2014

Web Service Transaction Correctness

Aspen Olmsted
University of South Carolina - Columbia

WEB SERVICE TRANSACTION CORRECTNESS

by

Aspen Olmsted

Bachelor of Science
State University of New York, 1989

Master of Business Administration
University of South Carolina, 2007

Master of Science
College of Charleston, 2009

Submitted in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy in

Computer Science

College of Engineering and Computing

University of South Carolina

2014

Accepted by:

Csilla Farkas, Major Professor

Michael Hodgson, Committee Member

Michael Huhns, Committee Member

Manton Matthews, Committee Member

John Rose, Committee Member

Lacy Ford, Vice Provost and Dean of Graduate Studies

Abstract

In our research we investigate the problem of providing consistency, availability and durability for Web Service transactions. First, 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.

Second, we consider Web Service transactions that consume anonymous and attribute based resources. We show that the availability of the popular lazy replica update propagation method can be achieved while increasing its durability and consistency. Our system provides a new consistency constraint, Capacity Constraint, which allows the system to guarantee that resources are not over consumed and allows for higher

distribution of the consumption. Our method provides: 1.) increased availability through the distribution of an element's master by using all available clusters, 2.) consistency by performing the complete transaction on a single set of clusters, and 3.) guaranteed durability by updating two clusters synchronously with the transaction.

Third, we consider each transaction as a black box. We model the corresponding metadata, i.e., transaction semantics, as UML specifications. We refer to these WS-transactions as coarse grained WS-transactions. We propose an approach that guarantees the availability of the popular lazy replica update propagation method while increasing the durability and consistency. In this section we extend the Buddy System to handle coarse-grained WS-transactions, using UML stereotypes that allow scheduling semantics to be embedded into the design model. This design model is then exported and consumed by a service dispatcher to provide: 1.) High availability by distributing service requests across all available clusters, 2.) Consistency by performing the complete transaction on a single set of clusters, 3.) Durability by updating two clusters synchronously.

Finally, we consider enforcement of integrity constraints in a way that increases availability while guaranteeing the correctness specified in the constraint. We organize these integrity constraints into three categories: entity, domain and hierarchical constraints. Hierarchical constraints offer an opportunity for optimization because of an expensive aggregation calculation required in the enforcement of the constraint. We propose an approach that guarantees the constraints enforcement. Our approach also distributes the write operations among many clusters to increase availability. Our experimental results show increased performance when compared to the lazy update

propagation algorithm.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

1SR.....One Copy Serializability

2PC.....Two Phase Commit

ACID..... Atomic, Consistent, Isolated, Durable

CAP.....Consistent, Available, Partition Tolerant

RDBMS.....Relational Database Management System

SI.....Snapshot Isolation

SOA.....Service Oriented Architecture

WS..... Web Service

Chapter 1

Introduction

Modern web based transaction systems need to support many concurrent clients simultaneously consuming a limited quantity of resources. These applications are often developed using a Service Oriented Architecture (SOA). 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 concurrency. We can think of as the availability of a service to all requesting clients requesting services. A common way to increase service availability is through replication. This requires replication of services and their corresponding resources. Unfortunately consistency and durability are often sacrificed to achieve this availability. The CAP theory [1, 2], (which states that distributed database designers can achieve at most two of the following properties: consistency (C), availability (A), and partition tolerance (P)) has influenced distributed database design in a way that often causes the designer to give up on immediate consistency.

The standard architecture used to increase the availability of a system is through a Web Service (WS) farm. The 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 high availability. 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. Figure 1.1 illustrates a simple WS farm. It is often required to replicate the database to support high availability and geographic distribution for low latency response time. This architectural solution solves the problem of increasing availability by increasing the capacity of servers but decreases data consistency. WS farms often use lazy replicated update propagation methods.

An example of a transaction time correctness guarantee that is lost in this high availability architecture is referential integrity. In a simple web shopping cart you may have an orders table with a foreign key to a customer table. Referential integrity would

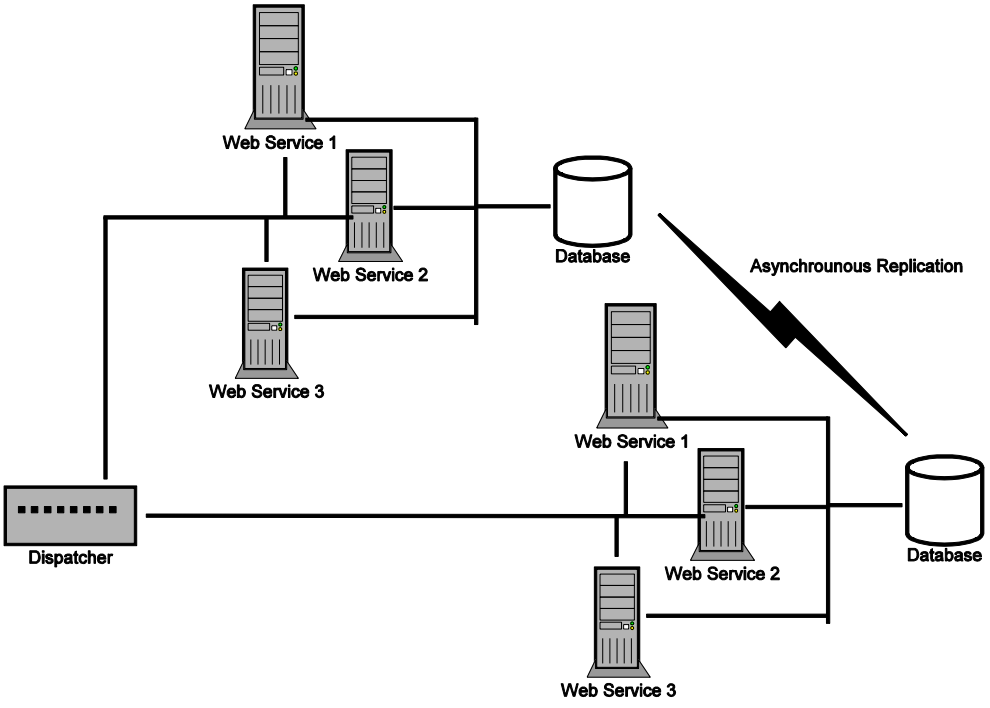

Figure 1.1 Example Web Service Farm

guarantee that an order cannot be committed unless the customer existed. With lazy replication the customer table may have one cluster as the master, and the order table may have a separate table as the master. This does not allow the database to ensure that the integrity existed at transaction time and would force the integrity to be resolved post transaction time.

Our research addresses the issues related to increasing availability while still guaranteeing durability and consistency of replicated databases in the context of SOA. We will provide algorithms and architectures that guarantee one-copy serializability and ensure that data is distributed in a way that provides enforcement of referential integrity, redundancy for higher durability, and high levels of availability.

1.1 Research Overview

The continuous connectivity introduced by the internet has created a demand for applications that can serve a large numbers of users. Many developers have given up on traditional relational database systems, with their associated guarantees of consistency and durability, to increase the availability of their systems. In this context availability is a measure of the number of concurrent users that can be serviced by a system without system downtime or users experiencing error messages. The goal of our research is to develop new algorithms and architectures that will increase the availability of distributed systems while maintaining the consistency and durability that users were guaranteed by traditional database management systems.

1.2 Motivation

Industry has moved away from requiring all transactions to the ACID (Atomicity, consistency, integrity, durability) properties. This relaxed requirement is motivated by the need to increase data availability. Unfortunately users experience incorrect data which causes confusion. An example of this problem is a web based banking interface that uses a replicated copy of a user's account activity. If the user provided a payment over the phone the transaction may have been executed on one system but not replicated to all systems at any point in time. The users will not see this payment in an online system and will be confused as to the real state of their bank account.

This kind of confusion may be tolerable in some industries but not in others such as health care or security. When a decision, based on incorrect data, could cost someone their life, the correctness of the data becomes more important.

1.3 Problem

The problem is to develop algorithms and architectures for distributed systems that increase availability over strict replication while preserving ACID guarantees.

The challenge for resource consumption in distributed systems is that once a resource in a transaction is used, it is not ever available to further transactions. Resources can be grouped into three categories:

- Serialized items – In this category each individual item has a unique identifier. An example of this type of item would be an assigned seating location for a performance. A user has a ticket for seat A 101 on the main floor.

- Anonymous items – In this category all items are interchangeable. An example of this type of item would be a general admission ticket where the ticket gets you into the event and you can pick any seat. The organization selling this resource knows the capacity they cannot exceed but the individual items are not distinguished.
- Attribute based items – Attribute based items have similarities to both serialized and anonymous resources. An attribute based item has blocks of capacity with an set of attributes that identify the block. An example of this type of item would be a general admission ticket to the floor for a concert. The ticket allows you into a specific section but within the section it is up to you to pick your seat.

1.4 Research Tasks

Four research tasks are addressed as follows:

Availability Increase in Serialized Resource Consumption

In this task 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. The results of this work were published in the proceedings of 2012 IEEE International Conference on Information Reuse and Integration [3] and the Journal of Internet Technology and Secured Transactions [4].

Availability Increase in Anonymous Resource Consumption

In this task we investigate the problem of providing consistency, availability and durability for Web Service transactions that consume anonymous and attribute based resources. We show that the availability of the popular lazy replica update propagation method can be achieved while increasing its durability and consistency. Our approach is based on an extension to the Buddy System, requiring that updates are preserved synchronously in two replicas, called buddies. Our system provides a new consistency constraint, Capacity Constraint, which allows the system to guarantee that resources are not over consumed and also allows for higher distribution of the consumption. Our method provides 1.) Higher availability through the distribution of a element's master by using all available clusters, 2.) Consistency by performing the complete transaction on a single set of clusters 3.) A guaranteed durability by updating two clusters synchronously with the transaction. The results of this work were published in the proceedings of 2012 IEEE Internet Technology and Secured Transactions [5] and the Journal of Internet Technology and Secured Transactions [4].

Availability Increase in Course Grained Web Service Scheduling

In this task we investigate the problem of providing consistency, availability and durability for Web Service-transactions. We consider each transaction as a black box, with only the corresponding metadata, expressed as UML specifications, as transaction semantics. We refer to these WS-transactions as coarse-grained WS-transactions. We propose an approach that guarantees the availability of the popular lazy replica update propagation method while increasing durability and consistency. In our previous work, we proposed a replica update propagation method, called Buddy System, which required that updates are preserved synchronously in two replicas. In this section we extend the Buddy System to handle course grained WS-transactions, using UML stereotypes that allow scheduling semantics to be embedded into the design model. This design model is then exported and consumed by a service dispatcher to provide: 1.) High availability by distributing service requests across all available clusters. 2.) Consistency by performing the complete transaction on a single set of clusters. 3.) Durability by updating two clusters synchronously. The results of this work were published in the proceedings of 2013 IEEE Web Services [6] and the Journal of Internet Technology and Secured Transactions [4].

Constraint Guarantees in Web Service Transactions

In this task we tackle the problem of designing and enforcing consistency guarantees in a distributed web service system. We use object constraint language to specify domain, entity, hierarchical and temporal constraints. We guarantee both client and server constraint using the semantics gained in the previous tasks to auto-generate

compensators to undo transactions if client constraints do not hold after completion of a service request. The results of this work were published in the proceedings of 2013 IEEE Internet Technology and Secured Transactions [7]

The organization of the dissertation is as follows. In chapter 2 we present related research. In chapter 3 we present our research findings on availability increase in serialized resource consumption. In chapter 4 we present our research findings on availability increases in anonymous resource consumption. In chapter 5 we present our research findings on availability increase in course grained web service scheduling, and in chapter 6 we present our research results on constraint guarantees in web service transactions.

