Component Replication in Distributed Systems: a Case study using Enterprise Java Beans

Achmad I. Kistijantoro, Graham Morgan, Santosh K. Shrivastava

School of Computing Science, Newcastle University, Newcastle upon Tyne, UK
{A.I.kistijantoro, Graham.Morgan, Santosh.Shrivastava}@newcastle.ac.uk

Mark C. Little

Arjuna Technologies Ltd., Newcastle upon Tyne, UK
Mark.Little@arjuna.com

Abstract

A recent trend has seen the extension of object-oriented middleware to component-oriented middleware. A major advantage components offer over objects is that only the business logic of an application needs to be addressed by a programmer with support services required incorporated into the application at deployment time. This is achieved via components (business logic of an application), containers that host components and are responsible for providing the underlying middleware services required by components and application servers that host containers. Well-known examples of component middleware architectures are Enterprise Java Beans (EJBs) and the CORBA Component Model (CCM). Two of the many services available at deployment time in most component architectures are component persistence and atomic transactions. This paper examines, using EJBs, how replication for availability can be supported by containers so that components that are transparently using persistence and transactions can also be made highly available.

Keywords: Availability, components, CORBA Component Model, Enterprise Java Beans, fault tolerance, middleware, replication, transactions

1. Introduction

Modern client-server distributed computing systems may be seen as implementations of an N-tier architecture. In a typical four tier architecture the first tier (*client tier*) consists of client applications containing browsers, with the remaining three tiers deployed within an enterprise representing the server side; the second tier (*Web tier*) consists of a Web server that receives client requests typically via HTTP and passes on the requests to specific applications residing in the third tier (*business tier*) that is capable of hosting distributed applications; the fourth tier (*enterprise information systems tier*) contains databases and legacy applications of the enterprise. The platform providing the Web tier plus business tier is usually called an *application server.*

An application server typically deploys a variety of object oriented middleware services using an object request broker (ORB) and provides to applications a higher level of abstraction of component oriented middleware. It is worth examining here briefly why this higher level of abstraction has been considered necessary. Although object oriented middleware provides type checked remote invocations and standard ways of using commonly required services (naming, persistence, transactions, etc.), a problem still remains that application developers have to worry about application logic as well as technically complex ways of using a collection of services. For example, use of transactions on distributed objects requires concurrency control, persistence and the transaction services to be used in a particularly intricate manner. Component oriented middleware alleviates this difficulty by the use of *components* that are composed of objects, and *containers* that host component instances. Containers take on the responsibility for using the underlying middleware services for communication, persistence, transactions, security and so forth and a developer's task is simplified to specifying the services required by components in a declarative manner. Thus, a major advantage components offer over objects is that only the business logic of an application (encoded in components) needs to be addressed by a developer. Well-known examples of component middleware architectures include Enterprise Java Beans (EJBs) and the CORBA Component Model (CCM).

In this paper we investigate how software implemented fault tolerance techniques can be applied to support component replication for high availability. We take the specific case of EJB components and consider *strict consistency* (that requires that the states of all available

copies of replicas be kept mutually consistent). We take EJBs primarily because not only are they used extensively in industry, but also because open source implementations of application servers for EJBs are available for experimentation. However, we believe that the ideas presented here are of interest to the general case of component middleware. In particular, as EJBs are closely related to the language independent CCM, the ideas presented here can be applied to the world of CORBA components.

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. In the spirit of component middleware, we prefer to delegate the responsibility of replication management to the container (*container managed replication*). We therefore examine how replication for availability can be supported by containers 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 (where computations take place) and databases (where persistent data is kept).

Ours is an engineering task that poses several problems that require careful resolution. In order to highlight these problems and possible solutions, we consider three replication approaches, beginning with a simple approach into which we incrementally incorporate additional sophistication. To explain these approaches, consider a simple transaction that, in a non-redundant system, operates on EJBs deployed within a single container, with the beans storing their persistent states on a database; in this system, we incorporate redundancy as follows:

(i) *State replication with single application server*: persistent state is stored on multiple databases; here database (but not the application server) 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.

(ii) *State replication with clustered application servers*: persistent state is stored on multiple databases; multiple application servers are used for load sharing the total number of transactions in the system; an individual transaction has the same reliability features as case (i).

(iii) *State and computation replication with clustered application servers*: persistent state is stored on multiple databases and instances of beans are replicated on the cluster of application servers; this is an ideal set of server side availability measures as it is able to mask a finite number of application server and database failures.

