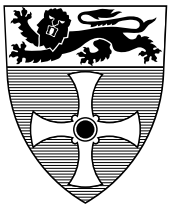
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Transaction Manager Failover: A Case Study Using JBOSS Application Server

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

TECHNICAL REPORT SERIES

No. CS-TR-975 June, 2006

TECHNICAL REPORT SERIES

No. CS-TR-975 June, 2006

Transaction Manager Failover: A Case Study Using JBOSS Application Server

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

Abstract

The extension of object-oriented middleware to component-oriented middleware is now commonplace, with many distributed applications structured as remote clients invoking services constructed from components. An advantage components offer over objects is that only the business logic of an application needs to be addressed by a programmer. An application server hosts components, managing supporting services to provide the execution environment for components. A transaction manager within an application server assumes responsibility for managing the execution of transactions. Failure of an application server instance could result in abortion of ongoing transactions that are being managed by the transaction manager on that server. 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.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

KISTIJANTORO, A. I., MORGAN, G., SHRIVASTAVA, S. K.

Transaction Manager Failover: A Case Study Using JBOSS Application Server
[By] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-975)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-975

Abstract

The extension of object-oriented middleware to component-oriented middleware is now commonplace, with many distributed applications structured as remote clients invoking services constructed from components. An advantage components offer over objects is that only the business logic of an application needs to be addressed by a programmer. An application server hosts components, managing supporting services to provide the execution environment for components. A transaction manager within an application server assumes responsibility for managing the execution of transactions. Failure of an application server instance could result in abortion of ongoing transactions that are being managed by the transaction manager on that server. 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.

About the author

Achmad Kistijantoro is a PhD student in the School of Computing Science at the University of Newcastle upon Tyne.

Graham Morgan is a lecturer in the School of Computing Science at the University of Newcastle upon Tyne. His interests are in the area of distributed applications, including web services, networked virtual environments, fault tolerance and group communications.

Santosh Shrivastava was appointed a Professor of Computing Science, University of Newcastle upon Tyne in 1986; he leads the Distributed Systems Research Group. He received his Ph.D. in computing science from Cambridge in 1975. His research interests are in the areas of distributed systems, fault tolerance and application of transaction and workflow technologies to e-commerce, virtual organisations and Grid-based systems. His group is well known as the developers of an innovative distributed transaction system, called Arjuna and a CORBA based dependable workflow system for the Internet. He has led and managed several European Union funded, multi-partner research projects, beginning with BROADCAST (1992) to a project on complex service provisioning on the Internet, TAPAS, that started in 2002. Together with his colleagues he set up a company in 1998 in Newcastle to productise Arjuna transaction and workflow technologies. Now based within the University campus, Arjuna Technologies is a centre of excellence in transaction technologies and is focusing on building products to support reliable Web Services-based applications. Close industry-university collaboration is guaranteed, and research projects on E-commerce platform and services have been initiated.

Suggested keywords

AVAILABILITY,
APPLICATION SERVERS,
COMPONENTS,
ENTERPRISE JAVA BEANS,
FAULT TOLERANCE,
MIDDLEWARE,
REPLICATION,
TRANSACTIONS

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

Abstract

The extension of object-oriented middleware to component-oriented middleware is now commonplace, with many distributed applications structured as remote clients invoking services constructed from components. An advantage components offer over objects is that only the business logic of an application needs to be addressed by a programmer. An application server hosts components, managing supporting services to provide the execution environment for components. A transaction manager within an application server assumes responsibility for managing the execution of transactions. Failure of an application server instance could result in abortion of ongoing transactions that are being managed by the transaction manager on that server. 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.

Key words

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

Contact Author:

Graham Morgan
graham.morgan@ncl.ac.uk

1. Introduction

Three-tier architecture is commonly used for hosting large-scale distributed applications. Typically the application is decomposed into three layers: front end, middle layer and back-end. Front-end ('Web server') is responsible for handling user interactions and acts as a client of the middle layer, while back-end provides storage facilities for applications. Middle layer ('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 can be introduced in each tier in an application specific manner. In this paper we concentrate on application server (middle tier) availability.

