A multi–layer architecture for high available *Enterprise JavaBeans**

Marcia Pasin[1] Taisy Silva Weber[1] Michel Riveill[2]

[1] Universidade Federal do Rio Grande do Sul
Instituto de Informática
Av. Bento Gonçalves, 9500 – Campus do Vale
CEP 91501–970 Caixa Postal 15064
Porto Alegre – RS – Brazil

[2] Université de Nice – Sophia Antipolis
Ecole Supérieure en Sciences Informatiques/
930 route des Colles – BP 145
06903 Sophia Antipolis Cedex — France

Abstract

EJB (Enterprise JavaBeans) *spec* does not describe high availability as one of its properties. If the application server fails, the service remains unavailable while it recovers. Some EJB server vendors claim to provide this desirable property implementing server replicas through centralized protocols. Unfortunately, these protocols could lead to an unavailable service if the coordinator server crashes. We are presenting a new architecture aiming high available EJB servers based on distributed concepts. Our replicas are modeled as state machines synchronized by group communication primitives. We achieve high availability to EJB application servers running *stateful* and *stateless session beans*.

Keywords – high availability, replication, group communication, Enterprise JavaBeans.

1 Introduction

The EJB (Enterprise JavaBeans) *spec* [9] describes a component system architecture based on a multi–tier framework. This architecture mainly comprises the client, the EJB application server and the database server. Programmers build EJB applications in a transparent and adaptable way. They *develop* their applications (the *enterprise service* or functional requirements) without worry with non–functional properties (as persistency, security, transaction management and scalability), and *deploy* their applications using a *deployment tool*. The deployment tool automatically includes non–functional properties.

EJB applications can be *transaction–aware*. A transaction could aggregate operations on multiple objects. It is a sequence of methods encapsulated by a *begin operation* and a *commit operation*. If at least one method inside a given transaction cannot be completed due to a system failure, all updates generated by this transaction will be undone. The transaction execution will be aborted and an exception will be thrown.

Objects from the EJB architecture are called *bean*s. EJB application servers host and manage *beans* through a component called *container*. Clients could request operations (read or update a *state*) to any *bean* just running an EJB application. The main kinds of *beans* are *session bean* and *entity beans*.

A *session bean* just retains state during a client-server session. When the session terminates, its state is lost. The *session bean state* is volatile and unique to each client (one individual thread is used to each client). *Session beans* are not sharable. Otherwise, *entity beans* maintain their persistent states through a database connection and could be sharable between different clients. This work provides high availability to *session beans*. High available persistent states will be treated in a future work.

* This work was supported by the French Ministry of Research through project RNTL Arcad and by the Brazilian Ministry of Education through CNPq contracts 142808/98–9 and 200594/00–1.

The EJB *spec* assures a safe state to *beans* despite failures. However it does not guarantee high availability. If the EJB server goes down, its service will be unavailable: the state of a *session bean* is volatile and will be lost. High availability requires replication and synchronization protocols. Group communication has proven to be a convenient abstraction for implementing distributed systems requirements, particularly for synchronizing replicas [4]. Implementing group communication concepts in transactional systems, as the EJB architecture, is quite different [11].

We are extending the EJB *spec* to allow building replicated *bean*s without changing the way that users develop and deploy their EJB applications. Users do not have to worry with replica management. They can build their EJB applications in the usual (non–replicated) way and, optionally, could specify which service will be replicated using the *deployment tool*. Our replicated service exploits group communication to minimize communication costs.

High availability is archived through a multi–layer architecture that comprises a group communication layer, a replication layer and a conventional EJB layer. An open EJB application server implementation [5] provides the EJB layer. The replication layer provides consistency to the replicated EJB application servers. Consistency is achieved through a synchronization protocol following the *state machine approach* [7]. A group communication system [2] provides suitable services to the replication layer. These services assure group membership, failure detection and reliable multicast primitives even in presence of failures.

