# Available, Consistent, Durable Multimedia Workflow Transactions

Aspen Olmsted
*Department of Computer Science*
*College of Charleston*
*Charleston, USA*

## Abstract

*In this paper, we investigate the problem of providing consistency, availability, and durability for web service transactions that return multimedia. The multimedia returned by the example web services are a visual representation of the state of a set of data. We propose an approach that matches the availability of the popular lazy replica update propagation method while increasing durability and consistency. Our replica update propagation method is called the Buddy System, which requires that updates are preserved synchronously in two replicas. Our first implementation schedules fine-grained WS transactions. In these transactions, each activity is a low-level database operation. Later 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. The Buddy System is able to handle these course grained WS transactions, using UML stereotypes that allow scheduling semantics to be embedded into the design model. In our final version, we add support for workflow transactions with human interaction. We show that our approach guarantees one-copy-serializability, matches the performance of the lazy update propagation methods, and increases durability in the presence of hardware failures.*

## 1. Introduction

Modern web-based transaction systems need to support many concurrent clients 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. SOA applications provide a high-level of concurrency; we can think of the measure of the concurrency as the availability of the service to all clients requesting services. Replication of the services and their corresponding resources increases availability. Unfortunately, designers sacrifice consistency and durability to achieve this availability. The CAP theory [1, 2] states that distributed database designers can achieve at most two of the properties: consistency (C), availability (A), and partition tolerance (P). Distributed database designers relax the consistency requirements under its influence.

The standard architecture used to increase the availability of a system is through a web service (WS) farm (see Figure 1). The WS farm hosts multiple replicas of the services and their resources. Service requests are distributed among the replicas within a WS farm to ensure a 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 share a single database, so all web service replicas have access to the same data. However, mutual consistency of the replicated database depends on the replica update propagation method, and satisfaction of integrity constraints is not guaranteed.

This paper addresses the issues related to increasing service and data availability while still guaranteeing durability and consistency of replicated databases in the context of SOA. Our approach requires that two of the replicated services must update their corresponding databases simultaneously, thus reducing the risk of data loss due to hardware failures. The selection of the web service pair, called *buddies*, supports consistency. We study the problem in two contexts. First, we model the web services as database transactions, requesting read and write access to the data items. Second, we model the services as black boxes, when only the input and output parameters of the services are known. We represent this information using the UML stereotype construct. Lastly, we consider human interaction interwoven in the transactional workflow. For each context, we develop algorithms for buddy selection and update propagation. O solutions guarantee one-copy-serializability, and our data distribution enforces referential integrity, protects against data loss due to hardware failures, and provides high levels of availability. We present our empirical results for both contexts.

The organization of the paper is as follows. Section 2 lays out an example transaction. Section 3 addresses the problem using fine-grained web services. Section 4 identifies a typical user-defined constraint that, when enforced, reduces availability. Our solution enforces this constraint maintains high availability. Section 5 looks at the problem at a

coarse-grained level where the services are abstract functions. Section 6 adds compensators to the system to support workflow transactions. Section 7 describes the related work and the limitations of current methods. We conclude and discuss future work in Section 8.

## 2. Example transaction

Consider a Ticket Reservation System (TRS). TRS uses web services to provide a variety of functionalities to the clients. Some of these web services return multimedia (images) instead of text back to the client application.

For example, clients may check ticket availability or request a ticket reservation. Upon receiving a client's request, the Availability Check service searches for ticket availability based on the client's requirements and preferences. For example, the client may be looking for aisle seats between May 1st and May 3rd, 2016 to a Mozart concert in Charleston, SC. The service returns an image representing the status of the seating layout including, pricing levels acoustics, etc. Replicated databases improve the Availability Check service's processing time.

When using a lazy-replication architecture, the Availability Check service may retrieve outdated data and, therefore, provide an incorrect response to the client. If strict replication propagation is supported and the request is concurrent with other availability requests, this request may time-out due to process contention. .

