A Customizable and Secure Software Architecture

Aspen Olmsted, RoxAnn Stalvey
Department of Computer Science
College of Charleston
Charleston, SC 29401

Abstract

In this paper, we investigate the problem of providing customization opportunities at implementation time while maintaining consistency, high availability and durability for distributed web-service transactions. We consider ways of adding runtime configurable interception points for after release customization while guaranteeing the correctness of the entire operation. We study integration hooks, that offer an opportunity for optimization by allowing the hook to pass data, writes, updates or consumption to the data tier or request additional data from the data tier. In our previous work, we proposed a replica update propagation method called the Buddy System, which guaranteed consistency, durability and increased availability of web services. In this paper, we extend the Buddy System to allow a pipe and filter architecture on incoming requests and the corresponding responses.

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. Web farms are used to host multiple replicas of the web application, web services, and their resources. Incoming requests get distributed among the replicas. Consistency and durability are often sacrificed to achieve increased availability. The CAP theory, stating that distributed database designers can guarantee 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 [1] [2]. In our previous papers, we have addressed issues related to increasing the availability while still guaranteeing durability and consistency of replicated databases. In this paper, we address issues related to maintaining high availability while adding customization hooks that allow secure transactional properties to hold for the entire transaction. Traditionally these after release customizations do not enforce secure transactional properties and add expensive run-time cost.

In our previous work 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 [3] [4]. The *Buddy System* executes a transaction on the primary replica. However, the transaction cannot commit until a secondary replica, " the buddy", also preserves the effects of the operation. The rest of the clusters are updated using one of the standard lazy update propagation protocols. This architecture allows the *Buddy System* to guarantee transactional durability. The 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. It also provides 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 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 [5]. A limitation of the *Buddy System* is there is no way to add end-user configurable business rules at implementation time without either sacrificing consistency or requiring a recompile of the client application. Some business rules require availability of additional data, not available in the original request, to make a decision. To ensure consistency, the *Buddy System* dispatcher needs to know of the extra data when it picks qualified clusters to service the request. We provide an approach that allows a business rule to modify the incoming request before the dispatcher selects the qualified buddies. The active cluster then sends back the additional data to the filter for the custom business rule to be applied.

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

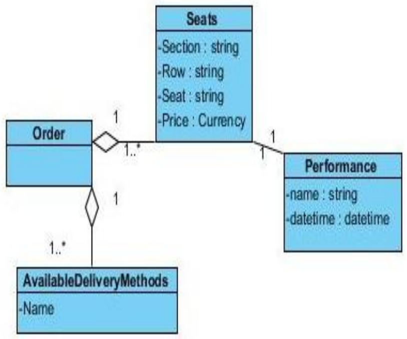

Figure 1. Original Response Data

supports dynamic master-slave site selection for data items and ensures correct transaction execution. Third, our method can easily be 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 in the customized transaction.

2. Example Transaction

The New York Philharmonic sells tickets for events on their self-service website (http://www.nyphil.org). Patrons can go online and purchase tickets to upcoming performances. The self–service web-site was built by assembling custom-made HTML pages along with a set of out of the box web-services. One of these web-services is used to add tickets to the shopping basket. The data returned from the web-service is shown in the UML diagram in Figure 1. The returned data holds all the tickets in the current basket along with delivery methods available for the customer for the current order.

The New York Philharmonic uses a set of organization defined business rules to control the delivery methods the patron has available to them in a transaction. A custom web-service integrates the organization's business logic into the transactional workflow. Execution of this service happens after the return from the out-of-the-box "AddTickets" web-service. Figure 2 shows a UML activity diagram for a sub-transaction that adds tickets to the shopping basket.

There are three problems with the custom web-service solution. First, there is extra latency caused by the extra round trip to the server. Second, there is a heavier load on the server, processing the additional web service request. Third, the process as a whole is more complicated to debug from a client side development perspective.