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 one 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; 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.

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., 18]. The interplay between replication and exactly once execution within the context of multi-tier architectures is examined in [19], whilst [20] 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 [5]; another implementation is reported in [6]. A replicated transaction coordinator to provide a non-blocking commit service has also been described in [21]. 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, 7, 8]. 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 [8] 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 [9] 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 to handle different client sessions. The state of a session is propagated to a backup after the computation has finished. When a server crashes, all sessions that it hosts can be migrated and continued on another server, regardless of 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 a resource which can be controlled by the resource manager from the server side, as shown in [10, 11]. One can also employ a transactional queue [12]. 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 a result, three transactions are required for processing each client request 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 [13] 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 described above.

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 (backend, database tier) that work with application servers have also been investigated by many researchers (see [14, 15]).

3. Background

In this section we provide background information on Enterprise Java Bean (EJB) components and transactions and how they are supported in the JBoss application server.

3.1. EJBs

Three types of EJBs have been specified in J2EE: (1) *Entity beans* represent and manipulate persistent data of an application, providing an object-oriented view of a data that is frequently stored in relational databases. (2) *Session beans* on the other hand do not use persistent data, and are instantiated on a per-client basis with an instance of a session bean available for use by only one client. A session bean may be *stateless* (does not maintain conversational state) or *stateful* (maintains conversational state). Conversational state is needed to share state information across multiple requests from a client. (3) *Message driven beans* provide asynchronous processing by acting as message listeners for Java Messaging Service (JMS).

A container is responsible for hosting components and ensuring that middleware services are made available to components at run time. Containers mediate all client/component interactions. An entity bean can either manage its state explicitly on a persistent store (bean managed persistence) or delegate it to the container (container managed persistence). All EJB types may participate in transactions. Like persistence, transactions can be bean managed or container managed. Use of container managed persistence and transactions are strongly recommended for entity beans; the remainder of this paper we assume container managed transactions are used.

EJBs present *home* and *remote* interfaces for use by clients. The home interface provides lifecycle services (e.g., create, destroy), and the remote interface allows clients to access the application logic supported by an EJB using method calls. Clients must first retrieve a reference to the home interface of the EJB which they wish to access. This is achieved via the *Java naming and*

directory interface (JNDI). The JNDI provides a naming service that allows clients to gain access to the home interface of the type of EJB they require. Once a reference to the home interface is gained, a client may instantiate instances of an EJB (gaining access to the remote interface).

3.2. EJBs in JBoss

In the J2EE specification *EJBHome* and *EJBObject* implement the home and remote interfaces respectively. In JBoss *dynamic proxies* aid in the EJBHome and EJBObject implementation and contribute to the overall aim of allowing the JBoss architecture to deploy components into a running JBoss application server. The dynamic proxies API allows a dynamic proxy class to implement an interface specified at runtime, a necessity if new components (presenting new interfaces) are to be introduced to a running application server (as Java is strongly typed this approach is required). From a client's perspective, a dynamic proxy is seen as implementing the interface they expose. Reflection may be used to determine which interfaces a dynamic proxy may support.

When EJBs are deployed a proxy factory is created that manages the creation of home and remote interface dynamic proxies. The home interfaces are bound into the JNDI. When a client contacts the JNDI to gain access to a home interface, the EJBHome dynamic proxy is downloaded to the client (the EJBHome dynamic proxy is serializable). Using this dynamic proxy interface a client may gain a reference to a remote interface associated with the home interface type. Gaining such a reference may require the creation of an EJB (this is the case for session beans) or the retrieval of an existing reference to a bean (only possible for entity beans as these may be shared amongst clients).

