Enhancing an Application Server to Support Available Components

Achmad I. Kistijantoro, Graham Morgan, Santosh K. Shrivastava, and Mark C. Little

Abstract—Three-tier middleware architecture is commonly used for hosting enterprise 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. One of the benefits of this architecture is that it allows flexible management of a cluster of computers for performance and scalability; further, availability measures, such as replication, can be introduced in each tier in an application specific manner. However, incorporation of availability measures in a multi-tier system poses challenging system design problems of integrating open, non proprietary solutions to transparent failover, exactly once execution of client requests, non-blocking transaction processing and an ability to work with clusters. This paper describes how replication for availability can be incorporated within the middle and back-end tiers meeting all these challenges. The paper develops an approach that requires enhancements to the middle tier only for supporting replication of both the middleware backend 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.

Index Terms— Application servers, availability, Enterprise Java Beans, fault tolerance, middleware, replication, transactions.

1 INTRODUCTION

Modern client-server distributed computing systems may be seen as implementations of N-tier architectures. 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 pass on the requests to specific applications residing in the third tier (middle tier) consisting of *application servers* where the computations implementing the business logic are performed; the fourth tier (database/backend tier) contains databases that maintain persistent data for applications (see fig. 1).

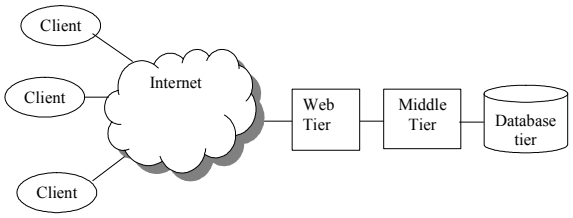


Fig. 1: N-tier architecture

Applications in this architecture are typically structured as a set of interrelated components hosted by *containers* within an application server. Various services required by 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 architecture also allows flexible configuration using clustering for improved performance and scalability. Availability measures, such as replication, can be introduced in each tier in an application specific manner. In a typical n-tier system, such as illustrated in fig. 1, the interactions between clients and the web tier are performed across the Internet. The infrastructures supporting these interactions are generally beyond the direct control of an application service provider. The middle and the database tiers are the most important, as it is on these tiers that the computations are performed and persistency provided. These two tiers are considered in this paper.

We investigate how software implemented fault tolerance techniques can be applied to support replication for availability within the middle and back-end tiers. We take the specific case of Enterprise Java Bean (EJB) components of Java 2, Enterprise Edition (J2EE) middleware and consider *strict consistency* (that requires that the states of all available copies of replicas be kept mutually consistent). We take EJBs as they are used extensively in industry with open source implementations available for experimentation. We believe that the ideas presented here are of interest to the general case of component middleware.

Data as well as object replication techniques have been studied extensively in the literature, so our task is not to invent new replication techniques for components, but to investigate how existing techniques can be migrated to components. Component-oriented middleware infrastructure provides clear separation between components that

- *A. I. Kistijantoro is a lecturer at the School of Electrical Engineering and Informatics, Bandung Institute of Technology, Indonesia.*
- *G. Morgan and S. K. Shrivastava are with the School of Computing Science, Newcastle University, UK.*
- *Mark Little is Senior Software Standards Engineer with RedHat.*

have persistent state and those that do not. 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. We examine how an application server can be enhanced to support replication for availability so that components that are transparently using persistence and transactions can also be made highly available, enabling a transaction involving EJBs to commit despite a finite number of failures involving application servers and databases.

A well engineered solution should meet a number of requirements stated below.

Exactly once execution: One important concept related to availability measures is that of *exactly once transaction* or *exactly once execution* [1]. The concept is particularly relevant in web-based e-services where the system must guarantee exactly once execution of user requests despite intervening failures. Problems arise as the clients in such systems are usually not transactional, thus they are not part of the recovery guarantee provided by the underlying transaction processing systems that support the web-based e-services. When failures occur, clients often do not know if their requests have been processed or not. Resubmitting the requests may result in duplication. Replication should hide such problems from clients, masking failures such that for each request submitted, a client receives exactly one response (safety) in a prompt manner (liveness).

Clustering and transparent failover: A common practice is to use application servers in a clustered configuration primarily for improving the performance (through load balancing) and scalability (deal with a large number of clients). Such a cluster system should provide transparent failover whilst preserving exactly once execution semantics allowing client requests to a failed server to be redirected automatically to another available server.

Non-blocking multi-database transaction processing: Multi-database transactions need to use the well-known two-phase commit protocol to maintain ACID properties (ACID: Atomicity, Consistency, Isolation, and Durability). In a non-replicated system, a crash of the application server that happens in the middle of the two phase commitment protocol can cause one or more participating backend databases to become blocked, waiting for the transaction coordinator (which may be located at the crashed application server) to be available again, so that they can find out the outcome of the transaction. A replicated system should prevent such blocking.

Modular design: Application server enhancements for replication should support (a) persistent state replication (database replication) and (b) computation replication (application server replication) in a modular fashion. By this we mean that it should be possible to deploy replication in any configuration, namely, just (a) or just (b) or both (a) and (b) with independently controlled replication levels within each tier.

Open, transparent solution: A solution should be open (non-propriety) and implementable using commodity hardware and software components. Furthermore, a solution should be transparent to the component middleware. The transparency requirement imposes the following constraints: (i) no modifications to the API (Application Programming Interface) between a client and a component; (ii) no modifications to the API between an EJB and the container hosting it; and no modification to databases.

We thus see that the introduction of availability measures in a multi-tier system poses challenging system design problems of integrating open, non proprietary solutions to transparent failover, exactly once execution of client requests, non-blocking transaction processing and the ability to work with clusters. As we discuss later, availability mechanisms of existing application servers fall short of meeting these requirements. Existing clustered application servers only support limited failover capability that does not guarantee exactly once execution.

For this reason, there has been much recent research on increasing availability of components hosted by commonly used application servers. However, we know of no research as of yet that meets all the stated requirements. In particular, experimental work reported so far for EJB components has dealt with transactions that update a single database only (as against multi-database transactions).

In this paper we describe how a widely used application server, JBoss [2] can be enhanced to support replication for availability that satisfy all the requirements stated here. Our approach requires enhancements to the middle tier only for supporting replication of both the tiers. The paper discusses the technical issues involved and presents design, implementation and performance evaluation. The paper is a unified and extended version of two earlier papers [3, 4].

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. We note that if crash failures can be detected accurately (synchronous environment), then a minimum of f+1 replicas will be required in a given tier to tolerate f replica failures; if accurate failure detection is not possible (asynchronous environment), then a minimum of 2f+1 replicas will be required to mask up to f failure suspicions, with the majority f+1 components not suspecting each other. In a clustered environment, it is generally possible to engineer a system with accurate failure detection. This is what we assume in our experimental setup for performance evaluation, whenever two replicas are used (f = 1) for a given tier. However, our design makes use of a group communication system for replica management that works in both asynchronous and synchronous environments.

The paper is structured as follows. Section two presents related work and points out how our work differs. Section three presents background information on EJBs and application servers. Section four presents de-

sign, implementation and performance evaluation of our persistent state replication (database replication) scheme. Here, persistent state of a component is stored on multiple databases; database (but not the application server) failures can be masked, so a transaction will be able to commit provided the application server can access a copy of the state on a database. This scheme has been designed to work transparently with whatever clustering scheme the application server employs. Section five presents design, implementation and performance evaluation of our computation replication (application server replication) scheme over a cluster of machines; it can work independently or in conjunction with the database replication scheme. We describe how a backup transaction manager within the cluster can complete two-phase commit for transactions that would otherwise be blocked. Concluding remarks are presented in section six.

