Component replication in application servers

Thesis by

Achmad Imam Kistijantoro

In partial fulfilment of the requirements

for the degree of

Doctor of Philosophy

NEWCASTLE UNIVERSITY LIBRARY

205 36626 6

Thesis L8377

University of Newcastle upon Tyne

Newcastle upon Tyne, UK

September 2006

Acknowledgement

First of all, I would like to thank my supervisor Santosh Shrivastava for his help and guidance throughout my Ph.D. work. I would also like to thank Graham Morgan for his advice and fruitful discussions, and Ms. Shirley Craig for her assistance in finding references for the thesis.

Personal thanks go to my parents, my brother and sister, and my parents-in-law for their constant supports throughout my life, and to my family, especially my wife Vera for her support and understanding during the difficult times we experienced for completing the thesis.

This work was supported by scholarship from QUE Project, Department of Informatics, ITB – Bandung, Indonesia.

Abstract

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.

On this architecture, availability measures, such as replication, can be introduced in each tier in an application specific manner. Among the three tier described above, the availability of the middle tier and the back-end tier are the most important, as these tiers provide the computation and the data for the applications.

This thesis investigates how replication for availability can be incorporated within the middle and back-end tiers. The replication mechanisms must guarantee exactly once execution of user request despite failures of application and database servers. The thesis develops an approach that requires enhancements to the middle tier only for supporting replication of both the tiers. The design, implementation and performance evaluation of such a middle tier based replication scheme for multi-database transactions on a widely deployed open source application server (JBoss) are presented.

Table of Content

1 Introduction **1**

1.1 Thesis organization **4**

2 Component Middleware Infrastructure **6**

2.1 Overview of component middleware architecture **8**

2.2 Containers and application servers **10**

2.3 Client interaction **12**

2.4 Component persistence **13**

2.5 Transactions **14**

2.6 Architecture of J2EE application servers implementation **17**
2.6.1 JBoss J2EE Application Server 3.2.x 18

2.7 Summary **22**

3 Availability measures for component-oriented middleware **23**

3.1 Background **23**

3.2 Assumptions and requirements **25**

3.3 Availability measures in existing application servers **27**
3.3.1 Limitation in existing application servers availability 30

3.4 State Replication **31**
3.4.1 Database replication in general 32
3.4.2 Middleware-based data replication 34

3.5 Computation replication **40**
3.5.1 Transactional queue 41
3.5.2 Transactional client 42
3.5.3 E-transactions 42
3.5.4 Interaction contract 45
3.5.5 J2EE replication framework 48
3.5.6 Stateful EJB replication 50

3.6 The approaches developed in this thesis **52**
3.6.1 Failure assumptions 53
3.6.2 State replication approach 53
3.6.3 Computation replication approach 54
3.6.4 Comparison with other approaches 55

3.7 Summary 57

4 State replication: improving the availability in database tier 58

4.1 Issues in state replication for component middleware 58

4.2 Single server implementation 61

4.3 Clustering configuration 62

4.4 Performance evaluation 64
4.4.1 Implementation and Setup 64

5 Computation replication: improving the availability of the application servers 69

5.1 Model 69
5.1.1 Different types of components: stateless, session and persistent component 71

5.2 Fail-over implementation 72
5.2.1 Client interface maintenance 74

5.3 Session state replication 75

5.4 Handling transaction 77
5.4.1 Determining when to replicate the transaction 79
5.4.2 Handling chained session invocations 79

5.5 Replication service implementation 80
5.5.1 Group membership 80
5.5.2 Replica state propagation 80
5.5.3 State transfer 81

5.6 Load balancing 81

5.7 Failure scenarios 82

5.8 Performance evaluation 83

6 Conclusions 88

6.1 Summary of thesis contributions 88

6.2 Further Work 90

List of Figures

Figure 1-1 N-tier architecture 1
Figure 2-1 Component middleware architecture 9
Figure 2-2 Interaction between the application server, the container and the components 11
Figure 2-3 EJB Remote access and local access 12
Figure 2-4 Java Transactions API 15
Figure 2-5 Transaction processing 16
Figure 2-6 EJB Container and its relationship with other components in JBoss 19
Figure 2-7 Proxy object 20
Figure 2-8 Processing client request in JBoss 21
Figure 3-1 Three-tier architecture 28
Figure 3-2 ADAPT framework (Babaoglu, Bartoli et al. 2004) 48
Figure 3-3 intercepting points on ADAPT framework (Babaoglu, Bartoli et al. 2004) 49
Figure 4-1 An application server without state replication 59
Figure 4-2 An application server with state replication 60
Figure 4-3 State replication in clustering configuration 62
Figure 4-4 Replica manager for maintaining available resource managers information 63
Figure 4-5 Configuration 64
Figure 4-6 Result 67
Figure 5-1 Client interaction 70
Figure 5-2 System Architecture 71
Figure 5-3 Replicated EJB implementation of a remote interface 73
Figure 5-4 Fail-over invoker proxy state chart 73
Figure 5-5 Server state 74
Figure 5-6 Augmenting application server with replication service 75
Figure 5-7 A typical interaction for a transaction processing in EJB 77
Figure 5-8 - Performance figures 84
Figure 5-9 Performance figures with failures 85
Figure 5-10 Throughput for entry order application with varying number of servers 85
Figure 5-11 Response time for entry order application with varying number of servers 86
Figure 5-12 Throughput for entry order application with varying number of servers in load balancing configuration 86
Figure 5-13 Response time for entry order application with varying number of servers in load balancing configuration 87

List of Tables

Table 3-1 E-transaction properties 43
Table 3-2 State replication approaches 55
Table 3-3 computation replication approaches 56
Table 6-1 State replication approaches 88
Table 6-2 computation replication approaches 88

1 Introduction

Modern client-server distributed computing systems may be seen as implementations of an N-tier architecture. Typically, the first tier consists of client applications containing browsers, with the remaining tiers deployed within an enterprise representing the server side; the second tier (Web tier) consists of web servers that receive requests from clients and passes on the requests to specific applications residing in the third tier (middle tier) consists of *application servers* where the computations implementing the business logic are performed; the fourth tier (database tier) contains databases that maintain persistent data for the applications (Figure 1-1).

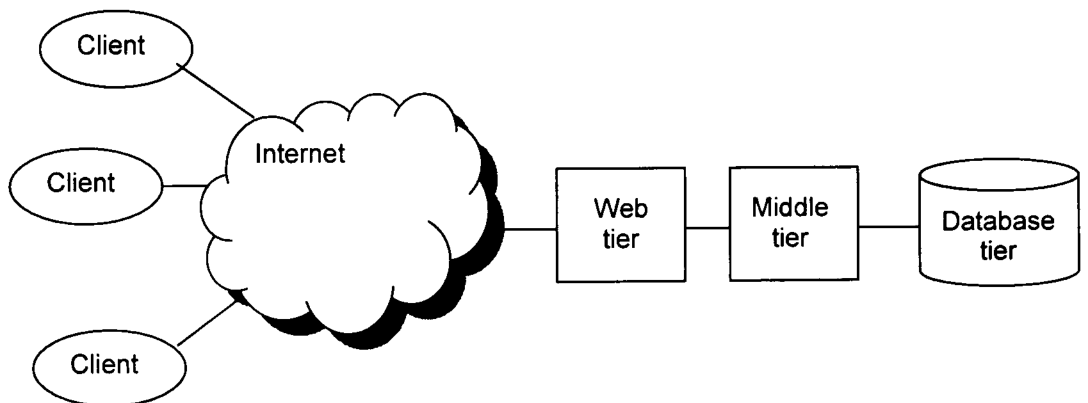


Figure 1-1 N-tier architecture

Applications in this architecture typically are structured as a set of interrelated components hosted by *containers* within an application server. Various services required by the applications, such as transaction, persistence, security, and concurrency control are provided via the containers, and a developer can simply specify the services required by components in a declarative manner. This relieves the developers from the complex task of handling them directly in the components code.

Furthermore, this architecture also allows flexible configuration such as partitioning and clustering for improved performance and scalability. Availability measures, such as replication, can also be introduced in each tier in an application specific manner. In a typical n-tier system such as illustrated in Figure 1-1, the interactions between clients to web server tier are performed across the Internet. The infrastructure supporting these interactions is generally beyond the direct control of an application service provider that normally only manages the web server tier and the tiers afterwards. The availability of a service can be increased by making each tier available. Among the tiers described above, the middle tier and the database tier are arguably the most important, as it is on these tiers the computations are performed and data storage and persistency are provided. Data as well as object replication techniques for improving the availability 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 and database tier.

One important concept related to availability measures is that of exactly once transaction or exactly once execution. For example, in a typical online shop where a customer may browse catalogue, select items into his/her shopping cart, and finally make an order and pay the order by his/her credit card. These activities involve transactionally accessing data (stock items, card payment validation, and customer details) which are hosted by different database servers. Here, the system must guarantee exactly once execution of user requests despite system failures. Without this, the client may end up with duplicated orders or he/she may lose the order without noticing it. One approach would be to make the entire interaction from client to databases transactional. Problem 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 applications. 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.

Many vendors for middle tier application servers provide clustering primarily as a solution for improving the performance (through load balancing) and scalability (ability to deal with large number of clients) and rely on propriety replication mechanisms of database vendors for database availability. Cluster systems allow the employment of multiple servers to service clients concurrently thus improve the throughput of the system. Cluster systems typically also provide limited *failover* capability, allowing client requests to a failed server to be redirected automatically to another available server. Stateless computation requests (i.e. requests that do not cause modification of the state stored on the application server) can safely be re-executed on another server. However, failover is not straightforward for stateful computation requests, where the application server may keep the session state produced by previously executed requests. One may circumvent this by requiring all state to be stored at a database, yet this makes unnecessary performance overhead, as the state is usually needed for short term, within the scope of current client interactions only. Some implementations provide session replication in application servers, removing the requirement to store session state in a database, but they are unable to salvage the active transactions from the failed server, causing the abortion of the client session. Furthermore, there is a more serious limitation: when an application server fails while in the process of committing a transaction. For single database access, the other application servers do not know whether the transaction from the failed application server has been committed successfully or not, and reprocessing the client request on another application server may violate the exactly once execution property. For multi-database access, this may result in databases and other shared resources becoming unavailable to the other servers[1], causing other available servers cannot continue servicing clients request even if they already have an updated session state for servicing the request. Normally this requires some manual operations to rectify the problem.

We thus see that introduction of availability measures in multi-tier systems poses challenging system design problems, namely transparent failover,

[1]A failure that happens in the middle of the two phase commitment protocol can cause the backend database blocked, waiting for the transaction coordinator (which is located at the crashed application server) to be available again, so that it can find out the outcome of the transaction.

exactly once execution of client requests, non-blocking distributed transaction processing, ability to work with load balancing and open, non proprietary solution.

For this reason, there has been much recent research work on increasing availability whilst supporting exactly once transactions through replication in commonly used application servers. However, none of the works that we know meets all the system design problems stated above.

This thesis investigates how replication for availability can be incorporated within the middle and back-end tiers. The replication mechanisms developed in the thesis guarantee exactly once execution of user requests, masking failures of application and database servers, ensuring continued client request processing despite failures during the committing process; furthermore they also support load balancing features for clustering configurations. In short, the design and implementation presented in the thesis solves all the system design problems stated above. The thesis develops an approach that requires enhancements to the middle tier only for supporting replication of both the tiers. Therefore, database server failure can also be masked without having to rely on the proprietary database vendor replication. The design, implementation and performance evaluation of such a middle tier based replication scheme for multi-database transactions on a widely deployed open source application server (JBoss) are presented.

1.1 Thesis organization

Chapter 2 discusses basic concepts of component middleware architecture and the platform that will be used as the basis for implementing our replication algorithms. It describes common components of an application server and how they interact with each other to provide service to clients.

Chapter 3 provides a literature review of existing approaches to availability on three-tier architecture. Each approach is evaluated and compared with the replication mechanisms described in this thesis. Those approaches are classified into two groups: the ones that focus on the availability of the

application state (backend-tier) and the ones that focus on the availability of the computations (middle-tier).

Chapter 4 describes application state replication approach implemented using mechanisms within the middle tier that works seamlessly with existing load balancing clustering configuration in application servers and presents the benchmark result of the implementation.

Chapter 5 describes the implementation of computation replication that is able to continue multi-databases transactions on backups when the primary fails. Exactly once semantics is ensured. The replication implementation also includes the load balancing version that distributes client requests among existing servers.

Chapter 6 summarises the contributions of the thesis and outlines areas into which the work done in this thesis could be extended and researched further.

2 Component Middleware Infrastructure

Large class of distributed applications map on to client-server architecture, where applications are separated into clients, in which computers are used for managing the interaction with the users, and into servers, in which the data is located and processed. This architecture gives flexibility as data can be shared among many clients running on heterogeneous platforms. For applications that need to store large amount of data and perform complex information processing of client requests, a server soon gets overloaded as the number of clients increase. To solve the problem, the function of the server is split into two tiers: one tier is for performing computation and another tier is for handling data. The computation tier, which is referred to as an application server, is also responsible for pooling the connections to the database servers, and shares these connections to service client requests, therefore reduce the number of connections required to the back-end servers. Application server belongs to a class of software known as *middleware*, a software layer between network operating systems and application components.

Many organizations today have a variety of computing applications running on heterogeneous hardware and software systems. These systems are often difficult to integrate as they have different platforms and architectures and support different protocols. Middleware can be used to help these systems to *interoperate*, by allowing a program on one system accessing data and programs provided by other systems. Middleware is also seen as a solution to speed up application development, by providing higher abstraction that hides the difficulties in dealing with many aspects of distributed systems, such as security, communication protocols, transaction management and concurrency control.

Software components, as a concept, has been in existence since a long time ago. Douglas McIlroy (Naur and Randell 1969) has proposed in 1968 that future software industry should be built based on software components, where

one can build software out of its components which can be provided from a number of software components suppliers, just like in other industries. such as electronics and mechanical industries.

The idea of software component is also reflected in *object oriented* approach to software, which became popular in 1990s. This approach views *software objects* as representation of real world entities, and are considered as the key to solve software crisis problem, as it enforces software developers to employ good design principles in developing software such as modularity, encapsulation and software reuse. Software objects are abstractions containing both data (which is called *object state* or *object attributes*) and a set of methods for manipulating the data. These objects are then used as the basic components to build applications. Objects communicate with other objects by message passing (this is implemented as *method calls* to other objects), and application can be seen as a set of interacting objects.

The nature of object oriented approach to software fits perfectly with the goal of software component, especially for supporting software reuse. However, the small granularity of objects makes them difficult to reuse on different projects and on different context. Object oriented approach also does not address the issue of composing software from software components coming from different platforms. Component oriented development, which is seen as the next drive after object oriented approach (Kozaczynski and Booch 1998), addresses this specific problem, focusing on how a component interacts with its environment and how a component can be used on different context by specifying its dependency explicitly as a contract. Component, in this context, is defined as a piece of software which has explicit specifications of what it requires and provides, can be independently deployable, and subject to third-party composition (Szyperski 2002).

Middleware adopts object oriented approach and software component technologies by developing standards for component oriented infrastructure, such as **Sun Enterprise JavaBeans (EJB)** specification and **Common Object Request Broker Architecture (CORBA)** specification.

2.1 Overview of component middleware architecture

We can look at component middleware architecture as n-tier architecture that consists of client, web servers, application servers and backend data stores. Figure 2-1 describes this architecture. A client can access the service provided by the system via the Internet, by using a web browser, which provide the display representation of the application to the client. Alternatively, the applications at the client's side may access directly to middle tier application servers. The interaction from the client applications to the middle-tier software can be either session-oriented or sessionless-oriented, using protocols such as Hypertext Transfer Protocol (HTTP), Java Remote Method Invocation (RMI), and Simple Object Access Protocol (SOAP).

Web servers manage the representation of the service so that the clients can access the service by using web browsers. Application servers constitute the middle-tier that provides containers for hosting the components and supporting services, such as security, communication, persistence and transactions. The backend databases provide the persistent state of the application.

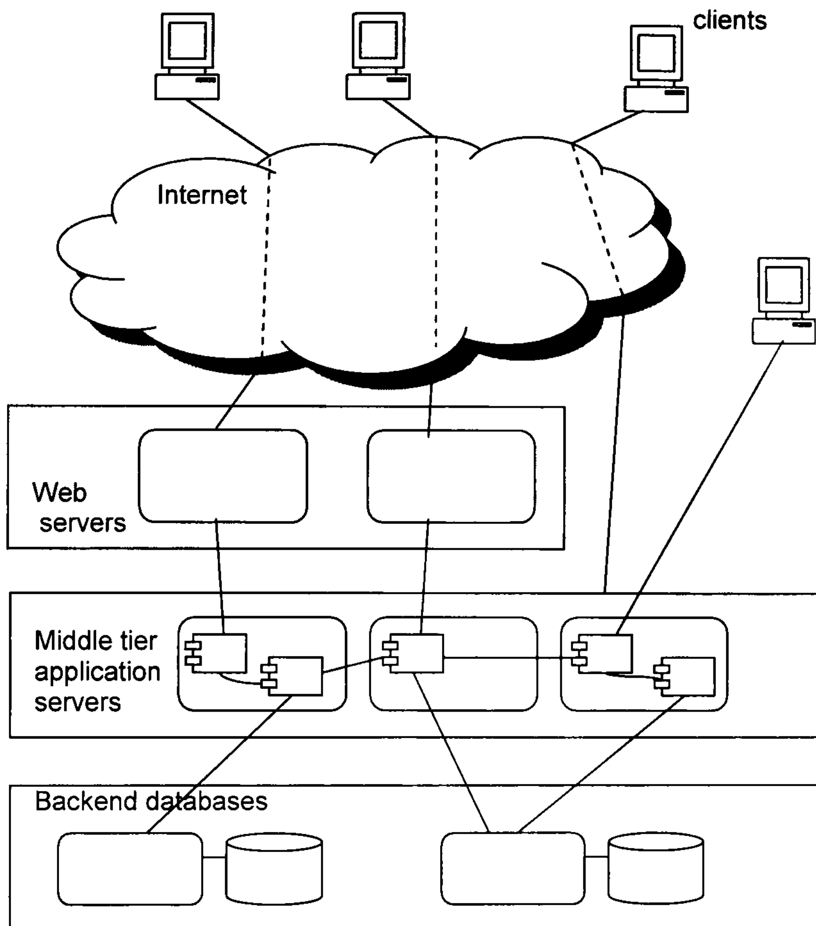


Figure 2-1 Component middleware architecture

All interactions between components and their environment, including the clients and backend datastores are mediated by the containers. To support various types of business logic, several types of components are normally provided. EJB specification defines components as stateless session beans, stateful session beans, entity beans and message-driven beans. Stateless session beans refer to components that represent processing client requests that does not require conversational state, therefore can be instantiated per client request basis. Stateful session beans are for processing conversational client requests, and maintain conversational state for requests from the same client. Entity beans represent and manipulate persistent data of an application, which are stored in the backend data store. Message driven beans allow developer to implement asynchronous components that act as listeners for Java Messaging Services (JMS).

The other specification, CORBA Component Model (CCM), provides similar categories: Service (corresponds to stateless session beans), Session (corresponds to stateful session beans), Entity (corresponds to entity beans), Process (unique to CCM, represents business process and has persistent state across multiple sessions). The rest of this chapter will focus on EJB model that will be used as the target implementation for replication mechanism.

2.2 Containers and application servers

As explained in section 2.1, containers manage all interaction between components and their environment. A container starts hosting a component through a process called **deployment**. In this process, the container gathers all information required for deploying the component from the **component descriptor**. The descriptor contains information such as the component type, access authorization, transaction attributes for each method, the name of the bean, persistence management, and external resources required. Based on this information, the container then creates an instance that represents the component home interface, and registers this instance to the naming service, so that clients can find the component via this service.

Home interface is an instance that allows clients to create, find and destroy an instance of the component. This is necessary as clients do not have direct access to the component instance, therefore cannot directly invoke the constructor of the component to create an instance. The home interface object is created by the container, and acts as the primary point of contact to the clients. Upon receiving create invocation, for example, the container can set or unset the transaction context, inspect access authorization, invoke necessary code on the component for creating a new instance, then create a representation for the bean instance, the bean interface, to return back to the client.

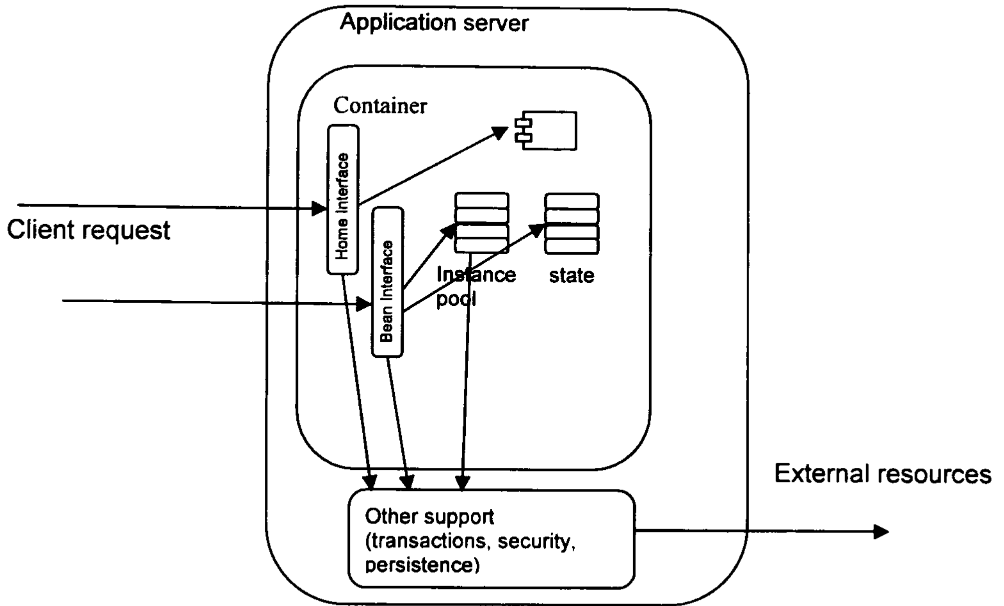


Figure 2-2 Interaction between the application server, the container and the components

As clients always access components indirectly, the container has various choices of implementation. Figure 2-2 depicts an example of typical interaction between application server, container and components. For components that do not maintain conversational state such as stateless session beans, a common implementation approach is by creating a pool of ready bean instances. These instances are allocated to clients requests, and are returned back to the pool after the processing finishes. For components that maintain state, such as stateful and entity beans, the states are kept separately, and the container attaches this state to a bean instance every time it has to service a specific client request.