Chapter 2

Related Work

Most of the distributed database research ignores resource consumption issues and assumes traditional locking mechanisms. Julian Jang et al [8] investigate ways to provide non-locking resource consumption for a longer duration than the transaction to avoid holding locks. Unfortunately this approach sacrifices serializable guarantees of ACID (Atomicity, consistency, integrity, durability) that are provided by traditional relational database management system (RDMS). One of the current application areas for replicated databases is Web Services applications. Lou and Yang [9] 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 of its efficiency and scalability. Lazy replication methods are also more partition tolerant than eager replication methods. However, lazy-replication protocols require additional considerations to ensure consistency. Research has been conducted for decades on strict and lazy replication in RDMS. Recent research can be grouped into one of three goals: 1.) trying to increase availability with strict replication, 2.) trying to increase consistency with lazy replication, and 3.) attempting to use a hybrid approach.

Increasing Availability with Strict Replication

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 [10]for an overview.) More recently, Snapshot Isolation (SI) [11, 12] 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.

Increasing Consistency in Lazy Replication

Breitbart and Korth [13], and Daudjee et al. [14] 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 multi-version serializability where different versions of data can be returned for read requests during the period that replication has not completed.

Hybrid Approach

Jajodia and Mutchler [15] and Long et al. [16] 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. [17] 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 reduces 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 [18] describe a hybrid system, called Cassandra, which 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.

2.1 Service Coordination, Composition and Transactions

Web service transaction management research shares many aspects with web service coordination and composition. Over the lifespan of a transaction, the web

services called will have specific sequencing requirements. Several standards have been created as the result of years of research in this area.

WS-Business Activity

WS-Business Activity [19] is an OASIS standard created for defining the coordination of long running transactions implemented with many web services. The goal of a WS-Business Activity transaction is to ensure that all participants agree on the outcome of a transaction. A WS-Business Activity Transaction can involve many different service providers in a single transaction. WS-Business Activity uses other OASIS standards in the WS* stack including WS-Coordination and WS-Policy to define the transactional behavior. WS-Coordination is used to coordinate the participants in the transaction. WS-Policy is used to define the behavior of the transaction.

Web Service Business Process Execution Language

WS-BPEL is a standard developed by OASIS [20] for designing the workflow between web services inside one realm of authority. Web Services are combined into a workflow expressed in WS-BPEL and the result is a web service that can be called by other clients to execute the workflow.

WS-BPEL Scope

To support Long Running Transactions (LRT) WS-BPEL implements Scopes. A WS-BPEL scope is a combination of a database transaction and a traditional scope in an imperative programming language. A compensation handler is available in the scope to

undue the results of the activities if not all the activities in the scope are successful. WS-BPEL also supports Isolated Scopes which hold locks on resources like an atomic transaction to ensure serialize-ability. A WS-BPEL scope does not support coordination of scopes beyond one BPEL engine.

WS-BPEL Compensation vs, WS-BusinessActivity Compensation

Both WS-BPEL and WS-BusinessActivity support Long Running Transactions (LRT) through compensation, but the management of the compensation is quite different in the two specifications. In WS-BPEL compensation is implemented and controlled at the workflow engine. In WS-BusinessActivity the compensation is implemented and controlled at the service provider. This allows each participating provider in a WS-BusinessActivity transaction to decide how it compensates separately. This separate decision making leads to the reduction of the atomic transaction property described above. Sauter and Melzer [21] study the combination of WS-BusinessActivity to manage separate WS-BPEL engines.

2.2 Long Running Transactions

Traditional ACID transactions use locks to guarantee the ACID properties. These transactions tend to take milliseconds to complete so the negative side of effects of the locks is often ignored in favor of the guaranteed benefits. Long running transactions run over longer periods of time and may involve human interaction in the middle of the transaction. This elongated time period makes the traditional method of using locks much less desire-able. At the highest level of isolation in a database transaction, serialize-able, all records in the range of reads are locked for the duration of the

transaction. For a long running transaction this can essentially shutdown a service provider.

Sagas

In Garcia-Molina and Salem [22] defined sagas as a solution to maintain some of the atomic properties over long running transactions. With Sagas, many small atomic transactions are wrapped by a larger longer running transaction. Each small atomic transaction is paired with a compensation handler that is capable of reversing the activity done in the atomic transaction. If the long running transaction needs to cancel before completion then it can call the compensators in reverse order for all completed atomic transactions. Unfortunately with most implementations of Sagas the compensators need to be hand coded to create a reverse operation of the atomic transaction. This hand coding leads to many opportunities for errors over the life time of a product.

Relaxation of Isolation

American National Standards Institute (ANSI) SQL compliant database systems support 4 levels of isolation; Serialize-able, Repeatable Read, Read Committed, Read Uncommitted. The database programmer is able to set the isolation level before a database transaction to achieve a higher level of availability in exchange for less isolation. Correctness is traditionally measured from the perspective how a transaction would behave if it was run in complete isolation from the other concurrent transactions. The highest isolation level, Serialize-able, will use many database locks so that each concurrent transaction is in complete isolation of other concurrent

transactions. The next level, Repeatable Read, relaxes the level of isolation down so that two executions of the same query may return different result sets but protects the serialization so that any records read in one concurrent transaction cannot be modified by another concurrent transaction. The third level, Read Committed, relaxes the level of isolation down further by allowing one concurrent transaction to modify rows previously read by another concurrent transaction. The lowest level of isolation in database transactions is achieved by setting the isolation level to Read Uncommitted. In Read Uncommitted, changes made to records in one concurrent transaction are immediately visible to other concurrent transactions.

Long running web transactions inherently operate at the same isolation level as the ANSI SQL Read Uncommitted level. As soon as one long running transaction updates a resource, the change will be visible to other long running transactions. The traditional way with long running web service transactions to not relax the isolation to this level is to hold locks on used resources for the duration of the long running transaction. A versioning manager could be used as an alternative, to serve different versions to different concurrent long running transactions. Versioning has been implemented in database systems to increase availability over the Serialize-able isolation level but to not relax the isolation. The versioning implemented in commercial database systems does relax the isolation a little without the knowledge or consent of the database designer. Fekete et al. [23, 24, 25] have contributed algorithms that allow transactions to still provide guarantees in spite of the isolation relaxation.

Transaction Compensation

To ensure a database transaction maintains it atomic property the database management software has the ability to undue all the activities done by one concurrent transaction to enforce the all or nothing principle of a transaction. This undoing is referred to as a rollback of the transaction in database software. Long running transactions many not have the ability to undue or many not want to undue the parts of a transaction that were completed at the point that a transaction decides to abort. With the relaxation of the isolation property discussed earlier, other actions may have possibly been taken based on the partial completion of the transaction. Web service transactions implement a concept of compensation where each service provider is able to decide if and how they want to handle the abortion of a transaction they are a participant in. Some service providers may try to completely undue the activities of the transaction similar to a rollback and others may choose to ignore the abort. Schafer et al. [26, 27, 28] have researched ways to use compensation to provide a level of guarantee of correctness for transactions.

Relaxation of Atomic

With transaction compensation a service provider may decide to not undue an activity that was part of an aborted transaction. It may also not be possible to completely undue a transaction because of activities that may have happened based on the exposed information from the partial transaction. This leads to a relaxation of the atomic principle of database transactions since part of a transaction may be left in place depending on the decisions made by a service provider in the compensation handler.

Open Nested Transactions

In some database management systems, transactions can be nested inside other transactions. This is done by issuing a begin transaction statement while already inside another transaction. The database management system will isolate other concurrent transaction from the results of the inner transactions until the outer transaction completes. If the outer transaction cannot complete, the inner transactions will be rolled back along with the outer transaction. With web service transactions this level of isolation is relaxed. Both WS-BusinessActivities and WS-BPEL LRT can have atomic transactions running inside the long running transactions. The compensation handler is responsible for undoing the results of the inner transactions when they compensate. Garcia-Molina and Salem [22] they define a transaction as a saga if it can be rearranged into an open nested transaction.

Chapter 3

Consistency, Availability & Durability Guarantees with Serialized Item Consumption

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 Preliminaries

- A *Web Service Farm* is composed of a single dispatcher, D, and a set of Web

Service Clusters WSF = (D,{WSC_1, . . . ,WSC_n}). The dispatcher receives requests from clients and distributes these requests to the WS-Clusters.

- A *WS-Cluster* is a group of WS-Replicas that share a single data store and a load balancer. Each WS- Cluster (WSC) is represented as a three tuple WSC = (WS, HW, DB), where WS is a web service, HW = {hw_1, . . . , hw_n} is a set of common, off-the-shelf (COTS) hardware devices running identical copies of WS. DB is a database. In this work, we consider relational databases. The load balancer distributes load to the service replicas in the cluster.
- *WS-Replica Buddies* are ws_i and ws_j, such that ws_i and ws_j are replicas and they belong to two different WS clusters.
- A *Database Transaction* is a partial order of read and write operations on data items, and a single abort or commit. We denote a transaction T as follows, T = {≤, r[d], w[d] | d ∈ DB, c/a }. The read-set of a transaction T denotes all the data items d ∈ DB such that there is a r[d] ∈ T. The write-set of a transaction T denotes the data items d ∈ DB such that there is a w[d] ∈ T.
- *Data item version* denotes a data value and its version number. Given a database DB = {d_1, . . . ,d_n} each data item d_i (i = 1,..,n) is associated with a single version number v_n. Initially each data item's version number is 0. Version numbers are incremented by one when a data item is updated by a transaction. For clarity we model the database as pairs of data item and version numbers, that is DB = {(d_1, v_1), . . . ,(d_n, v_n)}. In this dissertation we use the term Object and Data item interchangeably.
- Each *database* is associated with a *version number*. Given a database DB and the

data items $\{((d_1, v_1), \ldots, (d_n, v_n)\}$ in DB, we say the version numbers of DB is the vector $V=<v_1,\ldots, v_n>$.

- *DB-Replicas*, denoted as DBR = $\{dbr_1, \ldots, dbr_n\}$, are databases originating from the same database (i.e., version $<0_1, \ldots, 0_n>$). Given two replicas, dbr_1 and dbr_j, they will have the same data items but may or may not have the same version number.

  Note, for any two database replicas dbr_i and dbr_j if $v_i = v_j$ then the two replicas must have the same values for each data item.

3.2 Buddy System

As we have shown in the introduction, 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 up-dated and, therefore, preserves the updates.