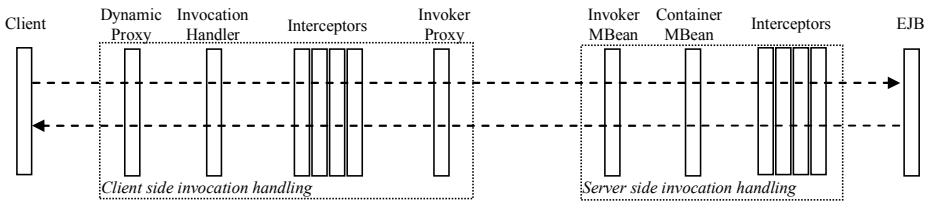


Figure 1 – Invocation handling in JBoss

Figure 1 provides an overview of the client and server side handling of an invocation in JBoss. When an invocation is retrieved from a client by the local proxy an attempt is made to identify if the client side may handle the invocation (i.e., server exists on the same *Java Virtual Machine*). This

prevents the unnecessary processing/latency overhead associated with remote invocation if invocations may be achieved on the same JVM. If remote invocation is required then the invocation is marshalled and forwarded to the appropriate container in the application server. The initial step in this remote invocation process is the marshalling of the invocation by the *invocation handler*, which in turn forwards the marshalled invocation through a series of client side interceptors to an *invoker proxy*. The interceptors may be used to provide additional services in a transparent manner to the client (e.g., transactions, security) and are the standard approach for service integration into the JBoss architecture. The invoker proxy handles protocol specific communications (common invoker proxy types are Java RMI or CORBA IIOP), sending marshalled client invocations and receiving associated replies to/from a JBoss server. The *invoker MBean* handles marshalled invocations, passing them to the appropriate container where the EJB component exists where ultimately the invocation is handled by application logic. As with the client side, interceptors may be present, incorporating additional services into the architecture. For our explanation of client/server invocation handling it suffices to say that the invoker MBean can be viewed as complementary to the client side invoker proxy.

3.3. Transactions in J2EE

Figure 2 shows the main elements of an application server that are responsible for supporting transactional applications. Application logic is implemented using EJBs, with persistence of application state provided by one or more resource managers. A resource manager is commonly implemented using a *relational database management system* (RDBMS). Transactional services that a container may manage on behalf of EJBs relate to persistence of state and the transaction participation status of a bean. *Java DataBase Connectivity* (JDBC) drivers are provided by resource manager vendors to allow the integration of their database products into J2EE implementations. To enable resource managers to participate in transactions an *XAResource interface* (shown as *XA* in figure 1) must be provided together with JDBC drivers (commonly bundled together and referred to as *resource adaptors*). A transaction manager is hosted by the application server and assumes responsibility for enabling transactional access to EJBs (e.g., coordinating the two phase commit protocol and managing resources afforded to transactions).