The home interface, the bean interface and the bean instance access external resources via the support provided by the container. On some cases, the interaction between container, components and external resources are defined through standard interfaces, such as transactions are standardized by Java Transaction API (JTA) and Java Transaction Service (JTS) specification, and databases access are standardized by Java Database Connectivity (JDBC) and J2EE Connector Architecture (JCA).

2.3 Client interaction

The EJB specification specifies three mechanisms to access EJBs: Java Remote Method Invocation (RMI), local access (for clients located on the same Java Virtual Machine (JVM)), and web services[2].

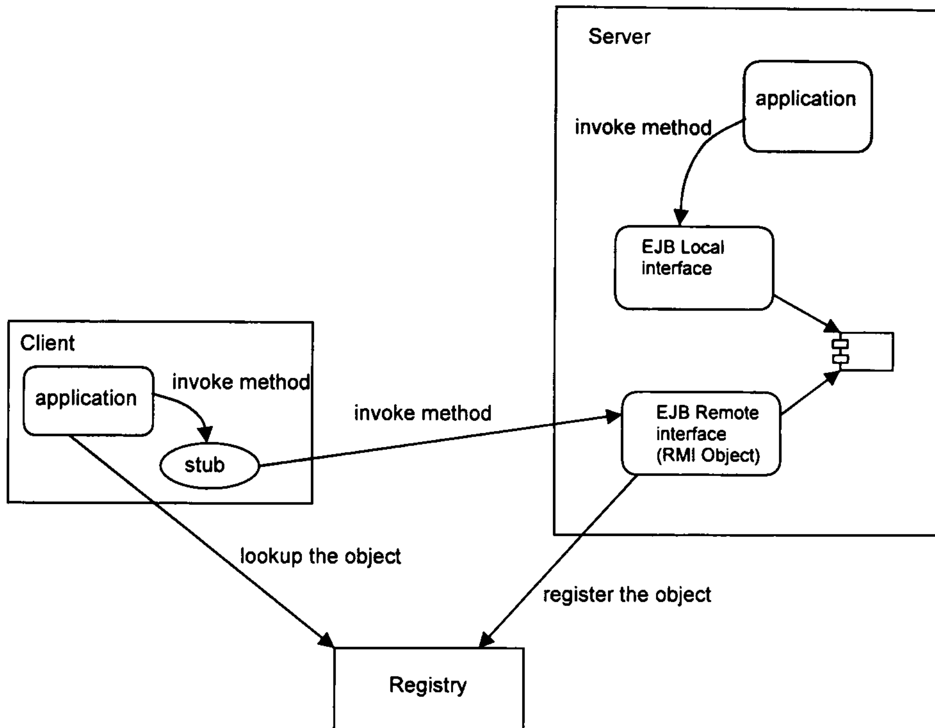


Figure 2-3 EJB Remote access and local access

RMI provides basic functionalities for invoking methods on remote objects. The server that provides objects for remote access makes the object stubs available via an object registry or a naming service. Clients then access the object via these stubs that will be responsible for delegating the invocation, marshalling the parameters and the result of the invocation (Figure 2-3). In most EJB implementations, vendors do not simply provide stubs for client access, but they also wrap the stub object as well with a client-side object implementation, to allow some computation performed on the client side. This implementation allows the stub to directly process some requests at the client side without having to contact the server.

---

[2] Only stateless session beans are allowed to provide access via web service

Clients that are located at the same process as the components can access the component using a **local interface** to that component. The local interface provides faster access than the remote interface such as RMI, as it removes the requirements for marshalling the parameters and also removes the need for messaging during the invocation.

As illustrated in Figure 2-1, clients can access middle tier components directly over the Internet, without the use of a web server and a web browser. These interactions are made possible via web services. A web service is a software component that can be accessed via network and has an interface that is described in a machine-processable format (e.g. WSDL – Web Services Description Language) (W3C W3C Consortium 2004). In EJB specification, web services client can access stateless session bean via SOAP/HTTP protocols. The application server is responsible for generating the class for the web service end point to access the bean and the service description (in WSDL) for the stateless session bean so that the web service client can find out about the services (how to access, interface/methods available for access, etc.).

2.4 Component persistence

There are two way to manage the state for entity beans: container managed persistence (CMP) and bean managed persistence (BMP). CMP allows the developer to delegate the maintenance of bean persistent state to the container. In this scheme, the container generates the SQL (Structured Query Language) commands and queries for retrieving and storing the bean state to the database. Alternatively, the developers can also specify or modify some of the SQL commands for some methods. On the other hand, BMP lets the developer to manage the bean persistent state directly by inserting explicit database SQL queries in the bean code.

From the component development perspective, CMP is preferable as it relieves the developer to code the SQL directly, thus make the component development and maintenance easier. However, the performance of CMP based components will be highly dependent on the application servers that are

used for the implementation (Cecchet, Marguerite et al. 2002; Gorton and Liu 2003).

In CMP, the containers are responsible for mapping the bean field into the database. This can either be done automatically by the container or by explicitly declaring the bean fields to database table mapping in the deployment descriptor. The containers are also responsible for synchronizing the bean state with the database, so that the component developers do not have to deal with this issue directly.

For managing consistency, the specification lays out three strategies depending whether the container has exclusive access to the database state (DeMichiel, Yalçinalp et al. 2000). For containers that have exclusive access to the database state (i.e. if the same database state is not used by other applications besides these containers), they can use **commit option A**, which keeps the instance ready in the containers, thus the containers does not have to reload it from the database again at the start of the next transaction. On the other hand, if the containers do not have exclusive access, they can still keep the instance ready in the cache, but they must reload it at the start of the next transaction. This scenario is called **commit option B**. Another implementation does not keep ready instances at all, and immediately passivates the instance as soon as the transaction finish. The last scenario is called **commit option C.** Brebner and Ran (2001) studied the performance of different commit options. They concludes that commit option A can give better performance up to 60% than commit option C, and commit option B gives better performance by 30% to commit option C. However, these results are product, version and application specific.

2.5 Transactions

A transaction manager is hosted by the application server and assumes responsibility for enabling transactional access to EJBs. The transaction manager does not necessarily have to reside in the same address space as the application server; however, this is frequently the case in practical systems.

The interface between containers, the transaction manager and database resources are specified via JTA (Cheung and Matena 2002) and the Connectors APIs .

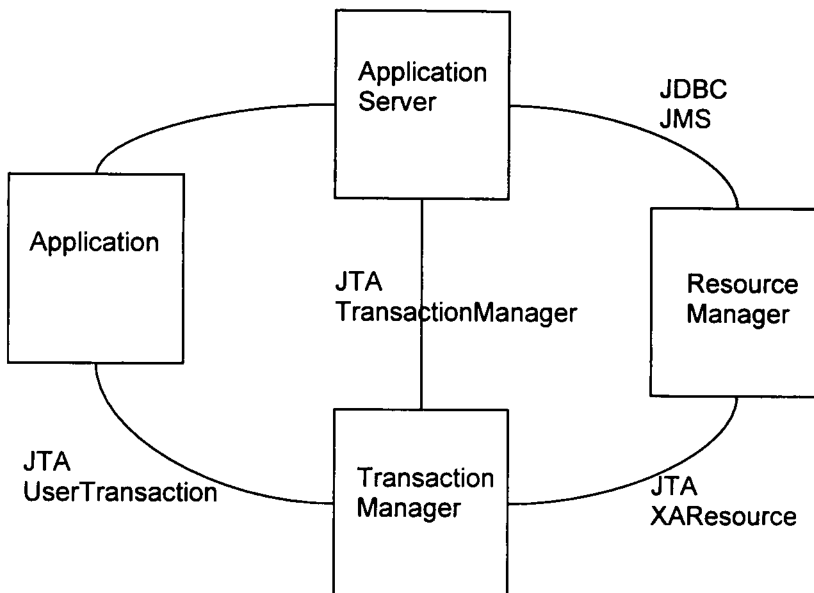


Figure 2-4 Java Transactions API

Figure 2-4 shows the interactions defined in JTA specification. JTA specifies four parties commonly involved in transactions: Application, Transaction Manager, Application Server and Resource Manager. In the three-tier architecture, the application is either a client that uses the system or a component that is hosted on an application server. The transaction manager and the application server are part of the middle tier; and the resource manager is the database on the back-end tier .There are three elements of API defined in JTA: `UserTransaction` is an interface for the application to control transaction boundaries, `TransactionManager` is an interface to allow the application server to control the transaction boundaries on behalf of the application and `XAResource` is a java mapping of the industry standard XA interface (X/Open 1992). `XAResource` allows the transaction manager to associate a global transaction to a transactional resource managed by the

resource manager. JTA also specifies `Transaction` interface, a representation of a transaction context that can be accessed by an application or an application server.

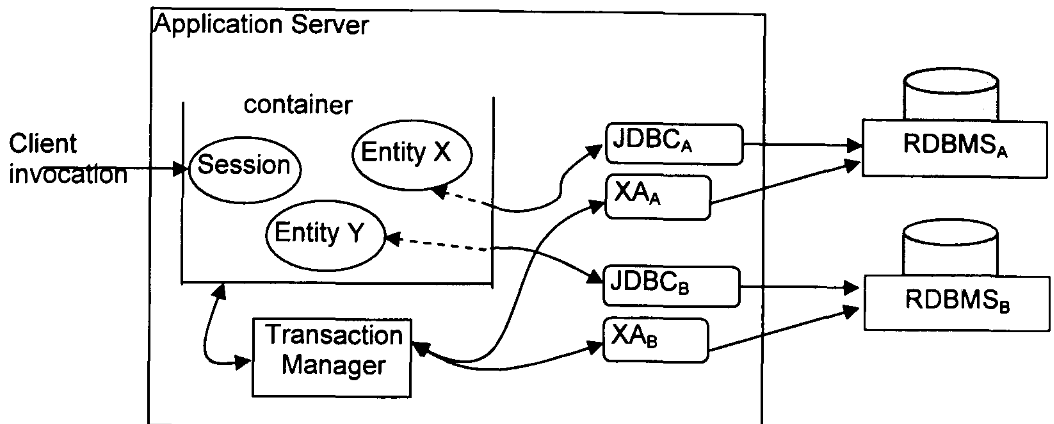


Figure 2-5 Transaction processing

An application server such as EJB server maintains information about components that it hosts, including the transaction attributes of each method of the components and resource managers that are responsible for the components. Figure 2-5 depicts a sample scenario of a single transaction involving three enterprise beans and two resource managers. A session bean receives a client invocation. The application server (via the container) determines[3] that this invocation should be executed within a transaction. The application server then instructs the transaction manager to create a transaction, say T1, and associate it with current thread. After that, an appropriate code from the session bean is executed. This execution, may for example issue a number of invocations on two entity beans (X and Y). Upon these invocations, the application server determines that those invocations are associated with the same transaction as the invoker, i.e. T1. The application server also enlists the resources used by the entity beans X and Y to transaction T1. The application server is also responsible for passing the

[3] The container determines this by inspecting the transaction attribute for the accessed method in the deployment descriptor

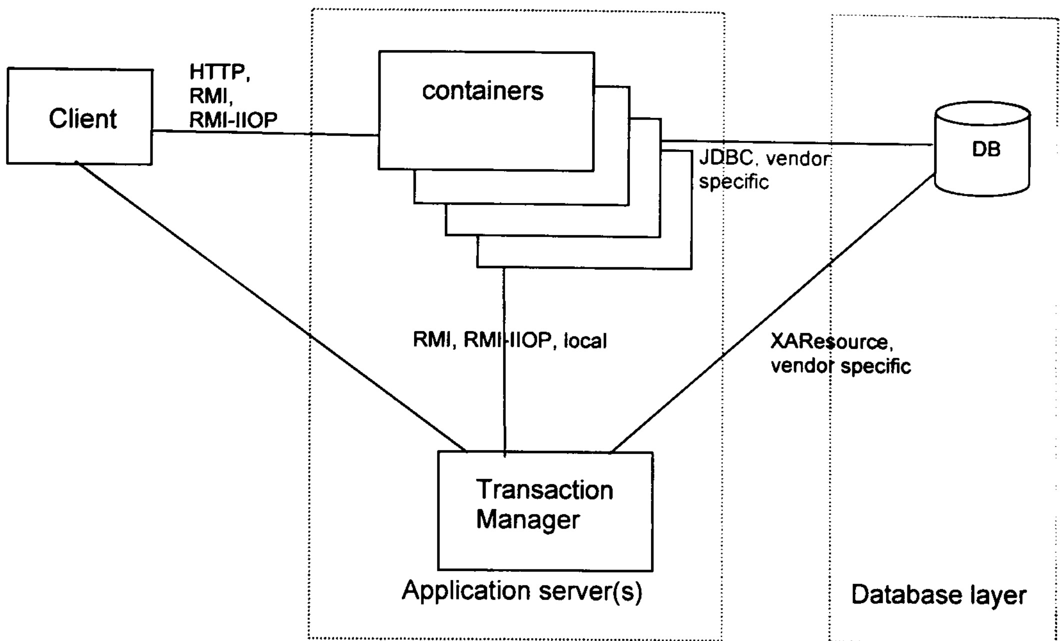
'transaction context' of T1 to the JDBC drivers in all its interactions, which in turn ensure that the resource managers are kept informed of transaction starts and ends. In particular: (i) retrieving the persistent state of X (Y) from $RDMS_A$ ($RDMS_B$) at the start of T1 will lead to that resource manager write locking the resource (the persistent state, stored as a row in a table); this prevents other transactions from accessing the resource until T1 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 T1 is at an end, the transaction manager attempts to carry out two phase commit to ensure all participants either commit or rollback T1. 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.

In this example, all the beans are in the same application server. Support for distributed transactions involving beans in multiple containers (on possibly distinct application servers) is also straightforward if the transaction manager is built atop a CORBA transaction service (Java Transaction Service), since such a service can coordinate both local and remote XA resources. Such a transaction manager will also be able to coordinate a transaction that is started within a client and spans EJBs, provided the client is CORBA enabled.

2.6 Architecture of J2EE application servers implementation

This section describes an example of J2EE application server using a popular open source J2EE application server, JBoss AS.



2.6.1 JBoss J2EE Application Server 3.2.x

JBoss application server is an implementation of J2EE specification developed by JBoss company. JBoss application server centres around Java Management Extension (JMX) (Sun Microsystems Inc 2002), where a microkernel called JMX MBean server hosts a number of pluggable services, presented as JMX MBeans. The role of the microkernel is to initialize all services and configure them in the right order. Other components of J2EE server, such as the Java Naming and Directory Interface (JNDI) server, EJB containers, transaction manager, persistence manager, database accesses are all implemented as MBean services, and these services can enquire the JMX MBean server about other available services so that they can interact with each other. When an MBean services relies on the service of provided by another MBean service, one can specify this dependency in a configuration file, and the JMX MBean server is responsible for creating and initializing those services in the right order.

2.6.1.1 JBoss EJB Containers

JBoss EJB containers host the EJB components and provide them access to other services in JBoss. The components are deployed by a service called EJBDeployer (*org.jboss.ejb.EJBDeployer*). The deployer first inspects the

deployment descriptor file[4] to find out the configuration of the component. Based from that information, the deployer then creates the container that will host that component, and initializes the container according to the specification of the component. The containers themselves are deployed as services that depend on other components such as transaction service, proxy factory (factory for creating proxy objects), persistence manager and lock manager (see Figure 2-6).

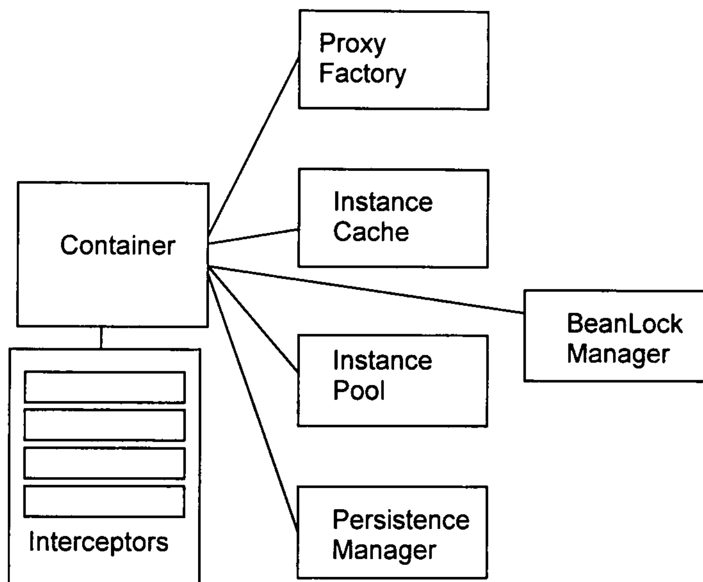


Figure 2-6 EJB Container and its relationship with other components in JBoss

During the initialization of the container (as part of component deployment), the container creates a representation of the component, an object proxy that will be used by clients to access the component (see Figure 2-7). The container creates the proxy by using a component named as Proxy Factory. The proxy contains a set of interceptors that intercept client's request at the client's side and an invocation context that contains the identity of the container, the JNDI name of the component, the EJB meta data, and the invoker. The invoker is the one that relays client's request to the server.

[4] In JBoss 3.2, the deployment descriptor is described in three files: ejb-jar.xml, jaws.xml and jboss.xml. These files are included in the jar file that contains the components and they describe the configuration of those components.

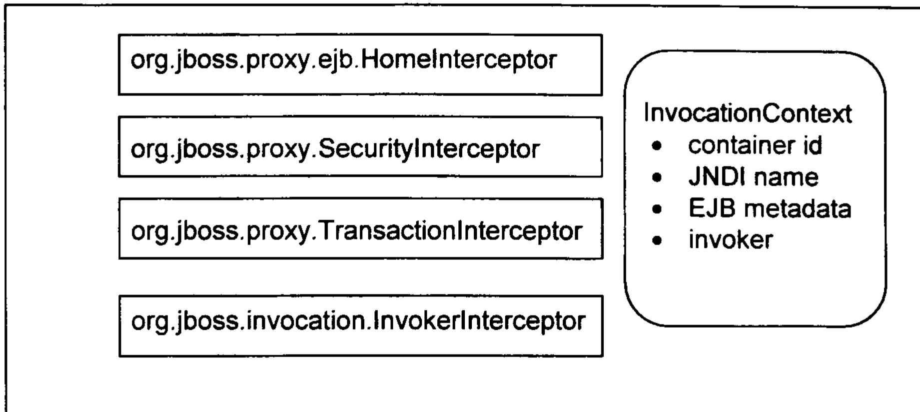


Figure 2-7 Proxy object

On the server side, requests from clients are handled by an invoker service. This service inspects the invocation context on each request, and delegates the processing of the request to the appropriate container.

In JBoss, A container hosts a specific type of enterprise bean component. A container has a set of interceptors, and different types of enterprise bean component (entity beans, stateless session beans etc.) have different set of interceptors. For example, a stateful session bean container has a set of the interceptors as below:

- ProxyFactoryFinderInterceptor: this interceptor is responsible for setting the proxy factory on the target container to match the proxy factory from the invocation.
- LogInterceptor: this interceptor is responsible for handling the logging for the invocation.
- TxInterceptorCMT: this interceptor is used for components with container managed transactions. It checks the transaction attribute of the invoked method, and sets a proper transaction context according to the transaction attribute of that method, then passes the invocation to the next interceptor.
- org.jboss.ejb.plugins.MetricsInterceptor: this interceptor measures the time for invoking the method and send the result via a message queue so that it can be read by other application.

- org.jboss.ejb.plugins.StatefulSessionInstanceInterceptor: this interceptor is specific to stateful session bean. It gets an instance context that contains the state of the accessed bean from the cache whenever available; otherwise it creates a new context for the invocation.
- org.jboss.resource.connectionmanager.CachedConnectionInterceptor: this interceptor checks for any previous connections opened during the previous invocations, and reopens the connections again so that they can be used in current invocation.
- org.jboss.ejb.plugins.SecurityInterceptor: this interceptor is responsible for checking the authorization for accessing this method and throws an exception if it is not authorized.

In the container, an invocation is processed first by interceptors, then the container invokes the appropriate method on the bean instance, and after that the result is passed back through the interceptors again, and sent back to the client. The details on how an client request is processed in JBoss are described in the next section.