The Ticket Reservation service reserves a ticket for the client based on the requirements and preferences given by the client. For example, the service may book two main floor tickets on May 3rd, 2016 to the Mozart concert in Charleston, SC at a rate of $99/ticket. Lazy update propagations, perform efficiently, but may lead to lost data (e.g., if the server where the update transaction runs crashes after the transaction is committed but before the other replicas are updated) and temporary data inconsistency (e.g., if referential integrity spans masters). Strict update propagation guarantees consistency and durability but reduces the availability.

## 3. Proposed system

Our proposed system has three benefits: decrease the risk of losing committed transactional data in case of a site failure, increase consistency of transactions, and increase the 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.

We adopt the WS-Farm architecture (**Figure 1**). Transactions arrive at the dispatcher at the OSI TCP/IP level 7. The dispatcher uses 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 set of Web Service Clusters WSF = $\{WSC_1, \ldots, WSC_n\}$. A single dispatcher receives requests from clients and distributes these requests to 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 HW = $\{hw_1, \ldots, hw_n\}$ is a set of common, off-the-shelf (COTS) hardware devices. DB is a database. In this work, we consider relational databases. The load balancer distributes load to the service replicas in the cluster.

A set of *WS-Replicas*, WS = $\{ws_1, \ldots, ws_n\}$ a set of replicated web services with the same functionality. WS-Replica Buddies or simply buddies are $ws_i$ and $ws_j$, such that $ws_i$ and $ws_j$ 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 = $\{\leq, r[d], w[d] \mid d \in DB, c/a\}$. The read-set of a transaction T denotes all the data items $d \in DB$ such that there is a $r[d] \in T$. The write-set of a transaction T denotes the data items $d \in DB$ such that there is a $w[d] \in T$.

*Data item version* denotes a data value and its version number. Given a database DB = $\{d_1, \ldots, d_n\}$ 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 the data item and version numbers, that is DB = $\{(d_1, v_1), \ldots, (d_n, v_n)\}$.

Each *database* is associated with a *version number*. Given a database DB and the data items $\{((d_1, v_1), (d_2, v_2), \ldots, (d_n, v_n)\}$ in DB, we say the version numbers of DB is the vector $<v_1, \ldots, v_n>$.

*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 may or may not have the same version number.

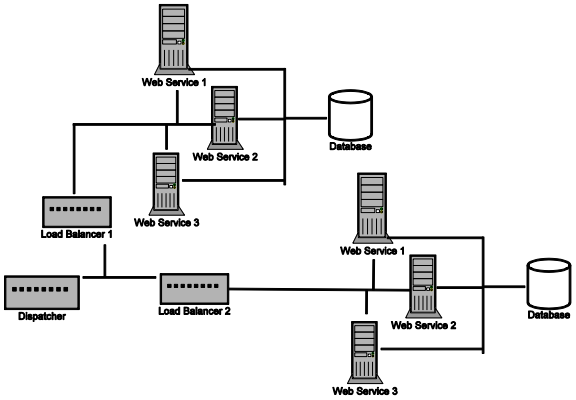


Figure 1. WS Farm with Buddy System

### 3.2. Buddy system

Lazy update propagation is vulnerable to loss of updates in the presence of a database server failure. 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." At least one replica in addition to the primary replica is updated and, therefore, preserves the updates.

**Figure 1** shows a WS farm (WSF) architecture where each cluster (WSC) has a load balancer (LB). 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 (b1) receives the transaction along with its buddy's (b2) IP address. The primary buddy (b1) becomes the coordinator in a simplified commit protocol between the two buddies. Both buddies perform the transaction and commit or abort together. The dispatcher maintains metadata about the freshness of data items in the different clusters. The dispatcher increments the version number for each data item after it has been modified. Any two service providers in two different clusters with the latest version of the requested data items can be selected as a buddy. Note, that the databases (DBR) maintained by the two clusters must have the same version of the requested data items but may not for the other data items.

