Service Constraint Guarantees

Aspen Olmsted, RoxAnn Stalvey

Department of Computer Science
College of Charleston
Charleston, SC 29401

Abstract

In this paper we investigate the problem of providing consistency, availability and durability for Web Service transactions. We consider enforcement of integrity constraints in a way that increases the availability while guaranteeing the correctness specified in the constraint. We study hierarchical constraints that offer an opportunity for optimization because of an expensive aggregation calculation required in the enforcement of the constraint. We propose an approach that guarantees enforcement of constraints and also allow the distribution of write operations among many clusters to increase the availability. In our previous work, we proposed a replica update propagation method, called the Buddy System, which guaranteed durability and increased availability of web services. In this paper, we extend the Buddy System to enforce hierarchical data integrity constraints.

1. Introduction

Enterprise web-based transaction systems need to support many concurrent clients simultaneously accessing shared 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 the measure of the concurrency as the availability of the service to all clients requesting services. A common way to increase the availability is through the replication of the services and their corresponding resources. Often a web farm is used to host multiple replicas of the web application, web services and their resources. Requests are distributed among the replicas. Consistency and durability are often sacrificed to achieve increased availability. The CAP theory [1] [2], stating that distributed database designers can achieve at most two of the 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. In our previous papers we have addressed issues related to increasing availability while still guaranteeing durability and consistency of replicated databases. In this paper we address issues related to maintaining high availability while adding guarantees of correctness by enforcing hierarchical constraints. Traditionally these hierarchical constraints are not enforced by the system due to the expensive run-time cost.

In our previous work [3] [4] we provided an extension to the lazy replica update propagation method to reduce the risk of data loss and provide high availability while maintaining consistency. The Buddy System executes a transaction on a primary replica. However, the transaction cannot commit until a secondary replica, “the buddy”, also preserves the effects of the transaction. The rest of the replicas are updated using one of the standard lazy update propagation protocols. The Buddy System provides a guarantee of transactional durability (i.e., effects of the transaction are preserved even if the server, hosting the primary replica crashes before the update can be propagated to the other replicas) and efficient update propagation (i.e., our approach requires the synchronized update between two replicas only, therefore adding minimal overhead to the lazy-replication protocol).

The Buddy System uses an application-layer dispatcher [5] to select the buddies based on the data items and the operations of the transactions, the data versions available, and the network characteristics of the WS farm. A limitation of the Buddy System is that integrity constraints that require different classes of the transactions, the data versions available, and the network also allow the enforcement of the constraint. We propose an approach that pulls enforced hierarchical data integrity constraints.

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 better distribute the 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.
2. Example Transaction

The Washington, DC transit system uses a smart card (SmarTrip) as a payment system. The card maintains the value that results from passenger activities (boarding, disembarking, adding value to the card). A centralized activity log records each activity. The activity is linked, to the smart card involved in the activity, in a centralized system. Some activities originate on the card (boarding, disembarking) and others originate in the central system (adding value). Error! Reference source not found. shows a sample UML class diagram for this example. The activity log relies upon a sequence number being maintained for each card to identify the ordering of activities. An incorrect sequence number can cause the system to not allow the 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 vulnerability. Imagine thousands of passengers unable to gain access to the public transportation system. Often this category of constraint is not enforced because of the expense of the runtime calculation. Figure 1 shows a simple example SQL check constraint that would enforce the constraint.

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

Figure 1. SQL Constraint

Unfortunately, most commercial SQL implementations do not allow sub-queries in the check constraint due to the potential expense. The lack of a sub-query makes this constraint impossible to enforce.

3. Integrity Constraints

Codd [16] 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 set.
- **Domain** - The model can define domains to represent valid values stored in entity attributes. Expression of the valid domains happens through the use of data types.
- **Column** - Each column of the entity set can specify a smaller set than the complete range for the data type. Expression of these smaller domains is accomplished using the ENUM feature of the database management system.
- **Foreign Key** - The DBMS can enforce that a parent related record exists in the database. If the parent does not exist then the child relationship cannot be added.
- **User defined** - A user-defined integrity constraint can express any user defined logic check. Check constraint syntax defines the invariant and the DBMS will enforce the invariant on inserts and update operations. DBMS SQL syntax often allows for the definition of both column level check constraints and tuple level check constraints. Tuple level check constraints can enforce integrity using any attributes of the tuple in comparisons. These comparisons may also use data available in sub-queries.