2.6.1.2 Processing client request in JBoss

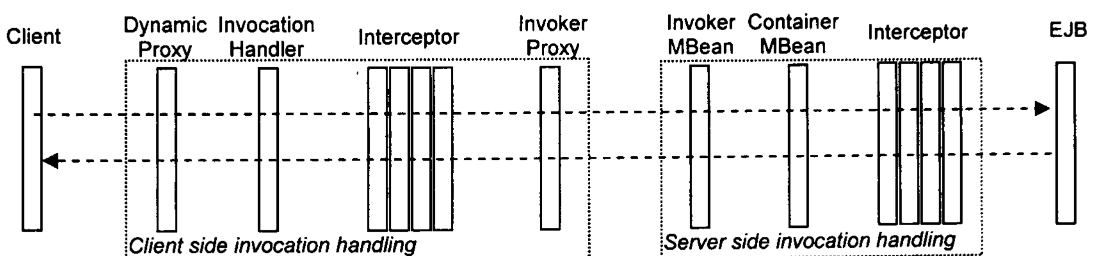


Figure 2-8 Processing client request in JBoss

Figure 2-1 provides an overview of the client and server side handling of an invocation in JBoss. When a proxy receives an invocation from a client, the proxy first checks whether the invocation is a local invocation or a remote invocation. Local invocations are optimized so that they do not have to pass through the invocation marshalling process. If the invocation is a remote one,

then the invocation is marshalled and forwarded to the appropriate container in the application server. The initial step in this 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 perform some necessary processing (see previous section) and 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. At the server side, the invoker MBean handles all incoming requests. It first unmarshalls the invocation, and passes them to the appropriate container where the EJB component exists where ultimately the invocation is handled by application logic. Each container may has its own set of interceptors, which will perform additional processing (see previous section)to the invocation prior and after being process by the bean.

2.7 Summary

This chapter provided an overview of component middleware concepts with particular references to the middleware platform that we will use as a target for designing and implementing the available components infrastructure. Among other things, we have described how a client request is processed in J2EE architecture, and the role of the transaction service in managing the transaction over this architecture. Based on this overview, an extension to JBoss to provide replicated components can be implemented as interceptors in both server and client side, and also as an invoker proxy and invoker MBean as well.

The following chapter deals with the problems and strategies for improving the availability of these platforms. We will provide a survey of existing techniques for improving the availability for systems similar to the one that we have described in this chapter.

3 Availability measures for component-oriented middleware

This chapter reviews strategies for improving availability in the component-oriented middleware infrastructure. Availability is improved by employing replication. As there are many participants involved in the infrastructure, an integrated approach on how to implement replication in a component-oriented middleware infrastructure is needed. The chapter provides a literature review on published work in this area and compare them to the work of this thesis

3.1 Background

Component-oriented middleware infrastructure, such as the one defined by J2EE standards, provides clear separation between components that have persistent state, and components for performing computations, that have only non persistent, session-oriented state. Therefore, it is natural to divide the replication support for these components into two categories: state replication and computation replication. State replication deals with masking data store failures to make persistent data highly available to components, while computation replication deals with masking application server failures where the computations are performed. The structure of this thesis and the literature review on this chapter therefore follows this observation.

There are two common approaches in implementing replication in distributed systems: active and passive replication. In **active replication** (Schneider 1990), each server replica processes every client invocation. As long as client requests are processed in the same order at all replicas, the replicas will produce the same output. This naturally requires replicas to be deterministic, that the state of a replica is determined only by its current state and the message it processes. On the other hand, **passive replication** (Budhiraja, Marzullo et al. 1993) allows nondeterminism by executing client requests only on one server, the primary, and distributing the updates to other replicas (backups) later on. Most server replication in component-oriented middleware

architecture follow the latter approach, due to non deterministic nature of most servers (e.g. because of thread scheduling, IO interrupts etc.).

A replication algorithm needs to make assumptions about the failure modes of components so that specific fault-tolerant measures can be deployed. Two widely used failure assumptions are *Byzantine failures* (Lamport, Shostak et al. 1982) and *crash failures*. A replica exhibits Byzantine failures when it can produce arbitrary (possibly malicious) output when a failure happens. Conversely, if a replica stops working when a failure happens and does not produce any output, it is called crash failure. By nature, Byzantine failures are more difficult to handle than crash failures. However, crash failures assumption is frequently considered sufficient for many applications, such as applications that are not life-critical (Schneider 1990).

The task of implementing a replication scheme is considerably simplified if all the correctly functioning replicas can be managed as a group with members having a mutually consistent view of the order in which events (such as message delivery, replica failures) have taken place. Design and implementation of fault-tolerant process group systems satisfying certain order properties has been a very active area of research during the eighties and nineties (Chang and Maxemchuk 1984; Peterson, Buchholz et al. 1989; Melliar-Smith, Moser et al. 1990; Birman, Schiper et al. 1991; Dolev, Kramer et al. 1993; Ezhilchelvan, Macedo et al. 1995). In an asynchronous environment where processing and communication delays cannot be estimated accurately, accurate detection of a failed component is not possible, as it is impossible to distinguish a failed component from a slow one (Fischer, Lynch et al. 1985). Process group systems circumvent this impossibility by relying on failure suspectors (Chandra and Toueg 1996). It is thus possible for a correctly functioning component to be eliminated from its group if other members suspect it to have failed. Well engineered systems can certainly minimise the probability of such occurrences.

Assuming components fail independently, to tolerate Byzantine failures, a minimum of 3f+1 components are needed to mask the failure of f components and output results need majority vote from 2f+1 components. For crash

failures, f+1 components are required to mask f components failures, assuming functioning components can communicate (i.e. there is no network partition) and component failures can be detected accurately; no output vote is required. If accurate failure detection is not possible, then 2f+1 replicas are necessary to mask up to f failure suspicions, with the majority f+1 components not suspecting each other.

3.2 Assumptions and requirements

Participants involved in component middleware infrastructure are: clients, web servers, application servers, and back-end databases. Application servers provide services to support components, such as transactions, persistence, security, etc. These services usually reside at the same process as the application server. Throughout this thesis we assume crash failure of computing nodes (application servers, database machines). After a crash, a node is repaired within a finite amount of time and is made active again. We choose to implement our replication approaches on application servers supporting J2EE architecture; hence it should provide the same correctness criteria as single J2EE application server. We also assume that back-end databases support ACID transactions.

One of the key concept of component middleware is to allow developers to focus on the specific business task of the components that they have to implement, thus the component developers should be relieved from the task such as managing access authorizations, persistence, and transactions, and the responsibility of managing replication to improve the availability as well. These tasks belong to the container implementers. Therefore, component replication should also be transparent to both clients and components. The transparency requirement imposes the following constraints: (a) no modifications to the API between client and component; (b) no modifications to the API between component and the container; (c) no modification to the existing middleware services and API. In later chapters we will explain in detail how replication can be introduced under above constraints.

One can take a look at each tier separately, and implement replication at each tier to enhance the availability of the system. Failure at the database tier makes

the persistent data becomes unavailable, and as a result, the whole system is also unavailable. Existing application servers as yet do not provide any support for managing persistent state replication. This means that they must rely on database vendor specific approaches for tolerating database failures (e.g. vendor specific database replication techniques).

At the middle tier, an application server failure causes the computations that were active to be lost. Commercial application servers make use of multiple application servers deployed over a cluster of machines to mask server failures so that these computations can be restarted on any other available server. For stateless computations, these scheme works perfectly, but for stateful computations, transactions are not supported and there are some limitations that are examined in detail in section 3.3.

At the client tier, a client crash usually does not pose critical issue to the availability of the system, as normally it does not cause other clients to be blocked from the service.

One important concept related to availability measures is that of **exactly once transactions** or **exactly once executions** (Little and Shrivastava 1998a; Little and Shrivastava 1998b; Tygar 1998; Frolund and Guerraoui 2000a; Barga, Lomet et al. 2002; Frolund and Guerraoui 2002). The concept is particularly relevant in web-based e-services where the system must guarantee exactly once executions on user requests despite system failures. The 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 at the server side. 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. 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 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.

The next section describes existing implementations for providing availability by employing multiple application servers over a cluster of machines, which is suitable for stateless computations. In here we explain why it is not sufficient for stateful computations and other limitations of the approach.

Several approaches on improving the availability by employing state replication are presented in section 3.4. The review focuses on approaches that improve the database availability by the use of middleware. The concept of exactly once transactions and several works that target this problem are discussed in section 3.5.

3.3 Availability measures in existing application servers

Most application server products available on the market currently support clustering. Clustering employs multiple servers (web servers, application servers) together with a load balancer to mask server failures. A cluster usually consists of several commodity low cost PCs located on the same LAN (Figure 2-1). At the front-end (PT – presentation tier), web servers receive queries from clients that usually access applications via web browsers. Before the requests reach web servers, typically they are processed by a web switch or load balancer (LB) to distribute the load among web servers (Cardellini, Casalicchio et al. 2002). In this approach, clients access a server by using a virtual IP, which is actually the address of the web switch or load balancer device. The device then distributes the load using a mechanism known as network address translation (NAT) (Egevang and Francis 1994). The mechanism works by editing the IP headers of packets to change the destination address for incoming packet and change the source address for outgoing packet. Load-balancing can also be employed at DNS (Domain Name System) level, by using round-robin DNS which gives different IP addresses for the same site name (Brisco 1995). More detail about this mechanism can be found in (Cardellini, Colajanni et al. 1999).

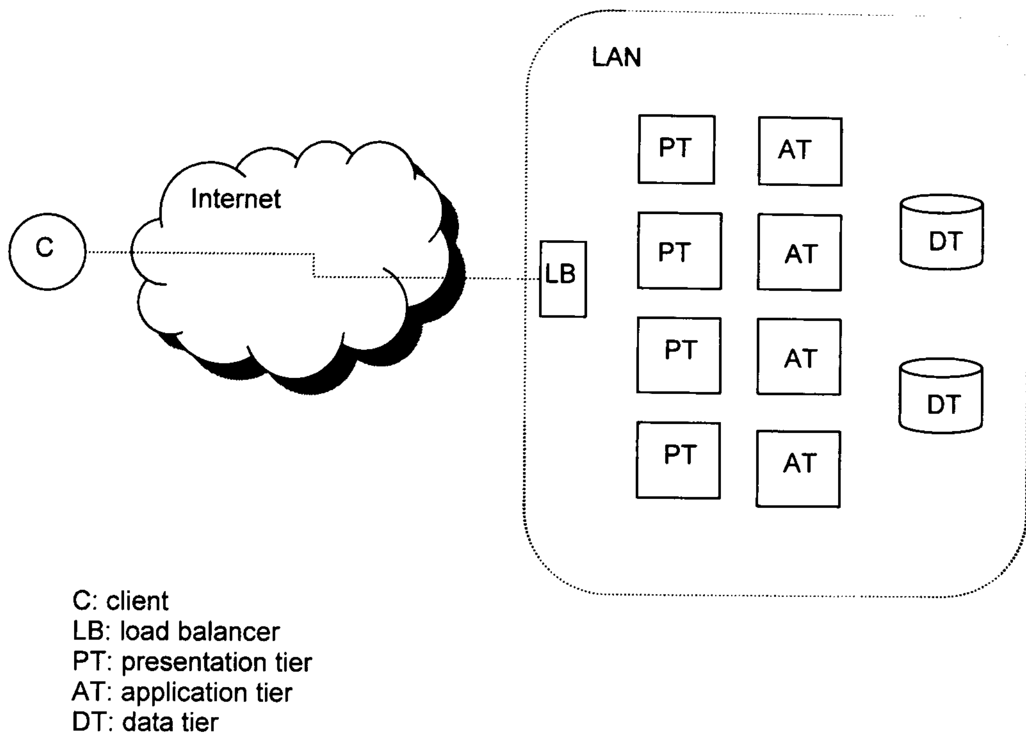


Figure 3-1 Three-tier architecture

Behind the load balancer components, there are several servers, which make up the n-tier system: presentation tier (PT) – web servers, application tier (AT) and data store tier (DT). On this system, one tier becomes a client for the next tier, for example, PT is the client for AT, and AT is the client for DT. This composition into tier is a logical one, and it is not necessarily reflected in hardware. One can, for example, configure a server machine as both presentation tier and application tier.

In practical situations one can either co-locate the presentation tier and the application tier on the same machine or separate the presentation tier and the application tier on different machines. The data store tier is usually located on a different machine. Co-locating the presentation tier and application tier reduces the latency for calls from the presentation tier to the application tier. However, one may need to scale the presentation tier only or the application tier only, and in this case, separating the presentation tier and the application tier is the only way to make this possible. The tier separation makes it more flexible to choose which tier needs to be scaled up.

At the application tier, a naming server or resource locator plays the similar role as the DNS for the Internet. For Java-based systems, such as J2EE application servers, this role is provided by a JNDI server. The JNDI server gives component home interface references to the web servers (the presentation tier), which are the clients of J2EE application servers, so that they can send requests to the components hosted by the application servers. Remote calls from the presentation tier are always initiated by queries to the JNDI server, in order to get a home interface of the accessed component.

There are three possibilities in configuring the JNDI server in a cluster environment. One can configure centralized servers (one or more) designated as JNDI servers, one can put independent JNDI server on each machine that host the application server and one can configure each JNDI server on each machine to share global reference to all components from all servers (Kang 2001). For the first approach, the main problem is if all of the JNDI servers crash, the system becomes unavailable even if there are some application servers still available. Furthermore, binding remote references from all servers to JNDI servers needs some significant time during recovery or initiation phase. On the other extreme, having each application server has its own JNDI server independently running on the same machine may increase the availability, but as each JNDI server does not know the existence of other application servers, there should be a mechanism to enable clients to find out other JNDI servers in case of failure. In third approach, each application server has its own JNDI server, like in the second approach, but each JNDI server maintains references to other application servers as well, thus clients can fail-over directly without contacting other JNDI server when the server that a client contacted fails. This approach has the same approach as the first one, i.e. requires significant time during the recovery or the initiation phase for the binding process.

Application server clustering typically are characterised for:

Scalability: the proposed configuration should allow the overall system to service a higher client load than that provided by the simple basic single

machine configuration. Ideally, it should be possible to service any given load. simply by adding the appropriate number of machines.

Load-balancing: the proposed configuration should ensure that each machine or server in the configuration processes a fair share of the overall client load that is being processed by the system as a whole. Furthermore, if the total load changes over time, the system should adapt itself to maintain this load-balancing properties

Failover: if any one machine or server in the system were to fail for any reason, the system should continue to operate with the remaining servers. Transparent failover (failures are masked from a client, who minimally might need to retransmit the current request) is an ideal, but rarely achievable with the current technology, for the reasons to be outline below. However, the important thing in current systems is that forward progress is possible eventually and in less time than would be the case if only a single machine were used.

3.3.1 Limitation in existing application servers availability

At the application server side, there are three types of components: stateless components, session-oriented conversational components and persistent components. In general, clustering provides limited fail-over capability for components hosted in application servers. For a stateless component, it is straightforward, as it has no state. The application server only needs to ensure that there is no duplication of execution when it tries to re-execute the query on a different server. For session components, many vendors employ in-memory replication for the session state among application servers. This is performed by propagating the session state to other servers on each invocation. However, handling session state replication is not a trivial task. Session components may involve in transactional invocation accessing database via entity beans, and when they fail, most clustering implementations simply create new sessions on a new server, and the client invocations are re-executed from the beginning as new transactions. This may result in inconsistency of the session state, as the updated state of the session from previous transaction is also wiped off, as the session is created as a new one. When the failure

happens in the middle of transaction commitment process, the outcome of this process is not known to other application servers. For single database access, the associated client invocation may either be executed twice (the transaction commits successfully in the failed application server and the backup assumes the transaction fails and re-executes the invocation) or may not be executed at all (the transaction fails to commit in the failed application server and the backup assumes it commits successfully), violating the exactly once execution property. For multi-database access, this may results in unnecessary blocking as the old transactions on the failed server may still have a hold on some resources. New sessions that are started may be blocked since they may require that same resources. The details of the problems and solution related to this approach will be discussed when we describe the implementation of computation replication in chapter 5. A summary of existing approaches that handle the same issue is provided in the literature review of exactly once execution in section 3.5.

One can also avoid in-memory replication for session state by always maintaining the state on an external database. The server failure is handled by restarting the component on a new server with the state from the database. This way, conversational components and persistent components are handled the same way, as persistent state. By this mechanism, the main focus now is how to make the database-tier fault tolerant. This will be the focus of chapter 4, when we discuss state replication on component middleware. A summary of existing approach for state replication is described in the section 3.4.

3.4 State Replication

Replication has long been researched in both database and distributed systems communities. Research on data replication in distributed systems has emphasized different targets and requirements than the one in database community (Wiesmann, Pedone et al. 2000b). In distributed systems research the focus has been mainly to provide fault tolerance. Database replication work has always assumed transactional data, while on distributed systems this is not a primary assumption. The following sections provide literature review

of database replication works that are relevant for our state replication approach, focusing on works that target middleware-based replication.

3.4.1 Database replication in general

Transactional data replication has also been widely researched (Bernstein, Hadzilacos et al. 1987; Gray, Helland et al. 1996). Many database replication strategies have been devised, but few only ever applied in commercial products (Kemme and Alonso 2000). The reason for this is that most approaches are not practical due to performance and scalability reasons. However, some vendors now provide database replication as part of their solution. A summary of these implementations and a discussion on how they can be used to support three tier architecture can be found in (Vaysburd 1999).

To guarantee consistency, the replicated system must appear to the client as if it only has a single copy. This correctness property is known as *one-copy serializability* (Bernstein and Goodman 1985). Many algorithms satisfy this property at all times by employing *eager replication* scheme, i.e. by propagating state update to all replicas eagerly before the transaction commit. In this way, all replicas will always be up to date with the latest committed transaction.

Available copies algorithm performs eager replication by sending write operations to all available replicas and sending read operation to any (nearby) replica. Performing transactions concurrently on different replica may lead to deadlocks that can only be detected later when the transaction is about to commit. To avoid this, one replica can be assigned as a *primary copy*, and all updates are performed first on this copy to determine any conflict with other transactions and then the updates are propagated to other replicas before the transaction commits.

As opposed to eager replication schemes, propagating state update to replicas can also be done lazily, i.e. after the transaction commit. This scheme is called *lazy replication scheme*. With lazy replication algorithms, it is possible that serializability violation is detected after some transactions commit, and these transactions cannot be aborted anymore. To keep the database consistent, the

effect of these transactions must be undone either by performing compensating transactions that can undo the effect of committed transactions or by performing manual intervention by a human operator. Despite this downside, many commercial products choose to implement lazy replication algorithms as they have higher performance compare to eager replication ones (Gray, Helland et al. 1996).

Recently, following the success of group communication approach in distributed systems, researchers have started to investigate the use of group communication for enhancing database replication (Agrawal, Alonso et al. 1997; Pedone, Guerraoui et al. 1998; Stanoi, Agrawal et al. 1998; Holliday, Agrawal et al. 1999; Kemme and Alonso 2000). A study in (Wiesmann, Pedone et al. 2000a) provides classifications of database replication schemes and suggests that group communications can be used for developing better eager replication schemes. Group communication, for example, can be used to define the order of conflicting transactions and to simplify the commitment protocol (Kemme 2003). (Kemme 2003) describes some of the issues in employing group communication for database replication as follows:

- The replication mechanism should allow concurrent execution of transactions as long as they do not conflict. This is a difficult task as it requires extensive access to other components within the database systems.
- Most implementations only allow a simplified transaction model, for example by only allowing a single SQL statement for each transaction. Multi-statements transaction execution requires the replication subsystem to track all changes performed by a transaction that might lead to a considerable overhead without adequate support from the database system.
- More investigations are still needed to define weaker transaction models that can still be acceptable for practical use and to develop a more adaptive failure-handling and recovery mechanism.

Integrating group communication sub-system into the database architecture requires access to the internals of the database, and this makes such schemes difficult to work with existing commercial databases. To relieve this problem, a middleware layer has been proposed to provide the required functionality for integrating group communication based eager replication schemes into existing commercial databases (Jimenez-Peris, Patino-Martinez et al. 2002).

However, middleware approach to database replications has some drawbacks. First, it cannot access the internal databases being replicated, meaning that it cannot access the locking mechanism used by the database, and possibly it has to perform redundant concurrency control mechanism that has already been performed at the database. The middleware layer must also have the ability to extract the concurrency unit information (e.g. tables, records) from the client queries (i.e. from SQL queries) so that it can perform its own concurrency control. But, on the other hand, middleware-based database replication is more suitable for middleware-based application architecture, such as web server clusters, J2EE-based and CORBA-based e-services.

3.4.2 Middleware-based data replication

Following the term used in (Patino-Martinez, Jimenez-Peris et al. 2005), we can classify middleware-based database replication into two categories: black box approach and grey box approach. Black box approach treats databases as black boxes and does not require any modification from existing database implementation, while grey box approach assumes the middleware replication has limited access to the database and therefore this approach may require some modification from existing database implementation. There are some research projects investigating middleware-based database replication, and these will be discussed in the next sub chapters.

3.4.2.1 Middle-R

Middle-R is a set of group communication based replication protocols that are implemented at the middleware layer developed as part of ADAPT project (Patino-Martinez, Jimenez-Peris et al. 2005). It requires some access to the internals of the database to support the replication, hence this approach is considered as *grey box* approach. In Middle-R, the database needs to be

modified to provide support for obtaining the write-set of a transaction and for applying the write-set into the database.