Figure 1.1 shows a WS farm architecture where each cluster has a load balancer. After receiving a transaction, the dispatcher picks the two clusters to form the buddy-system. The selection is based on versioning history. The primary buddy will receive the transaction along with its buddy's IP address. The primary buddy will become the

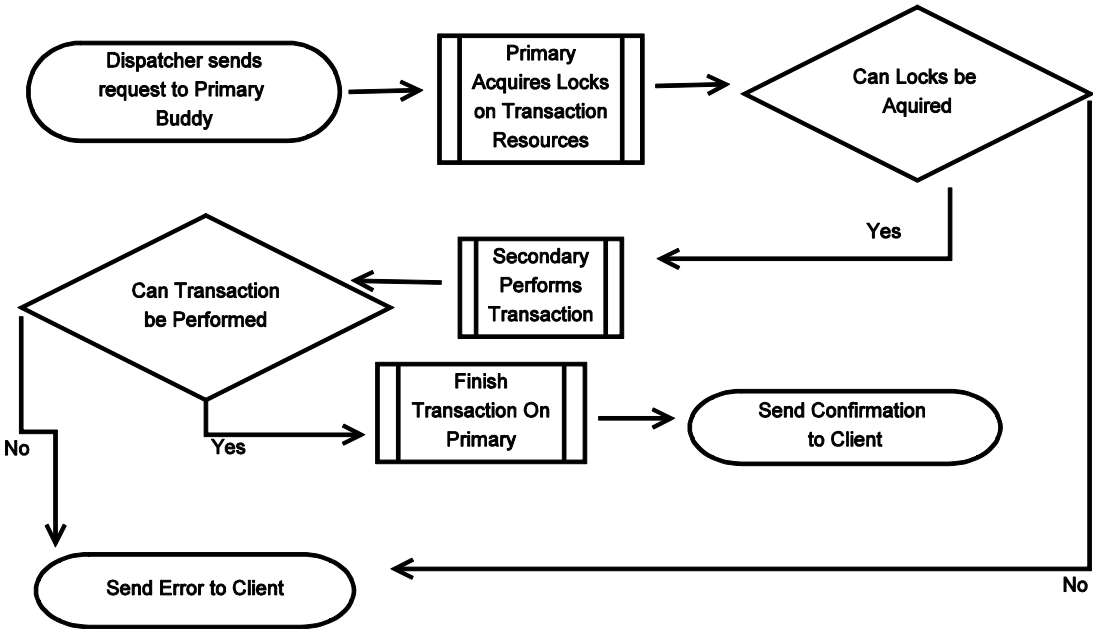

Figure 3.1 Flowchart of Communication between Primary and Secondary

coordinator in a simplified commit protocol between the two buddies. Both buddies perform the transaction and will commit or abort together. Figure shows the workflow of the transaction processing by the buddies. The dispatcher maintains metadata about the fresh-ness of data items in the different clusters. The dispatcher will increment a version counter for each data item after is has been modified. Any two service providers (clusters) with the latest version of the re-quested data items can be selected as a buddy. Note, that the database maintained by the two clusters must agree on the requested data items but may be different for the other data items.

3.3 Dispatcher Data Structures

The dispatcher will maintain a version table for every object modified by web services. Each service re-quest may include modification and read requests for several objects. When a service request is received, the dispatcher ensures that the request is

Table 3.1 Example Cluster List

Cluster	IP
1	192.168.3.1
2	192.168.3.2
3	192.168.3.4

delivered to the appropriate cluster.

If the request is read-only, the primary buddy must have the latest version of all com-mitted 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 cannot 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 analyze the read log to determine if an anomaly could take place. Operationally blind writes are writes that follow an earlier read operation

Table 3.3 Example Object Version Table

Object	Completed	In-Progress
A	1012	1014
B	954	954
C	2054	2054

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

Table 3.2 Example Mixed Transaction Table

Clusters	Read	Write
1,2	A,B	C
1,2	C	A
3,4	D	E

serializability. The dispatcher will maintain the following data structures for processing the algorithms:

- Cluster List - contains the names of the clusters and their IP addresses.
- Objects Version Table - contains the name of the data items and their version numbers, corresponding to the completed and in-progress transactions.
- Mixed Transaction Table – contains all open request with mixed (both read and write) operations

Table 3.4 Example Cluster Object Table

Cluster	Object	Version
1	A	1014
2	A	1014
3	A	1012
1	B	954
2	B	954
3	B	954
1	C	2054
2	C	2054
3	C	2054

- 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 3.1, Table 3.3, and Table 3.3) show that clusters 1 and 2 have two update operations on object A sent to them that are still in-progress.

3.4 Dispatcher Service Request Algorithm

The dispatcher service request algorithm (Algorithm 3.1) 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 cannot be found then the request is added to a queue for later processing. The algorithm has a special check for anti-dependency that will ensure that either the request is passed to the clusters updating the current records or waits for the dependent transaction to complete. For read only requests the dispatcher will execute the read only version of the algorithm (Algorithm 3.4). This version only requires a single cluster to respond to the request. The cluster must have completed versions for each object in the request

Algorithm 3.1 Dispatcher Service Request Algorithm –Writes

INPUT: requestedObjects = $\{O_1,\ldots, O_k\}$, where each O_i is a pair (O.id, O.action); O_i.id is the object identifier, O_i.action is the requested action.
OUTPUT: buddyList is a pair (B_1,B_2) of clusters to participate in the transaction.
TABLES USED: CL = cluster list table, OV =object version table, CO = cluster object table

```
buddyList = {}
available = all custer ids in CL
foreach O.id, O.action in requestObjects
    /* find latest version of an object */
    if NOT O.id in OV
        insert o.id into OV
        OV.complete=1,OV.inprogress=1
    set v.complete = OV.complete, v.inprogress = OV.inprogress \
      where ov.object = o.id
    /* eliminate unqualified clusters from potential buddies */
    foreach co.cluster, CO.object, CO.version in CO
        If co.version > V.complete && O.action==READ
            available.remove(co.cluster)
        elseif co.version < OV.inprogress && O.action==WRITE
            available.remove(co.cluster)
        elseif O.action==WRITE && antidependency(requestobjects,co.cluster)
            available.remove(co.cluster)
/* pick a pair of clusters */
foreach cl.cluster in CL
    if available(cl.cluster) and buddyList.count() < 2
        buddyList.add(cl.cluster)
if buddyList.count() > 1
    let b1,b2 denote two clusters in buddylist
    * update version information for write object
    foreach O.id, O.action in requestObjects
             if O.action==WRITE
          increment OV.inprogress for ov.object = o.id
          increment cO.version for cluster = b1 & co.object = o.id
          increment cO.version for cluster = b2 & co.object = o.id
    send buddyList,requestObjects to b1
else
    enqueue(requestObjects)
```

Load Balancing

Algorithm 3.1and Algorithm 3.4 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 (Table 3.1) with properties to represent sys-tem 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, bud-dies may be selected based on their geographical location and the reliability of the communication network.

Our current work extends Algorithm 3.1with the capability of incorporating these semantics.

3.5 Anti-dependency Detection Algorithm

The Anti-dependency detection algorithm (Algorithm 3.2) is executed by the dispatcher service request algorithm (Algorithm 3.1) to determine if a cluster should be eliminated from consideration for servicing a request. The algorithm will return a Boolean true if the request would have an anti-dependency with a pervious request if past to the current

3.6 Dispatcher Version Update Algorithm

The Dispatcher Version Update Algorithm (Algorithm 3.3) 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 3.2 Anti-Dependency Algorithm

INPUT: requestedObjects = {O_1,…, O_k}, where each O_i is a pair (O.id, O.action); O_i.id is the object identifier, O_i.action is the requested action. clusterId = is the id of the cluster being checked for anti-dependency
OUTPUT: boolean. True if there is an anti-dependency

```
antidependency = FALSE
foreach mt.read, mt.write in MT where mt.cluster NOT in clusters
      foreach O.id, O.action in requestObjects
            if o.action == WRITE && mt.read.contains(o.id)
                  antidependency = TRUE
            elseif o.action == READ && mt.write.contains(o.id)
                  antidependency = TRUE
return antidependency
```

Algorithm 3.3 Dispatcher Version Update Algorithm

INPUT: versionUpdates =(Triple of cluster, object, version
OUTPUT: buddyLis=(Paid of buddies or empty list if no pair available,

```
For each object, version in versioUpdates
update completed = version in objectVersions
Process requests from queue
```

3.7 Primary Buddy Service Algorithm

This section describes the interaction between the primary and secondary buddies. The primary buddy service algorithm (Algorithm 3.5) 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 re-quest 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.

Theorem 1: The Dispatcher Service Request Algorithms (Algorithm 3.1& Algorithm 3.4) guarantee one-copy serializability.

Proof:

Claim 1: H is one-copy serializable if the following three conditions hold:

1. The conflicting transactions are sent to the same pair of clusters (WSC)
2. Each cluster guarantees serializable transaction history on its local database.
3. Each request (transaction) is an atomic transaction

Proof of Claim 1:

For a transaction to be one-copy serializable, there must not exists a cycle among the committed transactions in the serialization graph of H [10]. For a cycle to exist the following must be true:

- An operation of T_i precedes a conflicting operation in T_j and an operation of T_j precedes a conflicting operation in T_i.

We show, that if the above 3 conditions hold, there cannot be a cycle in the serialization graph. Condition 1 ensures that the both transactions T_i and T_j are sent to the same cluster. Condition 2 ensures that the cluster will serialize the conflicting transactions T_i and T_j. Condition 3 ensures that the entire transaction T_i is in a single request to the dispatcher, allowing the local database to see the complete transaction at once. These three conditions ensure that any potential cycle is sent to the same pair of

clusters where local scheduling ensures serializability. So if these conditions hold we are guaranteed one-copy serializability.

To show that Algorithm 3.1 and Algorithm 3.4 guarantees one-copy serializability, show that it satisfies the 3 conditions above assume, by contradiction, that H is not one-copy serializable. Then, one of the 3 conditions must not be valid. Conditions 2 and 3 are guaranteed by the architecture. This leaves condition 1 as the only possible violation. Concurrent writes on the same data item or anti-dependent reads (transaction reads where a conflicting transaction has opposite read/write operations) must not be sent to the same cluster. There are five potential scenarios for this to happen. The five scenarios are:

Read Set/Write Set overlap – The transaction T_i, containing the read set, will be sent to any cluster containing the latest committed version of the elements in the transactions, effectively scheduling the transaction T_i before transaction T_j ($T_i <_H T_j$)

Write Set/Read Set overlap – The transaction T_j, containing the read set, will be sent to any cluster containing the latest committed version of the elements in the transactions, effectively scheduling the transaction T_j before transaction T_i ($T_j <_H T_i$)

Write Set/Write Set overlap (write dependency) – If the conflicting operation is on the same data element then both transactions (T_i, T_j) will be sent to the same cluster. The database management system guarantees serializable execution at that cluster, and, therefore, one-copy serializability.

Write Set/Write Set overlap (anti-dependency) – In the case where T_i reads an element written by T_j and T_i writes an element read by T_j then the requests will be sent to the same cluster or queued for processing after one of the two transactions complete. The

database management system guarantees serializable execution at that cluster, and, therefore, one-copy serializability.

Read Set/Read Set overlaps – If both transactions (T_i, T_j) only contain read operations then each will be sent to a cluster that has the latest version of the data elements in the set. There is no conflict. □

Algorithm 3.4 Dispatcher Service Request Algorithm -Read Only

INPUT: requestedObjects = {O_1,…, O_k}, where each O_i is a pair (O.id, O.action); O_i.id is the object identifier, O_i.action is the requested action but is always READ.
OUTPUT: buddyList is a single cluster to perform the transaction.
TABLES USED: CL = Table table, OV =object version table, CO = cluster object table

```
buddyList = {}
available = all custer ids in CL
foreach O.id in requestObjects
    * find latest version of an object
    if NOT O.id in OV
        insert o.id into OV
        OV.complete=1,OV.inprogress=1
    set v.complete = OV.complete, v.inprogress = OV.inprogress
    * eliminate unqualified clusters from potential buddies
    foreach CO.cluster, CO.version in CO
        If co.version > V.complete
            available.remove(co.cluster)
* pick a buddy
foreach cl.cluster in CL
    if available(cl.cluster)
        buddyList.add(cl.cluster)
if buddyList.count() > 0
    let b1 denote cluster in buddylist
    send requestObjects to b1
else
    enqueue(requestObjects)
```

3.8 Analysis of the Buddy System

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

Algorithm 3.5 Primary Buddy Service

INPUT: requestedObjects =(Request containg objects to be read and written).
OUTPUT: dataset (data requested in read operations), objectVersions

```
Initialize writelist to an empty list
For each object, action in requestObjects
      Update the latest completed version
      If action == WRITE
            Add object to writelist
            Lock object
            Log write operation
            Write undo log for write operation
      Else if action == READ
            Add data to dataset

If there are items in writelist
      Send writelist to secondary buddy
      If secondary buddy committed properly
            For each object in requestObjects
                  Complete write on object
                  Release lock on object
      Else if secondary buddy aborted
            For each object in requestObjects
                  Undo write on object
                  Release lock on object
Send response to client
Send version update to dispatcher
```

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 a 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.