### 3.3. Dispatcher data structures

The dispatcher maintains a version table for every data object modified by the web services. When a request is received for a service, the dispatcher ensures that the request is delivered to the appropriate cluster. If the request is read-only, the primary buddy (b1) must have the latest version of all committed objects in the request. If the request includes writes, the dispatcher determines whether there is any uncommitted transaction accessing the requested data objects. If there is such a transaction, 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, no suitable cluster can be found due to the distribution of the requested object, and the service request is queued until the currently active transactions complete, or the updates are propagated. To avoid snapshot isolation anomalies, we address blind writes and analyze the read log to determine if an anomaly could take place. Fekete et al. [3] documented anomalies that must be avoided to turn a snapshot isolation schedule into a serialized schedule. We incorporate these requirements to support serializable execution. The dispatcher maintains the following data structures for processing the algorithms:

- *Cluster List* - the names of the clusters (WSC) and their IP addresses.
- *Objects Version Table* - the name of the data objects and their version numbers, corresponding to the completed and in-progress transactions.
- *Cluster Object Table* - the cluster names, stored objects, and the version number of the objects at that cluster.

**Table 1**.a, b, and c show examples of the above tables. The Cluster List Table shows the three clusters 1, 2, and 3, the Object Version Table indicates that data objects B and C are not currently being updated, but object A has two update operations still in-progress at clusters 1 and 2.

### 3.4. Dispatcher service request algorithm

The dispatcher service request algorithm (Algorithm 1) is executed by the dispatcher for every incoming request containing write operations. The service request algorithm finds a pair of buddies that have the correct version for the incoming request, or the request is delayed for later processing. The algorithm has a special check for anti-dependency that ensures 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 executes a simplified version of the algorithm. This version only requires a single cluster, containing the latest committed values of the requested objects, to respond to the request. Due to space limitations, we omitted the read-only algorithm in this work.

**Theorem 1:** The Dispatcher Service Request Algorithm (Algorithm 1) guarantees one-copy-serializability.

**Proof.** Our proof is based on the following claim: Let H be a history over a set of transactions T, such that each transaction Ti ; i = 1, … ,n is made up of a set of read $R_i(d)$ and write $W_i(d)$ operations on data items d. 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 (WSC), and
3. Each cluster guarantees serializable transaction history on its local database.

To show that the claim holds by contradiction, assume that H is not one-copy serializable. Then there must exist a cycle among the committed transactions in the serialization graph of H [4]. Let $T_i$ and $T_j$ be the two transactions responsible for the cycle. **Algorithm 1** does not allow the serialization graph to contain a cycle for the five potential scenarios for these transactions. 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.

Transaction processing between the primary and secondary buddies is synchronous, and both will either commit or abort. After a committed update

INPUT: requestObjects (Request containing objects to be read and written), clusterObjects (Table1c), objectVersions (Table1b)
OUTPUT: buddyList (Pair of buddies or empty list if no pair available), clusterObjects (Table1c), objectVersions (Table1b)

```
Add all clusters to available
for each object O.o,action O.a in requestObjects
  if O in objectVersions
    get completed OV.c,inprogress OV.i from
      objectVersions
  else
    OV.c = 1, OV.i = 1
  for each cluster CO.c,version CO.v in
   clusterObjects
    if CO.v > OV.c && O.a == READ
      available.remove(CO)
    elseif CO.v < OV.i && O.a == WRITE
      available.remove(CO)
  for each cluster CO.c in clusterObjects
    if available(Co.c) and count(buddyList) < 2
      add Co.c to buddyList
  if count(buddyList) > 1
    Choose B1, B2 from buddyList
    for each O.o,action O.a in requestObjects
      get inprogress OV1.i from objectVersions for B1
      get inprogress OV2.i from objectVersions for B2
      if O.action == WRITE
        increment OV1.i
        increment OV2.i
    send buddyList,requestObjects to B1
  else
    enqueue(requestObjects)
```

Algorithm 1. Dispatcher Service Request Algorithm

operation on a data object, the dispatcher updates the Object Version Table to represent the current version numbers. When a primary buddy or any lazy update cluster completes a transaction, it sends a version update request to the dispatcher. The dispatcher updates the latest completed version value for these clusters. After the version is updated all requests in the queue are reprocessed in hopes that the dispatcher can now find a pair of buddies with the correct versions.