Existing application servers as yet do not provide any suppport 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 technique). Computation replication is well supported for *stateless* computations, but not so well for *stateful* computations in that transactions are not supported. These limitations are examined in detail in section 2.3 of the paper.

This paper shows how support for managing redundancy can be incorporated within application servers. We require that our solutions must be open (non-propriety) and implementable in software. Furthermore, we require that our solutions be transparent to the component middleware. The transparency requirement imposes the following constraints on our solutions: (a) no modifications to the API (Application Programming Interface) between client and component; (b) no modifications to the API between an EJB and the container; and (c) no modifications to the existing middleware services and APIs. Given these constraints, the contributions of the paper are to show that approaches (i) and (ii) can be implemented with relative ease. We have implemented these approaches to validate this claim. However, it is not possible to implement approach (iii) without breaking constraint (c); we outline what enhancements would be necessary to the component middleware to support approach (iii).

The paper is structured as follows: background information on EJB component middleware is presented is section two; replication approaches are then discussed in section three; section four presents performance results; related work is presented in section five; with concluding remarks in six.

2. EJBs and application servers

In the first two sub-sections we describe the terminology and basic concepts of Java 2, Enterprise Edition (J2EE) middleware that should be sufficient for our purposes (for additional details, see [1]). The third sub-section describes availability measures currently provided in application servers.

2.1. EJBs

Three types of EJBs have been specified in J2EE: (1) *Entity beans* represent and manipulate persistent data of an application, providing an object-oriented view of data that is frequently stored in relational databases. (2)

Session beans on the other hand do not use persistent data, and are instantiated on a per-client basis with an instance of a session bean available for use by only one client. A session bean may be *stateless* (does not maintain conversational state) or *stateful* (maintains conversational state). Conversational state is needed to share state information across multiple requests from a client. (3) *Message driven beans* provide asynchronous processing by acting as message listeners for Java Messaging Service (JMS).

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

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

2.2 Transactional EJB applications

The main elements required for supporting transactional EJB applications deployed in an application server are shown in figure 1. 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 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 in practical systems. 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 (ACID: Atomicity, Consistency, Isolation, Durability).

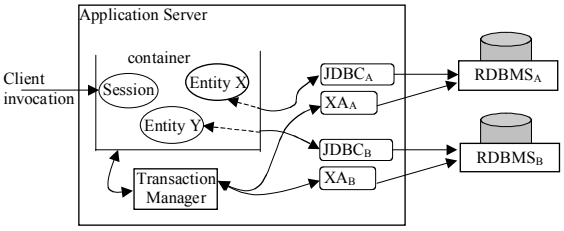


Figure 1 – EJB transactions.

Communications between an RDBMS and a container is via a Java DataBase Connectivity (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 J2EE architecture this interface is referred to as the *XAResource interface* (shown as XA in figure 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.

We now describe, with the aid of figure 1, 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 - we are assuming container managed persistence - retrieving their states from the appropriate resource managers for initialising 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 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 our example, all the beans are in the same container. Support for distributed transactions involving beans in multiple containers (on possibly distinct application servers) is 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. For this reason, in the rest of the paper, we need only consider the simple case of a transaction that, in a non redundant system, spans EJBs within a single container and a few resource managers.

2.3. Availability measures in current application servers

Commercial application servers make use of multiple application servers deployed over a cluster of machines with some specialist router hardware (see below) to mask server failures and rely on propriety replication mechanisms of database vendors for database availability (for example, some Oracle database products support a specific replication scheme). As we discuss below, these availability measures do not integrate properly with EJB initiated transactions.

Application servers are typically deployed over a cluster of machines. A locally distributed cluster of machines (a set of machines) with the illusion of a single IP address and capable of working together to host a Web site provides a practical way of scaling up processing power and sharing load at a given site. Commercially available application server clusters rely on a specially designed gateway router to distribute the load using a mechanism known as network address translation (NAT). The mechanism operates by editing the IP headers of packets so as to change the destination address before the IP to host address translation is performed. Similarly, return packets are edited to change their source IP address. Such translations can be performed on a per session basis so that all IP packets corresponding to a particular session are consistently redirected. Load distribution can also be performed using a process group communication system as first suggested by the ISIS system [2]; a recent open source application server has such a mechanism [3]. The two market leaders in the application server space, WebSphere [4] from IBM and WebLogic [5] from BEA, have very similar approaches to clustering. They typically characterise clustering 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 configurations 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 property.

