in Java EE using a Tomcat servlet container. The dispatcher uses class attributes to share hash tables, the internal data structures, across all request threads. Each cluster is also implemented in Java EE using a Tomcat servlet container. A separate MYSQL database is used by each cluster in serializable isolation. The cluster uses a JDBC connection pool communicating to its individual database.

A dataset with different sizes was generated with each transaction randomly selecting two items to read and one item to write. Buddy-100, Buddy-1000 and Buddy-10000 represent the performance of the Buddy algorithm with a dataset size of 100, 1000, and 10000 items, respectively. The same transactions were run against a single, master cluster system with lazy replication and a two clusters system with strict replica-
tion. Figure 4 shows that once the dataset size grew to 10,000 items the performance of the Buddy algorithm matches the performance of lazy replication, while increasing durability.

The severe performance penalty observed with small datasets is the result of the enqueue process and the overhead of selecting buddies. Our ongoing work aims to improve the buddy selection algorithm and to reduce the number of transactions that cannot be processed concurrently. Also, in the current implementation the dispatcher stores the version data structures in memory. Our future implementation will store these tables in secondary storage to increase redundancy and durability of the dispatchers’ data.

6 Conclusion

In this paper we propose an extension to the lazy replica update propagation methods. Our solution is based on using an application-layer dispatcher [1] to select the buddies based on the data items and the operations of the transactions, the data versions available, and the network characteristics of the WS farm. We show that although our solution can be viewed as a special case of hybrid update propagation method, it provides several advantages not addressed in traditional distributed database replica update protocols. First, our approach provides the scalability required by modern applications, such as web services, and is suitable for the architectures and technologies implementing these applications. The buddy-selection algorithm supports dynamic master-slave site selection for data items, ensures correct transaction execution, and aids load-balancing among the replicas. Finally, incorporating network specific characteristics, such as distance and bandwidth, reduces latency observed by the client. Our approach has a similar message complexity compared to a traditional lazy master-slave replication using group communication and less message complexity to a lazy master-slave replication without group communication.

The limitation of our work is that the dispatcher represent a single point of failure. Our ongoing work extends our solution to incorporate multiple dispatchers to eliminate this limitation.

References