The model assumes that the middleware layer is located as the same site as the database. Thus there is one to one correspondence between a middleware layer and a database instance. The system runs on an asynchronous system augmented with failure detectors that can provide reliable multicast with strong virtual synchrony.

Data in Middle-R is partitioned into *basic conflict classes* (Bernstein, Shipman et al. 1980). A conflict class is a unit of concurrency control; this can be a record, a table or even a database partition. Each basic conflict class is associated with a queue. A transaction that requires access to more than one basic conflict class is said to access a compound conflict class (a set of nonempty basic conflict classes). Each conflict class (either basic or compound) has a master site that will execute all transactions that access this conflict class. Therefore, each transaction will have a master site that executes the transaction and transactions that access the same set of conflict classes will always be executed at the same site.

The basic algorithm works as follow: a client can submit a request to any middleware layer. The request contains all operations that are to be performed within a transaction. Hence all accessed data items are also known. Upon receiving the request, the middleware layer broadcast the request to all sites. The master site for that transaction puts the request on a queue for the basic conflict class (or basic conflict classes) that is accessed by that transaction. The master site always executes the transaction that comes up as the first on all queues of basic conflict classes that it access. After the transaction finish, the master multicasts the update write set for that transaction to all sites. Read-only transactions are processed by reading from a database snapshot; therefore they can be executed directly at the site that receives the request from the client.

There are three different protocols described in (Patino-Martinez, Jimenez-Peris et al. 2005). The first protocol, DISCOR, limits transactions by only allowing them to access one basic conflict class only. The second protocol,

NODO, allows transactions to access multiple basic conflict classes, but it may abort transactions whenever a possible non serializable execution is detected. The third protocol, REORDERING, reduces the amount of transaction aborts by performing transactions reordering on non master sites so that those transactions are still serializable hence are not aborted.

The first protocol, DISCOR, orders all conflicting transactions at their master sites, and upon finishing those transactions, the master sites multicast the write set to other sites using FIFO ordering. As each transaction can only accesses one basic conflict class and each basic conflict class has a unique master site, there will be no overlapping/conflict between transactions that are executed on different sites. Hence, the execution is serializable since all conflicting transactions will always have the same order defined by their master sites.

The second protocol, NODO, allows transactions to access more than one basic conflict class by defining compound conflict classes that contain all basic conflict classes accessed by those transactions. In this protocol, requests from clients are multicast to all sites using total order, so that each server receives client requests in the same order. To allow executing transactions as soon as possible, the protocol employs optimistic total order multicast (Kemme, Pedone et al. 2003). Under optimistic total order multicast, each multicast message is delivered twice, i.e. the message is first delivered optimistically (OPT-delivered) as soon as possible after the message is received by a site and later on the message is delivered again for the second time (TO-delivered) after the total ordered has been computed. When the transaction is optimistically delivered on a site, the transaction is added to all basic conflict class queues that it accesses. The master site executes the transaction that is ready, i.e. it comes up as the first on the queues of all basic conflict classes that the transaction access. However, the transaction cannot be committed yet. It must wait for the TO-delivery for the transaction, the one that define its total order. If the TO-delivery for a transaction arrives but that transaction is still blocked by other transactions, then the blocking transactions are either aborted (if they are local transactions) or reordered (if they belong to other sites).

The third protocol, REORDERING, is a modification of NODO, by allowing some transactions that are executed at the same master site to follow different order from the one dictated by the total order multicast, in order to reduce the number of aborted transactions. Not all transactions on one master site can be reordered. If T2 blocks T1 and the set of T2's basic conflict classes is not a subset of the set of T1's basic conflict class, then when T1 arrives before T2 according to the total order delivery, T2 must still be aborted to allow the execution of T1. Otherwise, if T2's basic conflict classes are a subset of T1's basic conflict classes, then T1 can wait until T2 finish and the new order is multicast to other sites after T2 finish.

The middleware layer in Middle-R implementation runs on each database site, which is different from J2EE model, where there is no one-to-one correspondence between middleware layer server (i.e. the application servers) and the databases. The approach also assumes that all operations to be executed within a transaction are submitted as a request to the database. It cannot be used for applications that require multi-database transactions, as two phase commitment is not supported. Furthermore, it requires some database modifications to support the protocol. Therefore, the approach does not appear to be entirely suitable for J2EE application servers, such as the one being targeted in this thesis, i.e. replicating database from the application servers directly without any modification required for the database.

3.4.2.2 Ganymed

Ganymed (Plattner and Alonso 2004) is a middleware-based replication tool designed to support transactional web applications. Instead of 1-copy serialization, it supports *snapshot isolation* (Berenson, Bernstein et al. 1995), which is strictly weaker than 1 copy serializability. In snapshot isolation, transactions read from a database snapshot (based on the value of the data at the time the transaction started) so that they are not blocked from accessing data that being used by other transactions. As the result, sometimes a transaction must be aborted, i.e. when a transaction T reads data x and there is another transaction that commits while the transaction T is still active, and also accesses data x that is used by T.

Ganymed architecture consists of a scheduler and a set of database replicas that support snapshot isolation. One replica is designated as a primary copy while the rest are backups. The scheduler coordinates client requests by sending all update transactions to the primary copy and sending read only transactions to the backups. The scheduler also assigns a timestamp for each transaction at the start of the transaction, which will be used by database to determine which data version that will be accessed by that transaction. At commit time, after the primary copy successfully commits the transaction, the scheduler extracts the write set for that transaction from the primary and sends the write set to all backups.

The main limitation of Ganymed is that it requires database that support snapshot isolation. This makes the approach cannot be used directly in multi-database transactional setting (Schenkel, Weikum et al. 2000). Furthermore, it also requires database support for write set extraction. For implementing it on J2EE architecture, Ganymed scheduler can be placed as an intermediate layer between application servers and database servers.

3.4.2.3 Distributed versioning

Similar to Ganymed, Distributed versioning (Amza, Cox et al. 2003) also implements replication tool that targets transactional web applications. In this approach, the middleware layer performs its own concurrency control by maintaining the version number of each table on the database. Requests from clients are sent to databases only after they are cleared from conflicts, i.e. there is no other transaction that accessing the table with lower version number.

Distributed versioning requires the transaction to declare all the tables that it access at the beginning, which will be used to determine conflict among transactions. When a client starts a transaction, it sends a list of the table names that will be accessed by this transaction together with their access mode (read only or write). The middleware layer assigns a new version number for each of these tables. When the client send an operation to access a table, the middleware layer compares the version number of the table possessed by the transaction (the one that is assigned when the transaction starts) with the version number of the table that is currently on operation. The middleware

layer sends the access to the database only if the number matches. When the transaction commits, the scheduler increases the table versions accessed by this transaction.

Distributed versioning duplicates the concurrency control mechanism that already been performed at the databases. It avoids deadlock by ensuring all access to tables on the database will not contain circular. As it allows non strict execution of transactions (Bernstein, Hadzilacos et al. 1987; Breitbart, Georgakopoulos et al. 1991), the approach may not be suitable for distributed transaction settings. As with Ganymed approach, it requires a single scheduler that controls all accesses to the database, and since the scheduler performs more tasks compare to Ganymed, it may become the source of bottleneck. However, for implementing it on J2EE application servers, it has the same place as Ganymed, i.e. as an intermediate layer between application servers and database servers.

3.4.2.4 C-JDBC

Clustered JDBC (C-JDBC) (Cecchet, Marguerite et al. 2004) is a middleware for database clustering developed by ObjectWeb consortium. It provides transparency for clients to access a set of replicated and partitioned databases via a standard JDBC driver. C-JDBC architecture consists of C-JDBC drivers that run as part of clients' process, C-JDBC controller and backend databases. C-JDBC controller, via C-JDBC drivers, provides a virtual database to clients by relaying requests to appropriate databases transparently. C-JDBC schedules all requests from clients by sending read operation to any single database and sending update, commit or abort operation to all databases. C-JDBC also queues update access to the same virtual database so that at any given time only a single update, commit or abort operation is in progress. To allow faster response, C-JDBC also provides support for partial replication. In this setting, instead of replicating full database in all replicas, the database is partitioned into several databases, which will be provided by different backend databases. C-JDBC then parses client requests and sends those requests to the appropriate backend database.

Failed replica is handled by disabling it and removing it from further access. Before a failed replica is allowed to join in again, it must update its state to the latest state via external mechanism. C-JDBC allows configuration of several controllers that share information about their backend databases. When one of the controller fails, the other controllers can take over its backend databases so that it can be used. However, the paper does not provide information about whether the clients connected to failed controller can fail-over to the new controller.

Among the other implementations described above, C-JDBC is the most similar to our implementation for state replication in chapter 1. However, our implementation is developed independently from C-JDBC, and at that time the author did not have any knowledge about C-JDBC development. Compared to C-JDBC, we move a step further by implementing JDBC driver replication to allow clustering configuration, therefore the implementation described in chapter 4 does not require the existence of a single controller to intermediate access from application servers to databases.

3.5 Computation replication

As it has already been mentioned at the beginning of this chapter, one important requirement that availability measures must meet in multi-tier architecture is that of *exactly once executions*. The clustering approach described in section 3.3 and the replication mechanism supported in existing middleware products currently do not provide exactly once execution guarantee. If a failure happens just before the server (or cluster of servers) sends the reply back to the client, the client does not know whether its request has actually been executed or not, and if the client resends the same request, it may result in duplicated invocation.

If failure happens in the middle of request processing, say in application server, the clustering approach does not guarantee that the system will automatically failover correctly to another available server. Clustering can only help if a server fails exactly in-between requests from clients.

In the following sub chapters, we will describe several mechanisms and solutions to solve the exactly once execution problem.

3.5.1 Transactional queue

The most popular approach for solving this problem is by employing transactional queues (Bernstein, Hsu et al. 1990). In this approach, clients submit a request to a transactional queue as a transaction. The server then retrieves the client request from the queue, processes the request, and submits the result to the queue as a separate transaction. The client receives the result back from the queue as another transaction. By this way, the requests and their results cannot be lost, as they always move from one node to another under the control of transactions. If there is a failure, the transaction mechanism guarantees that the request (result) is either available at the original place or the new place.

One can make exactly-once execution of requests by making the whole process (client submits request, server process the request and client receives reply) as a transaction, however, this way makes the server holding all involved resources until the transaction finishes, thus increases resources contention. Transactional queue avoids unnecessary resource blocking by separating the request sending, request processing and reply into separate transactions. This approach has been supported in many On-Line Transaction Processing (OLTP) products (e.g. IBM MQ Series, BEA Tuxedo, Microsoft MTS).

Multi-transactions execution within one client request is supported by extending the model to store the temporary result in-between transactions in separate queues. For example, if an execution of a client request r1 is to be executed as three transactions t1, t2 and t3, then the client first submits the request to the queue for client request, then the server retrieves that request to be executed as t1, and submits the result to another queue, which then will be retrieved again for the execution of t2, and so on. One problem with this approach is that if the client wants to abort the execution after the first transaction of the multi-transactions request has committed. The only way to cancel that committed transaction is by executing compensating

transactions(Korth, Levy et al. 1990) to undo the effect of committed transaction, which on some cases may violate the ACID properties of the transactions.

However, this approach requires developers to develop applications in such way that no state kept in the application servers between successive requests from clients. All the state either kept in the requests themselves or in the backend database servers. From the performance point of view, this approach also requires at least three transactions for each client request: client submits the request, the server processes the request and the client retrieves the result from the queue.

3.5.2 Transactional client

(Little and Shrivastava 1998a) extends the transactional guarantee to the client side by making the web browsers as part of the transaction processing. This is done by making the browser as a resource which can be controlled by the resource manager from the server side. The browser also should be able to store its state persistently, and this can be implemented via mechanism such as cookies. In this way, the exactly once execution problem is guaranteed as whenever there is a failure (either on the server or on the client), the transaction will be aborted, and all the changes at the client side will be cancelled automatically. The drawback of this approach is that it requires the clients to be transactional.

CORBA with Object Transaction Service also allows client to be transactional. Felber and Narasimhan (Felber and Narasimhan 2002) implement failover capability on the client-side by employing multi-profile IOR (object references). This IOR enlists all available server replicas for that object. When the primary server fails, the client-side ORB will transparently find new available server based on the list of addresses contained in the IOR.

3.5.3 E-transactions

Frolund and Guerraoui (2002) approach the problem by first defining e-transaction (Exactly-once transaction) that describes the semantic of transaction execution in three-tier architecture. The specification extends the

transactional guarantee to the client by requiring that the client will always receive the result of its request as long as it does not fail. A client submits requests to an application server; the application server processes requests, interacts with databases and sends the result back to the client. Application servers are replicated to guarantee that they are always able to compute the result, to update the state from the database and to return the result back to the client.

Formally, e-transactions specification is as follows. An e-transaction has three properties: *Termination*, *Agreement* and *Validity*. Termination guarantees liveness, meaning that every request will always have a result, as long as the client that issues them does not crash. Agreement guarantees safety, that every computation gives correct result, and Validity excludes trivial solutions to the specification.

Properties of e-transactions	
Termination.	(T.1) If the client issues a request, then, unless it crashes, the client eventually delivers a result. (T2) If any database server votes for a result, then the database server eventually commits or aborts the result.
Agreement.	(A.1) No result is delivered by the client unless the result is committed by all database servers. (A.2) No database server commits more than one result for the same request. (A.3) No two database servers decide differently on the same result
Validity.	(V.1) If the client issues a request and delivers a result, then the result has been computed by an application server with the request as a parameter. (V.2) No database server commits a result unless all database servers have voted yes for that result.

Table 3-1 E-transaction properties

The specification relieves the client from being transactional while still guaranteeing end-to-end reliability by requiring the correctness of e-Transactions applies only as long as the client does not crash.

There are three different implementations described for the e-transaction. In these implementations, application servers are modelled as stateless servers. They do not maintain state across multiple client requests. They also do not model chained invocations, where an application server invokes services from other application servers. The first one (Frolund and Guerraoui 2002) solves the problem by making the application servers available as primary backup replicated servers. When the primary fails, the client reissues the request to the backup after timeout period and one of the backup takes over the computation and returns the result back to the client. This solves the transaction commitment blocking problem as the application servers take the role as replicated transaction coordinator as in (Reddy and Kitsuregawa 1998). As long as there is an application server available (with the assumption that the databases are always available or will be recovered within a reasonable time), the algorithm solves the e-transactions problem above. The protocol assumes the existence of perfect failure detectors[5] (Chandra and Toueg 1996) among application servers, as it is required for the primary backup scheme (Budhiraja, Marzullo et al. 1993).

The second implementation (Frolund and Guerraoui 2001) employs application servers that relaxes the perfect failure detector requirements. This protocol implements e-transactions on asynchronous environment augmented with eventually perfect failure detectors between application servers. The abstraction used for the synchronization is write-once register objects (wo-registers). This objects record once only and subsequent read will consistently result the first value written. There are two wo-registers used, say *regA* and *regD*. for Upon receiving a client request, an application server (say *appA*) writes a record containing the application server identity into *regA*. This step is necessary to avoid other application servers execute the same request, and also to record the fact that *appA* is computing the result. After computing the result and preparing the transaction to commit, *appA* writes the transaction commit outcome into *regD* and sends the result back to the client. If at later time *appA* fails, another application server (say appB) that takes over the computation

5 Perfect failure detectors has strong accuracy and strong completeness. They never suspect correct processes and all correct processes will eventually suspects all crashed processes.

knows from *regA* that *appA* has probably performed the computation and *appB* must check *regD* to find out if there is any result for that request.

The third implementation (Frolund and Guerraoui 2000b) discusses a pragmatic approach in single database context. In this situation, the protocol removes the need for coordination directly at the application servers, and pushes it back to the database layer via testable-transaction abstraction. Testable-transaction abstraction is an abstraction mechanism that allows the application server to check the outcome of a transaction, even after the transaction finishes. Thus, application servers become independent against each other, and as the result this, the system becomes more scalable.

The key issue in e-transactions and its solution lies on the model of application servers as stateless servers. In this context, the only state that needs to be handle when the application server fails is the transaction context (and the transaction outcome decision, if exist). Therefore, when the system has only one database, such as the one in the pragmatic approach above, one can even push all the required state information to the database. This model choice also removes the difficulties to handle chain invocation between application servers, as they don't handle the state by themselves. Any failures can be transferred to another application server, as long as the database has not committed the transaction yet.

Stateless server approach is suitable for some of internet-based applications. However, for many applications, this requires developers to structure the applications so that no state is kept in the application server between client's requests. The session state, if any, must also be kept in the backend database, which incurs additional overhead. In our approach, we model the application servers as stateful servers, capable of maintaining session state between requests which is encapsulated in stateful session beans, in addition to the transactional data maintained by the database.

3.5.4 Interaction contract

(Barga, Lomet et al. 2004) takes a different approach on how to guarantee exactly once execution on internet-based e-services. They focus the study on

how to do the recovery when there is a failure. In this approach, there is no replication of servers or databases, but their solution is focused on how to properly perform checkpoints and message logging. In order to do this, components are classified into three types: **persistent components** (e.g. web servers, application servers, and browsers), **transactional components** (e.g. database servers) and **external components** (user displays, printers). Persistent components have persistent state, transactional components have persistent guarantee only when transactions commit, and external components do not have persistent guarantee at all. Browsers are considered as persistent components; therefore they must have logging capabilities to support the recovery process. All components are required to be **piecewise deterministic** (PWD) (Strom and Yemini 1985), meaning that components can recover by restoring their state from the previous checkpoints and replaying all input messages that arrive after the last checkpoints.

The baseline algorithm for implementing recovery guarantees for component applications is based pessimistic logging. Components log all incoming and outgoing messages and immediately store them in stable storage. On recovery, a component can therefore start its state from its last stored state and replay the incoming messages received after that. This baseline algorithm can be optimised by avoiding the logging on the receiving components. When a component fails, on recovery it can asks other components to resend the lost messages.

Interactions between these components are specified as contracts, so that each component can manage its logs and checkpoints independently as long as it conforms to the specified contracts. There are four types of contracts specified between the above three components: **Committed Interaction Contract** (CIC) and **Immediately Committed Interaction Contract** (ICIC) for the interaction between persistent components; **External Interaction Contract** (XIC) for the interaction between external components and persistent components; and **Transactional Interaction Contract** (TIC) for the interaction between persistent components and transactional components. CIC reduces the number of force-logging required for the interactions between persistent components, while ICIC is similar to the baseline algorithm

described above. XIC allows duplication for messages sent to external components, and also allows best effort guarantee only for input messages from external components. TIC guarantees the persistent state in persistent components only after the transaction is expected to commit, but if the transaction is aborted or rollback, only the transaction's effect on the transaction component that is erased, and there is no guarantee at all at the persistent component state.

Although this approach has the benefit of reduced numbers of forced logs required, compared to pessimistic message logging (Alvisi and Marzullo 1998), the contract does not specify clearly about the state of the persistent components when the transaction is not committed (aborted or cancelled). An implementation that is simply let the uncommitted state 'poison' the persistent component is fine, according to the specification.

The notion of component in this approach refers to a software application as a whole, such as web servers, application servers, database servers while our approach uses the notion of component as a smaller piece of software such as EJB beans and servlets. To compare both approaches, we can assume that the state and the log of a software application (e.g. an application server) as the composite of all state and logs of the smaller components (e.g. EJB beans) that it hosts. Therefore, the composite of EJB beans in our approach becomes the persistent component in the interaction contract approach.

The interaction contract also assumes the interaction between persistent components and transactional components are always on one-to-one basis. Furthermore, it also assumes that the commitment process consists of single commit request from the persistent component and a commit reply from the transaction component, which rules out the two phase commitment required for the distributed transaction setting. One can assume that two phase commitment can be seen as a single commit request-reply interaction, i.e. the first request that initiate the commitment process (the prepare message) is considered as the commit request from the persistent server and the last reply that concludes the commitment process (the transaction commit/abort confirmation message) as the commit reply from the transactional component.

But this does not govern the interaction between a persistent component and two or more transactional components, which can happen in distributed transactions. For example, when a persistent component fails in the middle of two phase commitment which involve two other transactional components, the transactional interaction contract (TIC) fails to distinguish the state of transaction commitment between before taking a decision for the transaction outcome (before the first phase of the 2PC finishes) and after taking the decision (after the second phase of the 2PC starts). This distinction is necessary when we replicate the persistent component (e.g. the application server), and can avoid unnecessary blocking on the resources held by the transactional components.

3.5.5 J2EE replication framework

As part of ADAPT project[6], Babaoglu et al. developed a framework for prototyping J2EE replication (Babaoglu, Bartoli et al. 2004). They approach J2EE replication by creating a framework that provides hooks for implementing replication algorithms on top of existing J2EE servers. The framework intercepts client invocations, both at the client's side and also at the server's side, performs some processing, passes the control to the replication algorithm and returns the execution control back to the J2EE server (see Figure 3-2).

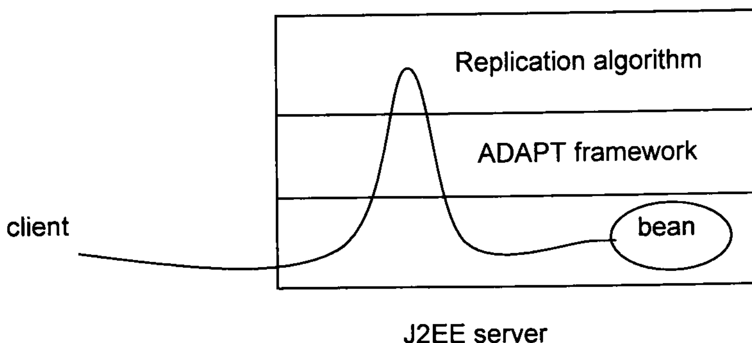


Figure 3-2 ADAPT framework (Babaoglu, Bartoli et al. 2004)

6 ADAPT: Middleware Technologies for Adaptive and Composable Distributed Components.

The framework provides hooks for intercepting request processing at three points (Figure 3-3): at the client's side just before control leaves the caller (1), at the client's side but inside the server provided stub (2), and at the server's side before the request is processed by the component (3a and 3b).