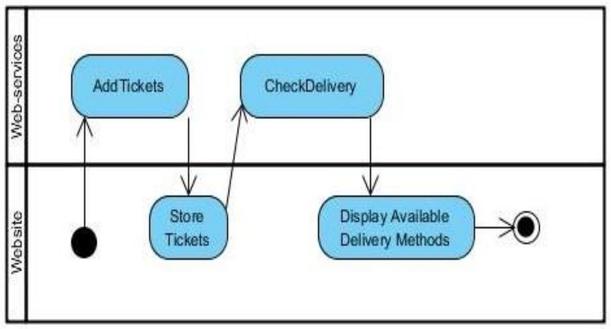

Figure 2. Add-Ticket Sub-Transaction Workflow

3. Server-Side Pipe & Filter Architecture

The JAVA programming language allows for a pipe and filter architecture using the servlet specification [6]. A Servlet is a JAVA object that is instantiated to process a request. This process happens in a servlet container (such as TomCat) running on the server. The servlet container passes a request object and a response object to the servlet. The request object holds that data sent from the client. Often the servlet will access a relational database to obtain data used to build the response. The response is then passed back to the client.

The Servlet Specification allows an unlimited number of filter classes to be defined to intercept the request before the servlet receives the request object. Each filter receives a copy of the request object and can choose to modify it before passing it along to the next filter. The process works like a stack with the actual servlet as the top of the stack. As each object processes the request, the object is popped off the stack, and the previous filter is given the response from the last object. This process allows the filter to see the answer on the way back and modify the response content before sending it to the client.

The problem with this architecture is that any business logic that requires data not in the original request or the original response needs to query the database directly. To enforce the secure ACID transactional guarantees the filter and the servlet need to share a JDBC connection. It is possible to incorporate the filter's SQL query into the Servlet's database transaction. Unfortunately, there is not a standard solution in the servlet specification. The lack of standardization leads developers to solutions that do not guarantee secure transactional properties.

4. Web-Service'S Output Manipulation

Often an organization business logic hook needs to modify the data returned by the web service. There are many possible ways a business logic hook can manipulate the data returned from the database. The following list provides examples of the various types of manipulations utilizing the New York Philharmonic example presented earlier.

1. *Modify an individual attribute in the return result.* An adjustment of the price of a particular seating location tuple would represent this type of customization. The modification may occur because of a particular discount for patrons who have donated at a high level.
2. *Remove elements from the return result.* The removal of a delivery method would be an example of this type of modification. An example is a particular discount requiring validation of the ticket user's age. The addition of the manual check should not allow print at home as a delivery option.
3. *Add elements to the return result.* The addition of another seating record to a transaction is an example of this type of modification. An example would be a handicapped seat may require two seats to be pulled to fit a wheelchair. A customization, which enforces this rule needs to consume the adjacent seat consumed by the web-service.
4. *Define new attributes in the return result.* The addition of an attribute to the seat tuples would represent this type of customization. An example of this is a flag added to a seat that a patron requires a hearing amplifier.
5. *Define new elements in the return result.* The definition of a new set of tuples would represent this type of modification. An example would be a tuple to store rewards for a customer that a customer earned in the transaction such as a bottle of wine.

5. Schemaless Web-Services

There are two main standards used for developing web services; SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). SOAP web-services have a well-defined schema for input and output to the web service. The schema is defined using a WSDL file having an XML definition for the input and output passed to and from the web-service [7]. REST web-services are less structured and do not have a standard for the definition of the schema for input and output.

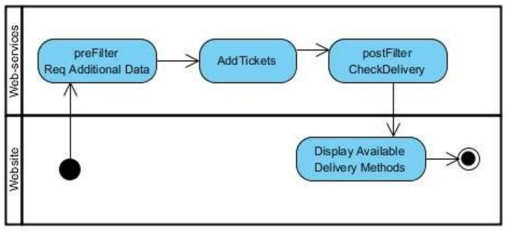
Web-services
Website
preFilter Req Additional Data
AddTickets
postFilter CheckDelivery
Display Available Delivery Methods

Figure 4. Workflow with Custom Filter

The first three methods described earlier for a business hook to manipulate data returned in the output of a web-service would work with either web-service technology. Only, the first three methods work with SOAP. They will work with SOAP because these do not change the output schema. They either suppress elements

