Transaction Manager Failover: A Case Study Using JBOSS Application Server

A. I. Kistijantoro, G. Morgan, S. K. Shrivastava

School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
{A.I.Kistijantoro, Graham.Morgan, S.K.Shrivastava}@ncl.ac.uk

Abstract. This paper describes, for the case of Enterprise Java Bean components and JBoss application server, how replication for availability can be supported to tolerate application server/transaction manager failures. Replicating the state associated with the progression of a transaction (i.e., which phase of two-phase commit is enacted and the transactional resources involved) provides an opportunity to continue a transaction using a backup transaction manager if the transaction manager of the primary fails. Existing application servers do not support this functionality. The paper discusses the technical issues involved and shows how a solution can be engineered.

Keywords: Availability, application servers, components, Enterprise Java Beans, fault tolerance, middleware, replication, transactions

1. Introduction

Three-tier middleware architecture is commonly used for hosting large-scale distributed applications. Typically the application is decomposed into three layers: front-end, middle tier and back-end. Front-end ('Web server') is responsible for handling user interactions and acts as a client of the middle tier, while back-end provides storage facilities for applications. Middle tier ('Application Server') is usually the place where all computations are performed, so this layer provides middleware services for transactions, security and so forth. The benefit of this architecture is that it allows flexible configuration such as partitioning and clustering for improved performance and scalability. Furthermore, availability measures, such as replication, can be introduced in each tier in an application specific manner. In this paper we concentrate on application server (middle tier) replication. Data as well as object replication techniques have been studied extensively in the literature, so our task is not to invent new replication techniques, but to investigate how existing techniques can be migrated to middle tier.

One important concept related to availability measures is that of *exactly once transaction* or *exactly once execution* [1,2]. The concept is particularly relevant in web-based e-services where the system must guarantee exactly once execution of user requests despite system failures. Problems arise as the clients in such systems are

usually not transactional, thus they are not part of the recovery guarantee provided by the underlying transaction processing systems that support the web-based e-services. When failures occur, clients often do not know whether their requests have been processed or not. Resubmitting the requests may result in duplication, and on the other hand it is also possible the requests have not been processed at all. This problem can be handled by replicating the application server to achieve availability. As we discuss in the next section, while existing application servers for Enterprise Java Bean (EJB) components do use replication, they do not adequately support exactly once transaction capability. For this reason, there has been much recent research works on replication for supporting exactly once transactions over commonly used application servers. However, implementation work reported so far has dealt with transactions that update a single database only, so do not require two-phase commit.

In this paper we go a step further and present design, implementation and performance evaluation of a middle tier replication scheme for multi-database transactions using a widely deployed application server (JBoss). We describe how a backup transaction manager can complete two-phase commit for transactions that would otherwise be blocked. The paper discusses the technical issues involved and shows how a solution can be engineered. Our case study can be used by other designers intending to enhance application servers in a similar manner.

2. Related Work

The classic text [3] discusses replicated data management techniques that go hand in hand with transactions. Object replication using group communication, originally developed in the ISIS system [4], has been studied extensively [e.g., 5]. The interplay between replication and exactly once execution within the context of multi-tier architectures is examined in [6], whilst [7] describes how replication and transactions can be incorporated in three-tier CORBA architecture. The approach of using a backup transaction monitor was implemented as early as 1980 in the SDD-1 distributed database system [8]; another implementation is reported in [9]. A replicated transaction coordinator to provide non-blocking commit service has also been described in [10]. Our paper deals with the case of replicating transaction managers in the context of standards compliant Java application servers (J2EE servers).

There are several studies that deal with replication of application servers as a mechanism to improve availability [1,2,11,12]. In [2], the authors precisely describe the concept of exactly once transaction (*e-transaction*) and develop server replication mechanisms; their model assumes *stateless application servers* (no session state is maintained by servers) that can access multiple databases. Their algorithm handles the transaction commitment blocking problem by making the backup server take on the role of transaction coordinator. As their model limits the application servers to be stateless, the solution cannot be directly implemented on stateful server architectures such as J2EE servers.