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. The load-balancing property should ensure that the client load gets redistributed to the remaining servers, each of which will henceforth process a proportionately slightly higher percentage of the total load. 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 outlined 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.

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. Failover support in this case is trivial: 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 bean is required to serialize its state to a datastore at the end of each client request and for the subsequent bean instance in the other server to deserialize the state before servicing the new request (obviously the servers must have access to the same datastore). The replication of the datastore is assumed to be the domain of the datastore

itself. This way, some of the functionality available for stateless sessions can be regained. However, even in this case, a failure during serialization of the bean's state (which could result in the state being corrupted) is not addressed. There is a more serious limitation: transactions cannot be supported. if transactional access to a bean is used, then the same server instance must be used for every invocation on that bean.

3. Component replication

The three approaches mentioned at the start of the paper will be considered in some detail in this section. We will nevertheless highlight only the essential aspects of our designs, glossing over minute details of replica management. We begin by stating the main assumptions. We will assume that an application server either works as specified or simply stops working (*crash*). After a crash, a server is repaired within a finite amount of time and made active again. We assume that resource managers support ACID transactions. As stated earlier, in this paper we consider availability measures for session and entity beans and consider strict consistency.

3.1. State replication with 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 in the scheme shown in figure 1 and use 'available copies' approach to data replication ('read from any, write to all') [6]

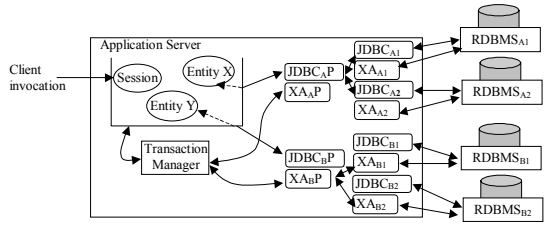


Figure 2 – Replication of state.

Figure 2 depicts an approach to resource manager replication that leaves the container, transaction managers internal to 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.

To ensure resource manager replica states 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 correctly functioning 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, $RDBMS_A$ 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).

The outline design presented here provides a simple and practical way of introducing data replication into component middleware.

3.2. State replication with clustered servers

A single transaction manager may be used to manage transactions across clustered application servers. However, to ensure a transaction manager does not present a bottleneck in the system, and a single point of failure, we assume that application servers are replicated complete with transaction managers (this is the case in practical systems). Figure 3 depicts a scenario where a cluster contains two application servers (AS1 and AS2) that are accessing shared resource manager replicas. We have maintained our architecture for providing state replication as described in the previous subsection. Only the resource adaptor proxies are shown to make the diagram simple; further, arrowed communication lines between resource managers and proxies - not all such lines have been shown - are dashed to emphasise that the

communication is actually through resource adapters and XAResources (that are not shown in the figure).

The possibility of multiple transactions running on different application servers each accessing shared resource manager replicas increases the difficulty of ensuring resource manager replicas remain mutually consistent. For example, assume 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. We want to prevent the situation that enables T1 to proceed as the container of AS1 manages to obtain the state of X from $RDBMS_{A1}$ and at the same time, T2 proceeds as the container of 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 that the ordering of load requests is serialized, causing conflicting transactions to block until locks are released. To ensure resource managers remain mutually consistent the store request (when an entity bean updates its persistent state in the resource manager) is issued to all resource manager replicas.

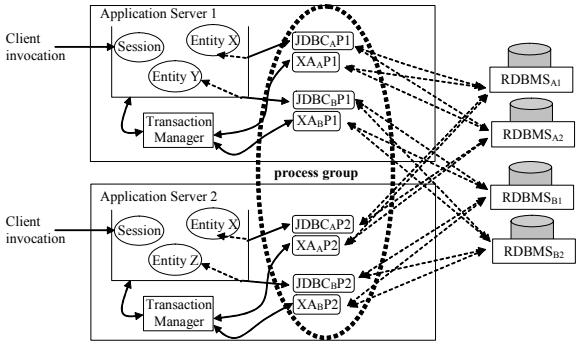


Figure 3 – Clustering and state replication.

Resource adaptor proxies from different application servers that need to access the same resource manager replicas have to agree on the primary read resource manager. In the presence of resource manager failure (and a possibility of incorrect suspicion of resource manager failure) an agreement protocol needs to be executed between resource adaptor proxies, so that they have the same view on the primary. We can obtain this facility by making use of a group communications system that supports the abstraction of a process group (and totally ordered atomic multicast, which is not required in this particular case) [2]. In our example, all the 'A' proxies need to be in a process group, and all the 'B' proxies need to be in a process group. For the sake of simplicity, in the figure, we have shown all the proxies to be the member of a single group, as this simple arrangement will work as well.