The paper follows presenting the section 2 with high available *session beans* requirements. The section 3 describes the distributed system model with the state machine approach. The section 4 describes our system design. The section 5 describes the replicated system implementation. The section 6 describes our preliminary results. The section 7 presents some related works. The paper ends with concluding remarks.

2 High available *session beans*

There are two kinds of *session beans*: *stateful* and *stateless*. *Stateful beans* retain state on behalf of an individual client. *Stateless bean*s are not aware of any client history. Recovery a server with *stateless session bean*s is straightforward because there is no information about the *bean* state stored in the server side. It requires the client reissue the request to another EJB server – it means *failover*.

Achieving high availability to *stateful session beans* requires replicate the *bean state* held during a *bean* method execution. Here we multicast this state to a replication group using group communication concepts. Implementing *failover* and maintaining the *exactly–once semantic* despite failures are also required. In the exactly–once semantic, the client makes a request and is guaranteed by the reply that the request has been executed.

3 Distributed system model

Our distributed system model is composed by clients and servers (EJB application servers). Clients could request read and update operations to objects (*beans*) placed in servers. Servers compose a *replication group,* and follow the *state machine approach* [7]. Here the state machine approach uses group communication primitives to achieve consistency to all group members. Initially the client request is executed locally at one server (the primary) and then the new state is multicast to all group members.

The *state machine approach* defines a consistent behavior to a collection of distributed objects. These objects run identical state machines (here, the EJB application servers) and perform the same sequence of operations, producing the same sequence of outputs and transitioning through the same sequence of states.

The behavior of the replicated objects is indistinguishable from that of a single high available object. Each client knows only the primary address and issues the request directly to it. The primary executes the service locally and then forwards its new state to the backups using a multicast primitive [4]. This primitive assures that all objects in the system (the *object group* – or, in our case the replication group) receive messages (state updates) in the same order, so that all objects perform the same sequence of state updates**.**

When a client wants to use high available service, it first contacts a *name service* to receive a unique *primary server address*. The other servers in the replication group work as *backups*. Then the client issues requests to this unique primary server. A request could be a read or an update operation to an object state (or *bean* state).

A server could work as *primary server* to a client and as *backup* to another because the *name service* could provide different primary addresses to different clients. This approach allows having multiple primaries simultaneously and avoids bottlenecks, a typical drawback of primary–backup approach.

Faulty backups are transparent to clients. Faulty primaries are not transparent to its clients and require the clients selecting a new primary and reissue the last request. The new primary discards the already done operations related to the issued request. Clients detect faulty primaries using *timeouts*.

We assume an asynchronous distributed system where neither message delays nor computing speeds can be perfectly bounded. Messages between different servers cannot be lost because an underneath group communication system provides reliable messages as well group membership. Group members can be assumed as *fault–suspected,* because there is no way to distingue between overloaded and faulty members. The group communication system also provides a *failure detector* to remove fault–suspected state machines from the object group. A fault–suspected state machine could be (repaired and) restarted and rejoins the object group by means of *state transfer* from surviving members.

4 System design through a multi–layer architecture

The high available service is provided by a multi–layer architecture (figure 1). Each server (primary or backup) has a group communication layer, a replication layer and an application layer. These layers are previously used by *Amir et al.* [1]. Here the application layer is played by the JOnAS EJB server [5]. The JavaGroups communication system [2] provides support to the group communication layer. These systems are open source and implemented using Java language. As both systems are implemented as component abstractions, its integration was straightforward.

4.1 Application server layer

The application server layer follows the EJB *spec* and manages the *beans* through a component called *container*. The container provides all non–functional properties (as persistency, security, transaction management and scalability). The *bean components* [9] are the *remote interface*, the *home interface,* the *bean class* and the *deployment descriptor*. These components are developed by the programmer. The *remote interface* is the client view of the *bean*. It contains the signatures of all *bean methods* (functional properties). The *home interface* contains the signatures of all methods for the bean life cycle (creation, suppression) and for instances retrieval (finding one or several *beans*) used by the client. The *bean class* implements the functional properties, and all methods allowing the *bean* to be managed in the EJB application server. The *deployment descriptor* contains the *bean properties* that may be edited at *configuration time*. The *beans* properties could identify, for example, if a bean is *stateful* or *stateless*.