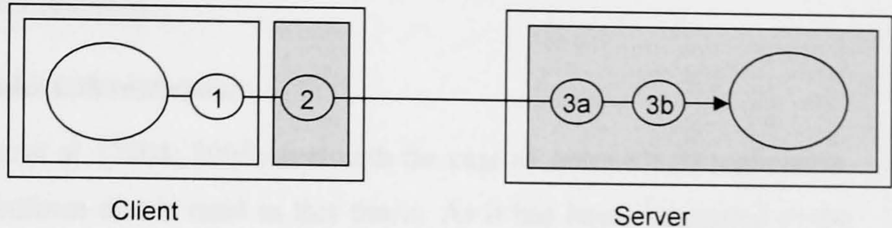


Figure 3-3 intercepting points on ADAPT framework (Babaoglu, Bartoli et al. 2004)

Interception point 1 is provided to handle client replication. When the client is replicated, the framework provides the replication algorithm to synchronize client's request with other client replicas. Interception point 2 is for handling failover when a target server fails. Interception point 3 is to control the state replication and propagation at the server's side. For EJB replication, interception point 3 is divided into two points: the first point (3a) is before the component reference is resolved, and the second point (3b) is after the reference and other properties such as security and transaction have been resolved. Interception point 3a is useful when the replication algorithm needs to instantiate a component itself, and interception point 3b is necessary as the replication algorithm may need the transaction context and other component properties that are only available after the component reference has been resolved. The framework also intercepts accesses to/from JNDI service and transaction service.

ADAPT framework has been used as the basis for a stateful EJB replication (Wu, Kemme et al. 2004) which will be described in the next section. Our approach is similar to this project, although we develop it independently. However, we think that the inception points provided by ADAPT framework is not sufficient if we want to handle replication that supports distributed transactions. For handling distributed transactions, the replication algorithm

must possess the information about the last status of the transaction outcome, especially during two phase commitment, and this information cannot be deducted by simply intercepting the transaction manager interaction with the resources and the application server. We argue that modifying the transaction manager is necessary if we want to properly handle replication that support distributed transactions.

3.5.6 Stateful EJB replication

Wu, Kemme et al. (2004; 2005) deal with the case of stateful EJB replication, the same platform that is used in this thesis. As it has been mentioned in the previous section, the algorithm is implemented using the ADAPT framework.

The algorithm assumes that the clients and EJBs are not multithreaded and are blocked when sending requests to the application servers or other EJBs. Therefore, there is no concurrency problems for stateful session beans as they can only be used by one client with no multithreading. Our approach follows the same assumption. The algorithm also assumes that when an application server fails, all active connections to the database will be lost and all active transactions are aborted by the database. While this might be incorrect for two phase commitment that involves more than one database, this is true for single database access with optimisation, i.e. when the transaction manager employs one phase optimization.

There are three variant algorithms being proposed: one client request is executed as one transaction (1-1), many client requests are executed within a single transaction (N-1) and one client request executes many transactions (1-N).

The algorithms implement primary backup replication, which consist of two parts: client-side (at the interception point 1 and 2 of ADAPT framework, see 3.5.5) and server-side (at the interception point 3 of ADAPT framework). The 1-1 algorithm proceeds as follows. The client replication algorithm (for now on this is referred as Client Replication Manager – CRM) intercepts each request, attaches a unique id, then forwards the request to the primary. The CRM maintains a list of server replicas and a pointer to the current primary. If

the current primary is down or does not give a positive response, the CRM queries other server replicas on the list asking whether they are the primary or not. The CRM then forwards the request to the server that responds back as the new current primary.

The primary processes the request, and before committing, it propagates the updates to all replicas, inserts the transaction identifier txid into the database. After receiving commit confirmation from the database, the primary again multicasts a committed message to replicas, and send the result back to the client.

For the N-1 algorithm, the client replication algorithm stores all requests submitted to the primary and their associated results. These requests are associated with the transaction id within which they are executed. When the primary fails, the client will try to replay the requests execution at the backup site and compare the results with the ones it has. If there is no nondeterminism (the results are the same), the computation can continue at backup site and the client perceives exactly once execution for its requests, otherwise the transaction must be aborted and the client is informed.

The N-1 algorithm above can guarantee exactly once execution as long as there is no nondeterminism when the replays of client requests are performed at the backup site. Nondeterminism can happen if another transaction has updated the same data after the first execution at the primary server but before the requests replay at the backup. (Wu and Kemme 2005) also proposes N-1-ordered extension for the algorithm above. In this algorithm, accesses to the database are ordered according to the order of the original transaction requests from the client. On replay, the requests are executed according to this order, so that the chance of exactly once execution increases.

The 1-N algorithm handles the replication similar to the 1-1 algorithm, only in this case one request can executes many transactions, and when there is a failure, the new primary must handle a case where some of the transactions have already committed while some other transactions are still active. To solve this problem, the algorithm assumes the existence of compensating

transactions, thus all committed transactions can be compensated and the request can be executed as a new request as is the one in the 1-1 algorithm.

The algorithms above assume single database access only. For multi databases access, the paper suggests a solution by intercepting messages (prepare and committing messages) from the transaction manager. In this case, the primary does not need to insert the transaction identifier to the database, but must send three multicasts to backup instead. Upon intercepting the first prepare message from the transaction manager, the primary multicast a *preparing* message to the backups. Upon intercepting the first commit message from the transaction manager, the primary again multicasts a *committing* message (in case of commit) to the backups. Finally, the primary multicasts a *commit*/abort message after the transaction has terminated. The new primary then can handle how to process a transaction according to the multicast messages that it has received for that transaction, either by sending aborting messages or commit messages to all involved databases. Therefore, this mechanism described in the paper assumes that the new primary *knows already* which databases are involved within a transaction, so that it can always follow up transactions from a failed primary. This assumption cannot be easily upheld in EJB application server, as it requires the developer/deployer to specify all involved databases for each invocation at configuration time, whereas normally they are determined at execution time.

3.6 The approaches developed in this thesis

We consider replication on application server supporting J2EE architecture with minimal change to the internal of the application server. As it has been outlined earlier in this chapter (see section 3.2), there should be no modification to the APIs that are used by clients, components and services on the application server.

The replication strategy is divided into two approach:

(i) *State replication for supporting persistent components*: persistent state is stored on multiple databases; here database failures can be masked, so the transaction will be able to commit provided the

application server can access a copy of the state on a database; the approach should support clustering configurations, where multiple application servers may execute different transactions at the same time.

(ii) *Computation replication for supporting session oriented components*: instances of session components are replicated on the cluster of application servers; here application server failures can be masked, and the sessions running on a failed application server should be able to continue on different application server, including continuing the transaction processing whenever possible.

3.6.1 Failure assumptions

We assume crash failures of application servers and databases. A transaction manager exists on each application server, controlling the transactions running on that application server. Hence, whenever an application server crashes, the associated transaction manager will also crash as well. Application server crashes can be accurately detected by the underlying group communication system. When accurate failure detection is not possible (asynchronous environment), a minimum of 2f+1 replicas will be required, otherwise f+1 replicas would be sufficient for the synchronous environment. In a clustered environment, it is possible to engineer a system where accurate failure detection is possible. Our design makes use of a group communication system that was designed for asynchronous environment, so will also work in synchronous environment as well.

3.6.2 State replication approach

To introduce state replication into a J2EE application server, an attractive approach that allows minimal modification to the existing application server is by implementing a database proxy that can be plugged in into the application server as a JDBC driver. This proxy intercepts all interaction between an application server and its external database resources; hence it can introduce state replication into the application server smoothly, without any modification to other parts of the application server. The proxy performs replication by using 'available copies' approach to maintain replicas of state on a set of

databases (Bernstein, Hadzilacos et al. 1987). For clustering configuration, the proxies on different application servers coordinate with each other to ensure the consistency of all replicas despite multiple transactions running on different application servers. This is done by ensuring that all proxies use the same replica to satisfy load requests for relevant entity beans. This replica determines the ordering of conflicting transactions; hence preserving the consistency. The state replication approach developed in this thesis works well in distributed transaction settings, as it also handles all interactions between application servers and databases for committing the transaction with two-phase commit.

3.6.3 Computation replication approach

Computation replication is implemented by using primary copy replication approach. The approach goes well with J2EE application server perfectly, as on this server, typically stateful session beans are only used by the same client, they are not shared among multiple clients. Session state check-pointing is performed at the end of each client request invocation; therefore client session can be continued on backup only by repeating the last unfinished invocation. Transactions failover is supported by including the transaction information (the transaction id, the information of all resources involved in that transaction and the transaction outcome decision) in the checkpoint.

To perform computation replication, modification of the internals of the application server is unavoidable. These modification includes:

(i) intercepting client invocations, before and after they are processed by the application server.

(ii) retrieving the state of a session bean within an application server, and installing it on another server

(iii) intercepting the commitment process; i.e. is right after the transaction takes a decision about the outcome of a transaction, prior to performing the second phase of the two phase commitment process.

(iv) retrieving the information about current active transaction within an invocation

Fortunately, the platform that is used for implementing the approach (JBoss) provides almost all the necessary hooks for the above requirements. Requirement (i) can be provided by implementing specific JBoss interceptors, which will be executed on each invocation. The state of a session bean (requirement (ii)) is also supplied as part of the interceptor mechanisms in JBoss. However, requirements (iii) and (iv) require modification of existing transaction manager within JBoss, as existing APIs for transactions (JTA and JTS) do not provide access to retrieve information about resources involved within a transaction, which is maintained by the transaction manager. The session state check-pointing and application server failure detection are performed using an open source group communication implementation from JGroups (Ban 1998).

3.6.4 Comparison with other approaches

As a summary, all the approaches are compared with each other in the following tables (Table 3-2 and Table 3-3). On the last column of each table is our own implementation that will be presented in chapters 4 and 5.

Aspects	Middle-R	Ganymed	Distributed versioning	C-JDBC	JDBC proxy (our approach)
Consistency criteria	1 copy serializability	Snapshot isolation	1 copy serializability	1 copy serializability	1 copy serializability
J2EE support	No	Possible	No	Yes	Yes
Backend database requirements	Modification required	Database supporting snapshot isolation with modification for write set extraction	Standard database	Standard database via JDBC driver	Standard database with strict 2PL via JDBC driver
Multi-databases transaction support	No	No	No	Yes	yes
Clustering support	Yes	Yes	Yes	No	yes

Table 3-2 State replication approaches

In Table 3-2 above, we consider Middle-R, Ganymed and Distributed versioning as do not provide support for J2EE application servers. Middle-R approach assumes all operations within a transaction are submitted to the database as one request, which is different from how a J2EE application server

interacts with databases in general. Distributed versioning approach also requires transactions to pre-declare all tables that they will access, forcing necessary modification on existing J2EE application servers to support this requirement. Ganymed may be used for J2EE application servers, with the exceptions that it works with a modified database and it supports snapshot isolation consistency criteria, instead of a standard 1-copy serializability.

Aspects	Transactional queue	Transactional client	e-transaction	Interaction contract	Stateful EJB replication	Stateful EJB replication (ours)
Transactional client requirement	Yes	Yes	No	No	No	No
Client persistency requirement	Yes	Yes	No	No	No	No
Stateful server	No	Yes	No	Yes	Yes	Yes
Multi databases access	Yes	Yes	Yes	No	No	Yes
Conformance to current standard	Yes	Yes	No	No	Yes	Yes
Flexibility for application developers	No	Yes	No	Yes	Yes	Yes

Table 3-3 computation replication approaches

Transactional queue (Table 3-3) decouples the client transactional requirements from the server's, by separating request submission and reply processing transactions from the server processing transactions. Although this approach reduces the contention for resources on the server (as oppose to putting all of them, the client request submission, server processing and client reply processing, into one transaction), it still incurs additional overhead by introducing two more transactions for each client requests.

Transactional client approach, especially the one that use CORBA standard, have full features that allows them to have many possibilities to handle client requests, such as 1-N and N-1 transactions and also multi databases accesses. The downside of these flexibilities is that it imposes heavy requirements on the client side, such as it dictates the client to support the transaction service and the persistent service.

We consider the interaction contract approach does not support multi databases approach, as the contract treats the interaction between persistent components and transactional components on bilateral basis. Therefore, there is no guarantee that governs interaction between one persistent component and two transactional components, for example, like the one that normally occurs in distributed transaction setting.

3.7 Summary

This chapter summarizes recent progresses in replication that are relevant for implementing replicated application servers to provide highly available components. Compare to the above research, this thesis deals with state replication implementation in middleware layer (chapter 4) that does not require any modification at the backend database and with minimum modification to existing application server implementation. Our target is to answer whether it is possible to do state replication at the application servers, especially on J2EE application servers, instead of having to use a specific replicated database system.

On chapter 5, a replication strategy of application server that support distributed transactions and provide exactly once guarantee is described. Here, our goal is to devise mechanism for replicating application servers that can avoid resource blocking and can fail-over the transactions to other application servers transparently.

4 State replication: improving the availability in database tier

Existing application servers do not yet provide support for managing persistent state replication. They must rely on the replication support provided by the database vendor, therefore they depend on the features provided by the database vendor, and not all database products support currently replication.

In this setting, we consider of masking database server failures only. Thus, an application server may continue to make forward progress despite failure of a database server, as long as another database server is available.

This chapter discusses the problems, techniques and possibilities when we want to improve the availability of the database tier without relying on the replication feature from the database. The design presented here provides a simple and practical way of introducing data replication into component middleware, with minimal modification into the internals of the application servers. The content of this chapter has also been presented as a conference paper (Kistijantoro, Morgan et al. 2003).

4.1 Issues in state replication for component middleware

In chapter 3 we have described some issues associated with data replication in middleware layer. Among these issues is that the middleware layer does not have access to the internal locking mechanisms employed by the database. However, this is unavoidable for the architecture that we consider in this thesis, i.e. data replication in component oriented application servers such as J2EE servers. In this environment, database is an external entity that must be assumed as a black box that can not be modified.

By replicating the data store tier, what we mean is that we want to use two or more databases (not necessarily from the same vendor) to store the same data. The J2EE application server accesses data stores by employing resource adapters. We modified these resource adapters so that they connect to identical

database copies and they can manage replication consistency by using the available copies approach (Bernstein, Hadzilacos et al. 1987).

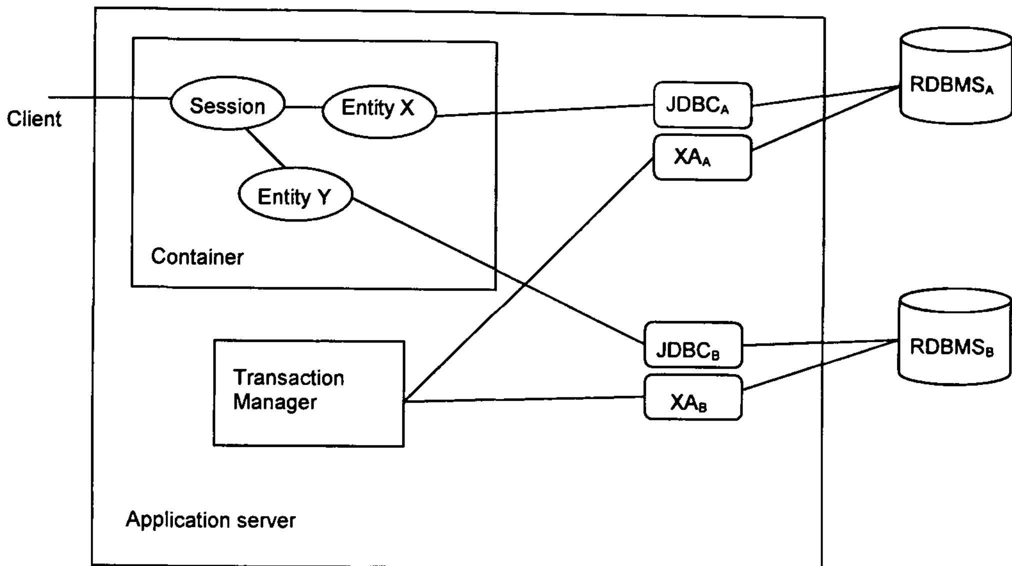


Figure 4-1 An application server without state replication

Figure 4-1 illustrates the interaction between components within an application server and external databases that store their state. The state of entity bean X is stored at RDBMS A, and the state of entity bean Y is stored at RDBMS B. Communications between an RDBMS and a container is via Java DataBase Connectivity (JDBC) driver, referred in the J2EE specification as a *resource adaptor*. To enable a resource manager to participate in transactions originated in EJBs, a further interface is required. In J2EE architecture this interface is referred to as the *XAResource interface* (shown as XA in Figure 4-1). A separation of concerns between transaction management via XAResource interface and resource manager read/write operations via JDBC is clearly defined. In simple terms, the transaction manager interoperates with the resource manager via the XAResource interface and the application interoperates with the resource manager via the JDBC driver.

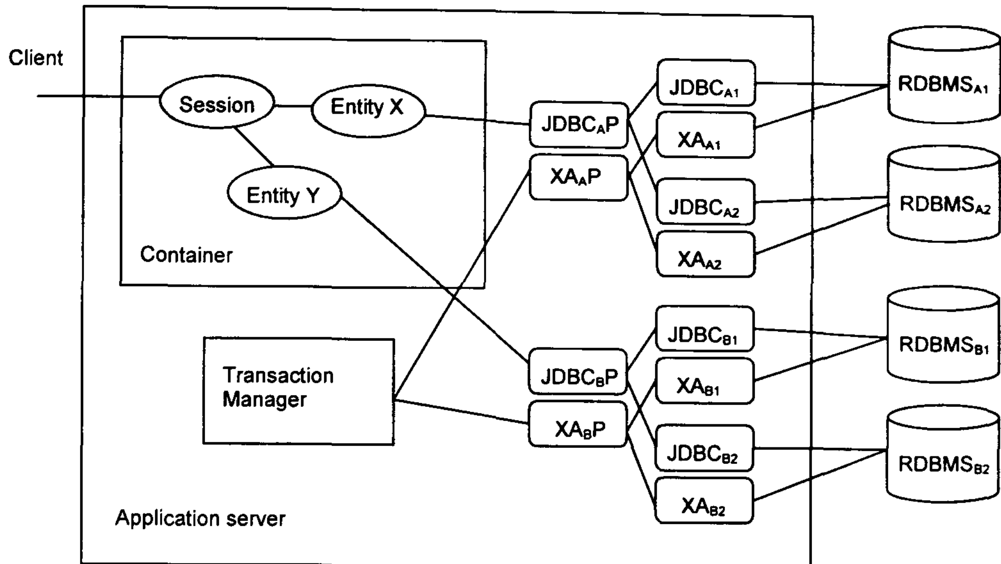


Figure 4-2 An application server with state replication

Figure 4-2 illustrate an approach to improve the availability of data store tier that leaves the container, transaction managers internal to resource managers and the transaction manager of the application server undisturbed. RDBMSs A and B (Figure 4-1) are now replicated (replicas A1, A2, and B1, B2). Proxy resource adaptors (JDBC driver and XAResource interface) have been introduced (identified by letter P appended to their labels in the diagram). In this diagram, the application consists of three beans: one session bean and two entity beans that each are stored to different data store. For this purpose, adapter proxies ($JDBC_xP$ and XA_xP) are introduced to handle replication for each data store.

We assume that the backend databases employ a standard concurrency control, i.e. two-phase locking based concurrency control. To correctly implement the replication, the adapter proxies must handle the following issues: interaction with the transaction manager, handling the data store failure, handling updates in the clustering configuration, and handling the application server failure in the clustering configuration. Each of these issues will be discussed in the following sub chapters.

4.2 Single server implementation

Available copies approach is the most common way to handle data replication (Bernstein, Hadzilacos et al. 1987). The replication manager maintains a list of all replicas/copies. On reading the data, one can read from any copy, and on writing the data, it must write the data to all available copies. When a copy is unavailable, it is removed from the list of available copies and it will not be accessed again, until the recovery process is performed on that copy.

Suppose during the execution of a transaction, say T1, one of the resource manager replicas say $RDBMS_{A1}$ fails. A failure would result in $JDBC_{A1}$ and/or XA_{A1} throwing an exception that is caught by $JDBC_{A}P$ and/or $XA_{A}P$. In a non replicated scheme, an exception would lead to the transaction manager issuing a rollback for T1. However, assuming that $RDBMS_{A2}$ is correctly functioning, such exceptions will not be propagated to the transaction manager, allowing T1 to continue on $RDBMS_{A2}$. Therefore $RDBMS_{A1}$ must be removed from the valid list of resource manager replicas until such a time when the states of $RDBMS_{A1}$ and $RDBMS_{A2}$ may be reconciled (possibly via administrative intervention during periods of system inactivity). Such a list of valid resource managers is maintained by $XA_{A}P$.

As the application server may have several active connections to the same database, there will be some write operations are executed concurrently. However, these operations will not be in conflict with each other, as the locking mechanism of the container controlling the bean would have prevented this from happening. Hence, the backup replicas will always receive conflicting operations in the same order.