Furthermore, the identification of the primary read resource manager needs to be available to an application server after a restart. This may be achieved by allowing a resource adaptor proxy of the newly restarted application server to gain the identity of the primary read resource manager from existing application servers. This requires the list of available resource adaptor proxies to be stored persistently by an application server. This type of persistent data may be stored within the XA element of a proxy resource adaptor.

The outline design presented here provides a practical way of introducing data replication for transactional EJB applications into a cluster of servers.

3.3. State and computation replication with clustered application servers

We now consider what is involved in masking application server failures. Since state replication can be incorporated transaparently in a cluster as discussed earlier, we can examine masking of application server failures independent of state replication. We need the ability to replicate containers on distinct application servers (with distinct transaction managers) and ensure the states of EJBs in container replicas and transactional states within the respective transaction managers are mutually consistent. One way of doing this would be to use the process group approach, and manage the replicas using active or passive replication techniques. Unfortunately, there are pitfalls:

Active replication: application servers are intrinsically multi-threaded and even working in a homogeneous application server environment (i.e., where application server replicas are copies of the same implementation) it is almost impossible to ensure that the same invocation received at container replicas will be dealt with identically. When using heterogeneous application servers, issues such as thread pooling add more complications.

Passive replication: when using primary-copy replication, it is theoretically possible in a homogenous environment to checkpoint the state of a container (e.g., at transaction commit time) such that a backup can take over in the event of primary failure: it is assumed that an application server implementation can create the checkpoint in such a format that another instance of the same implementation can later read it in. In a heterogeneous environment, such

state check-pointing requires intimate knowledge of the different internal and external state formats of the application server implementations involved. Most vendors do not make this kind of information publicly available (and especially not to competitors). Unfortunately, at present there is no open standard for defining state transfer between application server implementations, so any solution would be application server specific.

The same issues arise for replication of transaction managers. Although all transaction managers will have the concept of a transaction log (the persistent entity into which information about transactions is kept such that transactions may be completed in the event of recovery after failures), there is no open standard available for initialising a transaction manager from a checkpointed log, so state transfer mechanisms between transaction managers will be transaction manager specific.

Ideally, we would like a snapshot of the container (or possibly the application server) state to be taken such that another (potentially heterogeneous implementation) could take this snapshot and reconstitute the environment at the point it was originally taken. In order to do this, standard interfaces need to be defined (and used) by all compliant application server implementations. There is an attempt to address some of this in JSR 117 (J2EE APIs for Continuous Availability), which is developing the notion of a *Field Replaceable Unit* (FRU): an FRU is a collection of modules that can be deployed independently of other application modules that may be part of (for example) an application server [7]. The definer of an FRU is responsible for ensuring that state transfer between implementations is possible (by defining the format of an externalised state and specifying how this is created and can be used to initialise another FRU). This work is at an early stage.

4. Experiment and results

Experiments were carried out to determine the performance of our system over a single LAN. We consider the figures presented here as initial investigations into the performance of our system, further work is required to identify the performance of replicated state during a number of failure scenarios (e.g., application failure, resource manager failure). Javagroups [23] was used as the group communication sub-system in our clustered experiments.

4.1 Implementation and setup

Four experiments were carried out (configuration described in figure 4) 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 using a single resource manager without state replication (figure 4.i).

2. Single application server with state replication – Experiment 1 was repeated, with replica resource managers accessed by our resource adaptor proxy (figure 4.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.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.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) [20] 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.

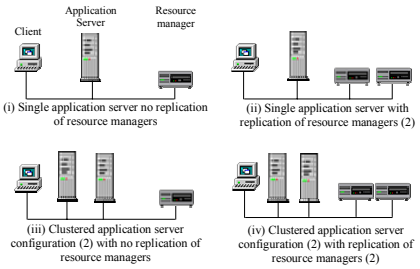


Figure 4 – Experimental setup

ECperf [21] was used as the demonstration application in our experiments. For the purposes of our experiments we now provide a brief description of ECperf. 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

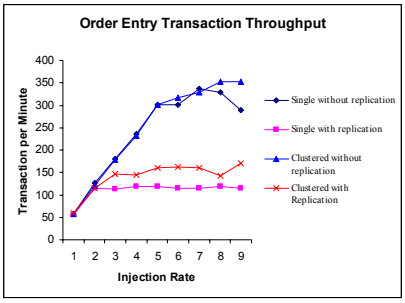
that provides a realistic approximation to what may be expected in a real-world scenario via a system that represents manufacturing, supply chain and customer order management. ECperf is particularly suited to modelling "Just in Time" manufacturing concepts, with goals of maximum efficiency coupled with minimum inventories. Therefore, the type and frequency of client orders has a direct impact on the manufacturing process which in turn influences stock ordering. 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.