2. Related Work

Group communications [5] plays a pivotal role in the evolution of availability solutions we see in a number of replication management schemes. The classic text [6] discusses replicated data management techniques that go hand in hand with transactions with the Arjuna system demonstrating the use of transactions in the support of replicated transactional objects [7, 8]. With the advent of object-oriented standards for distributed computing, a number of replication based solutions were developed, specifically in the area of CORBA making extensive use of group communications to ensure consistency of replicas (e.g., [9, 10, 11, 12]).

Recent works have seen transactions and group communications applied to Web Services to aid recoverability. For example, configurable durability to allow data persistence to survive crash failure [13] and utilizing group communications to provide replication schemes for increased availability [14]. Although there are similar techniques used in both approaches (as Web Services are often implemented using n-tier environments), the focus in this paper is an engineered solution for availability in transactional n-tier systems.

In the rest of this section we will examine prior work on availability measures for transactional data (objects) in n-tier architectures, beginning with current industrial practice for EJB application servers. We also describe how our work differs from the relevant research work reported so far.

2.1 Availability in current application servers

Commercial approaches make use of multiple applications servers deployed over a cluster of machines with some specialist router hardware that acts as a load balancer. If any server were to fail for any reason, the system is expected to continue to operate with the remaining servers, with the load-balancer ensuring that the client load is redistributed to the remaining servers, each of which will henceforth process a proportionately slightly higher percentage of the total load. Load balancing at the database tier is commonly achieved in a similar manner, using proprietary solutions offered by database vendors. This allows an individual application server cluster member's requests to be distributed across databases.

Transparent failover (failures are masked from a client, who minimally might need to retransmit the current request) is an ideal, but is rarely achievable with current technology for the reasons to be outline below. However, forward progress is possible and in less time than would be the case if only a single machine was used.

Transparent failover is easy to achieve for stateless sessions: any server in the cluster can service any request and if a client makes multiple requests in succession each may well be serviced by a different server. If a failure of the server occurs while it is doing work for the client then the client will get an exceptional response and will have to retransmit the request. The situation is more complicated for a stateful session, where the same server instance must be used for requests from the client, so the server failure will lead to loss of state. The approach adopted in commercial systems to avoid loss of state is to use the stateless session approach with a twist: the stateful session component is required to serialize its state to a database at the end of each client request and for the subsequent component instance in the other server to deserialize the state before servicing the new request (obviously the servers must have access to the same database). The replication of the database is assumed to be the domain of the database itself. This way, some of the functionality available for stateless sessions can be regained. However, a failure during serialization (which could result in the state being corrupted) is not addressed. Transactions that were active on a failed server will be left in an unresolved state unless the failed server can be re-started. Resolving such transactions without restarting the server will require manual intervention, but can be satisfied by allowing another server to resolve such transactions (hot failover of transaction manager). However, the hot failover approach in existing application servers may still require manual intervention as the following example identifies.

Although industrial attempts at hot failover of transaction managers may appear varied, they all follow the same essential approach. We use IBM's WebSphere to exemplify how commercial systems attempt to achieve transaction manager failover in application servers [15]. On detection of a failure (e.g., using timeouts associated to heartbeat messages) a correct server in a cluster assumes responsibility for attempting to resolve unfinished transactions that originated from a failed server using transaction logs created prior failure. However, no further processing of ongoing client requests from the failed server is achievable within the transaction. In addition, this mechanism requires all servers to have access to the same database. As databases recommended for use with commercial solutions provide their own failover mechanisms then one may assume the problem to be solved. However, this is not the case. We continue using WebSphere as our example to highlight the unresolved nature of transaction manager failover when considered together with database failover.

IBM's WebSphere coupled with Oracle's propriety load balancing and failover exemplifies the difficulties in allowing developers to make use of transactions across multiple databases in current state-of-the art commercial products [16]. Oracle can successfully load balance database connection requests by directing application servers to specific database nodes using its Real Application Cluster (RAC) technology. This approach can result in a transaction operating over multiple RAC nodes and may result in indeterminate state during two-phase commit if one or more RAC nodes fail. The following example, taken from [16], highlights such a scenario. Assume a single WebShpere node requests two database connections during a single transaction, say T1, that are satisfied by two different RAC nodes, say RAC1 and RAC2, resulting in two transaction branches TB1 and TB2. If RAC1 fails after prepare has been issued but before the issuing of commit by the WebSphere transaction manager, the WebSphere transaction manager will receive notice that RAC1 is unavailable. The WebSphere transaction manager will then attempt to get a new RAC connection, and as failover is provided by Oracle, will gain another connection and attempt to finish TB1 by issuing commit. However, Oracle may throw an exception indicating that the transaction does not exist as RAC2 may be unaware of TB1. In this scenario T1 will be unresolved, requiring manual intervention to release any locks held. To avoid this, WebShpere encourages developers to avoid multiple databases, in reality this means not using Oracle's load balancing and failover in such circumstances and thus restricts a developer to a single RAC node. In fact this problem is common, and is frequently referred to as the *unresolved transaction problem* in commercial products (such as SAP J2EE [17] and WebLogic), which all advocate manual intervention to resolve such issues.

In commercial systems the fact remains that failure of database alone may lead to inconsistencies requiring manual intervention to alleviate unresolved transactions. Turning off failover alleviates this problem, but only provides hot failover of transaction manager if database failover is not present.

Worth mentioning is JBoss's attempt to achieve failover within the application server cluster tier [18] by advocating session replication to enable failover of a component processing on one node to another. This approach load balances across replicas, allowing each replica to handle different client sessions. The state of a session is propagated to a backup after the computation finishes. When a server crashes, all sessions hosted on the crashed server can be migrated and continued on another server, regardless of the outcome of formerly active transactions on the crashed server, which may lead to inconsistencies.

In summary, JBoss does attempt to make forward progress with a session in the presence of application server failure, but without handling unresolved transactions.

2.2 Database availability

In transactional systems, strict consistency is also characterized as *one-copy serializability* (the replicated system must appear to the client as a single copy). Many algorithms satisfy this property by employing an *eager replication* scheme, i.e. by propagating state updates to all replicas eagerly before transactions commit. In contrast, propagating state update to replicas can also be done *lazily*, i.e. after transactions commit (however, one-copy serializability is typically not guaranteed in this approach).

Traditionally, for performance reasons, lazy replication schemes have been favored by commercial products. However, a classification of database replication techniques has been presented that suggests group communication can be utilized in support of eager replication protocols [19]. Recent research results of employing group communication for database replication identify eager schemes as a viable choice [20]. A drawback of these approaches is the need to integrate a group communication sub-system into the database architecture, a difficult task with existing commercial databases.

To overcome this drawback, a middleware layer – Middle-R - has been proposed that provides a way of integrating group communication based eager replication schemes into existing commercial databases [21]. However, there are two limitations: the approach as it stands cannot be used for applications that require multi-database transactions, as two phase commitment is not supported; furthermore, it still requires some database modifications to support the protocol (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 distributed versioning approach [22] overcomes the need for database modifications by requiring the middleware layer to perform its own concurrency control, but the approach has not been investigated within multi-database settings. Furthermore, non-modification of database comes at a scalability cost: Middle-R can implement optimizations in the database layer.