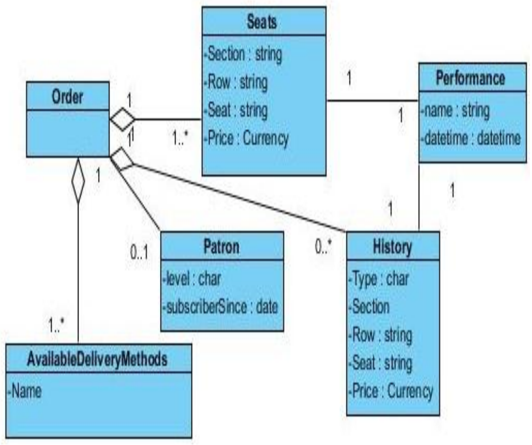

Figure 3. New SOAP to CRUD Mapping

or modify already defined attributes. The last two methods require changing the output schema. This change is more difficult with SOAP web-services because the WSDL file will change with each additional business logic hook in the customized pipeline.

For this research project, we did not need to add new data to the response sent back to the client. We needed to use additional data to suppress the elements returned to the client. In previous research, the *Buddy System* communicated with the client via SOAP requests [8]. The dispatcher would then translate the SOAP requests into a set of CRUD (create, read, update and delete) operations on basic objects. For these reasons, we decided to stay with SOAP web-services.

6. Soap Filters

We added a feature to the buddy system to allow a customized integration point to specify additional data in the SOAP to CRUD mapping. Figure 3 shows the UML of the modified "Add-Tickets" output. New tables are added to allow business rule filtering based on patron attributes and transactional history.

The additional data request is distributed with the original set of CRUD operations. The inclusion of the original CRUD with the customized CRUD allows the post-Filter to perform the suppression using transactional data that is guaranteed to be correct. Using the filter architecture also enables the elimination of the second round trip for the customized web service. Figure 4 shows the modified workflow using the pre and post filters to modify the CRUD data requested by the transaction.

On startup, the dispatcher creates a precedence graph based on the semantics of the XMI data (XML representation of the model) that comes from a UML model of the web service [8]. This model allowed the dispatcher to know the low-level CRUD operations of each web service. We added a configuration setting to control the pipe and filter architecture. Each filter can add CRUD semantics by including an XMI file containing at a minimum one of the classes from the previous filter or the original model. The dispatcher will merge the models adding the new relationships and attributes required by the business hook. There is no way to remove a relationship or an attribute. The architecture supports an unlimited number of filters in the pipeline to each web-service.

A limitation of our current work is there is currently no way to change the cardinality of a relationship. This restriction exists because the dispatcher just extends the original model instead of replacing components of the model. For example, if the original model or a previous filter had a one-to-many relationship between an ORDER class and a SEATS class there is no way for a business customization to change the mapping. This sort of change would require a change to the database schema that maps to the original model of the web-service.

7. Partitioning of Customization Data

Implementation time customizations can be deployed in any tier of the enterprise application. At a minimum, the partitioning layers include the view, business logic, and data tiers. In the *Buddy System,* we provide three integration points where data can be used.

New data classes can be added to the clusters (data tier) by adding the underlying database tables and relating them to current tables via the UML diagram as described in the previous section. UML design tools provide functionality output the matching SQL create table statements from the UML class diagrams. This generated SQL can be applied in the underlying relational database at each cluster.

The business logic tier incorporates the soap filters added to the SOAP web-service pipeline. The SOAP filter can pass data writes, updates and consumption to the data tier. The SOAP filter can request data for business logic built into the filter or to be passed along to the view tier.

View customizations are incorporated into the client. The client can be an HTML/CSS/JavaScript client, native mobile application or a fat client (native operating system application). Customizations in this partition can use original data returned from the web-service, additional data (custom or out of the box) or aggregate data returned by a business rule.

8. Service DATA vs. PIPE DATA

In the original functionality for course grained web-services in the *Buddy System,* the UML schema was used to map data used by the web-service [8]. The knowledge of the data read and created by a web-service allowed the dispatcher to map from the course-grained web service to the fine grained CRUD operations required by the dispatcher algorithm [3].

The business rule pipe and filter architecture needs a way to distinguish between the data read and written by the web service and the data required down the business filter pipeline. The *Buddy System* pulls semantics for the coarse-grained web services from the matching UML activity and UML class diagrams.