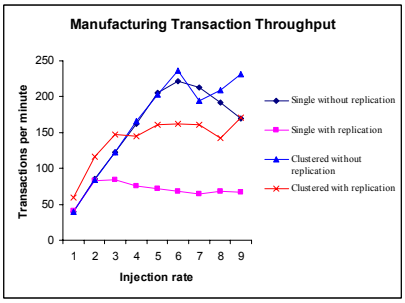
4.2 Performance results

Figure 5 presents three graphs that describe the throughput of the ECperf applications; figure 5(i) identifies the throughput for the entry order system, figure 5(ii) identifies throughput for the manufacturing application and figure 5(iii) identifies the cumulative throughput of the ECperf system (entry order and manufacturing combined).

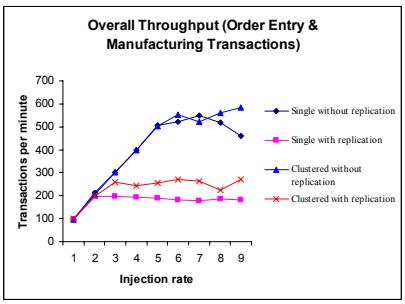
On first inspection we can see that the introduction of replicated resource managers lowers the throughput of clustered and non-clustered configurations when injection rates rise above 2 (figure 5(iii)). However, there is a difference when comparing order entry and manufacturing applications. The manufacturing 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 and the types of transactions used by each application. The dependency on a resource manager coupled with the supplier emulator (which is not replicated) for manufacturing is such that the impact of replication is not as prevalent for manufacturing as order entry.



(i) Order Entry



(ii) Manufacturing



(iii) System (order entry & manufacturing)

Figure 5 – Performance figures.

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. Related work

Replicated data management techniques that go hand in hand with transactions have been discussed in [6]. Object replication using process group communication, originally developed in the ISIS system [2], has been studied extensively [8,9]. The Eternal system [10] offers replicated, non-persistent CORBA objects. A complete fault tolerance framework for CORBA objects (FT-CORBA) is presented in [11].

Existing standards relating to object group approaches to replication, such as [11], have not considered the issue of integrating transactions and persistence. When using group communications to support active replication schemes there is an expectation that all replicas exhibit deterministic behaviour. However, as discussed in 3.3 this approach is not appropriate for guaranteeing replica states remain mutually consistent when considering multi-threaded J2EE application servers. A passive approach to replication in J2EE environments has been developed [3] in an attempt to enable a client session that consists of multiple client requests to continue processing on a backup application server when the primary fails without the need to abort the whole client session. There are two shortcomings to this scheme: (1) only stateful session beans are considered; (2) client invocations that make use of this scheme may not benefit from transactions. The limitations of this scheme reflect the difficulties associated to passive replication in J2EE as discussed earlier in 3.3.

The Arjuna system supported replication of transactional persistent objects [12, 13, 14]. The use of transactions and group communication for supporting persistent object replication is investigated in [15]. The replication approach presented in section 3.2 is based on the approach discussed in [15].

An approach to integrating replication and transactions suitable for n-tier systems has been proposed [22]. The authors base their solution on combining FT-CORBA and the CORBA Object Transaction Service (OTS). Nested transactions are a key feature of their protocol, allowing transactions to be rolled back in part and restarted on backup servers. However, applying such an approach in J2EE will be problematic as only flat transactions are supported by the J2EE specification at the moment. Additionally, the problems associated with transaction manager replication (as discussed earlier in 3.3) are not addressed.