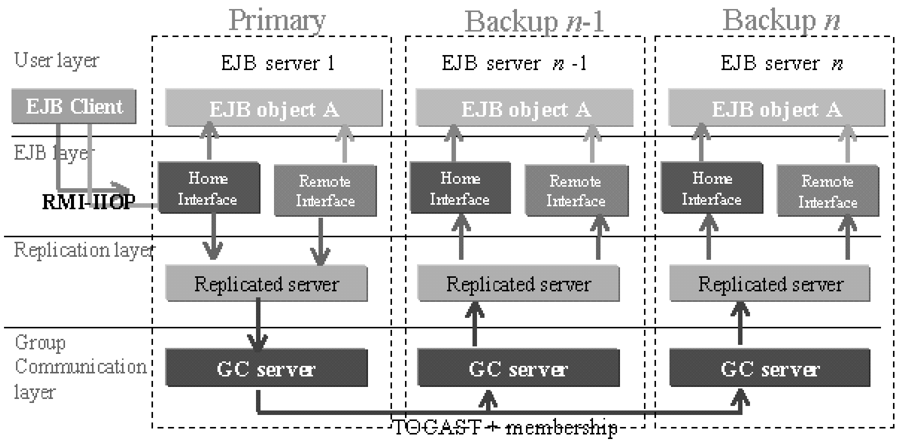

Figure 1 – The multi–layer architecture

4.2 Replication layer

The replication layer implements an interface between the EJB application server and the group communication layers. Whenever a local server receives a request from a client to execute an update method in a *stateful bean*, and after executing this method, it signals the replication layer. The replication layer generates a message containing the new bean state and forwards it to the group communication layer, which multicast this message to the group members using the TOCAST (*total–order multicast*) primitive [4]. Each group member receives the message in its local group communication layer and delivers it to the replication layer. By delivering the same set of messages using the TOCAST primitive, the algorithm guarantees that all servers hold the same bean state to a given client. If one application server in the group fails, clients are guaranteed access to the same state through the backups.

To minimize communication costs, only new states are multicast to backups. The replication layer could be able to distinguish updates from simple reads to *bean*s, either identifying the executed method or comparing the new signaling state with the stored previous one. Examining the *bean code* in pre–compiling time, it is possible to identify the methods that could potentially chance the *bean state*.

Update propagation and transactional attributes

Update propagation is an important drawback of distributed replicated systems because it can decide the system performance. There are two different strategies to propagate updates: *deferred update* and *immediate update*. The deferred update strategy processes all transactions locally at one server (the primary one) and forwards the last final result to the others at commit time. Immediate update synchronizes every update across all servers.

Although deferred update has advantages over immediate update, as reducing the number of distributed states, it requires executing the service from the last committed transaction when a failure happens. Both strategies could be implemented to analyze their efficiency in a comparative study.

The EJB *spec* supports both *transaction–aware* and *non–transaction–aware bean*s. *Non–transaction–aware beans* requires using immediate update whenever. *Transaction–aware beans* require analyzing the transactional attribute available in the *bean deployment descriptor*. The `NotSupported` and `Never` transactional attributes ever require immediate update. `Required, RequiresNew, Mandatory` and `Supports` attributes support both immediate update and deferred update. The `RequiresNew` attribute supports both immediate and deferred update. Both execution effects are the same because it requires a new transaction per each remote method invocation.

4.3 Group communication layer

The group communication layer is implemented by the JavaGroups group communication system [2]. The group communication system uses available protocol layers responsible for achieving group membership, total ordering of messages, and other properties as building blocks with which the high–level state machine replication semantics are obtained.

5 Implementation

This section describes the implementation of the replication layer. The implementation addresses the server–side and client–side, and both kinds of *session beans* (*stateless* and *stateful*). At the client–side, *failover* enables selecting a new server to provide access to *stateless* and *stateful session beans*. At the server–side, the replication layer overloads a non–replicated JOnAS server to multicast distributed state from *stateful session beans*.