The approach by Wu, Kemme et al in [12] specifically addressed the replication of J2EE application servers, where components may possess session state in addition to persistent state stored on a single database. The approach assumes that an active

transaction is always aborted by the database whenever an application server crashes. Therefore, it uses a mechanism similar to testable transaction abstraction developed in [1], and on failover, the backup server uses this mechanism to find out the outcomes of transactions performed on the crashed primary. Our approach assumes the more general case of access to multiple databases; hence two phase commitment (2PC) is necessary. Application server failures that occur during the 2PC process do not always cause abortion of active transactions, since the backup transaction manager can complete the commit process.

JBoss clustering [13] uses session replication to enable failover of a component processing on one node to another. The approach targets load balancing among replicas and it allows each replica handles different client sessions. The state of a session is propagated to backup after the computation finish. When a server crashes, all sessions that it hosts can be migrated and continued on another server, regardless the outcome of formerly active transactions on the crashed server, which may lead to inconsistencies.

Exactly once transaction execution can also be implemented by making the client transactional, and on web-based e-services, this can be done by making the browser as a resource which can be controlled by the resource manager from the server side, as shown in [14,15]. One can also employ transactional queue [16]. In this way, user requests are kept in a queue that are protected by transactions, and clients submit requests and retrieve results from the queue as separate transactions. As the result, three transactions are required for processing each client requests and developers must construct their application so that no state is kept in the application servers between successive requests from clients. The approach presented in [17] guarantees exactly once execution on internet-based e-services by employing message logging. The authors describe which messages require logging, and how to do the recovery on the application servers. The approach addresses stateful application servers with single database processing without replicating the application servers. The table below summarizes the differences between the various approaches; concentrating on exactly once transactions as such approaches consider similar requirements to our work.

Aspects	Transactional queue	Trans. client	Message logging [17]	e-transaction	Wu and Kemme	Our approach
App. server replication	No	No	No	Yes	Yes	Yes
Transactional client	Not required	Required	Not required	Not required	Not required	Not required
Stateful server	Supported	Supported	Supported	Not supported	Supported	Supported
Platform	TP monitors	Web	Web	Custom	J2EE	J2EE
Multi database	Supported	Supported	Not supported	Supported	Not supported	Supported

Table: exactly once transaction solutions

For the sake of completeness, we point out here that replication approaches for the third tier (back-end, database tier) that work with application servers have also been investigated by many researchers (see [18,19]).

3. Background

We assume the reader is familiar with EJB component model and how transactions are used through containers in J2EE servers (background details are available in the more detailed version of this paper [20]). We only provide a brief description of how services are integrated into JBoss via *interceptors*, *management beans* (MBeans) and *Java Management Extensions* JMX and then describe how this approach is used to implement transactions in JBoss middleware.

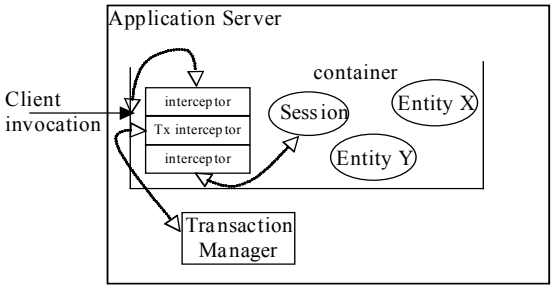


Figure 1 – Augmenting application server with transactions.

In JBoss invocations pass through a series of interceptors within a container. These interceptors enable the integration of additional services into a container to support EJB execution (e.g., security, transactions), with the final interceptor in the incoming chain of interceptors handling method invocation on the actual EJB itself. Services may be added to JBoss via MBeans. An MBean exposes a management interface, attributes and operations while adhering to the JMX specification and may be made available for use via the standard object location services in JBoss (JNDI). JMX provides an API for management and monitoring of resources, including remote access, so a remote application can manage and monitor applications.

JBoss implements transactions with the aid of *tx interceptors* and the transaction manager (figure 1). The tx interceptor inspects an incoming invocation with the aid of the transaction manager and determines the appropriate settings for the transaction context before the receiving bean processes the invocation. A transaction context is used to identify a transaction and determines the transaction an invocation belongs to (in particular, the thread of execution associated to an invocation), allowing transactional mechanisms to be enacted in line with invocation processing on transactional objects (e.g., mark for rollback, throw exception, commit).