UML provides an extensibility mechanism that allows the designer to add new semantics to the model. A stereotype is one of the three types of extensibility mechanisms in the UML that enables the developer to extend the vocabulary of UML. This extension is done to represent new model elements [17]. The *Buddy System* uses stereotypes to know the data read and data written by the web-service. We provide a profile to designers that combine all the stereotypes into a single package.

We extended the stereotypes available in the profile to include the following:

1. Data Read By Service – This stereotype represents the original *Read* stereotype that is used to mark UML classes that the web service will read in the transaction.
2. Data Written By Service – This stereotype represents the original *Write* stereotype that is used to mark UML classes that the web service will write in the transaction.
3. Data Consumed By Pipeline – This category represents a pair of stereotypes. One stereotype is an attribute stereotype to mark the capacity attribute. The second stereo type is the consumption journal class. These stereotypes consume a limited resource by a customized business filter. To guarantee the secure transactional properties, the sterotype includes the use in a single database transaction with the original web-service. The consumption log can be distributed using the capacity constraint feature provided by the *Buddy System* [8]. An example of this type of stereotype is a customization that consumes a limited resource such as a ticket or an inventory item. For example, imagine a customization to enable an organization to gave away a ticket to a concert if the patron purchases two tickets. The original web service would consume two of the seats, but the customization would need to consume one in the same transaction.
4. Data Written By Pipeline – This stereotype represents new tuples written as part of the

transaction. The customization would set the values of the attributes in the tuples. The business rule sends a collection to the dispatcher, representing the new tuples. An example of this type of customization is one where a log is written to the transaction table representing the data in the transaction. For instance, image, you want to have a printing record of tickets printed in the transaction.

5. Tuples Read By Pipeline – This stereotype represents data required by the business customization on an individual tuple level. The stereo type is on the class level. The business rule can pass the selection filter to the dispatcher to limit the tuples retuned from the class.
6. Aggregates Read By Pipeline - This stereotype represents data that required by the business customization in a summary level. Each attribute, which is part of the aggregation, is marked with the stereotype. The dispatcher will return the data to the business rule for further processing with one tuple per set of aggregate attributes. There are also attribute stereo types that represent the aggregate function of (sum, average or count). As in the previous stereotype, a filter can be passed by a business rule to limit the tuples included in the aggregation. An example of a customized business rule that would use this stereotype is a yield management policy. The rule would apply a change to the pricing of an individual product as its sales fluctuate. The customization would need to know the quantity of sales at different price points.
7. Data Updated By Pipeline – This stereotype represents a set of attributes that have their value updated. As in the previous two stereotypes, a filter can be passed by a business rule to limit the tuples that processed in the transaction. An example of this type of customization would be to have a post service filter mark a set of tickets as printed.

9. Additional Data Filter Activity

The Buddy System was designed for deployment in environments with high concurrency. It is best to limit data operations to pre-service business filters to maintain the high availability of the system. A single set of data operations allows the dispatcher to distribute one request and release all locks. Several of the stereotypes allow new data in a single data activity including Consume, Write and Update.

Unfortunately, some business customizations require post-service business filter data updates. In this case, there is a request attribute that allows the dispatcher to hold locks that span more than one request. These locks allow the business rule to pass an additional data request to the dispatcher in the same database transaction.

10. Post Service Rollback

A pre-service filter can stop the pipeline processing before the request reaches the original web-service. If a business rule needs to make the decision of rolling back the transaction after the initial web-service has executed, it must inform the dispatcher in advance. As in the additional data request, this is highly discouraged as it requires the clusters to maintain open transactions.

11. Business Rule Access Control

Previous versions of the *Buddy System* shared a single credential among all clients. There is an assumption that the clusters will use the relational database's role based access control (RBAC) to limit access to tables, columns and tuples in the underlying database. The credentials used by the web-services are kept private from the client. The web-service developer limits access to the underlying data through the web service interface.

The addition of the capability to request pipeline data for a customized business rule creates a new vulnerability where a user without proper credentials can use the web services credentials to access private data. To mitigate the vulnerability, we added a new role based access control that allows assignment of permissions to business rules.

Table 1 shows an example RBAC table for an individual web-service. Each row represents the stereotype functionality available to a customized rule. There is an additional row to represent the ability for a business rule to keep a connection open to enable the rollback for further database operations. The keep alive represent a significant vulnerability as it can allow a plug-in to cause deadlocks in the system. On startup, the dispatcher will check the RBAC table against the business rule to ensure the business rule has the proper authorization. The business rule will be disabled if it does not pass the RBAC table.