### 3.5. Load Balancing

**Algorithm 1** chooses the first available cluster for read only requests and the first available pair of clusters for requests containing write operations which may create an unbalanced use of the WS clusters. We can improve the selection by decorating the Cluster List Table (**Table 1**.a) with attributes to represent system properties (e.g., processing power, available applications, process wait-time, etc.) and network-related information (e.g., link properties, hop-distances, etc.). These attributes can be used to influence buddy selection, e.g., provide load balancing among the clusters or select clusters based on their geographical location and network reliability. This improved buddy selection is not presented in this work.

### 3.6. Analysis of the Buddy System

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

Table 1. Data Structures

(a) Cluster List

| Cluster | IP |
|---|---|
| 1 | 192.168.1.1 |
| 2 | 192.168.1.2 |
| 3 | 192.168.1.3 |

(b) Obj. Version Table

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

(c) 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 |

(d) Obj. Capacity Table

| Object | Capacity |
|---|---|
| B | 2500 |
| C | 4500 |

**Performance analysis in high volume scenarios:** Some of the transactions involve large volumes of data items of the same type. For example, multiple concert tickets have the same characteristics but different row and seat numbers. Performance characteristics of transactions involving a large number of similar objects can be improved by considering the type of resource consumption. We distinguish between three types of consumption patterns:

*Anonymous Item Consumption* when each object is interchangeable, for example, all seats for general admission for a concert. In this case, the B*uddy System* does not improve latency over simple master-slave replication since all concurrent resources requests need to be sent to the same buddy pair.

*Attribute Item Consumption* when each client's request has filters on the attributes, such as main-floor or balcony seating for a concert. The *Buddy System* improves latency over simple master-slave replication because each set of attributes is sent to a different buddy pair.

*Serialized Item Consumption* when each client's request is for a specific object. The *Buddy System* greatly improves latency over simple master-slave replication because each request is sent to a different buddy pair.

**Analysis of lost updates in the presence of failures:** 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 the vulnerability that a lost update can occur if hardware hosting the master replica fails before the lazy update propagation is initiated.

The window of vulnerability is removed by eager replication because the updating transaction cannot commit until all other replicas are also updated. However, the update cost of eager update propagation is high, and, therefore, it is not used frequently.

The *Buddy System* increases durability over lazy replication and maintains efficient performance. 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. For this reason, we require that the data structures maintained by the dispatcher are backed by traditional database durability support.

## 3.7. Empirical results

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. Asynchronous communication dictates that the client sends a request, and the dispatcher sends a response asynchronously. In synchronous communication, the client waits until the response is received. These two methods differ on how the enqueue process is handled when the dispatcher cannot fulfill the request with the current state of the clusters. **Figure 2** shows the empirical data from an implementation using synchronous requests. 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. Each cluster uses a separate MYSQL database in serializable isolation, and a JDBC connection pool is communicating to its individual database.

We generated a dataset with different sizes, and each transaction randomly selects 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. We ran the same transactions against a single master cluster system with lazy replication and a two cluster system with strict replication. **Figure 2** shows that once the dataset size grew to 10,000 items, the performance of the Buddy algorithm allowed the system to exceed the performance of lazy replication while increasing durability.

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

## 4. Anonymous resource consumption

To reduce the performance penalty observed in our initial implementation we modified our solution to guarantee that resource capacity is enforced in a way that we can distribute simultaneous writes to different clusters.

The problem of anonymous resource consumption has driven many system designers away from using an RDBMS because of the way resource contention is. The popular locking mechanism, used to ensure serializability, does not allow for an efficient standard solution to the anonymous resource consumption problem.

To solve the problem of anonymous resource consumption, we introduce a new constraint. The dispatcher keeps a capacity value for each resource which allows the buddy selection algorithm to treat updates to an item as separate writes. The original dispatcher **Algorithm 1** distinguished 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 checks the capacity of the requested resource. If there is available capacity, the update is converted to a write by using an initialization version number instead of the actual version.