4. Model

Our approach to component replication is based on a passive replication scheme, in that a primary services all client requests with a backup assuming the responsibility of servicing client requests when a primary fails. Crash failures of servers are assumed. In a configuration of server machines where the failure of a server can be detected with accuracy, a minimum of f+1 replicas are needed to tolerate up to f server failures; such a scheme can be engineered for a well managed cluster of machines connected by a high bandwidth LAN. Configurations where accurate failure detection is not possible

(e.g., the servers are widely distributed with arbitrary inter-communication delays), a minimum of 2f+1 replicas are needed. Performance evaluation that we present in section 6 are for a LAN configuration.

Recovery measures undertaken vary depending upon where the primary fails within a client session: (1) during non-transactional invocation phase, (2) during transactional phase. As entity beans access and change persistent state, the time taken to execute application logic via entity beans is longer than enacting the same logic using session beans. The reason for this is two fold: (1) the high cost of retrieving state on entity bean activation and writing state on entity bean deactivation; (2) the transactional management associated to persistent state updates. The structuring of an application to minimize the use of entity beans (and transactions) to speed up execution times is commonplace. This approach to development leads to scenarios in which a client enacts a "session" (a series of related invocations) on an application server, with the majority of invocations handled by session beans. Transactional manipulation of persistent state via entity beans is usually left to the last steps of processing in a client's session. The sequence diagram in figure 2 describes the style of interaction our model assumes. We are only showing application level logic invocations (as encoded in EJBs) in our diagram, therefore, we do not show the transaction manager and associated databases. The invocations that occur within a transaction are shown in the shaded area. As mentioned earlier, we assume a client is not part of the transaction.

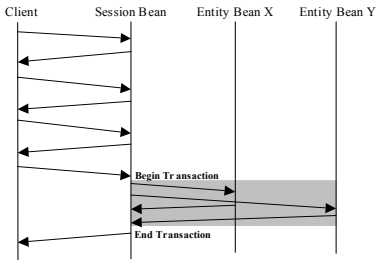


Figure 2 – Interactions between beans and client.

We assume a single stateful session bean is used to present a single interface for a client during a session. The creation and destruction of a stateful session bean by a client delimits the start and end of a session (i.e., lifetime of stateful session bean). We assume the existence of a single transaction during the handling of the last client invocation and such a transaction is initiated by the stateful session bean and involves one or more entity beans. The transaction is container managed and is scoped by this last method invocation.

Failure of the primary during a session will result in a backup assuming responsibility for continuing the session. This may require the replaying of the last invocation sent by a client if state changes and return parameters associated to the last invocation were not recorded at backups. If state changes and parameters were recorded then the backup will reply with the appropriate parameters. During the transactional phase the transaction may be completed at the backup if the commit stage had been reached by the primary and computation has finished between the entity beans. The backup will be required to replay the transaction if failure occurs during transactional computation.

5. JBoss Implementation

Figure 3 shows the interceptors and associated services that implement our replication scheme in the JBoss application server. The interceptors perform the following tasks: *retry interceptor* – identifies if a client request is a duplicate and handles duplicates appropriately; *txinspector interceptor* – determines how to handle invocations that are associated to transactions; *txinterceptor* – interacts with transaction manager to enable transactional invocations (unaltered existing interceptor shown for completeness); *replica interceptor* – ensures state changes associated with a completed invocation are propagated to backups.

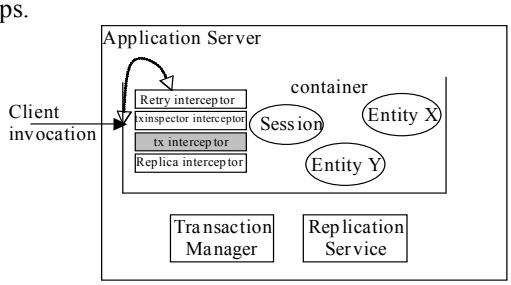


Figure 3 – Augmenting application server with replication service.

The txinterceptor together with the transaction manager accommodates transactions within the application server. The replication service supports inter-replica consistency and consensus services via the use of JGroups [21]. The replication service, retry interceptor, txinspector interceptor and the replica interceptor, implements our replication scheme.