The RBAC table is available at the class or attributes level. Access control on the attribute level gives the implementer more granular control over the security. The RBAC table can be placed at the dispatcher or the web-service. Setting the security at the web-service is helpful in situations where the web-services are in a separate domain of control from the dispatcher. The challenge with setting the RBAC table at the web-services is ensuring consistency in security across the web-service farm.

Table 1. Example RBAC for Web Service

Privilege	Class A	Class B	Class C
Consume	X	X	X
Read	X	X	X
Aggregate			
Write		X	
Update	X		X
Keep Alive			

12. Algorithm Correctness

Theorem 1: The Dispatcher Service Request Architecture guarantees one-copy serializability for a transaction consisting of the union of the original web-service request and the set of business rules CRUD (Create, Read, Update, Delete) operations.

Proof:
Claim 1: H is the graph of the transactions produced by the *Buddy System*. H is one-copy serializable if the following three conditions hold:

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

Proof of Claim 1:
For a transaction to be one-copy serializable, there must not exist a cycle among the committed transactions in the serialization graph of H [9]. 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 three 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 the same pair of clusters receives any potential cycle. Local scheduling on the cluster ensures serializability. So if these conditions hold there is a guarantee of one-copy serializability.

To show that the *Buddy System* architecture with business rules guarantees one-copy serializability, assume, by contradiction that H is not one-copy serializable. Then, one of the three conditions must not be valid. The architecture guarantees conditions 2 and 3. The only remaining possible violation is condition one. 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) are 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) – A write set/write set overlap happens when transaction T_i reads an element written by transaction T_j. Simultaneouly transaction T_i writes an element read by transaction T_j. When the dispatcher sees this pattern it will send the requests to the same cluster or queue the request 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. □

13. Empirical Results

We modeled a performing arts center with blocks of current users ranging in value from one hundred to one thousand. Each user had identical credentials so that the server processing load would be the same for each user. We compare the model against three different architectures;

1. Client is calling a SOAP Web-Service in sequence (Figure 2) using JavaEE on a Tomcat server and a single MySQL database. There are two issues with this architecture; the data used for the business hook is outside the transaction and therefore not guaranteed to be correct. Also, the additional latency for the extra round trip significantly reduces the availability of the service

2. Client is calling a SOAP Web-Service with business logic hook (Figure 4) using JavaEE on a Tomcat server and a single MySQL database. A JavaEE filter is used to intercept the request. The filter sends a second database call to retrieve the data needed to make a decision on which delivery methods to suppress in the results. The issue with this architecture is that the data used for the business hook is outside the transaction and therefore not guaranteed to be correct.
3. Client is calling a SOAP Web-Service with business logic hook (Figure 4) using the Buddy System Architecture with four clusters. This architecture guarantees the correctness of the data and removes the latency of the second round trip to the server.

With the Buddy System, higher availability was achieved by reducing the number of round trips from client to server and by distributing the inserts to all four clusters while guaranteeing the consistency. Figure 5 shows the empirical results.

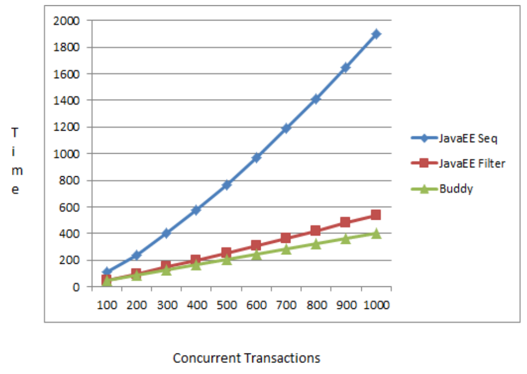

Figure 5. Empirical Results

14. Related Work

Developers have implemented business rules in software systems since the first software package was developed. Most research has been around developing expert systems to process large business rule trees efficiently. Charles Forgy developed the Rete Algorithm which has become the standard algorithm used in business rule engines [10]. Forgy has published several variations on the Rete algorithm over the past few decades.