We classify Codd’s [6] original five constraint types into three categories: entity, domain and hierarchical. This smaller category size allows us to separate constraints that can be run in parallel and those that need sequential executions. The domain and column constraints are both used to limit the domain of an attribute. Domain and column constraints can be enforced in parallel because all required information exists in the single tuple. Foreign key constraints are also a form of domain constraint. They allow a refinement of the domain of a column to just existing parent objects. User defined constraints are primarily used to express constraints on associations between relations that are more complex than the semantics that can be expressed with referential integrity. These associations are typically hierarchical and enforce an aggregate or require an iteration across children records in an association.

4. Object Constraint Language

Object Constraint Language (OCL) is part of the official Object Management Group (OMG) standard for UML. An OCL constraint formulates restrictions for the semantics of the UML specification. An OCL constraint is always true if the data is consistent. Each OCL constraint is a declarative expression in the design model that states correctness. Expression of the constraint happens on the level of the class,
and enforcement happens on the level of the object. OCL has operations to observe the system state but does not contain any operations to change the system state.

4.1 Kinds of OCL Constraints

- **Invariants**: An invariant is a condition that always holds. In a relational database management system, RDBMs, invariant maps to an assertion. The DBMS enforces the assertion on every action to the system.

- **Pre-conditions**: A pre-condition is a condition that is guaranteed to hold before an activity execution. RDBMs use check constraints to enforce the constraint. The check is only required when data is inserted or updated in a specific table.

- **Post-conditions**: A post-condition is a condition that is guaranteed to hold after an activity is executed. In RDBMs, the postcondition needs to be implemented in a trigger. The trigger allows the evaluation after the action.

OCL is integrated into many UML design environments. It fits well in a model driven architecture (MDA). Figure 3 shows a sample OCL to enforce that the sequence number on an inserted activity is greater than all previous sequence numbers for the same smartTrip card.

![Figure 3 - OCL Example](image)

5. Hierarchical Constraints

Hierarchical constraints are expressions of data integrity that involve more than one tuple in the correctness expression. 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; universal and existential quantification.

An example of a universal quantification in our metro card example is the requirement that the new sequence is greater in value than all previous sequences numbers (Figure 3). An example of existential quantification in our metro card example is that there must be at least one previous activity that added value to the card before any other activity.

With aggregate constraints, the functional aggregate calculation is often expensive to calculate at insertion time and is therefore ignored. In relational database systems, this enforcement is done with a check constraint or trigger. The former being less expensive as it is a declarative constraint. Unfortunately check constraints that can execute sub-queries are often not supported in the relational system. SQL-99 compliance names this feature F671 and none of the major relational vendors support this feature [19]. Triggers are a more expensive solution for enforcement of the hierarchical 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 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 compared 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 is a sale’s line item table with a quantity field. The sum of the quantity field is often used to ensure the new tuple does not surpass inventory quantity.

- **Count**: Count Aggregation constraints are used to ensure adding a new tuple does not surpass an upper bound on the quantity. An example would the capacity constraint added to the Buddy System in our previous work [1]. Referential Integrity [16] is a specific form of a count based aggregate constraint. Normally the count has a value of one when enforcing referential integrity. Having a guarantee of one related record ensures the parent record exists.

In this work, we show how to check an aggregate calculation to enforce a constraint without the cost of recalculation for each request. We also perform a check at a point in the transaction that allows for high distribution of transactions across clusters.

5.1 Aggregate Constraint Materialization

We recommend enforcing aggregate constraints on the dispatcher or a server dedicated to the constraint enforcement. The constraint server materializes the constraint by keeping a copy in memory of the aggregate calculation. As a transaction completes or aborts, the materialized aggregation is updated accordingly. For example, a non-completing transaction on a minimum or maximum aggregate only updates the materialized value if it is still the current value. These “undo” operations are a form of “role-back” without requiring locks that would reduce the availability. Table 1 shows example data that is maintained by the dispatcher to materialize a constraint. The value and parent of an object are stored per object along
with the quantity. The quantity is only used with aggregate operations, such as average, where the quantity of records in the hierarchy is part of the equation.