Replication logic at the server side makes use of four persistent logs that are maintained by the replication service: (i) current primary and backup configuration (group log), (ii) most recent state of session bean together with the last parameters sent back as a reply to a client invocation (bean log), (iii) invocation timestamp associated to most recent session bean state (timestamp log), (iv) state related to the progress of a transaction (transaction log). The replication service uses a single group via the JGroups service to ensure these logs are consistent across replicas.

We skip over the details of how a client side proxy has been enhanced with retry ability to backups as well as how session state checkpointing to backups is performed using group communication, as these techniques are well known (details can be found in [20]); instead we concentrate below on transaction failover management.

5.1. Transaction failover management

We assume container managed transaction demarcation. Via this approach to managing transactions the application developer specifies the transaction demarcation for each method via the transaction attribute in a bean deployment descriptor. Using this attribute a container decides how a transaction is to be handled. For example, if a new transaction has to be created for an invocation, or to process the invocation as part of an existing transaction (i.e., the transaction was started earlier in the execution chain). Based on this mechanism, a single invocation of a method can be: a single transaction unit (a transaction starts at the beginning of the invocation and ends at the

end of the invocation), a part of a transaction unit originated from other invocation, or non transactional (e.g. the container can suspend a transaction prior to executing a method, and resume the transaction afterwards). We assume that the processing of an invocation may involve one or more beans (both session beans and entity beans) and may accesses one or more databases, requiring two phase commitment.

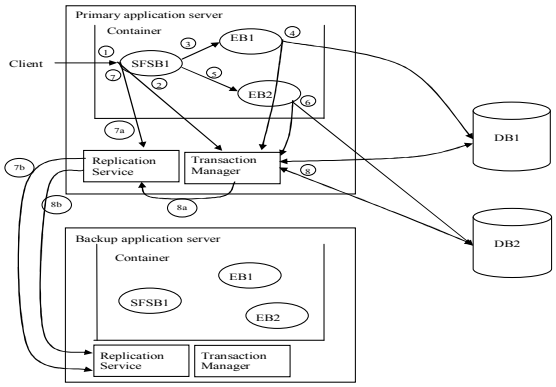


Figure 4 - A typical interaction for a transaction processing in EJB

Figure 4 illustrates the execution of a typical transaction (for brevity, we have not shown resource adaptors). We shall use this example as a comparison to highlight the enhancements we have provided to handle transaction failover (this example is represents the shaded area shown in figure 4). SFSB stands for a stateful session bean and EB stands for an entity bean. All methods on the beans have a *Required* tag as their transaction attribute, indicating to the container that they must be executed within a transaction. The invocation from the client initially does not contain a transaction context. At (1), a client invokes a method on a stateful session bean SFSB1. The container (e.g. the tx interceptor on JBoss app server) determines that the invocation requires a transaction and calls the transaction manager to create a transaction T1 for this invocation (2). The container proceeds to attach a transaction context for T1 to the invocation. The container does not have to create a new transaction for nested invocations (3) and (5). The invocation on EB1 requires access to a database DB1 (4) and at this point, the container registers DB1 to the transaction manager as a resource associated with T1. The same process happens at (6) where the container registers DB2 to be associated with T1. After the computation on SFSB1, EB1 and EB2 finishes, before returning the result to the client, the container completes the transaction by instructing the transaction manager to commit T1. The transaction manager then performs two phase commit with all resources associated with T1 (8) (not shown in detail here).

Our transaction failover mechanisms are performed at point (7) and (8). A multicast of the state update of all involved session beans together with the result parameter, the transaction id and information on all resources involved is made (7a) and (7b) to all backup replicas. If the primary fails after this point, a backup will try to finish the commit process. At point (8), a multicast of the decision taken by the transaction manager is made to all backup replica transaction managers via the replication service (8a) and (8b). If the primary fails after this point, a backup will try to finish the commit process according to the decision that has been taken by the failed primary.

A number of technical challenges needed to be overcome to provide an engineered solution. However, for brevity we do not go into such details here; the interested reader is referred to [20].

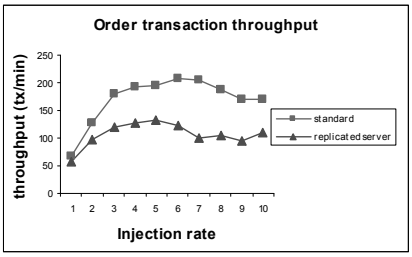
6. Experimental Evaluation

We carried out our experiments on the following configurations: (1) Single application server with no replication; (2) Two application server replicas with transaction failover. Both configurations use two databases, as we want to conduct experiments for distributed transaction setting.