Aspects	Middle-R	Distributed versioning	C-JDBC	Our approach
J2EE support	No	No	Yes	Yes
Backend database requirements	Modification required	Standard database	Standard database via JDBC driver	Standard database via JDBC driver
Multi-databases transaction support	No	No	Yes	yes
Clustering support	Yes	Yes	No	yes

TABLE 1: DATABASE REPLICATION

Clustered JDBC (C-JDBC) is middleware for database clustering [23]. It provides transparency for clients to access a set of replicated and partitioned databases via a standard JDBC (Java DataBase Connectivity) driver. The architecture consists of C-JDBC drivers that run as part of a client's process, a C-JDBC controller and backend databases. The 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 operations to

any single database and sending update, commit or abort operations to all databases. Worth mentioning is phoenix/ODBC [24], that provides a similar approach to C-JDBC but for ODBC drivers and their associated databases. However, phoenix/ODBC differs to C-JDBC in that application interactions with the database are logged (using the database) and used at a later date (once database has recovered) to instantiate the application session at the state it was prior to failure. In essence, the approach of phoenix/ODBC is to promote recoverability with minimum disruption to a client application.

Our approach requires minimal modifications to an application server and requires implementing a database proxy that can be plugged into the application server as a JDBC driver. This proxy intercepts all interaction between an application server and an external database; 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 [6]. 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 requests for relevant entity beans. This replica determines the ordering of conflicting transactions; hence preserving consistency. Our state replication approach works well in multi-database transaction settings, as it also handles all interactions between application servers and databases for committing transactions with two-phase commit.

Of all the schemes described here, C-JDBC comes closest to our work; developed independent to our work and described in [3]. However, we go a step further by implementing JDBC driver replication to allow application server clustering. Table 1 summarizes the discussion.

2.3 Mid-tier availability

The key requirement here is to ensure exactly once execution of transactional requests. The interplay between replication and exactly once execution within the context of multi-tier architectures is examined in [25], whilst [26] describes how replication and transactions can be incorporated into three-tier CORBA architectures. The approach of using a backup transaction monitor to prevent transaction blocking was implemented as early as 1980 in the SDD-1 distributed database system [27]; another implementation is reported in [28]. A replicated transaction coordinator to provide a non-blocking commit service has also been described in [29]. Our paper deals with the case of replicating transaction managers in the context of standards compliant Java application servers (J2EE servers).

There are several studies that deal with replication of application servers as a mechanism to improve availability [1, 30, and 31]. In [1], the authors precisely describe the concept of an exactly once transaction (*e-transaction*) and develop server replication mechanisms; their model assumes *stateless application servers* (no session state is maintained by servers) that can access multiple databases. Their algorithm handles the transaction commitment blocking problem by making the backup server take on the role of transaction coordinator. As their model limits the application servers to be stateless, the solution cannot be directly implemented on stateful server architectures such as J2EE.

The approach described in [31] specifically addressed the replication of J2EE application servers, where components may possess session state in addition to persistent state stored on a single database ([31] implements the framework described in [30], therefore we concentrate our discussion on the implementation details of [31] only). The approach assumes that an active transaction is always aborted by the database whenever an application server crashes. Our approach assumes the more general case of access to multiple databases; hence two phase commitment (2PC) is necessary. Application server failures that occur during the 2PC process do not always cause abortion of active transactions, since the backup transaction manager can complete the commit process.

An approach for handling primary/backup style replication with the ability of backup application servers to continue long running activities (as described in the Activity Service specification for J2EE) in addition to regular ACID transactions is presented in [32]. In this approach, a long running activity is considered an *open nested transaction* (ONT) with compensation of nested transactions allowed (instead of rollback). Using checkpoint data, backups may assume responsibility of continuing a long running activity; including committing or rolling back outstanding transactions. This, in principle, is similar to our approach when considering single transactions. However, this approach considers a replica unit as an application server and an associated database together; failure of database results in failure of associated application server, there is no database failover supported. In our approach we provide database failover independently of application server failover.

Exactly once transaction execution can also be implemented by making the client transactional, and on web-based e-services, this can be done by making the browser a resource which can be controlled by the resource manager from the server side, as shown in [33, 34]. One can also employ transactional queues to gain a similar result [35]. In this way, user requests are kept in a queue that are protected by transactions, and clients submit requests and retrieve results from the queue as separate transactions. As a result, three transactions are required for processing each client request and developers must construct their application so that no state is kept in the application servers between successive requests from clients. The approach presented in [36] guarantees exactly once execution on internet-based e-services by employing message logging. The authors describe which messages require logging, and how to do recovery on the application servers. The approach addresses stateful application servers with single database processing without replicating the application servers.

We use a primary copy replication approach. Session state check-pointing is performed at the end of each client

request invocation. Therefore, client session can be continued on a backup only by repeating the last unfinished invocation. Transaction failover is supported by including transactional information (the transaction id, the information of all resources involved in that transaction and the transaction outcome decision) in the checkpoint. Modification to the internals of the application server is unavoidable. These modifications include:

- Intercepting client invocations, before and after they are processed by the application server.
- Retrieving the state of a session bean within an application server, and installing it on another server
- Intercepting the commitment process; i.e. right after the transaction monitor takes a decision about the outcome of a transaction, prior to performing the second phase of the two phase commitment process.
- Retrieving the information about currently active transactions within an invocation.

Fortunately, the platform used for implementing the approach (JBoss) provides almost all the necessary hooks for the above requirements. Table 2 summarizes the differences between the various approaches described in this section and our approach that was first reported in [4].

Aspects	Transactional queue	Trans. client	Message logging	e-transaction	Reference [31]	Reference [32]	Our approach
App. server replication	No	No	No	Yes	Yes	Yes	Yes
Transactional client	Not required	Required	Not required	Not required	Not required	Not required	Not required
Stateful server	Supported	Supported	Supported	Not supported	Supported	Supported	Supported
Platform	TP monitors	Web	Web	Custom	J2EE	J2EE	J2EE
Multi database	Supported	Supported	Not supported	Supported	Not supported	Not Supported	Supported

TABLE 2: EXACTLY ONCE TRANSACTION SOLUTIONS

3 ENTERPRISE JAVA BEANS AND APPLICATION SERVERS

Background information on EJB component middleware is presented is this section. In the first two subsections we describe the terminology and basic concepts of transaction processing in Java 2, Enterprise Edition (J2EE) middleware that should be sufficient for our purposes[1].

3.1. Enterprise Java Beans

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

A container is responsible for hosting components and ensuring that middleware services are made available to components at runtime. Containers mediate all client/component interactions. An entity bean can either manage its state explicitly on a persistent store (*bean managed persistence*) or delegate it to the container (*container managed persistence*). All EJB types may participate in transactions. Like persistence, transactions can be bean managed or container managed.

EJBs present *home* and *remote* interfaces for use by clients. The home interface provides lifecycle services (e.g., create, destroy), and the remote interface allows clients to access the application logic supported by an EJB using method calls. Clients must first retrieve a reference to the home interface of the EJB which they wish to access. This is achieved via the *Java naming and directory interface* (JNDI). The JNDI provides a naming service that allows clients to gain access to the home interface of the type of EJB they require. Once a reference to the home interface is gained, a client may instantiate instances of an EJB (gaining access to the remote interface).

Use of container managed persistence and transactions are strongly recommended for entity beans. Below we describe how this particular combination works, as we will be assuming this combination for our replication schemes.

3.2. Transactional Enterprise Java Beans applications

The main elements required for supporting transactional EJB applications deployed in an application server are shown in figure 2. An application server usually manages a few containers, with each container hosting many (hundreds of) EJBs; only one container with three EJBs is shown in the figure. The application server is a multi-threaded application that runs in a single process (supported by a single Java Virtual Machine). Of the many middleware services provided by an application server to its containers, we explicitly show just the transaction service. 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. At least one *resource manager* (persistence store) is required to maintain persistent state of the entity beans supported by the application server; we show two in the figure. In particular, we have shown relational database management systems (RDBMS) as our resource managers (bean X stores its state on $RDMS_A$ and bean Y does the same on $RDMS_B$). We assume that resource managers support ACID transactions.