A classification of database replication techniques has been presented that suggests group communications can be utilised in the support of eager (strong consistency) replication protocols [16]. Traditionally, lazy replication schemes are favoured by commercial products because the communication overheads of eager schemes compared to lazy schemes is significant. However, the inconsistencies across database replicas that may result from lazy schemes (not an issues in eager schemes) coupled with promising initial results of integrating group communication into replica database schemes identify eager schemes as a viable choice in the future [17]. Group communication based mechanisms that allow new members of a database replica group to retrieve current state information from existing database replicas are discussed in [18]. Within the context of the study presented here, a drawback of these approaches is the need to integrate a group communications sub-system into the database architecture, making such schemes difficult to work with existing commercial databases. A second drawback is that the transaction model assumed is not distributed (a transaction can not span multiple databases). To overcome the first drawback, a middleware layer has been proposed that provides the required functionality to enable a more seamless integration of group communication based eager replication schemes into existing commercial databases [19]. If the second drawback can be removed, then these approaches will be very suitable for supporting state replication.

6. Concluding remarks

In this paper we have examined how for the case of EJB component middleware, replication for availability can be supported by containers. We have shown that whereas persistent state replication can be incorporated

transparently, the same is not yet possible for application servers. Enhancements required for enabling this have been pointed out and explained. We have demonstrated our approach to state replication in the clustered and non-clustered application server environments via a series of experiments using a well known benchmarking software tool for J2EE. Our experiments show that there is a significant overhead in implementing state replication. However, our experiments do indicate that clustering with state replication still provides a more scalable solution than a non-clustered approach to application server deployment. Future work will concentrate on the development of state and computation replication with clustered application servers (described in 3.3) and further experiments to determine the benefit of our system in the presence of failures.

Acknowledgements

This work is part-funded by European Union under Project IST-2001-34069: "TAPAS"; the UK DTI e-Science programme under project "GridMist"; Kistijantoro's work is funded by QUE Project Batch III, Department of Informatics Engineering, Institute of Technology Bandung, Indonesia.

7. References

[1] R. Monson-Haefel, "Enterprise Java Beans ", O'Reilly, 2001

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

[3] JBoss application server: www.jboss.org

[4] WebSphere Scalability: WLM and Clustering, September 2000, http://ibm.com/redbooks/sg246153.pdf

[5] BEA White Paper, "Achieving Scalability and High Availability for E-Business, ftp://edownload:BUY_ME@ftpna2.bea.com/pub/downloads/wls_clustering.pdf, September 2001.

[6] P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[7] JSR 117: J2EE APIs for Continuous Availability http://www.jcp.org/jsr/detail/117.jsp

[8] R. Guerraoui, P. Felber, B. Garbinato and K. Mazouni, "System support for object groups", Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications, OOPSLA 98.

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

[10] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan,"Consistent Object Replication in the Eternal System", Theory and Practice of Object Systems, vol. 4, no. 2 (1998).

[11] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "A Fault Tolerance Framework for CORBA," Proceedings of the IEEE International Symposium on Fault-Tolerant Computing, Madison, WI (June 1999), pp. 150-157.

[12] M.C. Little and S. K. Shrivastava, "Replicated K-resilient objects in Arjuna", Proceedings of the 1st IEEE Workshop on the Management of Replicated Data, Houston, November 1990, pp. 53-58.

[13] M. C. Little, D. McCue and S. K. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system", ICDCS-13, Pittsburgh, May 1993, pp. 491-498.

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

[15] M. C. Little and S. K. Shrivastava, "Integrating Group Communication with Transactions for Implementing Persistent Replicated Objects", Chapt. 10, Advances in Distributed Systems, LNCS Vol. No. 1752, 2000.

[16] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso, "Database Replication Techniques: a Three Parameter Classification", Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000), Nürnberg, October 2000.

[17] B. Kemme & G. Alonso, "A New Approach to Developing and Implementing Eager Database Replication Protocols", ACM Transactions on Database Systems (TODS), Volume 25, No. 3, September 2000, pp 333-379.

[18] B. Kemme, A. Bartoli, O. Babaouglu, "Online Reconfiguration in Replicated Databases Based on Group Communication", Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN 2001), Goteborg, Sweden, June 2001.

[19] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, G. Alonso, "Improving the Scalability of Fault-Tolerant Database Clusters: Early Results", Proc. of the IEEE 22nd Int. Conf. on Distributed Computing Systems, Vienna, July 2002.

[20]http://otn.oracle.com/products/oracle9i/pdf/Oracle9i_Database_summary.pdf, May 2002

[21]ftp://ftp.java.sun.com/pub/spec/j2ee_ecperf/maon1rgh/ecperf-1_1-fr-spec.pdf, April 2002

[22] P. Felber, P. Narasimhan, "Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications", Distributed Objects And Applications 2002, Oct 2002

[23] http://www.javagroups.com