### 4.1. Dispatcher data structures

The dispatcher maintains the three original data structures, **Table 1**.a, b, c from the *Buddy System* for processing the algorithms, along with a new structure **Table 1**.d to represent resource availability. The Object Capacity Table contains the name of the data items and their capacities.

### 4.2. Analysis of the Buddy System on resource consumption

**Figure 4** shows the performance of the new *Buddy System* algorithm compared against the lazy and eager replica update protocols. The new algorithm, using the capacity constraints, outperforms lazy replication on all types of resource consumption. Using the *Buddy System* in our earlier example transaction improves 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 also has a guarantee of consistency and durability.

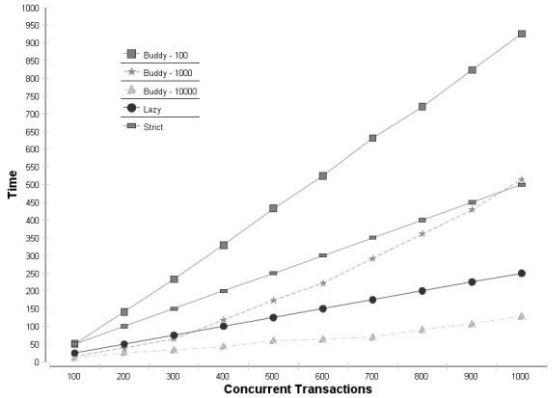


Figure 2. Performance Results

## 5. Coarse-grained web services

In the previous sections, we considered web services as fine- grained CRUD services similar to a database transaction. In this section, we target coarse-grained web services, where the only information the dispatcher has at runtime is the input and output parameters of the web service.

### 5.1. Example transaction expanded

Consider a Ticket Reservation System (TRS) presented in Section 3. Clients select a specific seat for the popular concert as a step in the ticket reservation transaction. **Figure 3**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 available seats
- Pricing for the current user
- Performance details

After the users have selected a set of seats, they would like to purchase, a web service is invoked that finalizes the seat reservation transaction. In Figure 3, we show the web services of the TRS.

- GetSession –retrieves session state based on a unique session id.
- LoginAnonymous –logs a user in so they can retrieve credentials for pricing and

seating location availability. If the session does not have a logged in credential, it gives the user the "anonymous" credentials.

- GetZones –retrieves the zone information for the space where the event takes place. This information is used to allow a user to navigate between zone information. This information does not typically change after a ticketed event has been setup.
- GetSeats –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 –retrieves state information for all the seats in the zone. This information changes when any seat is consumed by another user.
- GetPerformanceDetails –retrieves program details for the performance that is being sold. This information does not typically change after a performance has been setup.
- ReserveSeats –consumes the limited resource and changes the state of the previous GetSeatState web service.

The number of requests may vary drastically over 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 may need to handle hundreds of thousands of simultaneous requests.

The popular way to handle unknown workload is by manually partitioning the data across different servers. For example, each event could have its own ReserveSeats. However, this solution does not scale well; because, new hardware would be needed to handle higher levels of event concurrency, and it reduces consistency in the process.

## 5.2. UML semantics

Additional semantics for the coarse-grained web services can be acquired from the 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 allows a designer to extend the vocabulary of UML in order to represent new model elements [4]. Traditionally programmers manually translated these semantics into the program code.

```
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)
```

Algorithm 2. Coarse-Grained Buddy Selection

**Read and write semantics.** Figure 3 is an activity diagram with two stereotypes used to model web services that are read-only and web services that write and update data. The ReserveSeats service 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. The same data can be retrieved from the WSDL **[5]** message types. In this work, we use the data from the UML diagram.

An attribute level stereotype <<PK>> represents the unique identifier combination of the resources. For example in the *GetSeatStatus* web service, an individual seat's status can be uniquely identified in the response by the attribute set {Performance, Zone, SeatId}. The unique identifier is also used by the *ReserveSeatsRequest* service as the input parameter.