[1] Our discussion does not take into account changes introduced in the new release of Java components, EJB3, announced in 2006. EJB3 specification is available at jcp.org/en/jsr/detail?id=220

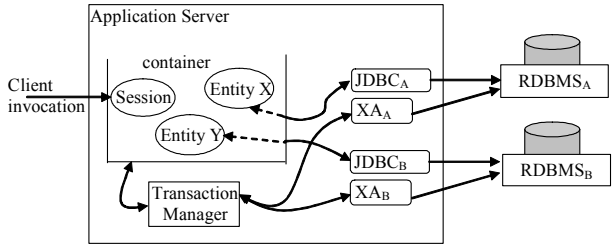


Fig. 2: EJB transactions.

Communications between an RDBMS and a container is via a JDBC driver, referred in the J2EE specification as a *resource adaptor*. A JDBC driver is primarily used for accessing relational databases via SQL statements. To enable a resource manager to participate in transactions originated in EJBs, a further interface is required. In the J2EE architecture this interface is referred to as the *XAResource interface* (shown as XA in figure 2). 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.

We now describe, with the aid of figure 2, a sample scenario of a single transaction involving three enterprise beans and two resource managers. A session bean receives a client invocation. The receiving of the client invocation results in the session bean starting a transaction, say T1, and issuing a number of invocations on two entity beans (X and Y). When entity beans are required by the session bean, first the session bean will have to 'activate' these beans via their home interfaces, which results in the container retrieving their states from the appropriate resource managers for initializing the instance variables of X and Y. The container is responsible for passing the 'transaction context' of T1 to the JDBC drivers in all its interactions, which in turn ensures 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 rollback 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 become persistent.

4 Persistent State Replication

We first describe our design for a single application server and then enhance it to work in a clustered environment. Our design can work with any load balancing strategy. Performance measures under clustering presented here make use of the standard JBoss load balancer. When we introduce mid-tier replication (section 5), we will need to modify cluster management in order for it to perform failover and load balancing whilst preserving exactly once execution, but this will not affect the workings of state replication.

4.1. Replication with a single server

By replicating state (resource managers) an application server may continue to make forward progress as long as a resource manager replica is correctly functioning and reachable by the application server. We consider how state replication may be incorporated into the scheme shown in figure 2 and use 'available copies' approach to data replication ('read from any, write to all') [6].

Figure 3 depicts an approach to resource manager replication that leaves the container, transaction manager interaction with resource managers, and the transaction manager of the application server undisturbed. RDBMSs A and B are now replicated (replicas A1, A2 and B1, B2). Proxy resource adaptors (JDBC driver and XAResource interface) have been introduced (identified by the letter P appended to their labels in the diagram; note that for clarity, not all arrowed lines indicating communication between proxy adaptors and their adaptors have been shown). The proxy resource adaptors reissue the operations arriving from the transaction manager and the container to all replica resource managers via their resource adaptors.

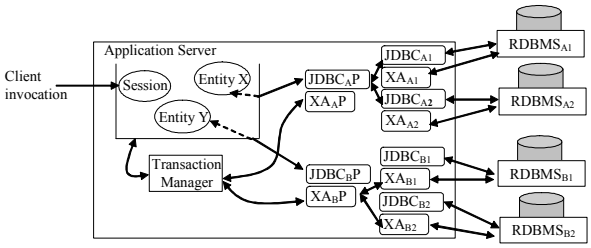


Fig. 3: An application server with state replication.

To ensure resource manager replicas remain mutually consistent, the resource adaptor proxy maintains the receive ordering of operation invocations when redirecting them to the appropriate resource adaptor replicas. This guarantees that each resource adaptor replica receives operations in the same order, thus guaranteeing consistent locking of resources across resource manager replicas.

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 $JDBC_AP$ and/or XA_AP. In an unreplicated scheme, an exception would lead to the transaction manager issuing a rollback for T1. However, assuming $RDBMS_{A2}$ is correct then such exceptions will

not be propagated to the transaction manager, allowing T1 to continue on $RDBMS_{A2}$. In such a scenario the states of the $RDBMS_{A1}$ and $RDBMS_{A2}$ may deviate if T1 commits 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 may be maintained by XA_AP (as is the case for XAResources, XA_AP is required to be persistent, with crash recovery procedures as required by the commit protocol).

4.2. Replication with clustered servers

In a clustered configuration, concurrent access to data by application servers may break the serializable property of the database. This problem is illustrated in the following scenario. In figure 4 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 (entity bean X's data is replicated across $RDBMS_{A1}$ and $RDBMS_{A2}$). Without appropriate 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 is because the JDBC/XA proxies located in each application server will operate independently of each other and just by chance the requests for X (locking X) may arrive at different resource manager replicas simultaneously, breaking the serializable property of transactions. To overcome this problem, coordination of JDBC/XA proxies across different application servers must be introduced. We achieve this in the manner described as follows.

A single resource manager replica that is the same for all application servers should satisfy requests for relevant entity beans (we call such a resource manager a *primary read resource manager*). This will ensure all requests are serialized, causing conflicting transactions to block until locks are released. To ensure resource managers remain mutually consistent the request is issued to all resource manager replicas.

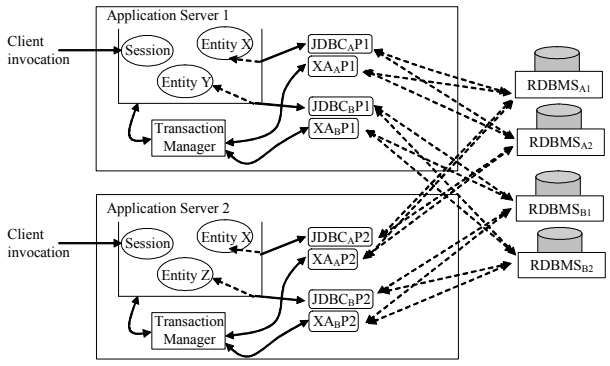


Fig. 4: Clustering and state replication.

Within a 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 primary and available resource managers for each adaptor proxy amongst application servers. A process group is used to manage copies of replica managers (see figure 5). We make use of JGroups group communication (also used by JBoss for cluster management [37]).

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 on the list as the primary, which will be the same for all resource adaptors. JGroups ensures that read attempts are ordered with respect to the installation of new views relating to identification of the primary read resource manager. If this was not the case then false suspicion of primary read resource manager may result in one or more primary read resource managers in existence at the same time.

The identification of the primary read resource manager needs to be available to an application server after a restart. The restarted application server may retrieve the list of available resource adapter proxies and the primary from existing application servers, if there are any. Otherwise, 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.

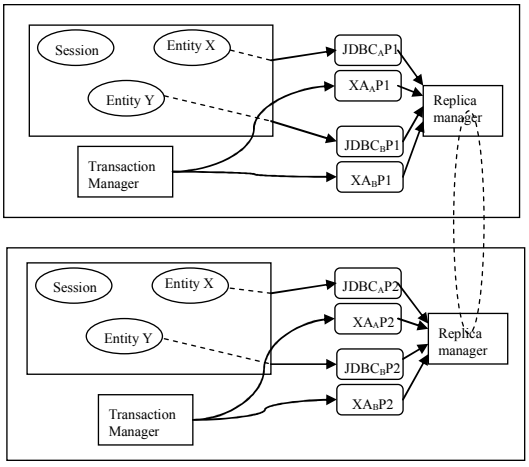


Fig. 5: Replica manager for maintaining available resource manager information.