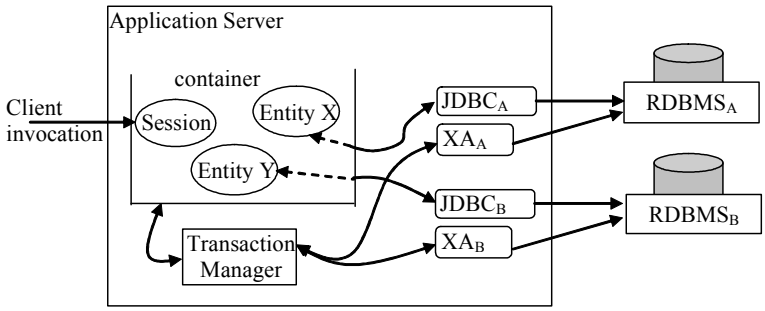


Figure 2 – EJB transactions

We now describe, with the aid of figure 2, a sample scenario of a single transaction involving three enterprise beans and two resource managers. A session bean receives a client invocation. The receiving of the client invocation results in the session bean starting a transaction, say T_1, and issuing a number of invocations on two entity beans (*X* and *Y*). When entity beans are required by the session bean, first the session bean will have to 'activate' these beans via their home interfaces, which results in the container - we are assuming container managed persistence - retrieving their states from the appropriate resource managers for initializing the instance variables of *X* and *Y*. The container is responsible for passing the 'transaction context' of T_1 to the JDBC drivers in all its interactions, which in turn ensure that the resource managers are kept informed of transaction starts and ends. In particular: (i) retrieving the persistent state of *X* (*Y*) from $RDMS_A$ ($RDMS_B$) at the start of T_1 will lead to that resource manager write locking the resource (the persistent state, stored as a row in a table); this prevents other transactions from accessing the resource until T_1 ends (commits or rolls back); and (ii) *XA* resources (XA_A and XA_B) 'register' themselves with the transaction manager, so that they can take part in two-phase commit.

Once the session bean has indicated that T_1 is at an end, the transaction manager attempts to carry out two phase commit to ensure all participants either commit or rollback T_1. In our example, the transaction manager will poll $RDBMS_A$ and $RDBMS_B$ (via XA_A and XA_B respectively) to ask if they are ready to commit. If a $RDBMS_A$ or $RDBMS_B$ cannot commit, they inform the transaction manager and roll back their own part of the transaction. If the transaction manager receives a positive reply from $RDBMS_A$ and $RDBMS_B$ it informs all participants to commit the transaction and the modified states of *X* and *Y* becomes the new persistent states.

3.4. Implementing transactions in JBoss

We 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.

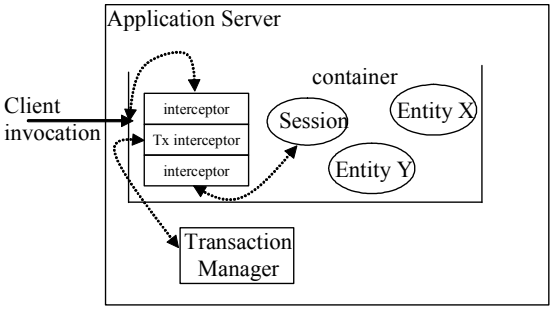


Figure 3 – Augmenting application server with transactions.

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 3; for brevity, we have not shown resource adaptors). 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). In our example described in figure 2, the same transaction context would be propagated to the transactional resources (session bean, entity beans, JDBC drivers) to ensure all participants are associated with a single transaction (transaction T_1 in our example).

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 is assumed. There are two different times within a client session when a primary may fail: (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 4 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.

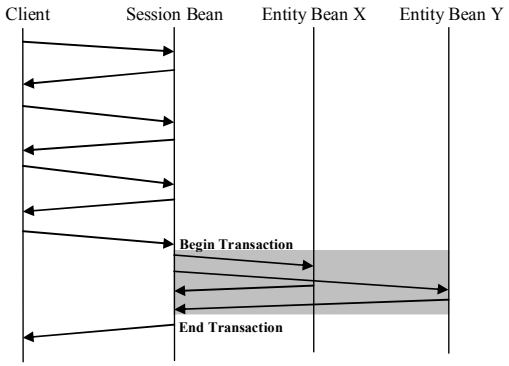


Figure 4 – Interactions between beans and client.

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

We use interceptors, MBeans and JMX technologies to integrate our replication service into JBoss. This approach ensures that we do not disturb the functionality of existing services.

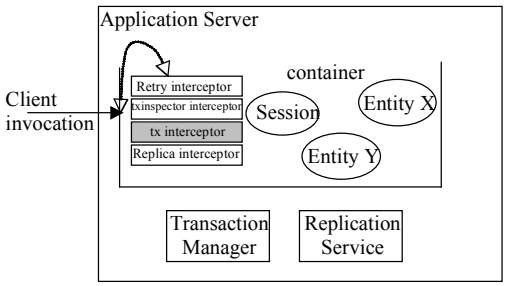


Figure 5 – Augmenting application server with replication service.

Figure 5 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.

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 [16]. JGroups provides a group communication subsystem that supports the abstraction of a process group (and totally ordered atomic multicast). 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.