## 5.3. Parallel scheduling semantics

The UML Activity diagram (**Figure 3**) provides us with the semantics required to know which services can be called in parallel. The GetSession

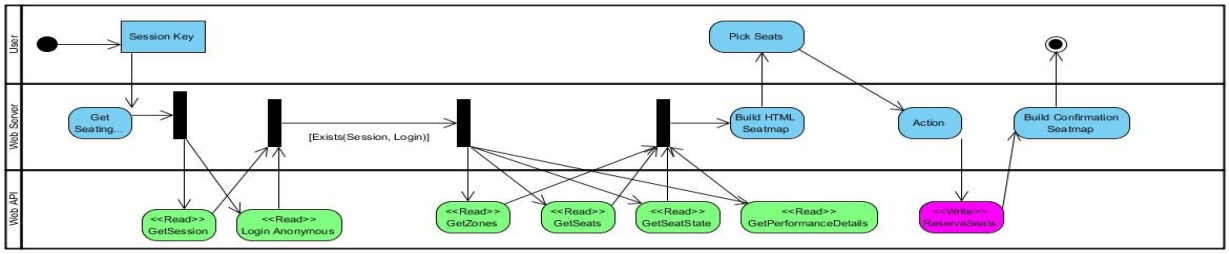


Figure 3. Activity diagram for ticketing seat selection

and LoginAnonymous services are required to be called before the remaining services because they change required state used by the later service. We export this data to an XMI file. The XMI is a standard XML layout to represent the UML diagram. The web services form a directed acyclic graph (DAG). A *fork* node separates a flow into several parallel control flows. A *join* node joins several parallel control flows once a token has arrived from each flow. The fork, join, and each web service is represented as *ownerMember* XML elements with a unique identifier. Each path leads to the join node where the dispatcher waits for all paths to complete. We employ a breadth first search algorithm that uses parallel traversal to follow all the parallel paths in the fork.

## 5.4. 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 1***)* needs visibility into all operations of the transaction at a single point in time. The client facilitates visibility by sending all requests as a batch. The dispatcher sequences the calls based on the semantics from the XMI data.

**Buddy selection algorithm.** Algorithm 2 is an updated buddy selection algorithm to select the appropriate pair of web services to perform the transaction. The algorithm iterates over the forks in the activity diagram and services 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 iterates over each web service and flattens 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.

**Theorem 2**: The Coarse-Grained Buddy Algorithm (**Algorithm 2**) guarantees one-copy-serializability.

**Proof**: 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 $r_i(d)$ or write $w_i$ (d) operations on data item d. H is one-copy serializable if the following three conditions hold:

1. Each request (transaction) is an atomic transaction

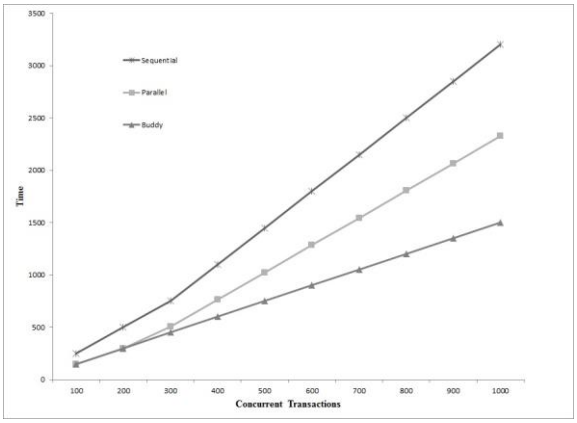


Figure 4. Empirical results for Coarse-Grained Implementation

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. Since Theorem 1 holds, and **Algorithm 2** maps from the coarse-grained web services to the fine-grained CURD operations of **Algorithm 1**, then the serialization graph cannot contain a cycle.

## 5.5. Coarse-grained 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 until completion. In **Figure 4**, 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 against three different architectures. First each web service was called sequentially using no UML semantic data, second web services were called in parallel using the semantic data from the UML Activity diagram, and third they were distributed using the semantic data from both the activity and the class 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 dispatcher schedules the services in a sequence on the same web service box.