4.3. Performance evaluation

Experiments were carried out to determine the performance of our system over a single LAN. Four experiments were carried out 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.
2) *Single application server with state replication* - Experiment 1 was repeated, with replica resource managers accessed by our resource adaptor proxy.
3) *Clustered application server with no replication* - Two application servers constituted the application server cluster with a single resource manager providing persistent storage.
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.

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) – providing snapshot isolation - 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.

ECperf [37] was used as the demonstration application in our experiments. ECperf is a benchmark application provided by Sun Microsystems 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 system that represents manufacturing, supply chain and customer order management. The system is deployed on a single application server. 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). A machine runs the ECperf driver to represent a number of clients and assumes responsibility for issuing appropriate requests to generate transactions.

ECperf 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.

Figures 6, 7 and 8 present three graphs that describe the throughput of the ECperf applications. On first inspection we can see that the introduction of replicated resource managers lowers the throughput of clustered and non-clustered configurations when injection rates rise above 2. However, there is a difference when comparing order entry and manufacturing applications. The manufacturing application does not suffer the same degree of performance slowdown as the order entry application when state replication is introduced. This observation is particularly prominent when clustered application servers are used. The reason for this is not obvious. However, such a difference may reflect the nature of the different application types (manufacturing, ordering).

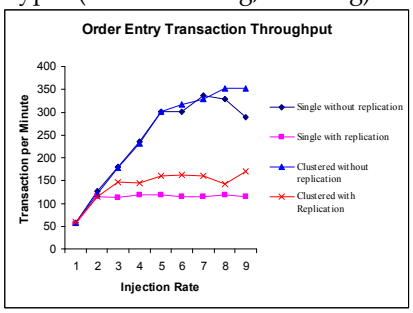


Fig. 6: Order Entry.

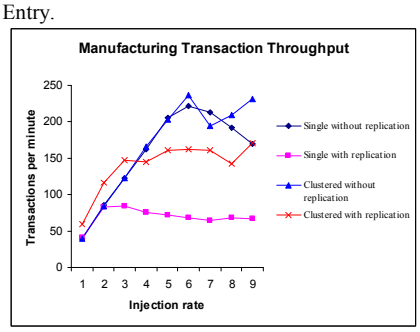


Fig. 7: Manufacturing.

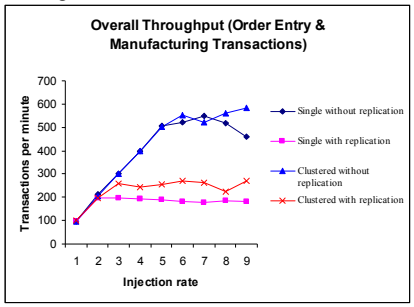


Fig. 8: Overall systems.

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 benefit 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

We next describe our mid-tier replication scheme. From the mid-tier replication perspective, a replicated database resource appears as a single entity, and all operations issued by the transaction manager or containers from either the primary or backups are translated by the proxy adapters to database replicas. So, the scheme described here can work seamlessly with the replication scheme of the previous section.

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 one-copy serializability (at database and application server tiers) and crash failures of servers. There are two different times within a client session when a primary may fail: (1) during non-transactional invocation phase, (2) during transactional phase.

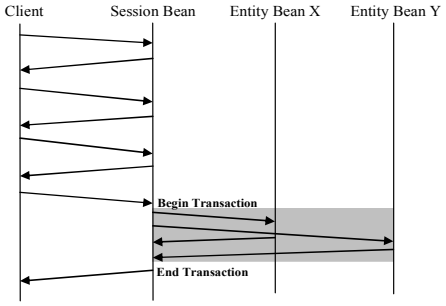


Fig. 9: Interactions between beans and client.

As entity beans access and change persistent state, the time taken to execute application logic via entity beans is longer than enacting the same logic using session beans. The reason for this is twofold: (1) the high cost of retrieving state on entity bean activation and writing state on entity bean deactivation; (2) the transactional management associated to persistent state updates. The structuring of an application to minimize the use of entity beans (and transactions) to speed up execution times is commonplace. This approach to development leads to scenarios in which a client enacts a 'session' (a series of related invocations) on an application server, with the majority of invocations handled by session beans. Transactional manipulation of persistent state via entity beans is usually left to the last steps of processing in a client's session. The sequence diagram in figure 9 describes the style of interaction our model assumes. We only show application level logic invocations (as encoded in EJBs) in our diagram, therefore, we do not show the transaction manager and associated databases. The invocations that occur within a transaction are shown in the shaded area. As mentioned earlier, we assume a client is not part of the transaction.

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

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

5.2. JBoss Implementation

We use interceptors, *management beans* (MBeans), and *Java Management Extensions* (JMX) technologies to integrate our replication service into JBoss. This is the standard approach used when adding services to JBoss: interceptors intercept JBoss invocations while MBeans and JMX combine to allow systems level deployment of services that act upon intercepted invocations. This approach ensures that we do not disturb the functionality of existing services. Figure 10 shows the interceptors and associated services that implement our replication scheme. The interceptors perform the following tasks: *retry interceptor* – identifies if a client request is a duplicate and handles duplicates appropriately; *txinspector interceptor* – determines how to handle invocations that are associated to transactions; *txinterceptor* – interacts with transaction manager to enable transactional invocations (unaltered existing interceptor shown for completeness); *replica interceptor* – ensures state changes associated with a completed invocation are propagated to backups.

The txinterceptor together with the transaction manager accommodates transactions within the application server. The replication service supports inter-replica consistency and consensus services via the use of JGroups [38]. The replication service, retry interceptor, txinspector interceptor and the replica interceptor, implements our replication scheme.

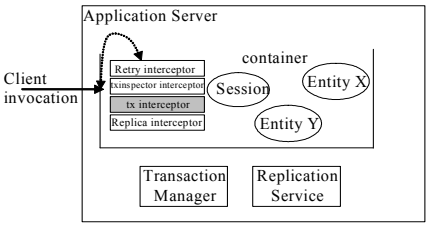


Fig 10: Augmenting application server with replication service.

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

5.2.1. Client side failure handling

To enable failover, instead of a single session bean reference being present in the client proxy interface, a list of references is provided, representing primary and backups (returned when client connects to the remote interface of the desired bean). Client invocations are directed to the primary. If the primary is non-responsive (proxy interface timeouts primary) then the invocation is repeated using each backup in turn until a backup acknowledges the invocation. If a backup is not the primary, it responds to a client invocation with a message indicating the current view of primary/backup; the client re-issues its invocation to the primary (this is achieved transparently without the application layer's knowledge). This process is repeated until: (i) primary responds or; (ii) application server becomes unreachable (no replies from all backups). The proxy interface of the client also maintains a logical clock which timestamps each invocation as it is received from the client. After each timestamp is issued the clock is incremented by one, uniquely identifying each invocation emanating from a client. This information is used by the application server to prevent duplicated processing of a client invocation.

In the JBoss application server alterations were made to enhance interface proxies for the client with the additional functionality required for our failover scheme. Alterations were also made on the invoker MBean at the server to allow the server to determine if the receiving bean is the primary or not (by checking local group log).

5.2.2. Session State Replication

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

If the invocation is not a retry and the receiving bean is the primary, then the invocation is executed by the bean. After bean execution (i.e., when a reply to an invocation is generated and progresses through the interceptor chain towards the client) the replica interceptor informs the replication service of the current snapshot of bean state, the return parameters and the invocation timestamp. Upon delivery confirmation received from the replication service, the primary and backups update their bean and timestamp logs appropriately. Once such an update has occurred, the invocation reply is returned to the client.