Deadlock may still be happened as it could happen on any database with two phase lock based concurrency control. When this happens, the database proxy receives aborting message from the primary database, and then the proxy informs other replicas to abort the same transaction on those replicas.

The transaction manager controls the execution of the transactions on the data store by sending instructions to start, to end, to commit or to abort the transactions. The XA connection proxy intercepts these instructions and treats

them in the same manner as the JDBC proxy handles the write operations to the data store.

4.3 Clustering configuration

For clustering configuration, each application server has its own database proxy for each database unit used by that application server. In this configuration, concurrent access to data stores from application servers may break the serializable property of the data store. This problem is illustrated in the following scenario.

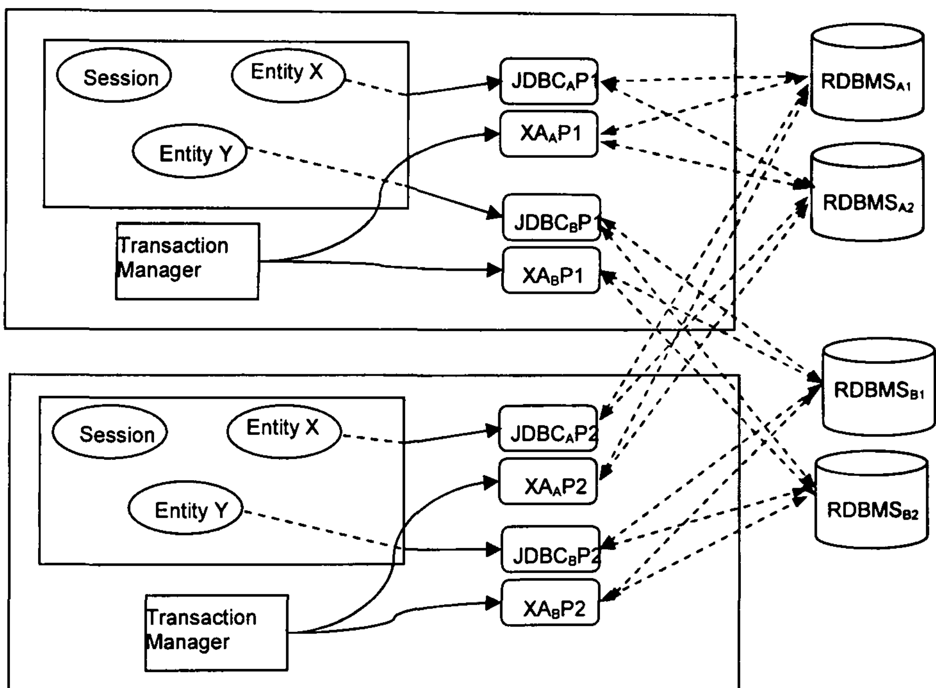


Figure 4-3 State replication in clustering configuration

In Figure 4-3, a cluster contains two application servers (AS1 and AS2) that are accessing shared resource manager replicas. To make the diagram simple, only the resource adaptor proxies are shown.

Let us assume that transaction T1 is executing on AS1 and T2 is executing on AS2 and both T1 and T2 require invocations to be issued on entity bean X. Without proper coordination, there is a possibility that AS1 manages to obtain the state of X from $RDBMS_{A1}$ while AS2 manages to obtain the state of X

from $RDBMS_{A2}$. This will break the serializable property of transactions. To overcome this problem, a single resource manager replica that is the same for all application servers should satisfy load requests for relevant entity beans (we call such a resource manager a *primary read resource manager*). This will ensure all load requests serialized, causing conflicting transactions to block until locks are released. To ensure resource managers remain mutually consistent the store request is issued to all resource manager replicas.

Within clustering configuration, resource adaptor proxies from different application servers have to agree on the primary read resource manager. This is done by introducing a replica manager that is responsible for propagating information about the primaries and available resource managers for each adaptor proxy among application servers. The replica manager multicast the resource managers' availability by making use of a group communication system that supports the abstraction of a virtually synchronous process group.

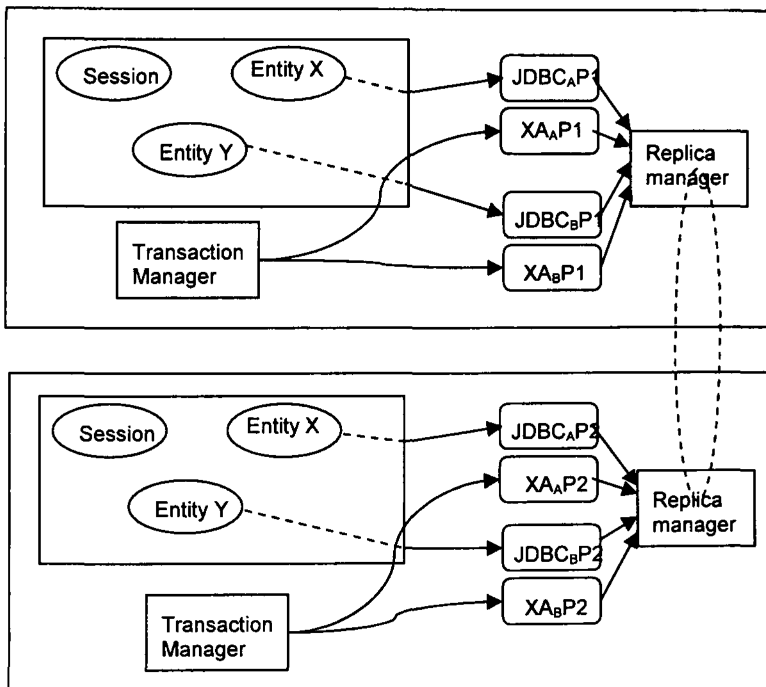


Figure 4-4 Replica manager for maintaining available resource managers information

When an adaptor proxy detects failure of a resource manager replica, the failure information is multicast to other application servers by the replica manager. All resource adaptors on all application servers will then remove the

failed resource manager replica from the list of available resource managers. The list of available resource managers is an ordered list with the primary read resource manager at the top. Therefore, when the primary resource manager of a resource adaptor fails, all other resource adaptors will choose the next available resource manager as the primary, which will be the same to all resource adaptors.

The identification of the primary read resource manager needs to be available to an application server after a restart. The new restarted application server may retrieve the list of available resource adapter proxies and the primary from existing application servers, if there are any. Otherwise, for the first application server to start, it should get the information from somewhere else, i.e. from a persistent store. We assume the set of resource managers are fixed, so that this information can be inserted as part of the application server's configuration.

4.4 Performance evaluation

Experiments were carried out to determine the performance of our system over a single LAN. JGroups (Ban 1998) was used as the group communication sub-system in our clustered experiments.

4.4.1 Implementation and Setup

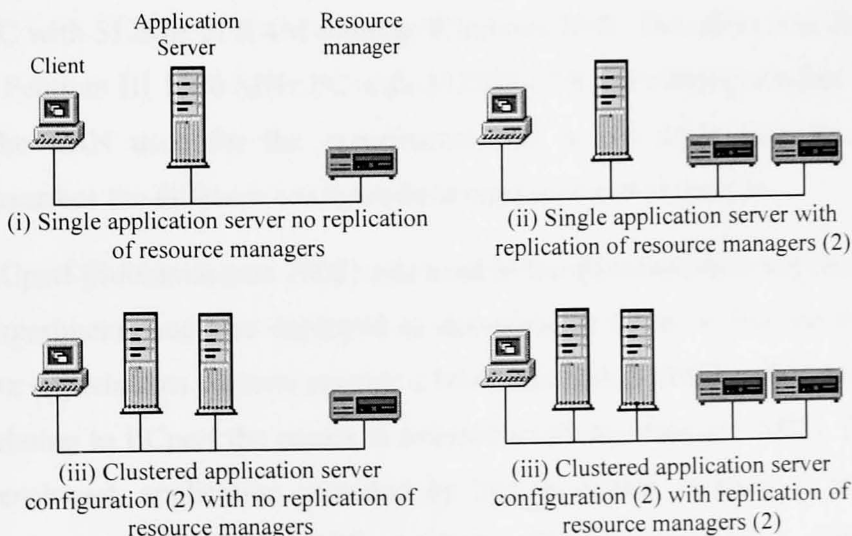


Figure 4-5 Configuration

Four experiments were carried out (configuration described in Figure 4-5) to determine the performance of the clustered (using JBoss clustering) and non-clustered approaches with and without state replication:

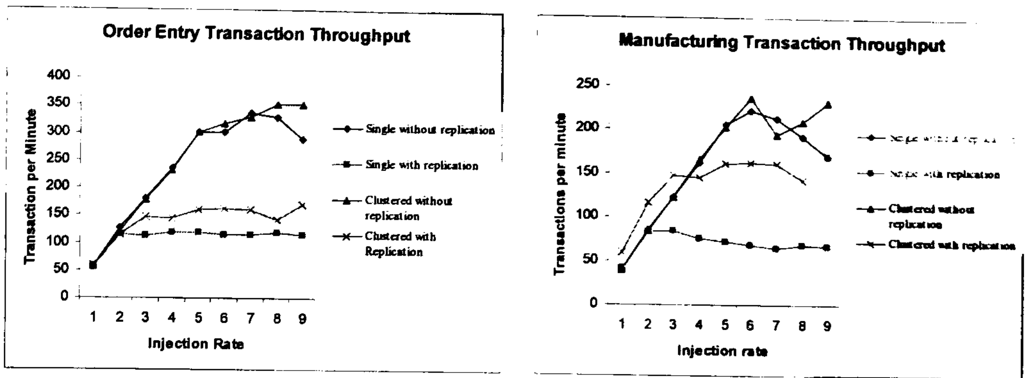
1. **Single application server with no replication** - To enable comparative analysis of the performance figures, an initial experiment was carried out to determine the time required to satisfy a client request issued to the application server using a single resource manager without state replication (Figure 4-5.i).
2. **Single application server with state replication** – Experiment 1 was repeated, with replica resource managers accessed by our resource adaptor proxy (Figure 4-5.ii).
3. **Clustered application server with no replication** – Two application servers constituted the application server cluster with a single resource manager providing persistent storage (Figure 4-5.iii).
4. **Clustered application server with state replication** – We repeated experiment 1 with replica resource managers accessed by resource adaptor proxies from each of the application servers (Figure 4-5.iv).

The application server used was JBoss 3.2.0 with each application server deployed on a Pentium III 1000 MHz PC with 512MB of RAM running Redhat Linux 7.2. The resource manager used was Oracle 9i release 2 (9.2.0.1.0) with each resource manager deployed on a Pentium III 600 MHz PC with 512MB of RAM running Windows 2000. The client was deployed on a Pentium III 1000 MHz PC with 512MB of RAM running Redhat Linux 7.2. The LAN used for the experiments was a 100 Mbit Ethernet. Figure 4 describes the different configurations used in our experiments.

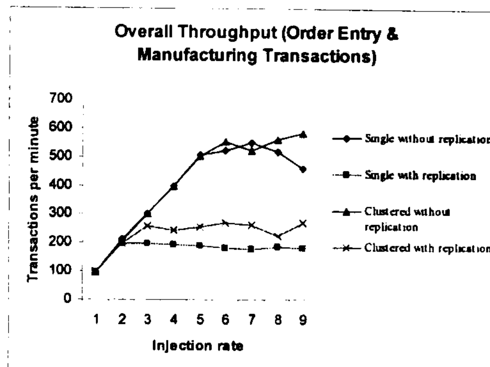
ECperf (Subramanyam 2002) was used as the demonstration application in our experiments and was deployed as described in figure 4. For the purposes of our experiments we now provide a brief description of ECperf. For more detail relating to ECperf the reader is referred to (Subramanyam 2002). ECperf is a benchmark application provided by Sun to enable vendors to measure the performance of their J2EE products. ECperf presents a demonstration application that provides a realistic approximation to what may be expected in a real-world scenario via a demonstration system that represents

manufacturing, supply chain and customer order management. The system is implemented using EJB components and deployed on a single application server (commonly described as the *system under test* (SUT)). In simple terms, an order entry application manages customer orders (e.g., accepting and changing orders) and a manufacturing application models the manufacturing of products associated to customer orders. The manufacturing application may issue requests for stock items to a supplier. The supplier is implemented as an emulator (deployed in a java enabled web server). In our configuration the supplier emulator is deployed on the same machine as the application server (when using the clustered approach only one of the application servers needs to run the supplier emulator). The client machine runs the ECperf driver. The driver may represent a number of clients and assumes responsibility for issuing appropriate requests to generate transactions within the order entry and manufacturing applications.

The ECperf driver was configured to run each experiment with 9 different injection rates (1 though 9 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.



(i) Order entry application throughput (ii) Manufacturing app. throughput



(iii) overall throughput

Figure 4-6 Result

The performance benefit associated with clustering is shown by our experiments as higher injection rates result in a lowering of transaction throughput for single application server scenarios (indicating application server overload). The introduction of clustering prevents such a slowdown. The slowdown experienced by the single server is most visible in the manufacturing application (where injection rates of over 5 reduce transaction throughput significantly). However, when state replication is introduced the single server does not experience such a slowdown in performance, indicating that saturation of the system has not yet been reached. In fact, the transaction throughput of the clustered approach with replication is similar to that of a single application server without state replication in the manufacturing application when the injection rate is 9.

The experiments show that clustering of application servers benefits systems that incorporate our state replication scheme. Clustering of application servers using state replication outperform single application servers that use state replication by approximately 25%. This is the most important observation, as state replication does not negate the benefits of scalability associated to the clustered approach to system deployment.

5 Computation replication: improving the availability of the application servers

This chapter discusses the replication of application servers to improve the availability of the system. As it has been explained on chapter 3, our focus is how to provide exactly once execution for invocation of components hosted on the application servers/middle tier. The content of this chapter is an expanded version of (Kistijantoro, Morgan et al. 2006)

5.1 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.

We assume servers with crash failures. Each EJB server hosts both EJB containers and a transaction manager within one process, which serve as a unit of failure. This means that whenever an EJB container fails, the associated transaction manager will also fail. Messages are reliable, and each message will eventually arrive at its destination, as long as both the sender and the receiver remain operational. Processes fail by stop executing (crash failures). Once it fails all sub component in that process, i.e. EJB containers and the transaction manager, will not send any message or do any action before they perform the necessary recovery procedure.

A typical scenario of interactions between a client and a server is that the client first enacts a session on an application server, then it sends a series of non transactional requests, and it ends the interaction by sending a transactional request to set the modification persistently (Figure 2-1).

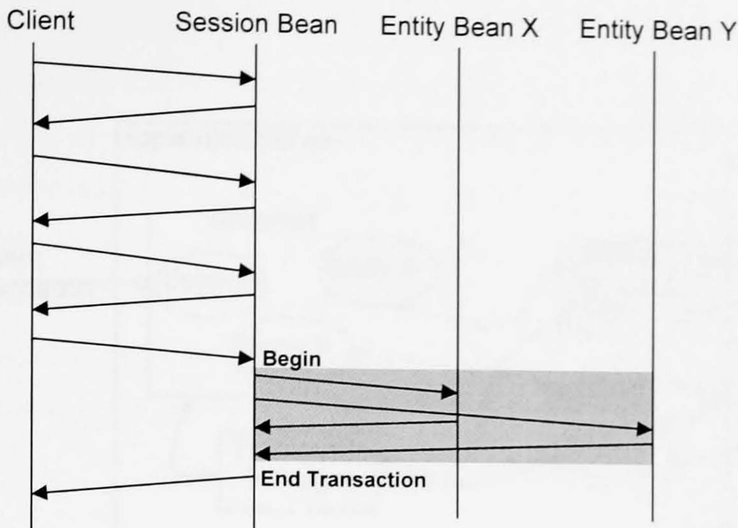


Figure 5-1 Client interaction

Transactions are managed by containers, i.e., all entity beans are assumed to have *container managed transactions*, which are specified per bean method invocation basis. Therefore, the unit of transaction always corresponds to a single invocation that encompasses that transaction, and this invocation may invoke other invocations which will be executed under the same transaction.

When the primary application server fails, the backup takes over the execution of unfinished requests from the primary. Each application server has its own transaction manager, which runs in the same address space as the application server. Therefore, the associated transaction manager will also fail when the application server fails. To allow minimal modification to existing code, we make use of interceptor mechanism available at JBoss server. In JBoss, interceptor is a piece of code that can be configured to be executed each time a client invokes a method on an EJB bean instance. One can configure several interceptors to be executed as a chain, before and after the application server invokes the method on the bean instance. The detail of the interceptor mechanism in JBoss can be found in section 2.6.1.2.

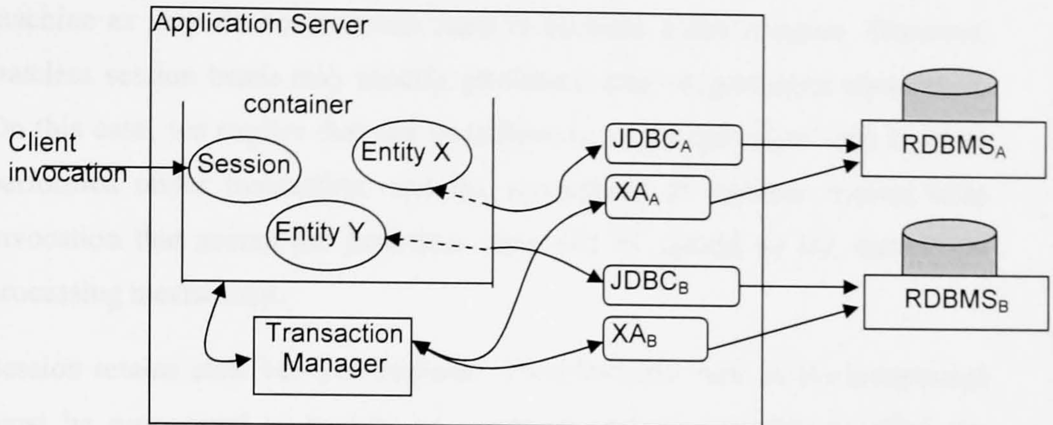


Figure 5-2 System Architecture

Figure 5-2 illustrates again the basic principles of how an application server processes a request. A client invokes a method on a session bean (which can be either a stateless session bean or a stateful session bean), and this invocation can invoke other invocations on other beans, which can be any type of beans (stateful, stateless or entity). The container intercepts all invocations on the beans and performs necessary actions required to process the invocation correctly. There are two interception points occur on JBoss, one is at the client side and another one is at the server side.

The interception point at the client side is used for implementing failover mechanism, which will be explained in section 5.2. The interception point at the server side is used for implementing state update propagation from the primary to the backup. There are two types of state update propagation: transactional and non transactional. Both types are described in section 5.6 and 5.3.

5.1.1 Different types of components: stateless, session and persistent component

There are three kinds of EJB objects: session beans, entity beans and message driven beans. Session beans can be either stateful or stateless. This chapter mainly focus on stateful session beans. Entity beans are not replicated explicitly, but they are supported by the backend database, hence they can

always be revived by the backup by retrieving their last state from the database. Stateless components can also be easily restarted on different machine as they do not maintain state in between client requests. However. stateless session beans may modify persistent state via persistent component. On this case, we require that any modification to the persistent state must be performed under transaction, and the correctness of stateless session bean invocation that access the persistent state will by upheld by the transaction processing mechanism.

Session retains state between requests. Therefore, the state of the component must be propagated to backup on every invocations in order to allow the backup to continue the session after the primary fails.

5.2 Fail-over implementation

The fail-over mechanism is implemented by utilising JBoss client-side interceptor mechanism. A new interceptor is implemented to give a unique id for each request submitted by a client. This id is needed to prevent duplication of execution at the backup when the primary fails. The invoker proxy on the client side is also modified to contain a list of available server (one primary and one or more backups) together with its version number of the group membership (Figure 5-3). The application servers are maintained as a view synchrony group. On each invocation, the server will check the version id with their own view-id. When there is a mismatch between the client group membership version number and the server's own view id, the new view of the server group is then piggybacked to the client.

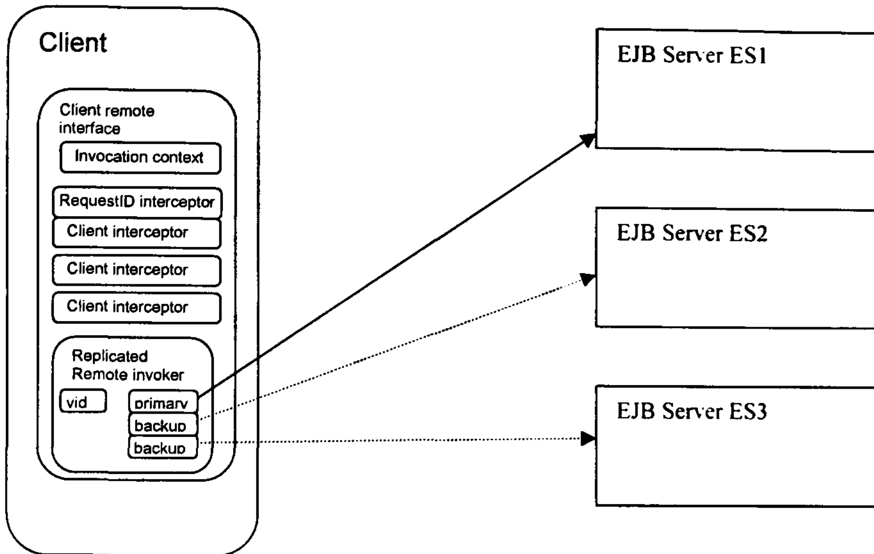


Figure 5-3 Replicated EJB implementation of a remote interface

The invoker proxy always sends invocations to the primary server. When the primary crashes, the proxy receives a time out and then resends the invocation to a backup. Meanwhile, the other group member reconfigures and installs a new view for the group. After a new view is installed, the backup contacted by the proxy informs the proxy about the new view and the new primary. If the contacted backup is the new primary, it processes the invocation directly; otherwise, the proxy must again resend the invocation to the new primary.