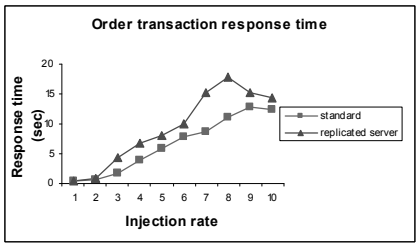
The application server used was JBoss 3.2.5. The database used was Oracle 9i release 2 (9.2.0.1.0) [20]. All clients, application servers and database servers were deployed using machines of a similar configuration (Pentium IV 2.8 GHz PC with 2048MB of RAM running Fedora Core 4). The LAN used for the experiments was a 100 Mbit Ethernet. ECperf [22] was used as the demonstration application in our experiments. ECperf is a benchmark application provided by Sun to enable vendors to measure the performance of their J2EE products. For our experiments, we configured the ECperf application to use two databases instead of just a single database (as is the default configuration).

Two experiments are performed. First, we measure the overhead of our replication scheme introduces into application performance. The ECperf driver was configured to run each experiment with 10 different injection rates (1 though 10 inclusive). At each of these increments a record of the overall throughput (transactions per minute) for both order entry and manufacturing applications is taken. The injection rate relates to the order entry and manufacturer requests generated per second. Due to the complexity of the system the relationship between injection rate and resulted transactions is not straightforward. The second experiment measures how our replicated algorithm performs in the presence of failures. In this experiment we ran the ECperf benchmark for 20 minutes, and the throughput of the system every 30 seconds is recorded. After the first 12 minutes, we kill the primary server to force the system to failover to the backup server.

Figure 5 presents two graphs that describe the throughput and response time of the ECperf applications; figure 5(i) identifies the throughput for the entry order system, figure 5(ii) identifies the response time for the entry order system. On first inspection we see that our replication scheme lowers the overall throughput of the system. This is to be expected as additional processing resources are required to maintain state consistency across components on a backup server.



(i) throughput for entry order app.



(ii) response time for entry order app.

Figure 5 – Performance figures.

Figure 6 presents a graph that describes the throughput of our system and the standard implementation over the time of the benchmark. After 720 seconds running (12 minutes), we crash the primary server. When no replication is present the failure of the application server results in throughput decreasing to zero, as there is no backup to continue the computation. When replication is present performance drops when failure of the primary is initiated. However, the backup assumes the role of the primary allowing for throughput to rise again. An interesting observation is that throughput on the new primary is higher than it was on the old primary. This may be explained by the fact that only one server exists and no replication is taking place. The initial peak in throughput may also be explained by the completion of transactions that started on the old primary but finish on the new primary. This adds an additional load above and beyond the regular load generated by injection rates.

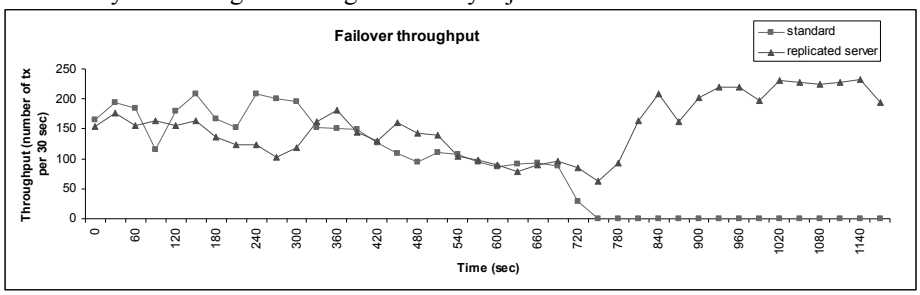


Figure 6 – Performance figures under a failure.

The experiments show that our replication scheme does not incur high overhead compared to a non replicated system, and is able to perform quick failover when the primary crashes.