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 be-cause 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 up-date 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 3.1 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

Table 3.5 Windows of Vulnerability

	InActive	During Transaction	After Transaction Committed
Web Server housing Web Service of primary buddy	No issue because of redundancy	2PC will roll back trans, Client will get error from primary	No issue because of redundancy
Web Server housing Web Service of secondary buddy	No issue because of redundancy	2PC will roll back trans, Client will get error from primary	No issue because of redundancy
Database Server	No issue. Dispatcher needs to be notified by web service that it is unavailable	2PC will roll back trans, Client will get error from primary	No issue because of redundancy
Dispatcher	New dispatcher, Each clusters will update dispatcher versions	After dispatch to primary no issue because primary responds directly to client	New dispatcher, Each clusters will update dispatcher versions

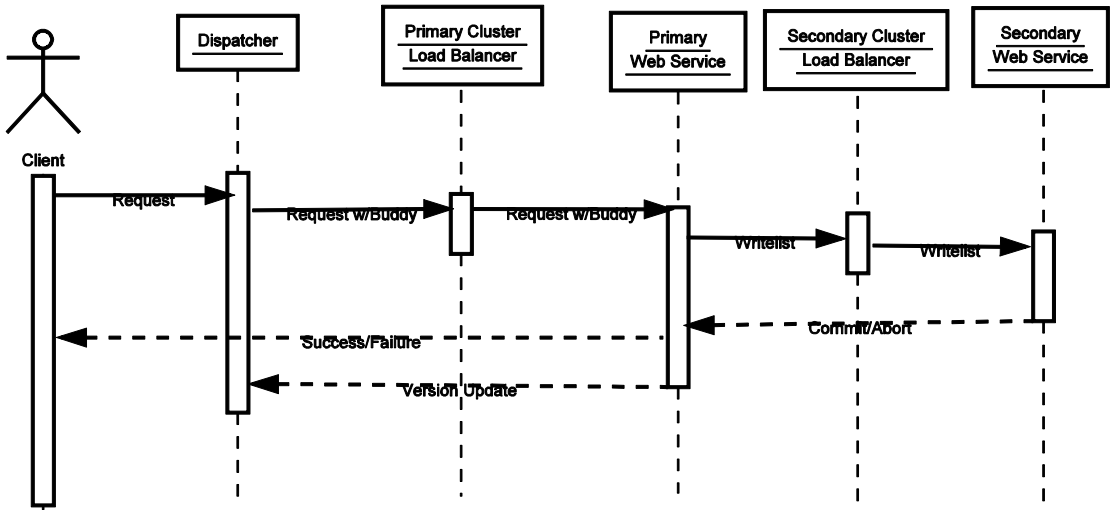

Figure 3.1 Buddy System Workflow

the original transaction so a hardware error on one replica will not result in the loss of update.

Table 3.5 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 trans-action started, during execution, and after the trans-action committed but before the update is propagated

3.9 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 cannot fulfill the request with the current state of the clusters. Figure 3.2 Implementation Data 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 replication. Figure 3.2 shows that once the dataset size grew to 10,000 items the performance of the Buddy algorithm matches the performance of lazy replication, while in-creasing 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

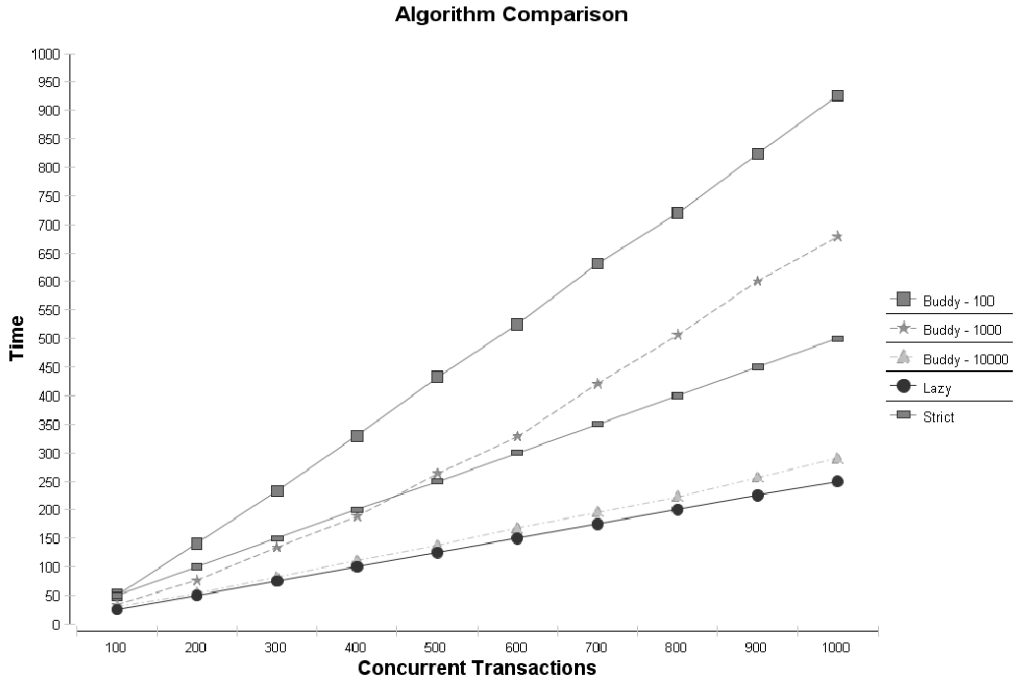

Figure 3.2 Implementation Data

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.

Chapter 4

Consistency, Availability & Durability Guarantees with Anonymous Resources

4.1. Anonymous Resource Consumption

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 a large number of tickets being purchased, we will discover the three types of consumption patterns Julian Jang et al [8] identified:

- 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.

Algorithm 4.1 SQL Implementation with One Record per Item

```
/* Table Creation */
Create table items (
       Id int identity,
       Item varchar(20),
       Status char(1)
)

/* Inventory Population */
Declare @id int
Set @id = 1
While @id <= 10000
Begin
       Insert into items (item, status)
              Values (‘Opening Night’, ‘A’)
       SET @id = @id + 1
End

/* Consumption Code */
Begin transaction
Select top 1 @myid = id from items
       Where status = ‘A’ and item = ‘Opening Night’
/* Item will be held in basket until transaction completes */
Update items set status = ‘S’ where id = @myid
Commit transaction
```

Figure 3.2 shows how the original Buddy System was compared against the lazy and eager replica update protocols. 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 cluster system with strict replication. Figure 3.2 shows that once the dataset size grew to 10,000 items, and enough clusters were made available, the performance of the Buddy algorithm matches, or exceeds, the performance of lazy-replication. This increase in availability came with an increased durability and consistency.

Anonymous Resource Consumption

The severe performance penalty observed with small datasets is the result of the enqueue process and the overhead of selecting buddies. To reduce this penalty the system needs to be able to guarantee that resource capacity is enforced in a way that can distribute simultaneous writes to different systems. Relational database programmers have had a problem with anonymous resource consumption for similar architectural issues. The locking mechanism in relational database systems behaves like a binary semaphore where only one transaction can get access to a record at a time. The resource consumption problem requires a constraint that can behave like a counting semaphore where a fixed number of concurrent processes can access a resource at a time. To solve this problem in relational systems the capacity updates need to be converted from an

Algorithm 4.2 SQL Implementation with One Record per Attribute

```
/* Table Creation */
Create table tickets (
       Id int identity,
       Item varchar(20),
       Int avail
)

/* Inventory Population */
Insert into items (item, avail)
       Values ('Opening Night',35000)

/* Consumption Code */
Begin transaction
/* Item will be held in basket until transaction completes */
Update items set avail =avail - 1 where id = @myid
Commit transaction
```

update activity with exclusive locks to a write operation. An outside process is required to ensure that the number of writes does not exceed capacity. This conversion would sacrifice the RDBMS ACID guarantees and force the developers to implement their own

set of guarantees.

Relational DBMS Implementation

The problem of anonymous resource consumption is a problem that has driven many system designers away from using a RDBMS because of the way resource contention is handled in traditional database system. The locking mechanism of RDBMS is designed to ensure serializability by isolating rows between concurrent transactions. Unfortunately this mechanism does not allow for a standard solution to the anonymous resource consumption problem. Algorithm and Algorithm show attempts to implement anonymous resource consumption in a Microsoft SQL database. Algorithm 4.1 attempts to model the resource in one record per item/attribute combination. Unfortunately only one concurrent transaction would gain access to the record. The other transactions are forced to wait on the lock until completion. Algorithm 4.2 attempts to model the problem by prepopulating a table with one row per record but unfortunately the locking mechanism again will block concurrent readers.

Capacity Constraints

To solve the outstanding issue of traditional relational databases and our Buddy System we introduce a new constraint. Allowing the dispatcher to keep a capacity value for each resource allows the algorithm to treat updates to an item as separate writes. The original dispatcher Algorithm 3.1distinguished between writes and updates by the data item version in the versions table. If an in-progress version was found it was considered an update otherwise it was considered a write. Our new dispatcher algorithm (Algorithm , checks capacity, and if there is available capacity converts the update to a write by using an initialization version number instead of the actual version.

Dispatcher Data Structures

The dispatcher will maintain the three original data structures (Table 3.1, Table 3.3, and Table 3.) from the Buddy System for processing the algorithms along with a new structure (Table 3.):

1. Cluster List - contains the names of the clusters and their IP addresses.
2. Objects Version Table -contains the name of the data items and their version numbers, corresponding to the completed and in-progress transactions.
3. Cluster Object Table - contains the cluster names, stored objects, and the version number of the objects at that cluster.
4. Object Capacity Table - containing the name of the data items and their capacities

Table 4.1 Example Object Capacity Table

Object	Capacity
B	2500
C	4500

4.2 Analysis of the Buddy System on Resource Consumption

Figure 4.1 shows how the new Buddy System algorithm compared against the lazy and eager replica update protocols. The new algorithm is able to easily outperform lazy-replication on all types of resource consumption using the new capacity constraints.

Using the Buddy System on our earlier example transaction would improve the availability of the TRS by allowing more clusters to participate in the transaction through the use of different masters for each seating location. The TRS would also have a guarantee of consistency and durability.