5.2.3. Transaction failover management

We assume container managed transaction demarcation. Via this approach to managing transactions the application developer specifies the transaction demarcation for each method via the transaction attribute in a bean deployment descriptor. Using this attribute a container decides how a transaction is to be handled. For example, if a new transaction has to be created for an invocation, or to process the invocation as part of an existing transaction (i.e., the transaction was started earlier in the execution chain). Based on this mechanism, a single invocation of a method can be: a single transaction unit (a transaction starts at the beginning of the invocation and ends at the end of the invocation), a part of a transaction unit originated from other invocation, or non transactional (e.g. the container can suspend a transaction prior to executing a method, and resume the transaction afterwards). We assume that the processing of an invocation may involve one or more beans (both session beans and entity beans) and may accesses one or more databases, requiring two phase commitment.

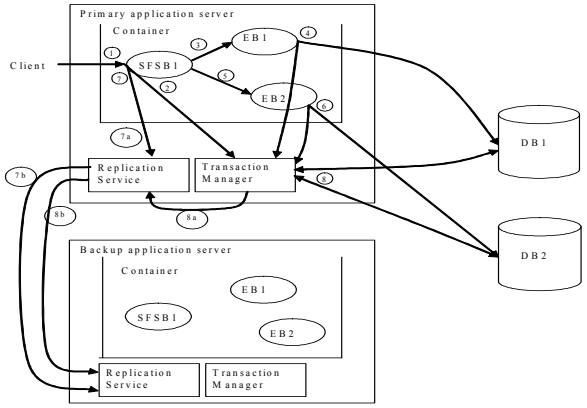


Fig 11: A typical interaction for a transaction processing in EJB

Figure 11 illustrates the execution of a typical transaction (for brevity, we have not shown resource adaptors). We shall use this example as a comparison to highlight the enhancements we have provided to handle transaction failover (this example represents the shaded area shown in figure 9). SFSB stands for a stateful session bean and EB stands for an entity bean. All methods on the beans have a *required* tag as their transaction attribute, indicating to the container that they must be executed within a transaction. The invocation from the client initially does not contain a transaction context. At (1), a client invokes a method on a stateful session bean SFSB1.

The container (on JBoss application server it is the tx interceptor that performs this task) determines that the invocation requires a transaction and calls the transaction manager to create a transaction T1 for this invocation (2). The container proceeds to attach a transaction context for T1 to the invocation. The execution of the method of SFSB1 generates another invocation (3) on EB1 and also an invocation (5) on EB2. At (3) and (5), the container determines that although the invocations need to be executed within a transaction, it does not have to create a new transaction for them as the invocation has already been associated with a transaction context. The invocation on EB1 requires access to a database DB1 (4) and at this point, the container registers DB1 to the transaction manager as a resource associated with T1. The same process happens at (6) where the container registers DB2 to be associated with T1. After the computation on SFSB1, EB1 and EB2 finishes, before returning the result to the client, the container completes the transaction by instructing the transaction manager to commit T1. The transaction manager then performs two phase commit with all resources associated with T1 (8) (not shown in detail here).

We now identify where in the transaction execution described in figure 11 we accommodate for transaction failover. A multicast of the state update of all involved session beans together with the transaction id and information on all resources involved at point (7) is made via their replication service (7a), (7b). That is, when application level logic has ceased we inform backup replicas of the states of resources involved in the transaction (when commit stage is about to commence). A multicast of the decision taken by the transaction manager is made to all backup replica transaction managers after the prepare phase at point (8) via the replication service (8a) and (8b). If the primary fails before reaching point (7), the transactional invocation will not complete, and the client will retry and the backup will execute the computation as a new transactional invocation; but if the primary fails after reaching point (7) the backup will already have the updated state and it will attempt to finish the transaction by continuing the two phase commitment process depending on whether the primary transaction manager has taken a decision or not at point (8).

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

Figure 12 displays the server side invocation handling, with focus on the three interceptors involved in transaction failover. On JBoss the tx interceptor is responsible for inspecting the transaction context from the incoming invocation, and replacing the transaction context when necessary with a new one. Interceptors that are located before the tx interceptor (on side A in figure 12) will see the original transaction context of the invocation while the interceptors that are located after the tx interceptor (on side B in figure 12) will see the new transaction context as defined by the tx interceptor. Therefore, in order to determine which invocation must be executed as a transaction unit, our txinspector interceptor must be placed before the tx interceptor so that it can inspect the transaction context from the incoming invocation and compare it with the transaction attribute of the method being invoked. When the txinspector interceptor determines that an invocation is a unit of a transaction, it flags that invocation with a TRANSACTIONUNIT attribute so that the replica interceptor knows that it has to propagate the state and the transaction information after the computation has finished: end of method execution will result in two phase commit. The txisnpector interceptor also flags non-transactional invocations with a NONTRANSACTIONAL attribute so that the replica interceptor knows that it has to propagate the state without the transaction information.

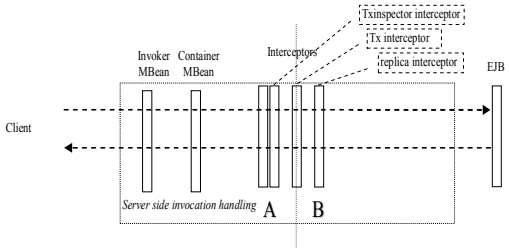


Fig 12: Server side invocation handling for transaction failover

The state update at (7a) includes all information necessary to attempt the transaction at a backup replica (e.g., application logic state, resources associated with transaction, transaction context). The replica interceptor does not have to propagate the state of a session bean after a partially executed transaction, as any failure that happens during the transaction requires a backup replica to execute the original transactional invocation from the beginning (e.g., (1) in figure 11). This follows our initial assumption regarding style of application execution where transactions predominantly consist of a limited number of executions that occur after non-transactional client/application server interactions.

If application level execution has ceased within a transaction and two-phase commit is to be attempted, we can complete a transaction at a backup replica if the primary fails. At (8) the transaction manager performs two phase commit by first sending a prepare message to all transaction participants. After all replies have been received, the transaction manager takes a decision on whether to commit or abort a transaction. We had to modify the JBoss transaction manager to ensure the decision is multicast (using the replication service) to all backup replicas. The multicast message contains the transaction id and the decision taken by the transaction manager. Once delivery notification of this multicast is received from backups by the transaction manager then the decision is sent to transaction participants.

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

nical details.

5.2.4. Load balancing

The scheme described so far assumes a single primary that services all clients. To allow the scalability from clustering while benefiting from mid-tier replication, our scheme must be extendable to support load balancing for processing client requests.

Extending our scheme to allow load balancing of client requests across a cluster of servers is straightforward. This is due to the nature of a session within J2EE: a session describes a relationship between a single client and a server, commonly denoted by the creation, usage and then deletion of a stateful session bean (instances of session beans are not shared by clients).

To support load balancing, a client is attached to a session bean on a server. The choice of server is made in the normal way by the load balancer. This server is the primary, with all other servers acting as backups.

The replication service maintains the mapping between session beans and their primaries, so that each server knows for which sessions they are primary and which they are acting as backup. What is actually happening is that sessions are replicated in a passive manner as opposed to servers themselves.

5.2.5. 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 settings.

Two experiments are performed. First, we measure the overhead our replication scheme introduces. The ECperf driver was configured to run each experiment with 10 different injection rates (1 through 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 replication scheme 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 one of the servers to force the system to failover to the backup server.