Business rule engines have sprung up to allow the separation of business rules from the core application code. These system are designed to allow the end users to change the business rules freely without changing the original application code. In 2007, International Data Corporation implemented a survey where they asked 'How often do you want to customize the business rules in your software?' Ninety percent of the respondents reported that they changed their business rules annually or more frequently. Thirty-four percent of the respondents reported that they changed their business rules monthly [11].

Researchers have studied replication for decades. There are two main types of replication in RDMS: strict and lazy replication. Strict replication includes all nodes in a transaction synchronously and lazy does so asynchronously. Recent research can be grouped into three goals. First, to increase the availability with strict replication. Second, to increase consistency with lazy replication. Finally, to use a hybrid approach to increasing availability. Our previous work, the *Buddy System*, increases availability while providing 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 to provide one-copy serializability (1SR) eventually. Transactions on traditional replicated databases can 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 strict or eager update propagation protocols; approaches that allow the propagation of the update asychronouly, after the transaction 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 [9] for an overview.) More recently, Snapshot Isolation (SI) [12, 13] 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 and Daudjee et al. propose frameworks for master-slave, lazy-replication updates that provide consistency guarantees [14] [15]. The approaches require all writes to be performed on the master replica. Updates are propagated to the other sites after the commit of updating transaction. Their framework provides a distributed serializable schedule where there is no guarantee of the ordering of updates.

The approach proposed by Daudjee et al. provides multi-version serializability where different versions of data are available for read requests during the period that replication has not completed.

- Hybrid Approach: Jajodia and Mutchler and Long et al. both define forms of hybrid replication that reduce the requirement that all replicas participate in eager update propagation [16] [17]. The proposed methods aim to increase the 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. [18] 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 describe a hybrid system, called Cassandra, which was built by Facebook to handle their inbox search [19]. Cassandra allows a configuration parameter that controls the number of nodes that are part of the synchronous update. The Cassandra system allows configuration of the nodes chosen for synchronous inclusion cross data center boundaries to increase durability and availability.

- Buddy System: In our previous work, we provide an architecture and algorithms that address three problems experienced in high volumne web service transactions [3, 4, 8]. First, the risk of losing committed transactional data in case of a site failure. Second, contention caused by a high volume of concurrent transactions consuming limited items. Finially, contention caused by a high volume of read requests. We called this system the Buddy System because it used pairs of clusters to update all transactions synchronously. 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.

To support the above components, an intelligent dispatcher is placed in front of all clusters. The dispatcher operated at the OSI Network level 7. This 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 selection is based on versioning history. If a version is in progress and the request is modifying the data, then the dispatcher chooses the set containing the same pair currently executing the other modify 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 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.

15. Conclusion

In this paper, we propose an extension to the buddy system to provide a pipe and filter architecture for web service transactions that guarantees correctness of data while increasing availability of other architectures. Our solution is based on extending UML with stereotypes to embed CRUD and data element semantics into the model for business logic hooks. The dispatcher can then extract the semantics from the model and distribute the requests to clusters including the additional data elements required to processes the customized business logic in the hook. The dispatcher is then able to distribute the requests to any cluster that is qualified to handle the original transactional data and the custom logic data. A limitation of our work is that we currently only support SOAP web services which limits the types of business logic hooks you can develop.

16. 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, "High Volume Web Service Resource Consumption," in Internet Technology and Secured Transactions, 2012. ICITST 2012, London, UK, 2012.

[4] 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.

[5] 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.

[6] Oracle Corporation, "JSR-000315 Java Servlet 3.0 Final Release," [Online]. Available:

http://download.oracle.com/otndocs/jcp/servlet-3.0-fr-eval-oth-JSpec/. [Accessed 18 08 2014].

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

[8] A. Olmsted and C. Farkas, "Coarse-Grained Web Service Availability, Consistency and Durability," in IEEE International Conference on Web Services, San Jose, CA, 2013.
[9] M. T. Ozsu and P. Valduriez, Principles of Distributed Database Systems, 3rd ed., Springer, 2011.

[10] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," Artificial Intelligence, vol. 19, no. 1, p. 17–37, 1982.

[11] Ceiton Technologies, "Introducing Workflow," [Online]. Available: http://ceiton.com/CMS/EN/workflow/introduction.html#Customization. [Accessed 15 09 2014].

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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