7. Concluding Remarks

We have presented a practical solution to the problem of incorporating availability through replication in application servers, specifically for the general case of multi-database transactions. Although our design and implementation have been for a specific component model (EJBs) and application server (JBoss), the ideas can be applied to other application servers. Thus our case study can be used by other designers intending to enhance application servers in a similar manner.

Acknowledgements

This work is funded by UK Engineering and Physical Sciences Research Council -Grant No. GR/S63199/01, “Trusted Coordination in Dynamic Virtual Organisations”, and Platform Grant No. EP/D037743/1, “Networked Computing in Inter-organisation Settings”; Kistijantoro's work is funded by QUE Project Batch III, Institute Teknologi Bandung, Indonesia.

References

1. S. Frolund and R. Guerraoui, “A pragmatic implementation of e-transactions”, 19th IEEE Symposium on Reliable Distributed Systems, SRDS 2000.
2. S. Frolund and R. Guerraoui, “e-transactions: End-to-end reliability for three-tier architectures”, IEEE Transactions on Software Engineering 28(4): 378-395, 2002.
3. P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
4. K. Birman , "The process group approach to reliable computing", CACM , 36, 12, pp. 37-53, December 1993.
5. P. Felber, R. Guerraoui, and A. Schiper, “The implementation of a CORBA object group service”, Theory and Practice of Object Systems, 4(2), 1998, pp. 93-105.
6. B. Kemme, R. Jimenez-Peris et al, “Exactly once Interaction in a Multi-tier Architecture”, VLDB Conf. Trondheim, Norway. Aug. 2005.
7. W. Zhao, et al., ”Unification of Transactions and Replication in Three-tier Architectures Based on CORBA”, IEEE transactions on Dependable and Secure Computing, Vol. 2, No. 1, 20- 33, 2005.
8. M. Hammer and D. Shipman, "Reliability mechanisms for SDD-1: A system for distributed databases" ACM Transactions on Database Systems 5(4): 431--466, 1980.
9. P.K. Reddy and M. Kitsuregawa, “Reducing the blocking in two-phase commit protocol employing backup sites”, Cooperative Information Systems (CoopIS'98), August 1998.
10. Jiménez-Peris, R., M. Patiño-Martínez, et al, “A Low-Latency Non-blocking Commit Service”, 15th International Conference on Distributed Computing (DISC), October 2001.
11. Ozalp Babaoglu et al, “A Framework for Prototyping J2EE Replication Algorithms”, Int. Symposium on Distributed Objects and Applications (DOA), Agia Napa, October 2004.
12. H. Wu, B. Kemme, V. Maverick, “Eager Replication for Stateful J2EE Servers”, Int. Symposium on Distributed Objects and Applications (DOA), Cyprus, October 2004.
13. S. Labourey and B. Burke, “ JBoss Clustering 2nd Edition”, 2002, www.jboss.org
14. M.C. Little and S K Shrivastava, “Integrating the Object Transaction Service with the Web”, Enterprise Distributed Object Computing Workshop (EDOC’98), pp. 194 – 205, November 1998.
15. M.C. Little and S K Shrivastava, “Java Transactions for the Internet”, Distributed Systems Engineering, 5 (4), December 1998, pp. 156-167.
16. P.A. Bernstein, M. Hsu, et al., “Implementing recoverable requests using queues”, ACM SIGMOD international conference on Management of data, 1990, Atlantic City, New Jersey.
17. R. Barga, D. Lomet, et al. ,"Recovery guarantees for Internet applications", ACM Trans. on Internet Tech. 4(3): 289-328, 2004.
18. A. I. Kistijantoro, et. al, “Component Replication in Distributed Systems: a Case study using Enterprise Java Beans”, 22nd IEEE Symposium on Reliable Distributed Systems, SRDS 2003
19. M. Patiño-Martínez, et. al, “Consistent Database Replication at the Middleware Level”, ACM Transactions on Computer Systems (TOCS). Volume 23, No. 4, 2005, pp 1-49.
20. A. I. Kistijantoro, et. al., “Transaction Manager Failover: A Case Study Using JBOSS Application Server”, Technical Report, School of Computing, Newcastle University, 2006.
21. B. Ban, “JavaGroups User’s Guide” http://www.javagroups.com
22. S. Subramanyam, “JSR 4: ECperf Benchmark Specification Java Community Process” http://www.jcp.org/en/jsr/detail?id=4