### Table 1 – Sample Constraint Materialization Data

<table>
<thead>
<tr>
<th>Object</th>
<th>Constraint</th>
<th>Parent</th>
<th>Value</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>smarTrip</td>
<td>sequenceOrd</td>
<td>1000120</td>
<td>408</td>
<td>408</td>
</tr>
</tbody>
</table>

All post-condition constraints are converted to pre-condition constraints to allow a check at dispatch time. An example of a post-condition check conversion to a pre-condition is shown in Figure 5. This OCL represents the same transaction semantics as Figure 3 but can be checked on the dispatcher. The serialization and atomic guarantees by the clusters allow this conversion to take place to increase the availability.

### 5.2 Iterative Constraint Materialization

Universal quantifications are expressed with a comparison against a scalar or 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 universal quantification using a comparison against an aggregate, we use the same materialization infrastructure from above.

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

**Figure 5. Pre-Condition Conversion**

Verification of existential quantifications happens on delete operations along with insert and update. There may be several records that are available to satisfy the constraint. The system maintains a tuple for each constraint. The tuple holds 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 2.

### Table 2. Sample Constraint Materialization Data

<table>
<thead>
<tr>
<th>Object</th>
<th>Constraint</th>
<th>Parent</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>smarTrip</td>
<td>paymentExists</td>
<td>1000120</td>
<td>3</td>
</tr>
</tbody>
</table>

### 6. New Constraint Types

We have grouped the original Codd [16] 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 attribute stereotypes that represent the primary keys. Examples of the stereotypes are found in our previous work [15]. Web services require two additional constraint types not handled in relational database systems. Temporal constraints and cross domain constraints.

#### 6.2 Temporal Constraints

Temporal constraints model the state before and after the web service. There are two perspectives to consider in the context of temporal constraints: client and server. Server temporal constraints guarantee the state of the server is consistent after the service is completed based on the actions of the service and the workflow the service is a part of. Client temporal constraints guarantee the state of the client after the service is completed. In this work we, use the @pre (allows access to previous state) 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 use transactions.

To enforce 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 can be implemented from the client to the server to allow the client to roll back the server transaction in cases where the client constraint does not pass. Unfortunately, this method doubles the message count for every transaction and reduces the improvements in availability we have already achieved.

Using the method from our previous work mapping course grained services to fine grained services [18] 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 a constraint fails on the client after a response has been returned by the 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 4 shows an activity diagram with post conditions on both the server and the client.

Figure 4 is a simple example where we care about client state relative to one web service request. In many service oriented designs, a session identifier is added to each web service call so the services can utilize server side state without forcing all previous results to be passed in the service call. The session identifier allows web services to be chained into transactional workflows. The transactional workflow can have split operations to allow simultaneous web service calls and join operations to wait for a set of web services to all complete before moving to the next stage in the workflow. In this type of transactional workflow, the correctness of the client side state is based on the results of several asynchronous services at a join operation.

Figure 6 is an expanded example activity diagram. In this workflow when a user adds value to their card they qualify for loyalty points. We model the loyalty point system as a cross-domain call in a separate realm of authority from the normal web-services used to record value transactions. On adding value, there is a split operation where two asynchronous web-service calls are made. One call is to the in-domain web-services to record the value added to the card. The second call is to an out of domain web-service to record loyalty point activity. There are several points of exception that can occur once the web-services calls are joined back together:

- Card is never touched – In this case, the web service calls may have taken too long, and the transaction needs to be undone through the use of the compensating web services
- Loyalty points on card and loyalty points returned from service are not in sync – In this case there may be a security issue where the loyalty points have been altered by a malicious user or program.
- Card number returned from either service does not match card number touched before and after – In this case the response may be coming from a previous call that took longer to process than expected.

For each of the exception cases with temporal constraints a violation of the invariant in the client requires the compensator on the server to be executed. The compensator is behind the buddy system and the operations are unknown to the client. To facilitate this process we enhanced the buddy system to return a URI for a web service that will execute the undo operation. The client can then call the web service in the case of an invariant failure.