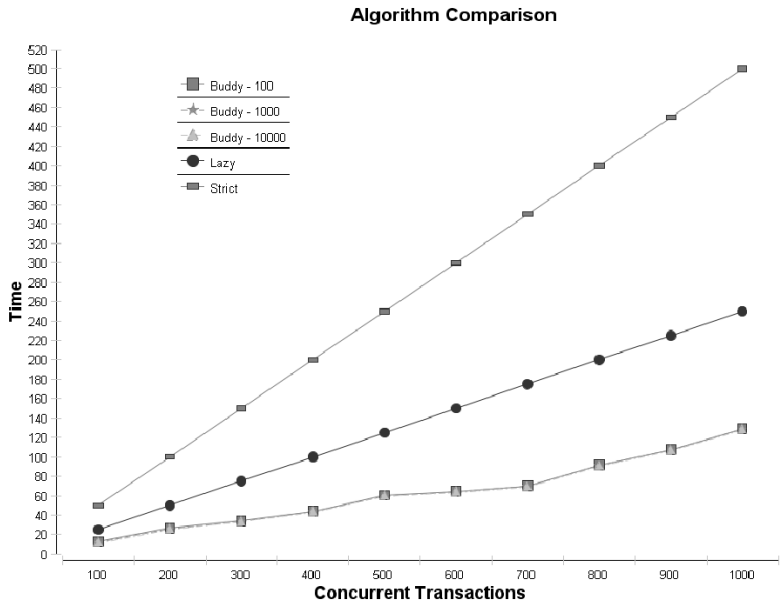

Figure 4.1 Implementation data with Capacity Constraint

Algorithm 4.3 Dispatcher Service Request Algorithm w/Capacity Constraint

INPUT: requestedObjects = {O_1,…, O_k}, where each O_i is a pair (O.id, O.action); O_i.id is the object identifier, O_i.action is the requested action.
OUTPUT: buddyList is a pair (B_1,B_2) of clusters to participate in the transaction.
TABLES USED: CL = cluster list table, OV =object version table, CO = cluster object table, OC = object capacity table

```
       buddyList = {}
       available = all custer ids in CL
       foreach O.id, O.action in requestObjects
         if  OC.availavility for O.id > 0
           * find latest version of an object
           if NOT O.id in OV
               insert o.id into OV
               OV.complete=1,OV.inprogress=1
           set v.complete = OV.complete, v.inprogress = OV.inprogress \
             where ov.object = o.id
           * eliminate unqualified clusters from potential buddies
           foreach co.cluster, CO.object, CO.version in CO
               If co.version > V.complete && O.action==READ
                   available.remove(co.cluster)
               elseif co.version < OV.inprogress && O.action==WRITE
                   available.remove(co.cluster)
               elseif O.action==WRITE &&
antidependency(requestobjects,co.cluster)
                   available.remove(co.cluster)
         else
           return no availability error
       * pick a pair of clusters
       foreach cl.cluster in CL
           if available(cl.cluster) and buddyList.count() < 2
               buddyList.add(cl.cluster)
       if buddyList.count() > 1
           let b1,b2 denote two clusters in buddylist
           * update version information for write object
           foreach O.id, O.action in requestObjects
                    if O.action==WRITE
                 increment OV.inprogress for ov.object = o.id
                 increment cO.version for cluster = b1 & co.object = o.id
                 increment cO.version for cluster = b2 & co.object = o.id
                 decrement OC.availability for O.id
           send buddyList,requestObjects to b1
       else
```

4.3 Conclusion

In this section we propose an extension to the buddy system to handle anonymous and attribute based resources. Our solution is based on a new constraint (Capacity

Constraint) that is enforced by the dispatcher. The constraint behaves as a counting semaphore where a limited capacity of concurrent transactions can gain access to the resource simultaneously.

This constraint allows distribution of the concurrent activity to multiple clusters increasing the availability of the system. Each individual transaction is applied to a pair of clusters synchronously allowing enforcement of consistency guarantees and durability.

The limitation of our work is that the element types need to be identified as anonymous or attribute based and the system cannot discover this from the semantics of the transaction. Our ongoing work extends our solution to incorporate semantic analysis of web service transactions to allow automatic

Chapter 5

Consistency, Availability & Durability Guarantees with Coarse Grained Web Services

In our earlier work, web services were fine grained CRUD services similar to a database SQL interface. Each request could contain several objects that would be updated, but the semantics of each object updated were available to the dispatcher in the request. These semantics are available because there is a limited set of operations and the detail level is atomic. Coarse grained web services are essentially distributed functions where the only information the dispatcher has at runtime is the input and output parameters of the web service. For the dispatcher to schedule the coarse grained web services properly it needs to map the coarse grained service to a limited set of operations on the atomic data item level.

5.1 Example Transaction

Consider a Ticket Reservation System (TRS). TRS uses web services to provide a variety of functionalities to the clients. For example, clients may want to select a specific seat for a popular concert in the ticket reservation. Figure 5.1 shows an implementation of this functionality.

Upon receiving a client request, the web application needs to communicate with a

set of web services to gather the data required to render the current seating map and allow the limited resource (the seats) to be consumed. The seating map needs to convey several pieces of information to the user, including:

- Visual representation of sold and available seats
- Pricing for the current user
- Performance details.

After the user has selected a set of seats they would like to purchase a web service is called to consume those seats and they will no longer be available for other users to consume. The following web services are used in Figure 5.1:

- GetSession – This web service will retrieve session state based on a unique session id.
- LoginAnonymous – This web services will login a user so they retrieve credentials for pricing and seating location availability. If the session does not have a logged in credential it will give the user the "anonymous" credentials.
- GetZones – This web service retrieves the zone information for the space where the event will take place. This information is used to allow a user to navigate

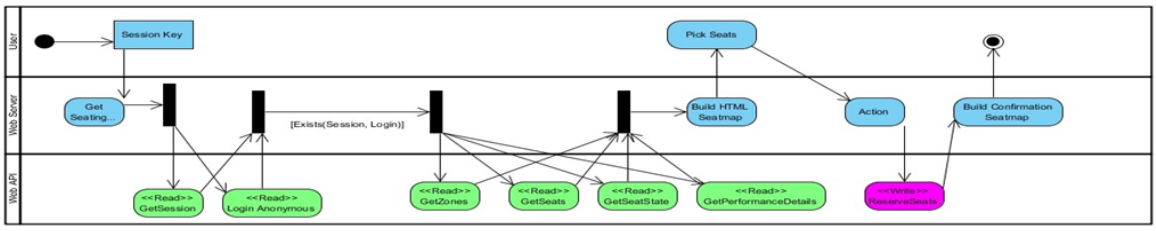

Figure 5.1 Activity Diagram for Self Service Seat Selection

between zone information. This information does not typically change after a

ticketed evented has been setup.

- GetSeats – This web service retrieves seating location for the current or default zone. Seating information is composed of a set of seats that have attributes for section, row and seat numbers. This information does not typically change after a performance has been setup.
- GetSeatState – This web service retrieves state information for all the seats in the zone. This information changes when any seat is consumed by another user.
- GetPerformanceDetails – This web service retrieves program details for the performance that is being sold. This information does not typically change after a performance has been setup.
- ReserveSeats – This web service consumes the limited resource and changes the state of the previous GetSeatState web service.

Unfortunately, it is not clear how many simultaneous requests will come from clients at a given time. During normal operations an organization may only have a few concurrent requests. When a popular event goes on sale, this number could rise to tens of thousands of requests. If several events go on sale at the same time then the services could need to handle hundreds of thousand, simultaneous requests.

To handle this unknown load at deployment time, implementers have resorted giving up consistency by manually partitioning the data across different servers. For example each event could have its own ReserveSeats server so that the load of many currently events would not impact performance. This solution does not scale well as new

hardware would be needed to handle higher levels of event concurrency.

5.2 UML Semantics

Additional semantics for the coarse grained web services can be acquired from integration of the matching UML Activity and Class diagrams. UML provides an extensibility mechanism that allows a designer to add new semantics to a model. A stereotype is one of three types of extensibility mechanisms in the UML that allows a designer to extend the vocabulary of UML in order to represent new model elements [29]. Traditionally these semantics were consumed by the programmer manually and translated into the program code in a hard coded fashion.

Read vs. Write Semantics

Figure 5.1 is an activity diagram with two stereotypes used to model web services that are read-only and web services that write and update data as part of the process. In the example the *ReserveSeats* services modifies data as part of its process and all other services just read data as part of their process.

Element Unique Identifier Semantics

Each Web Service in the Activity diagram has a matching UML Class diagram that shows the structure of the input and output messages. This same data can be retrieved from the WSDL [30] message types, though there is no natural link between the activity diagram and the WSDL services. So we ignore the WSDL at this time and use the data from the XMI file. Two of the matching class diagrams are shown in Figure 5.3 and Figure 5.4.

```
<wsdl:message name="GetSeatStateRequest">
    <wsdl:part name="event" type="xs:ID"/>
    <wsdl:part name="zone" type="xs:ID"/>
</wsdl:message>
<wsdl:message name="GetSeatStateResponse">
    <wsdl:part name="event" type="xs:ID"/>
    <wsdl:part name="zone" type="xs:ID"/>
    <wsdl:part name="seats" type="tns:Seats"/>
</wsdl:message>
<wsdl:message name="WriteReserveSeatsRequest">
    <wsdl:part name="event" type="xs:ID"/>
    <wsdl:part name="seats" type="tns:Seats"/>
</wsdl:message>
<wsdl:message name="WriteReserveSeatsResponse">
    <wsdl:part name="status" type="xs:Integer"/>
</wsdl:message>
<wsdl:portType name="TicketingSoapPort">
    <wsdl:operation name="GetSeatState">
        <wsdl:input message="tns:GetSeatStateRequest"/>
        <wsdl:output message="tns:GetSeatStateResponse"/>
    </wsdl:operation>
    <wsdl:operation name="WriteReserveSeats">
        <wsdl:input message="tns:WriteReserveSeatsRequest"/>
        <wsdl:output message="tns:WriteReserveSeatsResponse"/>
    </wsdl:operation>
</wsdl:portType>
```

Figure 5.2 WSDL for GetSeatState & WriteReserveSeats Web Service

An attribute level stereotype <<PK>> is used to represent the unique identifier combination of the attributes. For example in the GetSeatStatus web service (Figure 5.3), an individual seats status can be uniquely identified in the response by the attribute set {Performance, Zone, SeatId}. The *ReserveSeatsRequest* (Figure 5.4) has an input

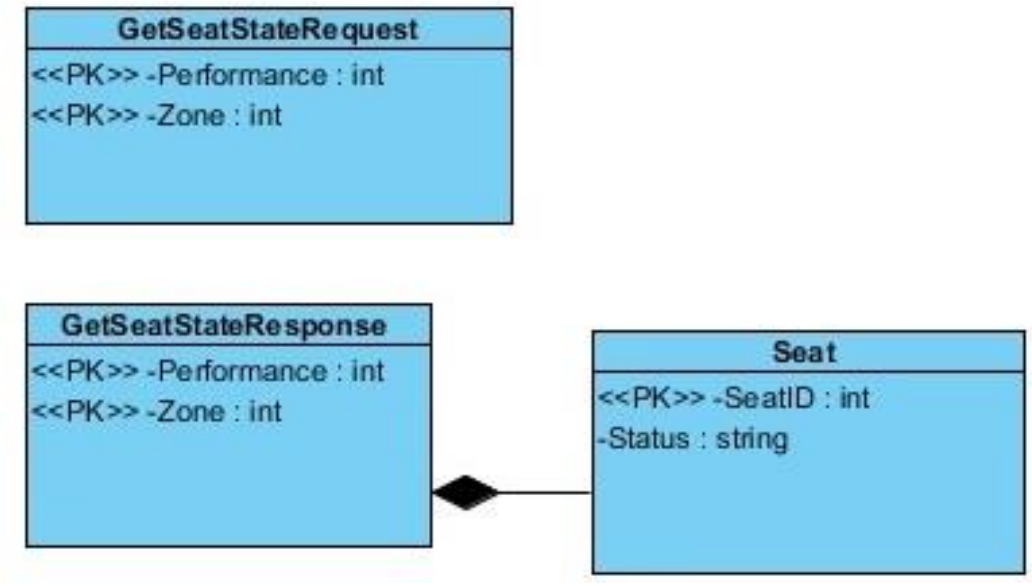

Figure 5.3 UML Class Diagram for GetSeatStatus Service

message that is a composition of seats with the same unique identified of the attribute set {Performance, Zone, SeatId}.