Figures 13 and 14 present graphs that describe the throughput and response time of the ECperf applications respectively. 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 15 presents a graph that shows the throughput of the replicated system and the standard implementation. After 720 seconds running (12 minutes), we crash a server. When no replication is present the failure of the server results in throughput of this server decreasing to zero, as there is no backup to continue the computation. When replication is present performance drops when failure of a server is initiated. However, the backup assumes the role of continuing the work of the failed server, allowing for throughput to rise again. An interesting observation is that throughput on the sole surviving server is higher than it was on the failed server prior to crash. This may be partially explained by the fact that only one server exists and no replication is taking place. However, the fact that throughput is actually higher is because the server continues with its own workload in addition to that of the failed server. The initial peak in throughput may be explained by the completion of transactions that started on the failed server but finish on the continuing server: an additional load above and beyond the regular load generated by injection rates.

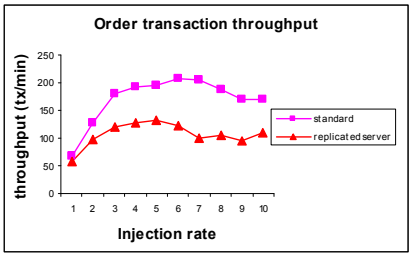


Fig. 13: Throughput for order entry

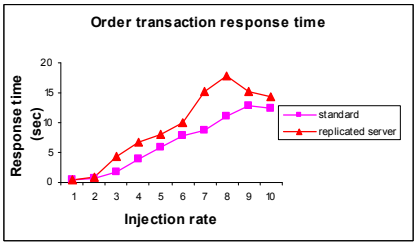


Fig. 14: Response time for order entry

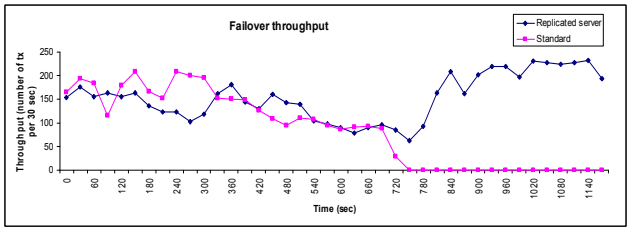


Fig 15: Performance figures with failure

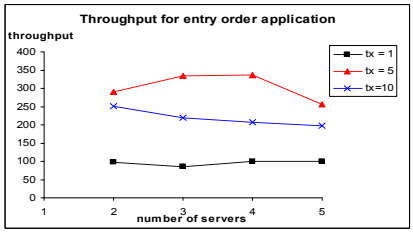


Fig 16: Throughput with varying server numbers

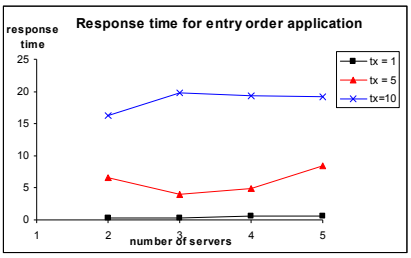


Fig 17: Response time with varying server numbers

Figures 16 and 17 present graphs that describe the performance of the ECperf applications, with varying numbers of servers as backups. In this configuration, only one server is the primary for all client requests, and the rest 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 rates, such as tx=5 and tx=10, the differences become obvious and may be related to an increase in server load when dealing with replication overhead and sequential execution of state updates.

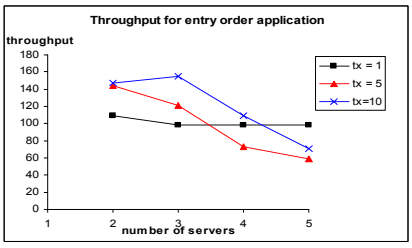


Fig 18: Throughput in load balancing configuration

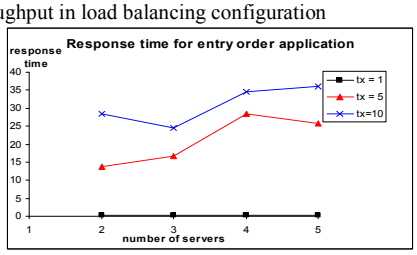


Fig 19: Response time in load balancing configuration

Figures 18 and 19 present 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: each session has a primary with remaining servers acting as backups. The new server for each session is determined randomly. When an application server fails, the remaining servers share the load.

When the transaction rate is low (tx = 1) throughput remains the same irrelevant of the number of servers employed. This indicates that such a low rate is comfortably handled by 2 servers. When transaction rates increase (tx = 5, 10), additional servers do afford a greater capacity for throughput. However, when server numbers increase beyond 2 (tx = 5) and 3 (tx = 10) there is a marked reduction of throughput. This indicates that the system, under higher transaction rates with more backup servers, reaches maximum capacity. This is reinforced by the observations of response times. This behavior is to be expected as the increased cost of the replication scheme amongst additional servers increases the burden on the overall system.

This suggests that in a cluster of n machines, it is better to have a configuration where each machine has just one backup rather than n-1 backups.

6 Conclusion

The replication schemes developed in this paper are based on a combination of several existing techniques that have been published. The focus of this work has been to combine existing techniques to provide a complete working solution on a real platform (J2EE application servers, in our case). 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 or separately without any difficulties.

There are a number of valuable lessons learned from our work presented here: (i) providing replication in the data store tier and the application server tier in a modular fashion such that they can work independently or together requires careful design; (ii) the interceptor based design of JBoss internals provide an ideal way of enhancing services with additional value without hindrance to existing services; (iii) to handle transaction failover we had no alternative but to alter the source code of the transaction manager, which is made possible due to the software being open source.

Acknowledgment

The work has been funded in part by European Union project TAPAS (IST-2001- 4069) and grants from the UK. Kistijantoro's work was funded by QUE Project Batch III, Department of Informatics Engineering Institute of Technology Bandung, Indonesia.

References

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

[2] www.jboss.org

[3] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava and M.C. Little, "Component Replication in Distributed Systems: a Case study using Enterprise Java Beans", Proc. IEEE Symp. on Reliable Distrinbuted Systems (SRDS), pp. 89 - 98, Florence, October 2003,

[4] A. I. Kistijantoro, G. Morgan and S. K. Shrivastava, "Transaction Manager Failover: A Case Study Using JBOSS Application Server", Proc. International Workshop on Reliability in Decentralized Distributed systems (RDDS), pp. 1555 - 1564, Montpellier, France, October 2006

[5] K. Birman, "The process group approach to reliable computing", CACM , 36(12), pp. 37 - 53, December 1993.