### 6.2 Cross Domain Constraints

As the internet continues to connect business and consumers, transactions begin to involve external entities in their transaction correctness. For example, many businesses have begun to reward social micro-blogging with loyalty currency. The correctness of these transactions depends on data available via web-services available at Twitter™ and Facebook™. To investigate the type of constraints required with these cross-domain loyalty transactions we investigated two aggressive programs.

The Marriott™ Hotel chain rewards patrons for micro-blogging on Twitter using a specific hashtag of #MRPoints. The reward for the micro-blog post or micro-blog re-post is twenty-five points [9]. The currency value of each post is valued at around $0.42. They cap the daily value of all micro-blog posts at one hundred points. There are no constraints placed on the tweets semantics, timing or the validity of the twitter account.

To evaluate the semantics of the micro-blogs we pulled a sample consisting of a day worth of micro-blogs. The sample consisted of 3,982 unique micro-blogs including the tag and 1,474 re-posts of these blogs. Only four percent of the blogs included reference to a specific hotel.

To evaluate the timing of the micro-blogs we used the same sample from above and compared the first and last post for the day for the user. Eighty-two percent of the users in the sample who posted multiple micro-blogs did so in a timespan of less than five minutes.

To evaluate the validity of the micro-blog user accounts we used the same sample from above. For each poster account, we examined the activity on their twitter account. We defined a valid account as one where the Marriott tweets represented less than twenty percent of the tweets. Seventy-four percent of the user accounts in the sample were classified as not valid in our study.

JetBlue™ Airlines also rewards its patrons with loyalty points for micro-blogging on Twitter [10]. The airlines program is a little different in a few perspectives; they automate the instantiation of the micro-blog so they can insure
the post includes links to the program, they reward badges as a middle layer to earning loyalty currency, and they utilize micro-blogging on Facebook as well as Twitter. As in our case with Twitter we considered three types of validation requires; the blogs semantics, timing of the blog and the validity of the bloggers account.

To evaluate the timing of the micro-blogs we pulled a sample consisting of a week’s worth of micro-blogs from twitter that used the tag #TrueBlueBadges. The sample only consisted of twenty-eight posts and zero re-posts (JetBlue did not reward badges for retweets). To understand why participation was so low we schedule a job that pulled the tweets every minute for a weeks’ time. To our amazement sixty-three percent of the posts were deleted shortly after the post. Unlike the Marriott program that rewarded the points at a later point in the day, the JetBlue program would reward the credit immediately on the posting. Our new sample data consisted of ninety-seven original posts. Sixty-one were deleted, and twenty-one were part of batches of posts done in less than five minute time periods. We concluded that only fifteen percent were valid based on the timing activities.

To evaluate the semantics of the micro-blogs we used the same sample from above and found that no posts included additional data from the link template created by the JetBlue website.

To evaluate the validity of the micro-blog user accounts we used the same sample from above. For each poster account, we examined the activity on their twitter account. We defined a valid account as one where the Marriott tweets represented less than twenty percent of the tweets. Twenty-eight percent of the users were classified as not valid in our study. We attribute the higher percentage of validity to the ability of a user to cleanse their twitter account after an immediate post.

7. 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. We loaded the model into a Microsoft SQL Server 2008 server. A function was developed that takes a single argument, the card id, and returns the maximum sequence for that card id. SQL Server does not support sub-queries in check constraints but does support functions. The function was called from inside the constraint to enforce that new tuples have a sequence greater the current maximum for that card.

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 the Buddy System implementing the constraint with four clusters (Figure 7). Without the Buddy System, the SQL Server implementation performed well as long as there was an index on the card id. The index allowed the system to seek on the index tree to the subset of records for one customer. The database system did not use synchronization when performing the check constraint. The asynchronous check means that current consistency with lazy replication and the SQL implementation is not guaranteed. With the Buddy System, higher availability was achieved by distributing the inserts to all four clusters while guarantee the consistency.

We added cross-domain constraints that would award a patron extra value if they had tweeted about their experience on the subway while on the trip. The tweet was validated to be between the start and end of the trip and required a special hashtag.