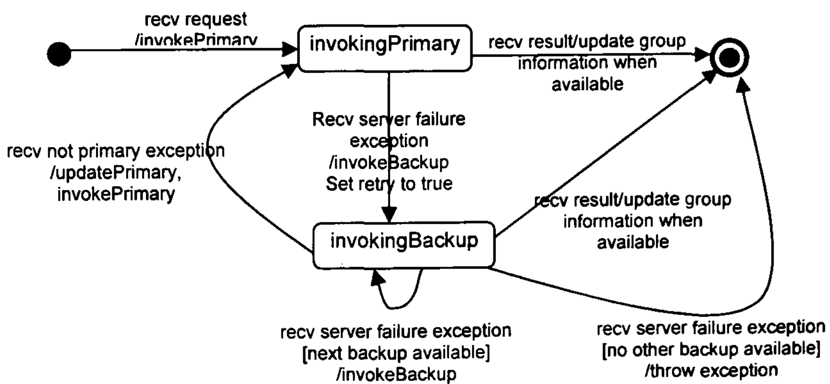


Figure 5-4 Fail-over invoker proxy state chart

Figure 5-4 displays the state chart of the invoker proxy implementation. On normal condition, it sends the invocation to the primary by invoking the

invokingPrimary method. If the primary is available and is able to respond with a result, the invoker proxy checks if there is any new piggybacked group information from the server, updates the list of available servers information, and returns the result back to the client. If the primary fails or cannot be contacted, the proxy receives a ServerFailureException[7], and invokes the invokingBackup method to send the invocation to a backup. The invokingBackup method iterates through the list of available servers, and sends the invocation to a backup server on the list one by one. If the contacted backup server has become a new primary server, it processes the invocation directly and returns the result. Otherwise, the backup server responds with a NotPrimaryException and provides the information about the new primary server. Figure 5-5 describes the state chart for a replica implementation.

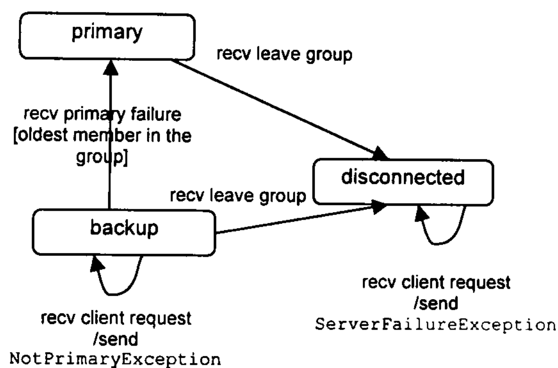


Figure 5-5 Server state

5.2.1 Client interface maintenance

The client retrieves an interface for accessing a bean from a naming server. Each application server maintains its own naming server, and a client can connect to any naming server from any application server that hosts the bean to access. When a client retrieves an interface from a naming server, the naming server serializes the interface via standard java serialization, and retrieves the current list available servers and primary server information from the application server, and includes this information as part of the serialization

[7] The ServerFailureException is raised by the system when a connection to a server cannot be made, or when an existing connection is lost.

data. Therefore, the interface retrieved from any server should point to the latest primary server information and the latest list of available servers at the time that the client retrieves the interface.

5.3 Session state replication

Session state replication is implemented on JBoss application server by utilising container interceptors, MBeans and JMX technologies available on JBoss.

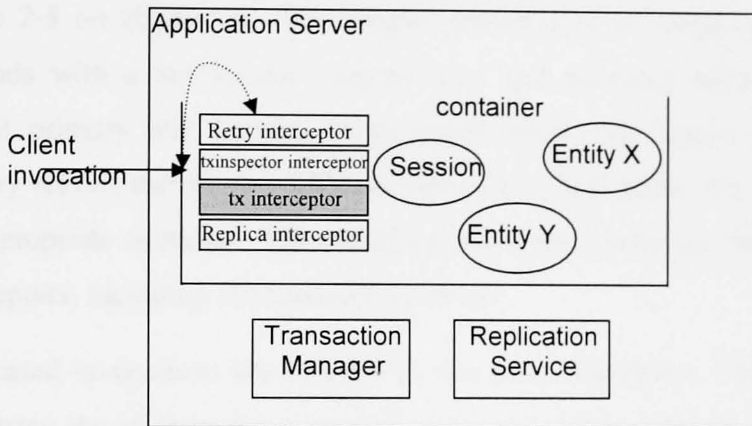


Figure 5-6 Augmenting application server with replication service

Figure 5-6 shows the interceptors and associated services that implement our replication scheme in 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
- Tx interceptor – existing interceptor that is implemented on JBoss to implement transaction handling
- Replica interceptor – ensures state changes associated with a completed invocation are propagated to backups.

The replication service distributes and processes session state updates to/from other application servers. To manage the session state, the service maintains several logs:

(i) invocation result log: each entry on this log contains an invocation id, the updated state of session beans resulted from the invocation and the result object returned to the client. When the invocation is transactional, the log also contains the transaction id together with information on all resources involved in that transaction.

(ii) Transaction log: each entry on this log contains a transaction id and a transaction outcome decision (commit or abort)

Invocations from clients are initially processed by the invoker MBeans (see Figure 2-8 on chapter 2). The invoker MBeans on a backup server always responds with a `NotPrimaryException` and provides information about current primary and current group membership (see Figure 5-5). On the primary server, the invoker MBeans determines and passes the invocation to the appropriate container. The container passes the invocation through a set of interceptors, including our interceptors above.

Duplicated invocations are handled by the retry interceptor. This is done by comparing the id from the incoming client invocation with the invocation id from the invocation result log. When the invocation id is found on the log, the invocation has been executed and the retry interceptor simply returns the result from the log to the client. Otherwise the invocation is processed through the next interceptor, and ultimately by the bean itself.

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 id. The replication service multicasts this information in one message (STATE_UPDATE message) to other backups. Upon delivery confirmation received from the group communication service, the replication service updates the invocation result log and returns the control back to the replication interceptor. Finally, the result is delivered to the client.

5.4 Handling transaction

We assume container managed demarcation. For each invocation, the container decides (based on the transaction attribute of the invoked method) whether or not to create a new transaction for the invocation. 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).

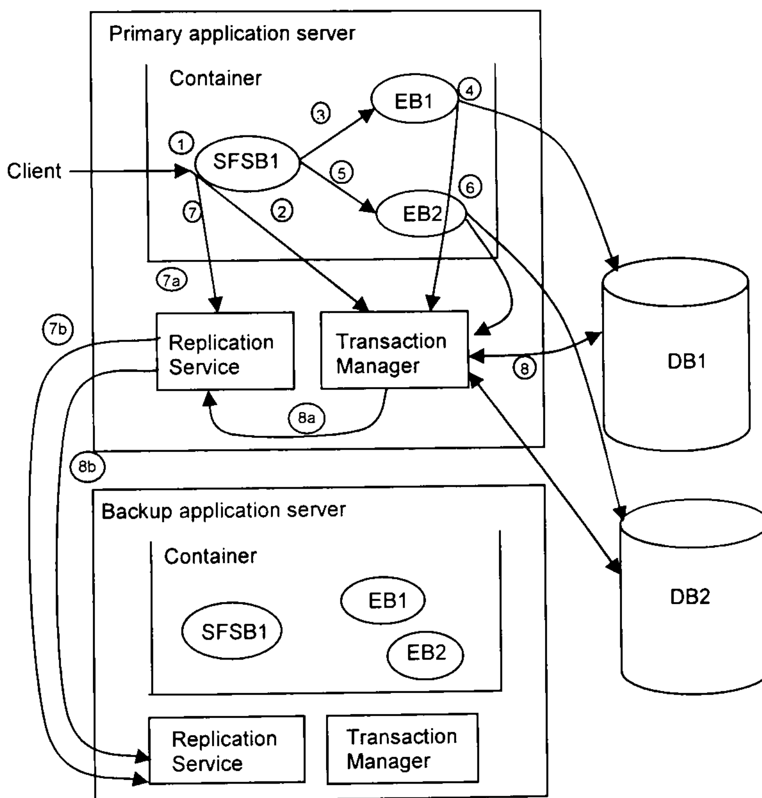


Figure 5-7 A typical interaction for a transaction processing in EJB

Figure 5-7 illustrates the execution of a typical transaction (for brevity, resource adaptors are not shown). This will be used as an example to highlight the enhancements made to handle transaction failover. 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 (i.e. the tx

interceptor on JBoss application 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 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). After the transaction commit, the result is then returned to the client.

Points (7a), (7b), (8a) and (8b) are the modifications to handle transaction failover. Right after the computation and prior to two-phase commitment process (7), the replica interceptor (7a) informs the replica manager to multicast (7b) the computation result (STATE UPDATE message) to backups. The multicast includes the transaction id, the latest state of all session beans involved in that transaction, the result object to return to the client and information on all resources involved in that transaction. This is to ensure that the backups have all necessary information to continue the computation should the primary fail. If the primary fails prior to this multicast, a backup must retry to execute the invocation from the beginning, otherwise the backup should try to perform the commitment process for the transaction.

During the two-phase commit, after the transaction manager takes a decision based on the vote result returned by the resources (8), the transaction manager informs the replica manager (8a) to multicast (8b) the transaction outcome decision (TX OUTCOME message) to backups. The multicast consists of the transaction id and the decision. 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.

5.4.1 Determining when to replicate the transaction

The replication service must be able to detect when the transaction starts and ends. It is possible to modify the transaction manager or existing JBoss interceptor to add a hook for this purpose. However, we choose to implement this mechanism as a separate set of interceptors. This task is performed by the txinspector interceptor (see Figure 5-6) that processes the invocation before the JBoss tx interceptor associates the invocation with a transaction. The txinspector interceptor compares the transaction context from the incoming invocation and the transaction attribute for the invoked method (which is available from the bean metadata). When a method with a *Required* transaction attribute matches an invocation that has no transaction context, this means that the container should create a transaction at the start of the invocation and it should end the transaction at the end of the invocation. The txinspector flags this invocation with a TRANSACTION_UNIT flag, so that later on the replica interceptor (at point 7a on Figure 5-7) knows that it has to multicast the state update together with the transaction information.

5.4.2 Handling chained session invocations

So far we have described how to handle transactional invocation that involves one session bean invocation and several entity beans invocations. Now the model is extended to include invocation on session bean that invoke another session bean. When a session bean SFSB2 is invoked by another session bean SFSB1, the state of SFSB2 is not immediately multicast to all backups after the invocation on SFSB2 finishes. The replica interceptor simply adds SFSB2 session state on the list of updated session, and later on when SFSB1 finishes, the state of SFSB1 together with the state of SFSB2 are multicast to backups. We do not handle concurrent access on SFSB explicitly, as the container has always serialized access on stateful session bean. Furthermore, the EJB specification also state that stateful session beans are not shared between clients, and clients are not allowed to make concurrent calls on the same bean (section 4.3.13 on EJB 3.0 specification (DeMichiel and Keith 2006)).

5.5 Replication service implementation

Replication service is an abstraction of group communication library for propagating messages and group membership information to all servers. The service is responsible to maintain group membership information, the replica state propagation and state transfer for new member.

5.5.1 Group membership

We use JGroups (Ban 2006), a virtual synchronous group communication library that provides the following features:

- Virtual synchrony: every non-faulty member of the group will receive the same sequence of views and they will receive the same set of messages between views. Every view is uniquely identified by view id.
- If a member is disconnected from a group, the next time it reconnects to the group, it will be treated as a new member. If the disconnected member is the primary, another member will become the new primary and when the old primary reconnects, it becomes a backup.

5.5.2 Replica state propagation

The replication service maintains logs associated with the replication (see the details on section 5.3 above), including current membership configuration. Upon requests from containers, the replica service multicasts the updates to other replicas and upon delivery it keeps the update in the appropriate log.

The replication service on backups also performs translation on each incoming session state received from the primary. The translation is necessary to convert all bean references found in the session state that originally point to other beans on the primary server to become references that point to the beans on the receiving server. For example, if SFSB1 on application server AP1 contains a reference to SFSB2 on AP1, then when the state of SFSB1 is received by application server AP2, the reference to SFSB2 must be translated so that it will refer to SFSB2 on AP2.

5.5.3 State transfer

Before a new replica (or formerly disconnected replica) can join as a backup, it must first perform a state transfer to bring its state to the most recent updates. This state consists of invocation result log and transaction log (see section 5.3). The state of a replica is defined as the request log and tx log. The state transfer mechanism is provided by the group communication library (JGroups). The group communication service is responsible for queuing messages delivered to the new member until it successfully retrieves the state.

5.6 Load balancing

The scheme described above assumes a single primary that services all clients. This scheme can be easily extended to support load balancing for processing client requests, as each session is accessed by a specific client only, and EJB specification does not allow concurrent access to a session bean. To support load balancing, each session bean has its own primary, which can be any of the available application server. The assignment of session beans to servers can be either decided by the application server or by the clients. This assignment is permanent, i.e. a session bean is attached to an application server as long as that server has not crashed.

The replication service maintains the mapping between session beans and their primaries, so that each server knows which sessions that use it as the primary. When a server fails, the replication service distributes all session beans on that server to other servers. To avoid duplicated executions of client requests, we use the following scenario. The invoker proxy on the client side (see 5.2) maintains a variable that represents current primary server for that session. For home interface invocation, the primary is determined on the first client invocation, and for remote interface invocation, it is determined when the session is created. If a client fails to connect to its primary server, the invoker proxy resends the request to any other server, together with the identity of the original primary of the session bean. The contacted server then inspects whether the original primary of that invocation is still a member of the current group view. As long as an application server is still a member of the current group view, other application servers cannot process any invocation that

belongs to that server, and they must respond to the invocation with a `NotPrimaryException`, as described in section 5.2. On the other hand, if an application server receives a failed over request from an application server that has been removed from the current group view, then it becomes the new primary for that session, and process the invocation as usual.

To see that the mechanism described work correctly, we have to ensure that for each session bean, at any time only one primary server exists. The primary server for a session is determined at the first invocation to the bean. The client will use the same primary server for this session, and as only one unique client accessing a certain session bean, this will guarantee the same primary server is used throughout the session bean lifetime, unless that client receives a failure exception from the invocation which can be caused either by a server failure or a temporary communication problem. The client changes to a different primary server only after the old primary is removed from the group view of the application servers which is happened when the application server process group decides that the old primary server has crashed.

5.7 Failure scenarios

As a result of the primary crashing a client that is waiting for a result will be timed out. The client then resends the request to a backup server. The contacted backup server then responds with the information on a new primary or it processes the request immediately if the contacted server itself is the new primary. For **non transactional** invocations, there are only two conditions that may happen: the primary fails before the STATE UPDATE message is delivered (section 5.3) or after the STATE UPDATE is delivered. If the primary fails before the message is delivered, the client request is re-executed on the new primary as a new request. The new primary has the previous state of the session bean, and other replicas have not received the result as well, so it will be safe to ignore the computation on the failed primary. If the primary fails after the message is delivered, the new primary simply returns the outcome of the computation previously done on the failed primary.

For transactional invocations, there are three cases to consider: (a) the primary fails before the STATE UPDATE message is delivered; (b) the primary fails

after the STATE UPDATE message is delivered but before the TX OUTCOME message is delivered; (c) the primary fails after the TX OUTCOME message is delivered. Case (a) is handled in similar way to the non transactional invocations; the request is re-executed on the new primary as a new request. For case (b) and (c), the new primary will try to finish the commitment process without violating the decision that may has been taken by the failed primary.

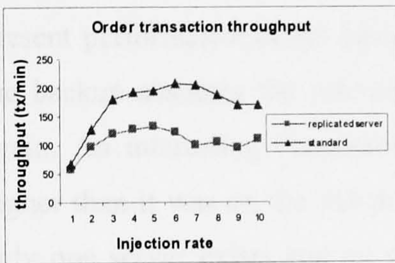
5.8 Performance evaluation

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. We also carried out experiments with up to five servers and experiments with load balancing configuration.

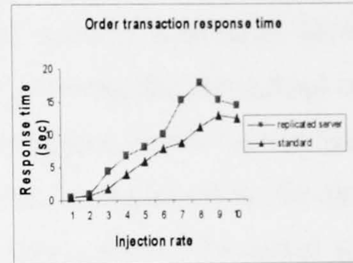
The application server used was JBoss 3.2.5 with each application server deployed on a Pentium IV 2.8 GHz PC with 2048MB of RAM running Fedora Core 4. The database used was Oracle 9i release 2 (9.2.0.1.0) (Cheevers 2002) with each database deployed on a Pentium IV 2.8 GHz PC with 2048MB of RAM running 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 (Subramanyam 2002) 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).

Four 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. The third experiment measures how our replication algorithm scales with additional backups. In this experiment we ran the ECperf benchmark with up to 5 servers' configuration. The fourth experiment measures the performance of the replication algorithm in a load balancing configuration. This experiment is similar to the third experiment, with up to 5 servers' configuration.



(i) throughput for entry order app.



(ii) response time for entry order app.

Figure 5-8 - Performance figures

Figure 5-8 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.

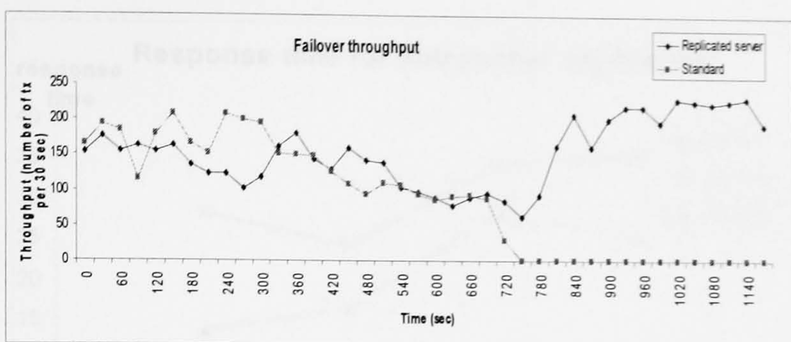


Figure 5-9 Performance figures with failures

Figure 5-9 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.

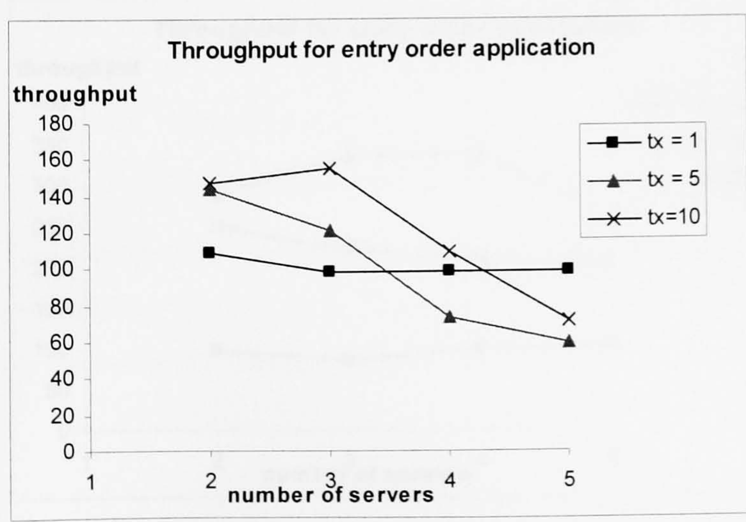


Figure 5-10 Throughput for entry order application with varying number of servers

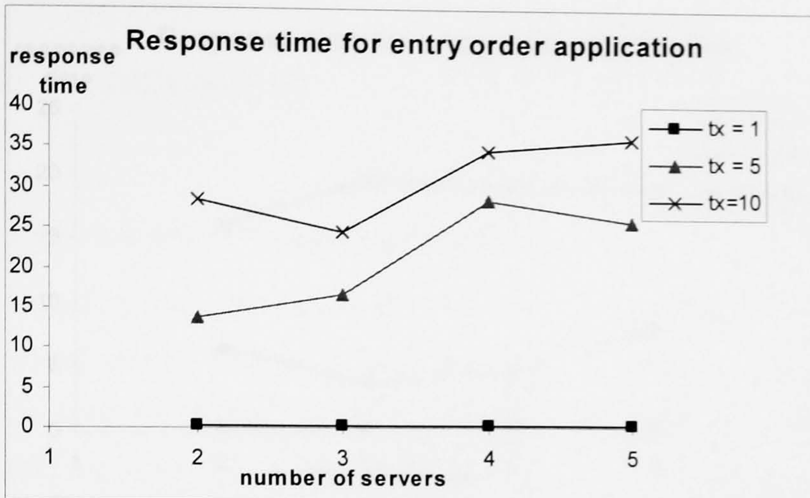


Figure 5-11 Response time for entry order application with varying number of servers

Figure 5-10 and Figure 5-11 present two graphs that describe the throughput and the response time of the ECperf applications, with varying number of servers as backups. In this configuration, only one server is the primary for all client requests, and the rests are backups. For low transaction rate (tx=1), adding backup servers does not incur much overhead, the throughput and response time differences between two server configuration and five server configuration are negligible. However, for higher transaction rate, such as tx=5 and tx=10, the differences become obvious. One explanation for this is that on higher transaction rate, the network becomes more saturated and takes longer time to multicast message to more servers.