[6] P. A. Bernstein, V. Hadzilacos and Nathan Goodman, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[7] M. C. Little, D. McCue and S. K. Shrivastava, "Maintaining

information about persistent replicated objects in a distributed system", Proc Int. Conf. on Distributed Computing Systems (ICDCS), pp. 491 - 498, Pittsburgh, May 1993

[8] M.C. Little and S K Shrivastava, "Implementing high availability CORBA applications with Java", Proc. IEEE Workshop on Internet Applications (WIAPP '99), pp. 112 - 119, San Jose, July 1999

[9] P. Felber, R. Guerraoui, and A. Schiper, "The implementation of a CORBA object group service", Theory and Practice of Object Systems, 4(2), pp. 93 - 105, April 1998

[10] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "Consistent Object Replication in the Eternal System", Theory and Practice of Object Systems, 4(2), pp. 81 - 92 April 1998

[11] R. Baldoni, C. Marchetti, "Three-tier replication for FT-CORBA infrastructures", Software Practice & Experience, 33(18), pp. 767 - 797, May 2003

[12] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "A Fault Tolerance Framework for CORBA", Proc. of the IEEE Int. Symp. on Fault-Tolerant Computing (FTCS), pp. 150 - 157, Madison, USA, June 1999

[13] X. Zhang, M. Hiltunen, K. Marzullo, R. Schlichting, "Customizable Service State Durability for Service Oriented Architectures", In Proc. of the Sixth European Dependable Computing Conf. (EDCC), pp. 119 - 128, Coimbra, Portugal, October 2006

[14] Salas, J., Perez-Sorrosal, F., Patiño-Martínez, M., and Jiménez-Peris, "WS-Replication: a Framework for Highly Available Web Services", In Proc. of the 15th Int. Conf. on World Wide Web (WWW), pp. 357 - 366, Edinburgh, Scotland, May 2006

[15] B. Roehm., "WebSphere Application Server V6 Scalability and Performance Handbook", IBM Red Books, ibm.com/redbooks

[16] S. Barghouthi, D. Banerjee, "Building a High Availability Database Environment using WebShpere Midddleware: Part 3: Handling Two-Phase Commit in WebSphere Application Server Using Oracle RAC", http://www.ibm.com/developerworks/websphere/techjournal/0710_barghouthi/0710_barghouthi.html

[17] A. Wilson, "Distributed Transactions and Two-Phase Commit", SAP White Paper SAP™ NetWeaver, https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/3732d690-0201-0010-a993-b92aab79701f

[18] S.Labourey and B.Burke, " JBoss Clustering 2nd Edition", www.jboss.org, 2002

[19] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso, "Database Replication Techniques: a Three Parameter Classification", Proc. of the 19th IEEE Symp. on Reliable Distributed Systems (SRDS), pp. 206 - 215, Nürnberg, October 2000

[20] B. Kemme and G. Alonso, "A New Approach to Developing and Implementing Eager Database Replication Protocols", ACM Trans. on Database Systems (TODS), 25(3), pp. 333 - 379, September 2000

[21] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme and G. Alonso, "Consistent Database Replication at the Middleware Level", ACM Trans. on Computer Systems (TOCS). 23(4) pp. 1 – 49, November 2005

[22] C. Amza, A. L. Cox, W. Zwaenepoel, "Distributed versioning: consistent replication for scaling back-end databases of dynamic content web sites", Proc. Middleware 2003, ACM/IFIP/USENIX Int. Middleware Conf., pp. 998 - 1008, Rio de Janeiro, Brazil, June 2003

[23] E. Cecchet, M. Julie, and W. Zwaenepoel, "C-JDBC: Flexible Database Clustering Middleware", USENIX Annual Technical Conf., pp. 89 - 102, Boston, June 2004

[24] R. S. Barga, D. B. Lomet, T. Baby, and S. Agrawal, "Persistent Client-Server Database Sessions", In Proc. of Advances in Database Technology - EDBT 2000: 7th International Conference on Extending Database Technology, pp. 462 - 477, Konstanz, Germany, March 2000

[25] B. Kemme, R. Jimenez-Peris et al, "Exactly once Interaction in a Multi-tier Architecture", VLDB Conf. Trondheim, Norway. Aug. 2005. Workshop on design, implementation, and deployment of database replication.

[26] W. Zhao, L. M. Moser and P. M. Melliar-Smith,"Unification of Transactions and Replication in Three-tier Architectures Based on CORBA", IEEE Trans. on Dependable and Secure Computing (TDSC), Vol. 2, No. 1, pp. 20 - 33, January 2005

[27] M. Hammer and D. Shipman, "Reliability mechanisms for SDD-1: A system for distributed databases" ACM Transactions on Database Systems 5(4) (TODS): pp. 431 - 466, 1980

[28] P.K. Reddy and M. Kitsuregawa, "Reducing the blocking in two-phase commit protocol employing backup sites", Proc. of Third IFCIS Conf. on Cooperative Information Systems (CoopIS'98), pp. 406 - 415, New York, August 1998

[29] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, S. Arévalo "A Low-Latency Non-blocking Commit Service", 15th Int. Conf. on Distributed Computing (DISC), pp. 93 - 107 , October 2001

[30] Ö. Babaoglu, A. Bartoli, V. Maverick, S. Patarin, J. Vuckovic and H. Wu, "A Framework for Prototyping J2EE Replication Algorithms", Int. Symp. on Distributed Objects and Applications (DOA), pp. 1413 - 1426, Agia Napa, October 2004

[31] H. Wu, B. Kemme, V. Maverick, "Eager Replication for Stateful J2EE Servers", Int. Symp. on Distributed Objects and Applications (DOA), pp. 1376 - 1394, Agia Napa, Cyprus, October 2004

[32] F. Perez-Sorrosal, J. Vuckovic, M. Patino-Martinez, R. Jimenez-Peris, "Highly Available Long Running Transactions and Actibities for J2EE Applications", In Proc. of the 26th IEEE Int. Conf. on Distributed Computing Systems (ICDCS), Lisboa, Portugal, July 2006

[33] M.C. Little and S K Shrivastava, "Integrating the Object Transaction Service with the Web", Proc. Of IEEE/OMG Second Enterprise Distributed Object Computing Workshop (EDOC), pp. 194 - 205, La Jolla, CA, November 1998

[34] M.C. Little and S K Shrivastava, "Java Transactions for the Internet", Distributed Systems Engineering, 5(4), pp. 156 - 167, December 1998,

[35] P. A. Bernstein, Meichun Hsu, B. Mann, "Implementing recoverable requests using queues", Proceedings of ACM SIGMOD Int. Conf. on Management of Data, pp. 112 - 122, Atlantic City, New Jersey, 1990

[36] R. Barga, D. Lomet, G. Shegalov, G. Weikum , "Recovery guarantees for Internet applications", ACM Trans. on Internet Tech. 4(3), pp. 289 - 328, August 2004

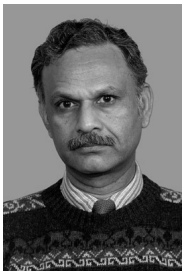
[37] http://java.sun.com/developer/earlyAccess/j2ee/ecperf/download.html

[38] http://www.jgroups.org

Achmad Imam Kistijantoro received the BSc degree in computer science from Institut Teknologi Bandung in 1996, the MSc degree in computer science from TU Delft in 1999, and the PhD degree in computer science from Newcastle University in 2006. He is currently a lecturer in Informatics Department of Institut Teknologi Bandung, Indonesia. His research interests are in the area of distributed systems and wireless/mobile computing.

Graham Morgan
University of New-
Graham has been a
School of Computing
Whenever he gets
techniques to ease
tolerant applications,
virtual worlds, and
dleware.

received his PhD from the
castle in 1999. Since 2000
faculty member in the
Science at Newcastle.
time he develops tools and
the development of fault-
multi-user interactive
inter-organizational mid-



Santosh Shrivastava received his Ph.D. from Cambridge in 1975. He was appointed a Professor of Computing Science, University of Newcastle upon Tyne in 1986 where he leads the Distributed Systems Research Group. His research interests span many areas of distributed computing including middleware, transaction processing and fault tolerance. Current focus is on middleware for supporting inter-organizational services where issues of trust, security, fault tolerance and ensuring compliance to service contracts are of great importance.

Dr Mark Little is Director of Standards and Engineering Director for Red Hat, where he is in charge of the SOA Platform. Prior to this Mark was Co-Founder and Chief Architect at Arjuna Technologies, an HP spin-off, where he lead the transaction teams. Mark was a Distinguished Engineer at HP before starting Arjuna Technologies. He has written many papers, several books and speaks regularly at conferences and workshops.