5.1. Client side failure handling

Clients query the naming service to retrieve a session bean reference. The naming service allows developers to use a naming scheme (similar to absolute file names) to ease the handling of EJB referencing. The naming service, when supplied with such an EJB name, supplies the appropriate reference to the *home interface* of the requested session bean. The home interface can be used by clients to manage the lifecycle services associated to a session bean type (e.g., create instance of session bean). Once a session bean instance has been created the client may invoke operations on the *remote interface* of the session bean instance. The remote interface provides access to the methods associated to the application logic of the EJBs.

Client invocations are handled by the client side proxy (providing the illusion of a local service) in a manner common with standard *remote procedure call* implementations. That is, a client issues a request to a proxy interface (presentation of session bean remote interface) that is within its own addressable space. The proxy interface manages inter-process communications between the

application server (instance of session bean), including returning any parameters (sent by the application server) to the client.

To enable failover, instead of a single session bean reference being present in the proxy interface, a list of references is provided, representing primary and backups. Client invocations are directed to the primary. If the primary is non-responsive (proxy interface timeouts primary) then the invocation is repeated using each backup in turn until a backup acknowledges the invocation. If a backup is not the primary, it responds to a client invocation with a message indicating the current view of primary/backup; the client re-issues its invocation to the primary (this is achieved transparently without the application layer's knowledge). This process is repeated until: (i) primary responds or; (ii) application server becomes unreachable (no replies from all backups).

The proxy interface of the client also maintains a logical clock which timestamps each invocation as it is received from the client. After each timestamp is issued the clock is incremented by one, and so uniquely identifying each invocation emanating from a client. This information is used by the application server to prevent duplicated processing of a client invocation.

In the JBoss application server alterations were made to enhance interface proxies for the client with the additional functionality required for our failover scheme. Alterations were also made on the invoker MBean at the server to allow the server to determine if the receiving bean is the primary or not (by checking the local group log). This proved to be quite straightforward as JBoss proxies allow the interception of invocations and additional processing to be introduced using interceptors.

5.2. Session State Replication

The retry interceptor first identifies if this is a duplicated invocation by comparing the timestamp on the incoming client invocation with that in the timestamp log. If the invocation timestamp is the same as the timestamp in the timestamp log then the parameters held in the bean log are sent back to the client. If the invocation timestamp is higher than the timestamp in the timestamp log then the invocation is passed, along the interceptor chain, towards the bean.

If the invocation is not a retry and the receiving bean is the primary, then the invocation is executed by the bean. After bean execution (i.e., when a reply to an invocation is generated and

progresses through the interceptor chain towards the client) the replica interceptor informs the replication service of the current snapshot of bean state, the return parameters and the invocation timestamp. Upon delivery confirmation received from the replication service, the primary and backups update their bean and timestamp logs appropriately. Once such an update has occurred, the invocation reply is returned to the client.

5.3. 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.