Using the semantic data from the UML Activity diagram **Figure 3**, the dispatcher determines that a sequence of two subsets is required:

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

Using these semantics, the dispatcher can schedule services in the same subsets in parallel for an improvement in performance over the original sequential schedule. **Algorithm 2** allows 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 stereotype in the UML activity diagram (Figure 3) indicates the read and write activities.
2. The individual items from the UML class diagrams represent the fine-grained items.

The <<PK>> stereotype in the UML class diagrams uniquely identifies each tuple in the fine-grained operations. Once these semantics have been identified, the original buddy algorithm (**Algorithm 1**) can be implemented on the coarse-grained services.

**Figure 4** 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.

## 6. Workflow Transactions

Previous versions of the *Buddy System* only supported a single web-services request per transaction. Workflow transactions or "Long Running Transactions" consist of several web services invoked in a sequence. Often there will be input gathered from human interaction interwoven between some of the web service requests. Unfortunately, the latency from human interaction is not predictable and can significantly lower availability if locks are held or if all future requests need to be scheduled on the same pair of buddies.

The workflow of the series of web-services shown in the UML diagram in **Figure 3** uses UML stereotypes to identify web services that mutate data. The two web services in the model could either be implemented with database insert operations or database update operations. Insert operations allow higher concurrency in the Buddy Architecture by allowing a higher distribution of web service scheduling among the clusters. Insert operations also allow for simpler compensator actions because they can be scheduled on any pair of buddies without being concerned with other tuple versions. The compensator for an insert is a delete operation. The compensator for an update is unknown If multiple updates occurred on the same data element during the time of the long running transaction.

### 6.1. Empirical Results

In this version of the *Buddy System*, we extend the system to support workflow transactions consisting of one or more requests in a transactional workflow. We limit the data activities to Read, Insert & Delete operations. Insert and Delete operations do not add or remove version information in the dispatcher until the workflow has committed. The *Buddy System* guarantees one-copy-serializability with an isolation level of serializable for individual requests. For multiple request workflows, the system will guarantee one-copy-serializability with an isolation level of read committed.

### 6.2. Empirical Results

A performing art center is modeled with blocks of current users accessing the system ranging in value of one hundred from one hundred to one thousand. Each user had identical credentials so that the server processing load would be the same for each user. The model was compared to three different architectures;

1. Fat client application written in JAVA SE using JDBC against a single MySQL database. The application opens a database transaction at the beginning of the workflow and holds the transaction through the entire workflow. The database is using "Read Committed" isolation level so it will see availability decreases when a transaction is completed. The inventory transactions are modeled with update operations with a capacity counter. The primary issue for this architecture is that the data used for the availability count will be locked by the first process updating the availability. Therefore, the architecture can only handle one concurrent thread.
2. Fat client application written in JAVA SE using JDBC against a single MySQL database. The application opens a database transaction at the beginning of the workflow and holds the transaction through the entire workflow. The database is using "Read Committed" isolation level so that it will see availability decreases when a transaction is completed. The inventory transactions are modeled with insert operations into a journal table. The availability is calculated using an aggregate query. There are two issues with this architecture; the data used for the availability count will be incorrect if there is concurrency as transactions will not see consumed inventory until transactions are

completed. Also, there were numerous failed transactions because a trigger was used to ensure at the end of the workflow that the inventory is not oversold. This resulted in a

3. The client is calling a SOAP Web-Service running under the *Buddy System* Architecture with four clusters. The *Buddy System* is using a capacity constraint [3] to ensure that inventory is not oversold. The inventory transactions are modeled with insert operations into a journal table as in the previous experiment. The *Buddy System* creates compensators to delete these inserts if the transaction is rolled-back. This architecture guarantees the correctness of the data and allows a much higher availability.

With the *Buddy System*, higher availability was achieved by reducing the locks and the failed transaction while guaranteeing the consistency.