All of the cross-domain constraints can be distributed and do not require checking at the buddy dispatcher. Therefore the cross-domain calls to twitter and Facebook could be made from the RDBMS system behind the cluster. Out of the box no SQL database allows for social micro-blogging constraints. We developed these constraints using SQL Servers .NET assembly technology which allows us to embedded .NET that calls out to twitter and Facebook to validate the constraints.

8. Related Work

Constraint specification and enforcement have been a part of relational database model research since Codd [6] originally wrote the specification. Recently work on auto generation of SQL code to enforce these constraints from the UML model has been done by Heidenreich, et al. [7] and Demuth, et al. [8]. In both these works, the focus is on the generation of the SQL code for relational databases for the invariants. Distributed databases and web services require additional work to ensure the constraints can be guaranteed while increasing availability of the service and data through the distribution of the load among the clusters available. Services also have constraints that need to be true before a service is called and after a service is completed.

Research has been conducted for decades on strict and lazy replication in RDMS. Recent research can be grouped into one of three goals: 1.) to increase the availability with strict replication, 2.) to increase consistency with lazy replication, and 3.) to use a hybrid approach to increase the availability. Our previous work, The Buddy System, increases availability will provide consistency and durability.

- Increasing Availability with Strict Replication: Several methods have been developed to ensure mutual consistency in replicated databases. The aim of these methods is eventually to provide one-copy serializability (ISR). 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 [6] for an overview.) More recently, Snapshot Isolation (SI) [7, 8] has been proposed to provide concurrency control in replicated databases. The aim of this approach is to provide global one-copy serializability using SI at each replica. The advantage is that SI provides scalability and is supported by most database management systems.

- **Increasing Consistency in Lazy Replication:** Breitbart and Korth [9] and Daudjee, et al. [10] propose frameworks for master-slave, lazy-replication updates that provide consistency guarantees. These approaches are based on requiring all writes to be performed on the master replica. Updates are propagated to the other sites after the updating transaction is committed. Their framework provides a distributed serializable schedule where the ordering of updates is not guaranteed.

The approach proposed by Daudjee et al. provides multiversion serializability where different versions of data can be returned for read requests during the period that replication has not completed.

- **Hybrid Approach:** Jajodia and Mutchler [11] and Long, et al. [12] both define forms of hybrid replication that reduce the requirement that all replicas participate in eager update propagation. The proposed methods aim to increase availability in the presence of network isolations or hardware failures. Both approaches have limited scalability because they require a majority of replicas to participate in eager update propagation. Most recently, Irun-Briz, et al. [13] 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 [14] 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.

- **Buddy System:** In our previous work [3, 4, 15], we provide an architecture and algorithms that address three problems: the risk of losing committed transactional data in case of a site failure, contention caused by a high volume of concurrent transactions consuming limited items, and contention caused by a high volume of read requests. We called this system the Buddy System because it used pairs of clusters to synchronously update all transactions. The pairs of buddies can change for each request allowing increased availability by fully utilizing all server resources available. Consistency is increased over lazy-replication because all transactional elements are updated in the same cluster allowing for transaction time referential integrity and atomicity.

An intelligent dispatcher was placed, in front of all clusters, to support the above components. The dispatcher operated at the OSI Network level 7. The high OSI level allowed 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.

After receiving a transaction, the dispatcher picks the two clusters to form the buddy pair. The dispatcher selects the pair of clusters based on versioning history. If a version is in progress and the request is modifying the data, then the dispatcher chooses set containing the same pair currently executing the other modifier transactions. Otherwise, the set contains any pair with the last completed version. The primary buddy receives the transaction along with its buddy’s IP address. The primary buddy 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 a version counter for each data item after it has been modified. Any two service providers (clusters) with the latest version of

- **Figure 7. Empirical Results**

![Figure 7. Empirical Results](image-url)
the requested data items can be selected as a buddy. Note, that the database maintained by the two clusters must agree on the requested data item versions but may be different for the other data items.

9. Conclusion

In this paper, 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 constraint’s 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 the request passes the constraint check. We also defined new constraint types required for web service transactions (temporal and cross-domain). A limitation of our work is that we currently only support a subset of possible OCL notation for the expression of the aggregate constraints.

10. References