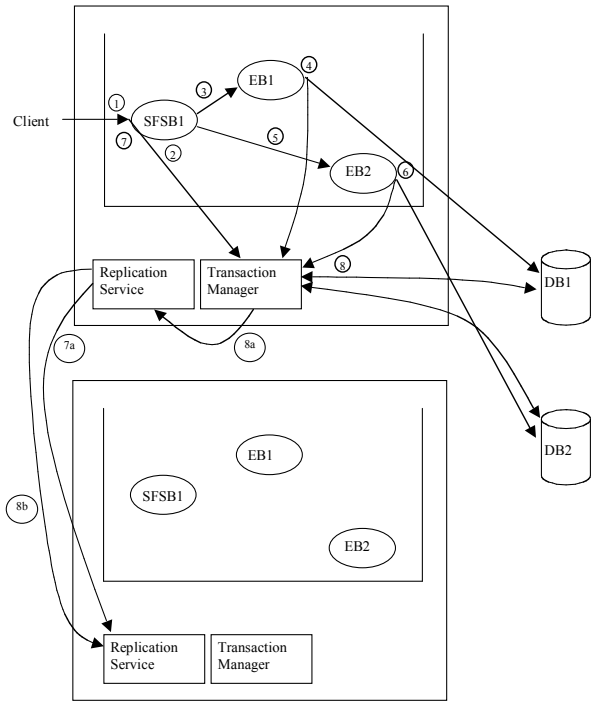


Figure 6 - A typical interaction for a transaction processing in EJB

Figure 6 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 (on JBoss application server it is the tx interceptor that performs this task) 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 invocation of the method on SFSB1 calls another invocation (3) on EB1 and also an invocation (5) on EB2. At (3) and (5), the container determines that although the invocations need to be executed within a transaction, it does not have to create a new transaction for them as the invocation has already been associated with a transaction context. 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).

We now identify where in the transaction execution described in figure 6 we accommodate for transaction failover. A multicast of the state update of all involved session beans together with the transaction id and information on all resources involved at point (7) is made (7a). That is, when application level logic has ceased and prior to entering two-phase commit we inform backup replicas of the states of resources involved in the transaction (when commit stage is about to commence). A multicast of the decision taken by the transaction manager is made to all backup replica transaction managers after the prepare phase at point (8) via the replication service (8a) and (8b). If the primary fails before reaching point (7), the invocation will not complete, and the client will retry and the backup will execute the computation as a new invocation; but if the primary fails after reaching point (7) the backup will already have the updated state and it will attempt to finish the transaction by continuing the two phase commitment process depending on whether the primary transaction manager has taken a decision or not at point (8).

In order to implement the above scenario, we must be able to detect which invocation is to be executed as a single transaction unit (e.g. (1)), and which invocation is part of a transaction unit defined elsewhere (e.g. (3) and (5). This distinction is necessary as we will only propagate the state update at the end of an invocation that is executed as a transaction unit.

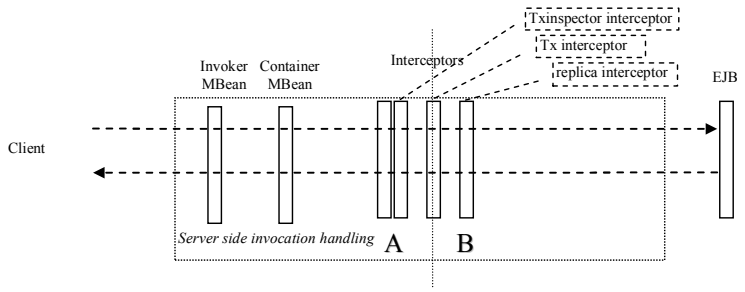


Figure 7 - Server side invocation handling for transaction failover

Figure 7 displays the server side invocation handling, with the focus on three interceptors involved in transaction failover. On JBoss the tx interceptor is responsible for inspecting the transaction context from the incoming invocation, and replacing the transaction context when necessary with a new one. Interceptors that are located before the tx interceptor (on A side in the figure 7) will see the original transaction context on the invocation while the interceptors that are located after the tx interceptor (on B side in the figure 7) will see the new transaction context as defined by the tx interceptor. Therefore, in order to determine which invocation must be executed as a transaction unit, our txinspector interceptor must be placed before the tx interceptor so that it can inspect the transaction context from the incoming invocation and compare it with the transaction attribute of the method being invoked. When the txinspector interceptor determines that an invocation is a unit of a transaction, it flags that invocation with a TRANSACTIONUNIT attribute so that the replica interceptor knows that it has to propagate the state and the transaction information after the computation has finished: the end of method execution will result in two phase commit. The txisnpector interceptor also flags non transactional invocation with a NONTRANSACTIONAL attribute so that the replica interceptor knows that it has to propagate the state without the transaction information.

The state update at (7a) includes all information necessary to attempt to complete the transaction at a backup replica (e.g., application logic state, resources associated with transaction, transaction context). The replica interceptor does not have to propagate the state of a session bean after a partially executed transaction, as any failure that happens during the transaction requires a backup replica to execute the original transactional invocation from the beginning (e.g., (1) in figure 6). This follows our initial assumption regarding style of application execution where transactions predominantly consist of a limited number of executions that occur after non-transactional client/application server interactions.