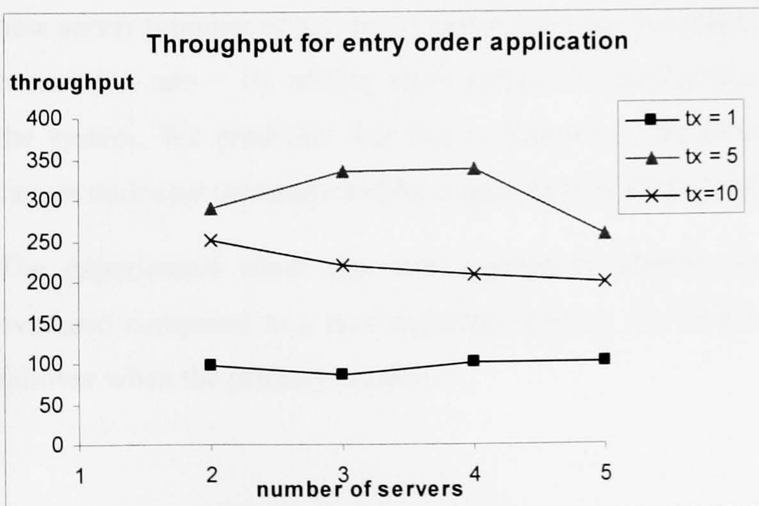


Figure 5-12 Throughput for entry order application with varying number of servers in load balancing configuration

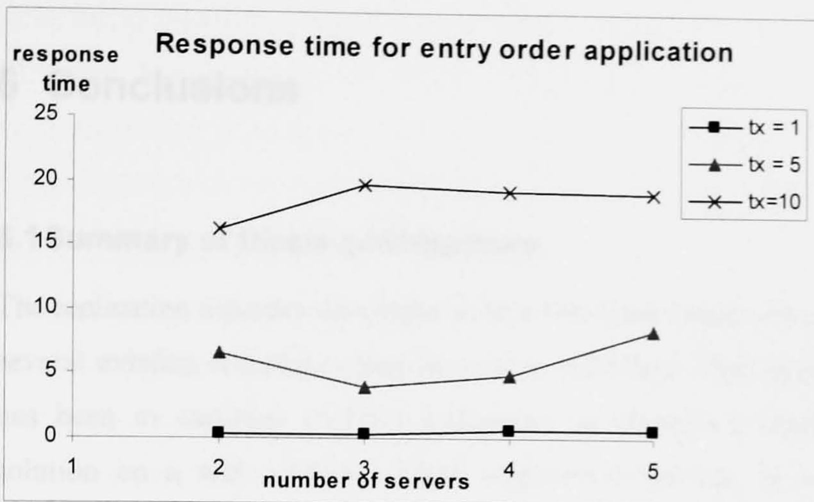


Figure 5-13 Response time for entry order application with varying number of servers in load balancing configuration

Figure 5-12 and Figure 5-13 present two graphs that describe the throughput and the response time of the ECperf applications, with varying number of servers in a load balancing configuration. In this setup, client sessions are distributed among available servers, and when an application server fails, other available servers have the same probability to become the new server for the sessions hosted on the failed server (the new primary server for each session is determined randomly). For low transaction rate (tx rate=1), adding more servers do not increase the performance. Two servers are enough to handle this rate. For the transaction rate = 5, the total throughput of the system increases when new servers were added up to four servers. However, adding a new server (number of server=5) makes the total throughput decreases. For the transaction rate = 10, adding more servers do not increase the throughput of the system. We predicted that this is because of the network saturation that causes multicast messages to take longer time to arrive to their destination.

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.

6 Conclusions

6.1 Summary of thesis contributions

The replication schemes developed in this thesis are based on a combination of several existing techniques that have been published. The focus of this thesis has been to combine existing techniques to provide a complete working solution on a real platform (J2EE application servers, in our case). The solution solved the system design problems described in chapter 1, namely, transparent failover, exactly once execution of client requests, non-blocking distributed transaction processing, ability to work with load balancing, and open, non proprietary solution. Tables Table 6-1 and Table 6-2 from chapter 3 are again presented here to provide a summary of comparison between our approaches and others.

Aspects	Middle-R	Ganymed	Distributed versioning	C-JDBC	JDBC proxy (our approach)
Consistency criteria	1 copy serializability	Snapshot isolation	1 copy serializability	1 copy serializability	1 copy serializability
J2EE support	No	Possible	No	Yes	Yes
Backend database requirements	Modification required	Database supporting snapshot isolation with modification for write set extraction	Standard database	Standard database via JDBC driver	Standard database with strict 2PL via JDBC driver
Multi-databases support	No	No	No	Yes	yes
Clustering support	Yes	Yes	Yes	No	yes

Table 6-1 State replication approaches

Aspects	Transactional queue	Transactional client	e-transaction	Interaction contract	Stateful EJB replication	Stateful EJB replication (ours)
Transactional client requirement	Yes	Yes	No	No	No	No
Client persistency requirement	Yes	Yes	No	No	No	No
Stateful server	No	Yes	No	Yes	Yes	Yes
Multi databases access	Yes	Yes	Yes	No	No	Yes
Conformance to current standard	Yes	Yes	No	No	Yes	Yes
Flexibility for application developers	No	Yes	No	Yes	Yes	Yes

Table 6-2 computation replication approaches

First, the problem of replication of the backend database tier was investigated. Our goal was to implement black-box database replication entirely using mechanisms of the middle tier. Our solution essentially required the state replication to be introduced into the application server as a resource adaptor.

While it is feasible to replicate the backend database tier solely at the middle tier level, performing recovery on failed database will require techniques to transfer the state from a functioning database to the recovering one. These were not investigated, but there are well known solutions that are available (Gray and Reuter 1993; Weikum and Vossen 2002).

Second, the thesis investigated the problem of replicating the middle tier itself, focusing on how to handle different types of states managed by the application servers. There are four different types of states:

(1) transactional state cached by the application servers.

(2) session state which is client specific.

(3) the state that is related to the transaction processing, such as the transaction id and involved resources.

(4) volatile state which exists only within a single request.

The strategy for the middle tier replication is to discard the volatile state and the cache of transactional state on the failed server, and to replicate only the session state and the transaction related state to backups by employing primary backup mechanism.

The thesis described precisely when and how to propagate the session state associated with clients and the state associated with the transaction processing, and how the backup can use that information to continue the processing without aborting the transaction. Session state is propagated using a standard primary backup algorithm, i.e. the state update is first multicast to backups after the primary computes the result, then the result is delivered to client. Transaction related state is divided into two parts: (i) the transaction id and information on involved resources that are multicast together with the other session state; (ii) the transaction outcome information that is multicast after the

transaction manager on the primary takes the decision in the first phase of the two phase commit protocol.

By multicasting the transaction related state to backups, our replication algorithm allows the backup to continue the transaction that was previously executed on the failed primary. This technique is similar to the idea described in several papers (Hammer and Shipman 1980; Reddy and Kitsuregawa 1998; Jiménez-Peris, Patiño-Martínez et al. 2001; Zhao, Moser et al. 2002) for implementing a non blocking commitment protocol by replicating the transaction coordinator. However, it has some differences as they target different platforms and employ different assumptions. For example, in (Jiménez-Peris, Patiño-Martínez et al. 2001) algorithm, it is the transaction participants that multicast their votes to a group of transaction coordinators in the first phase of the two phase commit protocol, not the primary as in our algorithm. Similar approach is also used in (Zhao, Moser et al. 2002). We assume the backend databases do not know the identity of the transaction managers/coordinators; hence they cannot multicast their vote directly to the coordinators.

Our techniques for replicating the database tier and that for replicating the middle tier are not dependent on each other, and can be used together without any difficulties. From the middle tier replication perspective, a replicated database resource appears as a single entity, and all operations issue by the transaction manager or containers from either the primary or backups are translated by the proxy wrappers to database replicas.

6.2 Further Work

The replication approaches described in this thesis assume *container managed transaction* only, where transactions are always initiated and completed within a scope of a single client request. When *bean managed transaction* is supported, it is possible to have a scenario where a client invokes a method which initiates a transaction, and afterwards the client invokes a series of invocations, and finally the client invokes another method that commits the transaction. In this situation, the fact that a client may see the state resulting from a partially executed transaction makes it difficult to provide consistent

state when an application server fails in the middle of this interaction. The problem lies in that the transaction processing management always automatically aborts the transaction, and we cannot mask this abortion from the client as executing the request as another transaction on another server may produce result that is different from the one which has already seen by the client. To avoid this, one could investigate mechanism so that the transaction processing management does not abort the transaction automatically, by developing a *fault tolerant resource adaptor*, which can survive the transaction even if the connection to the database is lost in the middle of a transaction. Another alternative is by investigating different transactional semantic, such as transactions with *savepoints (Gray and Reuter 1993),* so that the backup can continue the transaction from the last savepoint (which is made on each bean invocation), and use the state of the bean from the previous invocation.

One possible further work is to incorporating the replication approaches on other open source J2EE application servers, such as Apache Geronimo and Jonas. Although we believe that the approaches can be implemented on any other application servers, it would be interesting to see how it could be adapted to others.

References

Agrawal, D., G. Alonso, et al. (1997). Exploiting atomic broadcast in replicated databases. Euro-Par '97 Parallel Processing, Third International Euro-Par Conference, Passau, Germany, Springer.

Alvisi, L. and K. Marzullo (1998). "Message logging: Pessimistic, optimistic, causal, and optimal." Ieee Transactions on Software Engineering **24**(2): 149-159.

Amza, C., A. L. Cox, et al. (2003). Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites. Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings.

Babaoglu, O., A. Bartoli, et al. (2004). A framework for prototyping J2EE replication algorithms. Distributed Objects and Application.

Ban, B. (1998). JavaGroups - group communication patterns in Java Department of Computer Science, Cornell University,

Ban, B. (2006). JGroups User Manual: Reliable Multicasting with the JGroups Toolkit accessed on July 9, 2006 http://www.jgroups.org/javagroupsnew/docs/manual/html/index.html

Barga, R., D. Lomet, et al. (2004). "Recovery guarantees for Internet applications." ACM Trans. Inter. Tech. **4**(3): 289-328.

Barga, R., D. Lomet, et al. (2002). Recovery guarantees for general multi-tier applications. 18th International Conference on Data Engineering, 2002.

Berenson, H., P. A. Bernstein, et al. (1995). A Critique of ANSI SQL Isolation Levels. Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995, San Jose, California.

Bernstein, P., D. Shipman, et al. (1980). "Concurrency control in a system for distributed databases (SDD-1)." ACM Trans. Database Syst. **5**(1): 18-51.

Bernstein, P. A. and N. Goodman (1985). "Serializability theory for replicated databases." Journal of Computer and System Sciences **31**(3): 355-374.

Bernstein, P. A., V. Hadzilacos, et al. (1987). Concurrency control and recovery in database systems, Addison-Wesley.

Bernstein, P. A., M. Hsu, et al. (1990). Implementing recoverable requests using queues. Proceedings of the 1990 ACM SIGMOD international conference on Management of data, Atlantic City, New Jersey, United States, ACM Press.

Birman, K., A. Schiper, et al. (1991). "Lightweight causal and atomic group multicast." Acm Transactions on Computer Systems **9**(3): 272-314.

Brebner, P. and S. Ran (2001). Entity Bean A, B, C's: Enterprise Java Beans Commit Options and Caching.

Breitbart, Y., D. Georgakopoulos, et al. (1991). "On rigorous Transaction Scheduling." IEEE Transactions on Software Engineering **17**(9): 954-960.

Brisco, T. (1995). DNS support for load balancing.RFC 1794

Budhiraja, N., K. Marzullo, et al. (1993). The Primary Backup Approach. Distributed Systems. S. Mullender, Addison-Wesley.
Cardellini, V., E. Casalicchio, et al. (2002). "The state of the art in locally distributed Web-server systems." Acm Computing Surveys 34(2): 263-311.
Cardellini, V., M. Colajanni, et al. (1999). "Dynamic load balancing on web-server systems." Internet Computing 3(3): 28-39.
Cecchet, E., J. Marguerite, et al. (2002). Performance and scalability of EJB applications. OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM.
Cecchet, E., J. Marguerite, et al. (2004). C-JDBC: Flexible database clustering middleware. USENIX Conference.
Chandra, T. D. and S. Toueg (1996). "Unreliable failure detectors for reliable distributed systems." Journal of the Acm 43(2): 225-267.
Chang, J.-M. and N. F. Maxemchuk (1984). "Reliable broadcast protocols." ACM Transactions on Computer Systems (TOCS) 2(3): 251--273.
Cheevers, S. (2002). Oracle 9i Database Summary. An Oracle White Paper http://otn.oracle.com/products/oracle9i/pdf/Oracle9i_Database_summary.pdf
Cheung, S. and V. Matena (2002). Java Transactions API Specifications, version 1.0.1B, Sun Microsystems, Inc.
DeMichiel, L. and M. Keith (2006). JSR 220: Enterprise JavaBeans version 3.0. EJB Core Contracts and Requirements, Sun Microsystems.
DeMichiel, L. G., L. Ü. Yalçinalp, et al. (2000). Enterprise Java Beans Specification 2.0 Proposed Final Draft, Sun Microsystems Inc.
Dolev, D., S. Kramer, et al. (1993). Early delivery totally ordered multicast in asynchronous environments. The Twenty-Third Annual International Symposium on Fault-Tolerant Computing (FTCS).
Egevang, K. and P. Francis (1994). The IP Network Address Translator (NAT). RFC 1631.
Ezhilchelvan, P. D., R. A. Macedo, et al. (1995). Newtop: a fault-tolerant group communication protocol. Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS).
Felber, P. and P. Narasimhan (2002). Reconciling replication and transactions for the end-to-end reliability of CORBA applications. On the Move to Meaningful Internet Systems 2002. CoopIS, DOA, and ODBASE. Confederated International Conferences CoopIS, DOA, and ODBASE 2002 Proceedings (Lecture Notes in Computer Science Vol.2519). Springer-Verlag. 2002.
Fischer, M. J., N. A. Lynch, et al. (1985). "Impossibility of distributed consensus with one faulty process." Journal of the Acm 32(2): 374-382.
Frolund, S. and R. Guerraoui (2000a). Implementing e-transactions with asynchronous replication. Dsn 2000 : International Conference on Dependable Systems and Networks, Proceedings: 449-458.
Frolund, S. and R. Guerraoui (2000b). A pragmatic implementation of e-transactions. Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on.

Frolund, S. and R. Guerraoui (2001). "Implementing e-Transactions with asynchronous replication." Ieee Transactions on Parallel and Distributed Systems **12**(2): 133-146.

Frolund, S. and R. Guerraoui (2002). "e-transactions: End-to-end reliability for three-tier architectures." Ieee Transactions on Software Engineering **28**(4): 378-395.

Gorton, I. and A. Liu (2003). "Evaluating the Performance of EJB Components." IEEE Internet Computing **7**(3): 18-23.

Gray, J., P. Helland, et al. (1996). The dangers of replication and a solution. SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data.

Gray, J. and A. Reuter (1993). Transaction Processing: Concepts and Techniques, Morgan Kaufmann.

Hammer, M. and D. Shipman (1980). "Reliability mechanisms for SDD-1:A system for distributed databases." Acm Transactions on Database Systems **5**(4): 431--466.

Holliday, J., D. Agrawal, et al. (1999). The performance of database replication with group multicast. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing.

Jiménez-Peris, R., M. Patiño-Martínez, et al. (2001). A Low-Latency Non-blocking Commit Service. 15th International Conference on Distributed Computing, DISC.

Jimenez-Peris, R., M. Patino-Martinez, et al. (2002). Improving the scalability of fault-tolerant database clusters. Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on.

Kang, A. (2001). J2EE clustering, Part 1: Clustering technology is crucial to good website design; do you know the basics? JavaWorld. February 2001 http://www.javaworld.com/jw-02-2001/jw-0223-extremescale.html

Kemme, B. (2003). Database replication based on group communication: implementation issues. Future Directions in Distributed Computing. Research and Position Papers (Lecture Notes in Computer Science Vol.2584). Springer-Verlag. 2003.

Kemme, B. and G. Alonso (2000). "A new approach to developing and implementing eager database replication protocols." Acm Transactions on Database Systems **25**(3): 333-379.

Kemme, B., F. Pedone, et al. (2003). "Using optimistic atomic broadcast in transaction processing systems." Ieee Transactions on Knowledge and Data Engineering **15**(4): 1018-1032.

Kistijantoro, A. I., G. Morgan, et al. (2006). Transaction manager failover: a case study using JBOSS application server. Reliability in Decentralized Distributed Systems (RDDS 2006), Montpellier, France.

Kistijantoro, A. I., G. Morgan, et al. (2003). Component replication in distributed systems: a case study using Enterprise Java Beans. 22nd International Symposium on Reliable Distributed Systems, Florence, Italy, IEEE Computer Society.

Korth, H., E. Levy, et al. (1990). A formal approach to recovery by compensating transactions. The VLDB.

Kozaczynski, W. and G. Booch (1998). "Component-based software engineering." Software, IEEE **15**(5): 34-36.

Lamport, L., R. Shostak, et al. (1982). "The Byzantine generals problem." Acm Transactions on Programming Languages and Systems 4(3): 382-401.
Little, M. C. and S. K. Shrivastava (1998a). Integrating the object transaction service with the web. IEEE Proc. Second Int'l Workshop Enterprise Distributed Object (EDOC).
Little, M. C. and S. K. Shrivastava (1998b). "Java Transactions for the Internet." IEE Distributed Systems Engineering 5(4): 156-167.
Melliar-Smith, P., L. Moser, et al. (1990). "Broadcast protocols for distributed systems." IEEE Transactions on Parallel and Distributed Systems 1(1): 17-25.
Naur, P. and B. Randell (1969). Proceedings, NATO Conference on Software Engineering, Garmish, Germany, October 1968. Brussels, NATO Science Committee.
Patino-Martinez, M., R. Jimenez-Peris, et al. (2005). "MIDDLE-R: Consistent database replication at the middleware level." ACM Trans. Comput. Syst. **23**(4): 375--423.
Pedone, F., R. Guerraoui, et al. (1998). Exploiting atomic broadcast in replicated databases. Euro-Par '98 Parallel Processing. **1470:** 513-520.
Peterson, L. L., N. C. Buchholz, et al. (1989). "Preserving and using context information in interprocess communication." Acm Transactions on Computer Systems 7(3): 217-246.
Plattner, C. and G. Alonso (2004). Ganymed: scalable replication for transactional web applications. Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware, Toronto, Canada, Springer-Verlag New York, Inc.
Reddy, P. K. and M. Kitsuregawa (1998). Reducing the blocking in two-phase commit protocol employing backup sites. Proceedings of Third IFCIS Conference on Cooperative Information Systems (CoopIS'98). IEEE.
Schenkel, R., G. Weikum, et al. (2000). Federated transaction management with snapshot isolation. Transactions and Database Dynamics. Berlin, Springer-Verlag Berlin. **1773:** 1-25.
Schneider, F. B. (1990). "Implementing fault-tolerant services using the state machine approach: a tutorial." Acm Computing Surveys **22**(4): 299-319.
Stanoi, I., D. Agrawal, et al. (1998). Using broadcast primitives in replicated databases. Proceedings of 18th International Conference on Distributed Computing Systems. Amsterdam, Netherlands.
Strom, R. and S. Yemini (1985). "Optimistic recovery in distributed systems." ACM Trans. Comput. Syst. 3(3): 204-226.
Subramanyam, S. (2002). JSR 4: ECperf Benchmark Specification Java Community Process. 16 March, 2006 http://jcp.org/en/jsr/detail?id=4
Sun Microsystems Inc (2002). Java Management Extensions Instrumentation and Agent Specification, v1.2, Sun Microsystems, Inc.
Szyperski, C. (2002). Component Software: Beyond Object-Oriented Programming. London, Addison-Wesley.
Tygar, J. (1998). Atomicity versus Anonymity: Distributed Transactions for Electronic Commerce. VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases, Morgan.

Vaysburd, A. (1999). Fault tolerance in three-tier applications: focusing on the database tier. Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems., Lausanne, Switzerland, IEEE.
W3C Consortium (2004). Web Service Architecture W3 Consortium. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/
Weikum, G. and G. Vossen (2002). Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery, Academic Press.
Wiesmann, M., F. Pedone, et al. (2000a). "Database replication techniques: a three parameter classification." Proceedings of the IEEE Symposium on Reliable Distributed Systems: 206-215.
Wiesmann, M., F. Pedone, et al. (2000b). Understanding replication in databases and distributed systems. Proceedings - International Conference on Distributed Computing Systems, IEEE: 464-474.
Wu, H. and B. Kemme (2005). Fault-tolerance for Stateful Application Servers in the Presence of Advanced Transactions Patterns. SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), IEEE.
Wu, H., B. Kemme, et al. (2004). Eager Replication for Stateful J2EE Servers. Distributed Objects and Application.
X/Open (1992). Distributed Transaction Processing: The XA Specification, X/Open Company Ltd. The Open Group.
Zhao, W., L. E. Moser, et al. (2002). Unification of replication and transaction processing in three-tier architectures. Proceedings 22nd International Conference on Distributed Computing Systems.