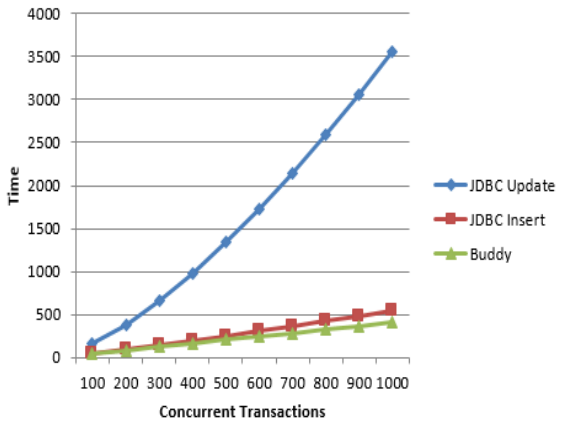


Figure 5. Empirical Results

Figure 5 shows the empirical results.

## 7. Related work

Most distributed database research ignores resource consumption issues and assumes traditional locking mechanisms. Julian Jang et al. [6], investigate ways to provide non-locking resource consumption for a longer duration than the transaction to avoid holding locks. Unfortunately, this approach sacrifices ACID guarantees that are provided by traditional RDMS. One of the current application areas for replicated databases is web services applications. Lou and Yang [7] 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. 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 for strict replication, 2.) trying to increase consistency for lazy replication, and 3.) attempting to use a hybrid approach.

### 7.1. Higher availability with strict replication

Several methods have been developed to ensure mutual consistency in replicated databases. These methods aim to, eventually, provide one-copy serializability (1SR). Transactions on traditional replicated databases read any copy and write (update) 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 violate mutual consistency. During the last decade, several methods have been proposed to ensure mutual consistency in the presence of lazy update propagation (see [8]for an overview.) More recently, Snapshot Isolation (SI) [9, 10] 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.

### 7.2 Higher consistency in lazy replication

Breitbart and Korth [11], and Daudjee et al. [12] propose frameworks for master-slave, lazy-replication updates that provide consistency guarantees. These approaches require 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.

### 7.3. Hybrid approach

Jajodia and Mutchler [13], and Long et al. [14], 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. [15] 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 [16] 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.

### 7.4. Sagas

Traditional ACID transactions use locks to guarantee the ACID properties. These transactions tend to take milliseconds to complete, so the negative side effects of the locks are 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 period makes the traditional method of using locks much less desirable. 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 shut down a service provider.

Garcia-Molina and Salem [18] define sagas as a solution to maintain some of the atomic properties of ACID transactions when performing 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. The manual coding solution of developing compensators increases the potential for logic errors introduced into the application.

## 8. Conclusion

In this paper, we propose an extension to the lazy replica update propagation methods. Our solution is based on using an application-layer dispatcher to select two web service clusters, called buddies, to perform a transaction simultaneously. The buddy selection is based on the data items and the operations of the transactions, the data versions available, and the network characteristics of the WS farm.

We provide an example transactional workflow with web services that return images representing the current state of the data. We show that although our solution can be viewed as a special case of hybrid update propagation method, it provides several advantages over existing solutions.

First, our approach provides the scalability required by modern applications, such as web services, and is suitable for the architectures and technologies implementing these applications. The buddy selection algorithm supports dynamic master-slave site selection for data items, ensures correct transaction execution, and aids load-balancing among the replicas. Finally, incorporating network specific characteristics, such as distance and bandwidth, reduces latency observed by the client. Our approach has a similar message complexity compared to a traditional lazy master-slave replication using group communication and less message complexity compared to a lazy master-slave replication without group communication.

We further extend the *Buddy System* to handle coarse-grained web services. Our solution is based on extending UML specifications 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.

Lastly, we extend the *Buddy System* to handle workflow transactions. The workflow transaction allows multiple requests to be included in the same transaction without lowering the availability of the system. The *Buddy System* auto-generates compensators to undo operations in the case of a rolled-back transaction instead of holding locks.

## 9. 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. Fekete, S. N. Goldrei and J. P. Asenjo, "Quantifying isolation anomalies," in *Proceedings of the VLDB Endowment*, 2009.

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

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

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

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

[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] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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 Concurrencia y Sistemas Distribuidos*, 2005.

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

[18] 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.

[19] 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.