If application level execution has ceased within a transaction and two phase commit is to be attempted, we can complete a transaction at a backup replica if the primary fails. At (8) the transaction manager performs two phase commit by first sending a prepare message to all transaction participants. After all replies have been received, the transaction manager takes a decision on

whether to commit or abort a transaction. We had to modify the JBoss transaction manager to ensure the decision is multicast (using the replication service) to all backup replicas. The multicast message contains the transaction id and the decision taken by the transaction manager. Once delivery notification of this multicast is received from backup by the transaction manager then the decision is sent to transaction participants.

A number of other technical challenges needed to be overcome to provide an engineered solution. For example, handling differences in bean referencing from primary to backup (local references for same bean types vary across servers). However, for brevity and to concentrate on our approach we do not go into such technical detail.

6. Experimental Evaluation

Experiments were carried out to determine the performance of our system over a single LAN. 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.

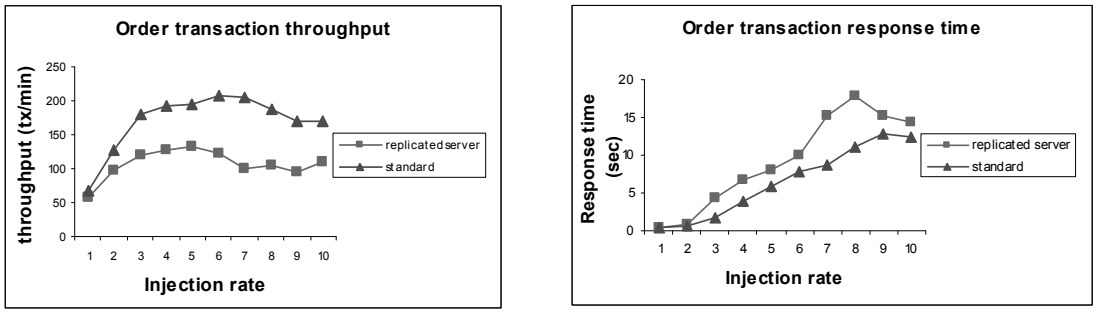
The application server used was JBoss 3.2.5 with each application server deployed on a Pentium IV 2.8 GHz PC with 2048MB of RAM running Fedora Core 4. The database used was Oracle 9i release 2 (9.2.0.1.0) [20] with each database deployed on a Pentium IV 2.8 GHz PC with 2048MB of RAM running Fedora Core 4. The client was deployed on a 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 [17] 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 8 presents two graphs that describe the throughput and response time of the ECperf applications; figure 8(i) identifies the throughput for the entry order system, figure 8(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 application

(ii) response time for entry order application

Figure 8 – Performance figures.

Figure 9 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.

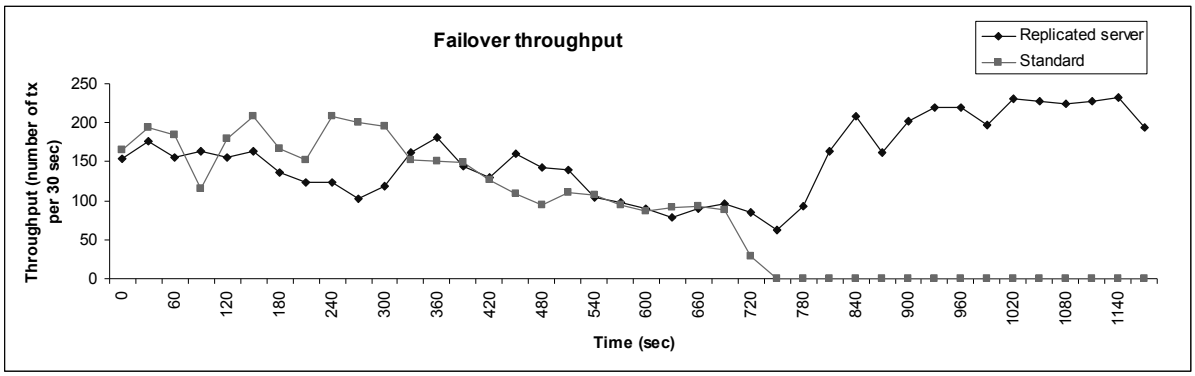


Figure 9 – Performance figures under a failure.