Parallel Scheduling Semantics

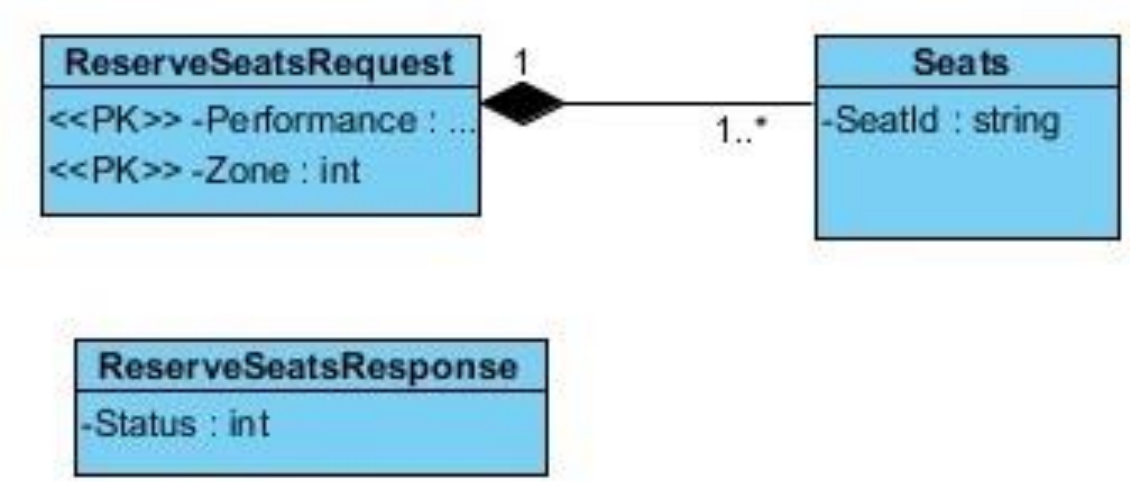

Figure 5.4 UML Class Diagram for Reserve Seat Service

The UML Activity diagram (Figure) also provides us with the semantics required to know which services can be called in parallel. The getSession and loginAnonymous

services are required to be called before the remaining services as they change required state used by the later service. Figure 5.5 shows a fragment of the XMI file used for extracting the parallel scheduling semantics. The file is organized in XML and the web services form a directed acyclic graph (DAG). The fork, join and each web service are represented as *ownerMember* XML elements with a unique identifier that can be traced to the graph edges. Each graph edge has a target for every path. Each path leads to the join node where the dispatcher will wait for all paths to complete. A breadth first search algorithm that uses parallel traversal is used to follow all the parallel paths in the fork.

5.3 Buddy System Changes to Handle Coarse Grained Services

The original buddy system received a single packet of the fine grained operations in the transaction. In normal web service operations, a client application is responsible for calling each operation individually. The *Dispatcher Service Request Algorithm (*Algorithm 3.1*)* needs visibility into all operations of the transaction at a single point in time. To facilitate this visibility, the client sends all requests as a batch and the dispatcher sequences the calls based on the semantics from the XMI data.

```
1   <ownedMember kind="fork" xmi:id="kd_vADyFS_ha7gch" xmi:type="uml:Pseudostate"></ownedMember>
2   <ownedMember name="GetSession" xmi:id="uFivADyFS_ha7gah" xmi:type="uml:CallBehaviorAction"></ownedMember>
3   <ownedMember name="Login Anonymous" xmi:id="fvqvADyFS_ha7gav" xmi:type="uml:CallBehaviorAction"></ownedMember>
4   <ownedMember kind="join" xmi:id="6OSfADyFS_ha7gde" xmi:type="uml:Pseudostate"></ownedMember>
5
6   <edge isLeaf="false" source="kd_vADyFS_ha7gch" target="uFivADyFS_ha7gah" xmi:id="DCEfADyFS_ha7gc9" xmi:type="uml:ControlFlow"></edge>
7   <edge isLeaf="false" source="kd_vADyFS_ha7gch" target="fvqvADyFS_ha7gav" xmi:id="SYkfADyFS_ha7gdl" xmi:type="uml:ControlFlow"></edge>
8   <edge isLeaf="false" source="6OSfADyFS_ha7gde" target="gHBfADyFS_ha7geg" xmi:id="gltpaDyFS_ha7gYy" xmi:type="uml:ControlFlow">
9       <guard body="Exists(Session, Login)" xmi:id="gltpaDyFS_ha7gYy_guard" xmi:type="uml:OpaqueExpression"/>
10  </edge>
```

Figure 5.5 XMI Snippet

Buddy Selection Algorithm

Algorithm 5.1 is an updated buddy selection algorithm to select the appropriate pair of web services to perform the transaction. The algorithm will iterate over the forks in the activity diagram to service the items that can be done in parallel. A fork is a point in the activity diagram where the flow is split and can run in parallel. Within each fork the algorithm will iterate over each web service and flatten the class diagram to get one instance per aggregation. Each instance is then iterated over and its current version is checked in the version tables to determine its current version. The algorithm then determines eligible buddies that can service the batch of web service requests and randomly chooses two to do so.

Theorem 1: The Buddy Algorithm (Algorithm 3.1) guarantees 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, . . . , n} is made up of a set of web services $WS_{i.}$ Each web service is made up with a setup of operations that are either read R_i (A) or 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_i and T_j be the two transactions responsible for the cycle. We show that the serialization graph cannot contain a cycle for the three potential scenarios. The three scenarios are: Read Set/Write Set overlap, Write Set/Write Set overlap, and Read Set/Read Set overlap.

- Read Set/Write Set overlap – in this scenario one transaction reads items that overlap with items being updated in another transaction. If T_i is the transaction reading items and T_j is the transaction writing items then the dispatcher will always schedule T_i before T_j by serving T_i with the previous version of the data items. This ensures that this scenario cannot contain a cycle.
- Write Set/Write Set overlap. If T_i is a transaction updating the same items as transaction T_j then both transactions will be sent to the same cluster. Since the cluster is guaranteeing serializability then this scenario cannot contain a cycle.
- Read Set/Read overlaps. Since both transaction T_i and transaction T_j are reading the same data items then they will scheduled in any order using the latest completed using version of the data items. This ensures that this scenario cannot contain a cycle.

5.4 Implementation

We used Visual Paradigm™ for the UML diagrams and exported the diagrams to XMI using the built in export functionality. On startup the dispatcher created a precedence graph based on the semantics of the XMI data. We ran the results against a concurrent load of users and measured the time till completion. Figure shows the results where we compare three different modes of operation against the time it takes for blocks of users to complete the requests. The users were tested in blocks of 50 and tested against three different architectures, where each web service was called sequentially using no UML semantic data, in parallel using the semantic data from the UML Activity diagram, and distributed using the semantic data from both the activity and the class

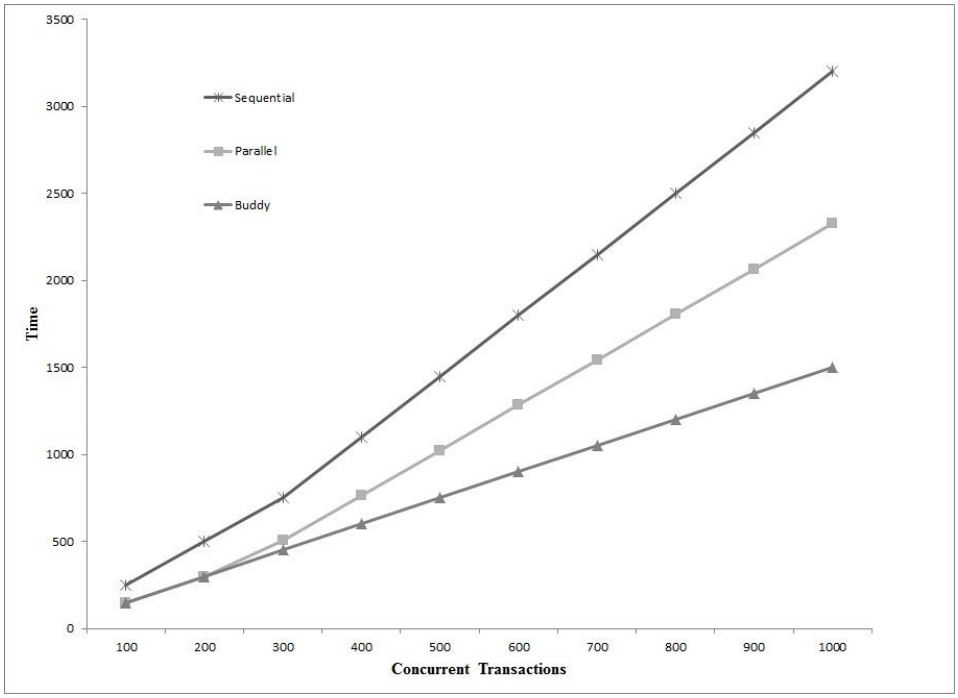

Figure 5.6 Availability Improvements under Coarse-Grained Scheduling

diagrams.

Transaction Details

In the example transaction the web application sends the set of web service requests {GetSession, LoginAnonymous, GetZones, GetSeats, GetSeatState, GetPerformanceDetails} to the dispatcher. In sequential mode the services would be scheduled in a sequence on the same web service box.

Using the semantic data from the UML Activity diagram Figure 5.1 Activity Diagram for Self Service Seat Selection, the dispatcher will determine that a sequence of two subsets is required:

1. {GetSession, LoginAnonymous}
2. {GetZones, GetSeats, GetSeatState, GetPerformanceDetails}

Using these semantics, the services in the same subsets can be scheduled in parallel for an improvement in performance over the original sequential schedule.

Algorithm 5.1 allowed the dispatcher to take this a step further by looping through fine grained objects read or written by the individual web service. This information is gained from two places:

1. The action of read or write comes from the stereotype in the UML activity diagram (Figure 5.1).
2. The individual items from the UML class diagrams represent the fined grained items.

The <<PK>> stereotype in the UML class diagrams allows us to uniquely identify each tuple in the fined grained operations. One these semantics have been identified the

Algorithm 5.1 Coarse Grained Buddy Selection

INPUT: activity (XMI from activity diagram & class diagram), clusterObjects, objectVersions
OUTPUT: buddyList (Pair of buddies or empty list if no pairavailable), clusterObjects, objectVersions

```
Add all clusters to available list
foreach fork in activity
    foreach ws in fork
        foreach O in ws //iterate over aggregate
            If O in objectVersions
                Getcompleted OV.c, OV.i from objectVersions
            else
                OV.c=1,OV.i=1
            foreach CO.c, CO.v in clusterObjects
                If CO.v > OV.c && O.a==READ
                    available.remove(CO)
                elseif CO.v < OV.i && WS.a==WRITE
                    available.remove(CO)
    foreach CO.c in available
        if count(buddyList)<2
            add Co.c to buddyList
    if count(buddyList)>1
        foreach B in buddyList
            foreach ws in fork
                foreach O in ws //iterate over aggregate
                    getinprogress OV.i from objectVersions for B
                    if WS.action==WRITE
                        increment OV.i
        send buddyList,requestObjects to B1
    else
        enqueue(requestObjects)
```

original buddy algorithm (Algorithm 3.1) can be implemented on the coarse grained services.

Figure 5.6 shows the performance results of the implementation where the additional semantics gained from the UML data allows the buddy system to almost double the availability of the original sequential schedule.

WSDL Parameter Partitioning

If data is constantly being updated by one service and retrieved by another service then the *buddy system* will partition the data on a natural level. For example in Figure the GetSeatState service has two input parameters (event, zone) and in Figure the WriteReserveSeats service has two input parameters (event, collection of seats). If a large stadium were selling an extremely popular concert without the buddy system they may want to partition the load based on the zone of the stadium. Unfortunately, the web services would need to be consistent in the parameter data to enable a dispatcher to distribute the requests based on the data.

The buddy system does this partitioning as part of the process of finding a pair of buddies. If a current transaction is progress that affects a data tuple, for example: zone availability), then all requests that use this tuple will be sent to the same cluster.

5.5 Conclusion

In this chapter we propose an extension to the buddy system to handle coarse grained web services. Our solution is based on extending UML with stereotypes to embed CRUD, Parallel and data element semantics into the model. The dispatcher can then extract the semantics from the model and distribute the requests to clusters as it did with the fine grained web service. Each individual transaction is applied to a pair of clusters synchronously allowing enforcement of consistency guarantees and durability. The

limitation of our work is that the dispatcher needs to understand all semantics at startup time and cannot discover new service semantics as they evolve.

Chapter 6

Web Service Constraint Optimization

A limitation of our earlier work on the *Buddy System* is that integrity constraints that required different classes in the calculation could not be guaranteed. For example, if an address required a valid owner in the person class. These integrity constraints could not be enforced because data mutation could happen on different clusters simultaneously. In this section we address that limitation. We provide an approach that pulls the UML constraints expressed in OCL from the design model and incrementally maintains the data that allow the dispatcher to enforce the constraint, and once successful it is free to distribute requests to several clusters concurrently.

Our solution provides several advantages not addressed in traditional distributed database replica update protocols. First, our approach provides the scalability required by modern n-tier applications, such as web farms, and is suitable for the architectures and technologies implementing these applications in cloud computing environments. 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 while also ensuring the integrity of the data.

6.1 Example Transaction

The Washington, DC transit system uses a smart card (*SmarTrip*) as a payment system. The card maintains the value on it resulting from passenger activities (boarding, disembarking, adding value to card). Each activity is recorded in a centralized activity log that is linked to the smart card involved in the activity on a central system. Some activities originate on the card (boarding, disembarking) and others originate in the central system (adding value). Figure 6.1 shows a sample UML class diagram for this example. This activity log relies on a sequence number to identify the ordering of activities. An incorrect sequence number can cause the system to not allow a card to receive added value despite a transaction occurring on the centralized system.

Corruption of the sequence numbers makes the sequence number data integrity issue a potential large scale denial of service issue. Imagine thousand passengers unable to gain access to the public transportation system. Often this type of constraint is not enforced because of the expense of runtime calculation. A simple example SQL check constraint that would enforce the constraint is shown in Figure 6.2. Unfortunately most commercial SQL implementations do not allow sub-queries in the check constraint. So this constraint becomes impossible to enforce.

6.2 Integrity Constraints

Codd [31] defined five types of integrity constraints to guarantee the consistency in relational databases:

- Entity - Every entity needs a primary key that will uniquely identify each tuple in the entity.
- Domain - The model can define domains to represent valid values stored in entity attributes. This is done through the use of data types.
- Column - Each column of the entity can specify a smaller set then the complete range for the data type. This is normally done through the ENUM feature of the database management system.

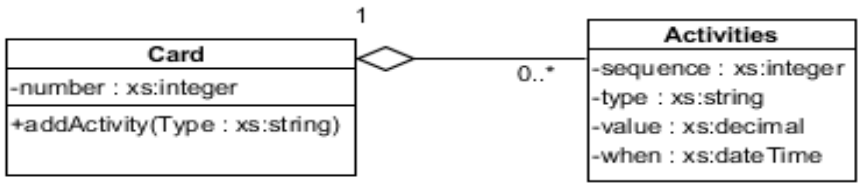

Figure 6.1 UML Class diagram

- Foreign Key - The DBMS can enforce that a parent related record exists in the database or the child relationship cannot be added.
- User defined - A user defined integrity constraint can express any user defined logic checks. This is normally done through the check constraint syntax of the DBMS. DBMS languages often allow for the definition of both column level check constraints and tuple level check constrains. Tuple level check constraints can enforce integrity using any attributes of the tuple in comparisons including sub-queries.

```
CREATE TABLE activity (
  smartripId int,
  sequence int,
  when datetime,
  activity char(1),
  amount money,
  CHECK ((select max(sequence) from activity a2
    where a2.smartripId = smartripId) < sequence)
)
```

Figure 6.2 SQL Constraint

These five types of constraints can be grouped into three categories: Entity, Domain and hierarchical. The Domain and Column constraints are both used to limit the domain of an attribute. Foreign key constraints are also a form of domain constraint. They allow a refinement of the domain of a column to limit to existing parent objects. User defined constraints are primarily used to express constraints on associations between relations that are more complex. These associates are typically hierarchical and enforce an aggregate or require an iteration across children records in an association.

6.3 Object Constraint Language

Object Constraint Language (OCL) is part of the official OMG standard for UML. An OCL constraint formulates restrictions for the semantics of the UML specification.

```
context SmarTrip::addActivity(inSequence: Integer)
    self.activities@pre->forAll( a|a.sequence < inSequence )
```

Figure 6.3 OCL Example

An OCL constraint is a guarantee that is always true if the data is consistent. A constraint is expressed on the level of classes, but it is applied on the level of objects. OCL has

operations to observe the system state but does not contain any operations to change the system state.

Kinds of OCL Constraints

- Invariants. An invariant is a condition which always holds. In a relational database management system RDBMs an invariant maps to an assertion because the assertion will be enforced by the RDBMS on every action to the system.
- Pre-conditions. A pre-condition is a condition that is guaranteed to hold before an activity is executed. In RDBMs a check constraint would be used to enforce the constraint as it would only check on the insertion and updating of data in the specific table.
- Post-conditions. A post-condition is a condition that is guaranteed to hold after an activity is executed. In a RDBMS the post condition would need to be implemented in a Trigger to force the evaluation to after the action.

OCL can navigate an association and provides functions that aggregate over collections. We considered predicate logic as the specification language of the constraints. Unfortunately it lacks the ability to express aggregate calculations. We also considered relational algebra for the specification of the constraints but it lacks the support is design tools. OCL is integrated into many UML design environments and fits well in a model driven architecture (MDA). Figure 6.3 shows sample OCL to enforce that the sequence number on currently inserted activity is greater than all others sequence numbers for the same smarTrip card.

6.4 Hierarchical Constraints

Hierarchical constraints are expressions of data integrity that involve more than one tuple. The association can be between two classes of data or self-referential over one class of data. These constraints fall into two categories; aggregates and iterative. Aggregate constraints involve functional calculations that are calculated over all the records in the association relationship. Iterative constraints require iteration over the association to enforce the constraint. Iterative constraints fall into two categories; existential and universal quantification.

With aggregate constraints the functional aggregate calculation is often expensive to calculate at insertion time and is therefore ignored due to the expensive operations. In relational database systems this enforcement is done with a check constraint or a trigger. The former being less expensive as it is a declarative constraint. Unfortunately check constraints that can use sub-queries are often not supported in the relational system. Triggers are a more expensive solution for enforcement of the constraints as they are procedural and offer less opportunity for optimization. There are several common aggregate calculations used in constraints:

- **Maximum**

  Maximum aggregation constraints are used to ensure a new tuple has a value in relation to the current maximum. This relationship is often a greater than or less than comparison. Our example above with the sequence number is an example of a maximum aggregate association constraint.

- **Minimum**

Minimum aggregation constraints are used to ensure a new tuple has a value in relation to the current minimum. This relationship is often a greater than or less than comparison.

- **Sum**

  Sum aggregation constraints are used to ensure a new tuple's value does not surpass an upper bound. An example would a sales line item table that has a quantity field. You could use the sum of the quantity field to ensure the new tuple does not surpass and inventory quantity.

- **Count**

  Count aggregation constraints are used to ensure adding a new tuple does not surpass an upper bound on quantity. An example would the capacity constraint added to the Buddy System in our previous work [5]. Referential Integrity [31] is a specific form of a count based aggregate constraint. Normally the count is one for referential integrity to ensure the parent record exists.

6.5 Aggregate Constraint Materialization

The dispatcher materializes the constraints by keeping a copy in memory of the aggregate calculation. As new tuples arrive at the dispatcher the materialized aggregation is updated incrementally. If a transaction does not complete the dispatcher will decrement count aggregates or subtract sum aggregate to undo the operation. Non-completing transactions on minimum and maximum aggregates only update the materialized value if they are still the current value. Table 6.1 shows example data that is maintained by the dispatcher to materialize a constraint. The value and parent are stored

per object along with the quantity which is only used with aggregate operations such as average where the quantity of records in the hierarchy matter.

All post-condition constraints are converted to pre-condition constraints to allow a check dispatch time. The serialization and atomic guarantees by the clusters allow this conversion to take place to increase availability.

Table 6.1 Sample Constraint Materialization Data w/Aggregates

Object	***Constraint***	***Parent***	***Value***	***Quantity***
smarTrip	sequenceOrd	1000120	408	408

6.6 Iterative Constraint Materialization

Universal quantifications are expressed with a comparison against a scalar or an aggregate. In the case of the scalar comparison the dispatcher can apply the constraint on all incoming requests that insert or update the object. If the constraint does not hold we can reject the request. In the case of a universal quantification using a comparison against an aggregate we use the same materialization infrastructure from above.

Existential quantifications need to be verified on delete operations along with insert and update. There may be several records are available to satisfy the constraint. To materialize this constraint check the system maintains a tuple for each constraint that records the number of records that are available to satisfy the constraint. Insert and update operations will increment the quantity and delete operations will decrement the quantity. If the quantity is greater than zero then the operation succeeds. An example of the data maintained by the dispatcher is shown in Table 6.1.

6.7 Temporal Constraints

We have grouped the original Codd [31] constraint types into 3 categories: entity, domain and hierarchical. Domain constraints can be modeled in the UML with data types and enumerations. Entity integrity can be modeled with UML stereotypes representing the primary keys as we have done in our previous work [6]. Web services require an additional constraint type not handled in relational database systems. This constraint type models the state before and after the web service. There are two perspectives to consider around temporal constraints: client and server. Server temporal constraints guarantee the state of the server is consistent after the service is completed

Table 6.2 Sample Constraint Materialization Data

Object	***Constraint***	***Parent***	***Quantity***
smarTrip	paymentExists	1000120	3

based on the actions of the service. Client temporal constraints guarantee the state of the client after the service is completed. Ziemann and Gogolla [32] have worked to extend OCL to support syntax to specific additional changes to state over the life of an application from instantiation to termination. For this work we were able to stick with the out of the box OCL and use the @pre tags in post-condition constraints to guarantee that the effects of the web service change the state of the web server correctly. Client temporal constraints are useful in the example transaction above. The smart card needs to guarantee that the balance after the use (reduce) transactions is equal to the original balance minus the sum of all the removes.

To enforce both client and server side temporal constraints the client needs a mechanism to undo the transaction after the server has returned the service response. A two phase commit could be implemented from the client to the server to allow the client to roll back the server transaction in the case where the client constraint does not pass. Unfortunately this method would double the message count for every transaction and reduce the improvements in availability we have already achieved.

Using the method from our previous work mapping course grained services to fine grained services [6] we are able to auto generate compensators. The use of the compensator allows a single round trip message from the client to the server when the constraints pass on both client and server. When a client side constraint fails the compensator is invoked to “undo” the state change that was performed by the service on the server. Figure 5.1 shows an activity diagram with post conditions on both the server

and the client.