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 models (e.g. CORBA Component Model) and application servers. Our design can be easily adapted to work with a cluster of servers if we assume a homogenous deployment where each application server in a cluster supports the same application logic (this is the most commonly used approach). Components and transaction managers may act simultaneously as both primary and backup for clients. A client session is allocated a primary node, with all other nodes in the cluster assuming the role of backup nodes. Different client sessions may have different primary nodes, so utilizing the processing power of the whole cluster for satisfying client sessions.

References

[1] Frolund, S. and R. Guerraoui, “A pragmatic implementation of e-transactions”, Proceedings the 19th IEEE Symposium on.Reliable Distributed Systems, SRDS-2000.
[2] Frolund, S. 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] 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.
[6] P.K. Reddy and M. Kitsuregawa, “Reducing the blocking in two-phase commit protocol employing backup sites”, Proceedings of Third IFCIS Conference on Cooperative Information Systems (CoopIS'98), August 1998, New York.
[7] Ozalp Babaoglu, Alberto Bartoli, Vance Maverick, Simon Patarin, Jaksa Vuckovic, Huaigu Wu., “A Framework for Prototyping J2EE Replication Algorithms”, Int. Symposium on Distributed Objects and Applications (DOA), Agia Napa, Cyprus, October 2004.
[8] H. Wu, B. Kemme, V. Maverick, “Eager Replication for Stateful J2EE Servers”, Int. Symposium on Distributed Objects and Applications (DOA), Agia Napa, Cyprus, October 2004.
[9] S.Labourey and B.Burke, “ JBoss Clustering 2nd Edition”, 2002, www.jboss.org
[10] M.C. Little and S K Shrivastava, “Integrating the Object Transaction Service with the Web”, Proc. Of IEEE/OMG Second Enterprise Distributed Object Computing Workshop (EDOC’98), La Jolla, CA, November 1998, pp. 194 – 205.
[11] M.C. Little and S K Shrivastava, “Java Transactions for the Internet”, Distributed Systems Engineering, 5 (4), December 1998, pp. 156-167.
[12] P.A. Bernstein, M. Hsu, et al. , “Implementing recoverable requests using queues”, Proceedings of ACM SIGMOD international conference on Management of data, 1990, Atlantic City, New Jersey.
[13] R. Barga, D. Lomet, et al. ,"Recovery guarantees for Internet applications", ACM Trans. on Internet Tech. 4(3): 289-328, 2004.
[14] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava and M.C. Little, “Component Replication in Distributed Systems: a Case study using Enterprise Java Beans”, 22nd IEEE/IFIP Symposium on Reliable Distrinbuted Systems (SRDS2003), Florence, October 2003, pp. 89-98.
[15] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, G. Alonso, “Consistent Database Replication at the Middleware Level”, ACM Transactions on Computer Systems (TOCS). Volume 23, No. 4, 2005, pp 1-49.
[16] Ban B., “JavaGroups User’s Guide” http://www.javagroups.com
[17] http://www.jcp.org/en/jsr/detail?id=4
[18] 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.
[19] B. Kemme, R. Jimenez-Peris et al, “Exactly once Interaction in a Multi-tier Architecture”, VLDB Conf. Trondheim, Norway. Aug. 2005. Workshop on design, implementation, and deployment of database replication.
[20] W. Zhao, L. M. Moser and P. M. melliar-Smith,”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.
[21] Jiménez-Peris, R., M. Patiño-Martínez, et al, “A Low-Latency Non-blocking Commit Service”, 15th International Conference on Distributed Computing (DISC), October 2001.