6.8 Empirical Results

We modeled a small urban transportation system with 100,000 users averaging 2 trips a day for 50 weeks a year. Each user is assumed to replenish his or her value once a week. The model was loaded into a Microsoft SQL Server 2008 server. We wrote a function with a single argument of the card id that returned the maximum sequence for that card id. SQL Server does not support sub-queries in check constraints but does support functions. The function was placed inside a constraint to enforce that new tuples

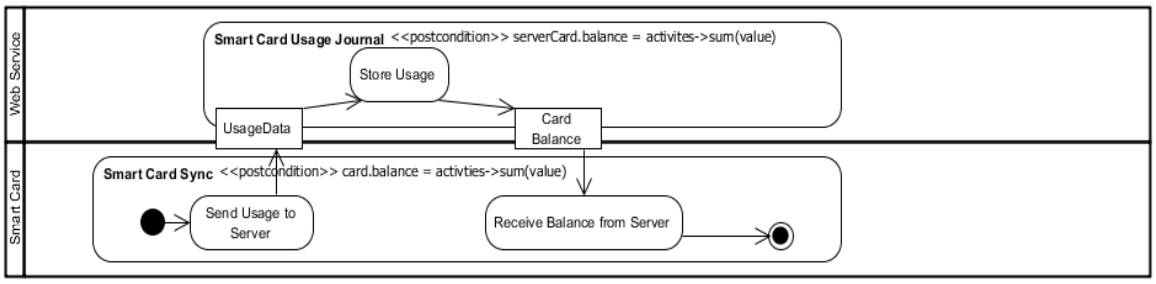

Figure 6.4 Service Activity Diagram

have a sequence greater the current maximum for that card.

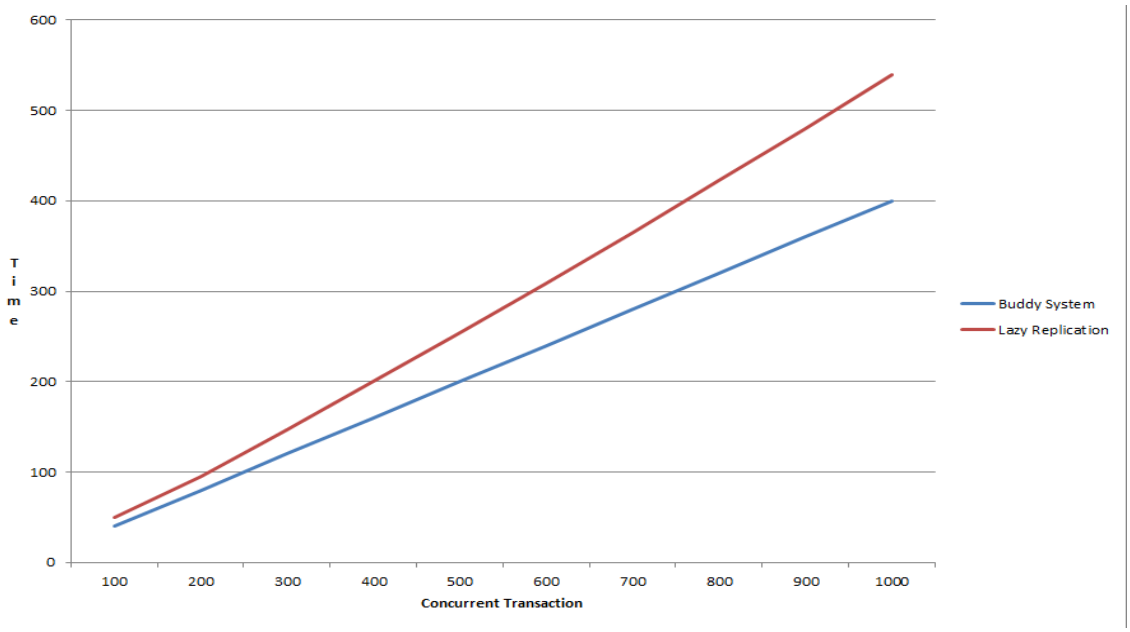

Figure 6.5 Empirical Results

We tested insert timings of loads of concurrent transactions in blocks of 100 with the constraint implemented in the SQL Server with lazy replication and with the *Buddy System* implementing the constraint with four clusters. Without the *Buddy System* the SQL Server implementation performed well as long as there was an index on the card id. This allowed the system to seek on the index to the subset of records for one customer. The database system did not use synchronization when performing the check constraint. This means that current consistency with lazy replication and the SQL implementation was not guaranteed. With the Buddy System higher availability was achieved by distributing the inserts to all four clusters while guaranteeing the consistency.

6.9 Conclusion

In this chapter we propose an extension to the buddy system to handle integrity

constraint guarantees. Our solution is based on extracting OCL design constraints from the UML models of the system. The dispatcher can then enforce these constraints using materialized aggregates. Each constraints aggregate value is updating incrementally as new tuples are inserted into the database. The dispatcher is then able to distribute the requests to any cluster after passing the constraint check. The limitation of our work is that we currently only support a subset of possible OCL notation for expressing the aggregate constraints.

Conclusion and Future Work

In our research we investigate the problem of providing consistency, availability and durability for Web Service transactions. We show that the popular lazy replica update propagation method is vulnerable to loss of transactional updates in the presence of hardware failures. We also show that strict replica update propagation method reduces availability beyond what is required for providing the necessary transactional guarantees. Our approach, called the "buddy" system, requires that updates are preserved synchronously in two replicas. 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.

Future research tasks in this area include:

- Application Partition Constraints – integrity constraints involving more than one application partition.

- Service CRUD Security – integrity constraints guaranteeing security in CRUD operations.

References

[1] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News,* vol. 33, pp. 51-59, 2002.

[2] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer,* vol. 45, pp. 37-42, 2012.

[3] A. Olmsted and C. Farkas, "The cost of increased transactional cor-rectness and durability in distributed databases," in *13th International Conference on Information Reuse and*, Los Vegas, NV, 2012.

[4] A. Olmsted and C. Farkas, " Buddy System: Available, Consistent, Durable Web Service Transactions," *Journal of Internet Technology and Secured Transactions,* vol. 3, 2013.

[5] A. Olmsted and C. Farkas, "High Volume Web Service Resource Consumption," in *Internet Technology and Secured Transactions, 2012. ICITST 2012*, London, UK, 2012.

[6] A. Olmsted and C. Farkas, "Coarse-Grained Web Service Availability, Consistency and Durability," in *IEEE International Conference on Web Services,* San Jose, CA, 2013.

[7] A. Olmsted and C. Farkas, "Web Service Constraint Optimization," in *Internet Technology and Secured Transactions, 2013. ICITST 2013,* London, UK, 2013.

[8] J. Jang, A. Fekete and P. Greenfield, "Delivering promises for web," in *Web Services, IEEE International Conference on*, 2007.

[9] M. Y. Lou and C. S. Yang, "Constructing zero-loss web services," *INFOCOM,* pp. 1781-1790, 2001.

[10] M. T. Ozsu and P. Valduriez, Principles of Distributed Database Systems, 3rd ed., Springer, 2011.

[11] Y. Lin, B. Kemme, M. Patino Martiınez and R. Jimenez-Peris, "Middleware based data replication providing snapshot isolation," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data, ser. SIGMOD '05*, New York, NY, 2005.

[12] H. Jung, H. Han, A. Fekete and U. Rhm, "Serializable snapshot isolation," *PVLDB,* pp. 783-794, 2011.

[13] Y. Breitbart and H. F. Korth, "Replication and consistency: being lazy helps sometimes," *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, ser. PODS '97,* pp. 173-184, 1997.

[14] K. Daudjee and K. Salem, "Lazy database replication with ordering," in *Data Engineering, International Conference on*, 2004.

[15] S. Jajodia and D. Mutchler, "A hybrid replica control algorithm combin-ing static and dynamic voting," *IEEE Transactions on Knowledge and Data Engineering,* vol. 1, pp. 459-469, 1989.

[16] D. Long, J. Carroll and K. Stewart, "Estimating the reliability of," *IEEE Transactions on,* vol. 38, pp. 1691-1702, 1989.

[17] L. Irun-Briz, F. Castro-Company, A. Garcia-Nevia, A. Calero-Monteagudo and F. D. Munoz-Escoi, "Lazy recovery in a hybrid database replication protocol," in *In Proc. of XII Jornadas de Concur-rencia y Sistemas Distribuidos*, 2005.

[18] A. Lakshman and P. Malik, "Cassandra: a decentralized structured," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35-40, 2010.

[19] "OASIS Standard, Web Services Business Activity (WS-BusinessActivity) 1.2," February 2009. [Online]. Available: http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os.doc. [Accessed 10 12 2012].

[20] "OASIS Standard, Web Services Business Process Execution Language (WS-BBPEL) 2.0," April 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. [Accessed 10 12 2012].

[21] P. Sauter and I. Melzer, "A Comparison of WS-BusinessActivity and BPEL4WS Long-Running Transaction," in *KIVS*, Kaiserslautern, Germany, 2005.

[22] H. Garcia-Molina and K. Salem, "Sagas," in *In Proceedings of the 1987 ACM SIGMOD international Conference on Management of Data*, San Francisco, California, 1987.

[23] A. Fekete, S. N. Goldrei and J. P. Asenjo, "Quantifying isolation anomalies," in *Proceedings of the VLDB Endowment*, 2009.

[24] H. E. Ramadan, I. Roy, M. Herlihy and E. Witchel, "Committing conflicting transactions n an STM," *SIGPLAN*, vol. 44, pp. 163-172, 2009.

[25] M. Cahill, U. Roehm and A. Fekete, "Serializable Isolation for Snapshot Databases," in *SIGMOD*, 2008.

[26] M. Schafer, P. Dolog and W. Nejdl, "An Environment for Flexible Advanced Compensations of Web Service Transactions," *ACM Transactions on the Web*, 2008.

[27] A. Fekete, P. Greenfield, D. Kuo and J. Jang, "Transactions in loosely coupled distributed systems," in *Proceedings of the 14th Australasian database conference*, Adelaide, Australia, 1003.

[28] S. Choi, H. Jang, H. Kim, J. Kim, S. M. Kim, J. Song and Y. J. Lee, "Maintaining consistency under isolation relaxation of Web services transactions," *Lecture Notes in Computer Science*, vol. 3806, pp. 245--257, 2005.

[29] Object Management Group, "Unified Modeling Language: Supersturcture," 05 02 2007. [Online]. Available: http://www.omg.org/spec/UML/2.1.1/. [Accessed 08 01 2013].

[30] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web service definition language (WSDL)," 2001. [Online]. Available: http://www.w3.org/TR/wsdl.

[31] E. F. Codd, The Relational Model for Database Management, Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[32] P. Ziemann and M. Gogolla, "Ocl extended with temporal logic," *Perspectives of System Informatics*, 2003.

[33] M. Aron, D. Sanders, P. Druschel and W. Zwaenepoel, "Scalable content-aware request distribution in cluster-based networks servers," in *Proceedings of the annual conference on USENIX Annual Technical Conference, ser. ATEC '00*, Berkeley, CA, USA, 2000.

[34] A. Olmsted and C. Farkas, "The cost of increased transactional correctness and durability in distributed databases," in *13th International Conference on Information Reuse and*, Los Vegas, NV, 2012.

[35] D. Long, J. Carroll and K. Stewart, "Estimating the reliability of regeneration-based replica control protocols," *IEEE Transactions on*, vol. 38, pp. 1691-1702, 1989.

[36] F. Heidenreich, C. Wende and B. Demuth, "A Framework for Generating Query Language Code," *Electronic Communications of the EASST*, 2007.

[37] B. Demuth, H. Hußmann and S. Loecher, "OCL as a Specification Language for Business Rules in Database Applications," in *The Unified Modeling Language. Modeling Languages, Concepts, and Tools.*, Springer, 2001, pp. 104-117.