5.1 Failover

In the EJB *spec*, before a new *bean instance* is created, the client executes a *lookup operation* using a JNDI (Java Naming and Directory Interface) server to contacts the name service. The JNDI server provides an application server address that can be used by the client to create a *bean instance* in the EJB server using the *home interface*. Then the client could access the *bean methods* using the *remote interface*. To finalize, the client destroys the *bean instance*.

Implementing a high available service requires locating the *bean* in a backup when a failure occurs – it means *failover*. The *lookup operation* is reexecuted by the client to locate another *bean copy* in the backup server. Then a *new bean instance* is created in this backup. *Stateful beans* require state update from the last state received by the multicast message.

We implement failover redirecting faulty requests in the client–side. *Hooks* could be transparently included in the client–side by a pre–compiler. A *hook* is a concept borrowed from the EMACS editor, which allows executing arbitrary commands before performing some operation. We change Java exceptions to both *lookup* and *create bean* operations by new operations to allow failover. This service is transparent to the programmer.

5.2 Replicated servers

As we saw, our replicated server implements a new replication layer between JavaGroups and JOnAS. This layer is responsible to manage the replication group. This is done by overloading classes of the non–replicated JOnAS server to include the TOCAST primitive and introducing *hooks* in the *bean class*. Replicated servers join the group and use this primitive to setting the distributed state to *stateful session beans*. *Stateful* and *stateless session beans* are distinguished by means of code inspection using the *bean deployment descriptor*.

Code inspection is also required to distinguish read from update methods. It could be done observing by introspection the method signature in the *bean remote interface*. A non–void class could be interpreted as an update operation. Classes with void as return parameter could be assumed as a read operation. Optionally, the user could specify which methods will be replicated using the deployment tool.

To the programmer, high available *beans* are built in a transparent way, by using a pre–compiler. This pre–compiler changes the *bean* code to include *hooks* that enable the high available service takes place. This approach is also used by the GenIC compiler from JOnAS [5], which includes non–functional properties using the bean interfaces.

Distributed state

A *distributed state* contains information about active *stateful bean*s in the replication group. Each distributed state holds information about the last *bean state*, the client who is using this *bean* and the primary server who is proving access to the *bean methods*.

The EJB *spec* enables one or more *create methods* to a *bean*. These create methods can differ by the parameters sent to each method. To implement the replicated service, we could overload all create methods with a new parameter called *failover*. Having the parameter failover as true value means that the new *bean instance* should be update with the value stored in the last distributed state, because the *bean* execution was failover. Once the service is done and the client disconnects, a particular *bean instance* is destroyed. Then all group members remove all distributed states from this client–server session.

Remote invocation in the presence of failures

The current EJB *spec* supports transactional services over non–reliable infrastructures. It implements the *best–effort execution semantic*. In the *best–effort semantic*, the client sends the message, and the client and infrastructure do not attempt retransmissions. High available services require a more sophisticated approach despite failures. *Five different classes of failures* [10] can occur in *remote procedure call* systems. These same failures could also occur in *remote method invocation* systems. They are required be treated by our replicated system because the EJB architecture is based on *remote method invocation.*

The first one, *the client is unable to locate the server*, here is treated providing failover. *Stateless session beans* require the *at–least–once semantic*: the client makes a request and retries the request until it receives the response. If a failure occurs, the client is enable try all group member candidates, which are specified in a list by the system administrator. If none is available, in the worst case, the client finally throws an exception. Duplicate message processing by the client is not a problem, because the *bean state* is not retained in the server–side.

Stateful session beans require the *at–most–once* or the *exactly-once semantics*. In the *exactly–once semantic*, the client makes a request and is guaranteed by the reply that the request has been executed. An approach to assure *exactly-once semantic* in replicated transactional systems is showed in *Frolund et al* [3]. The *at–most–once* builds on the *at–least–once* scenario. The client retries the request until it gets a response. A mechanism like *message identifiers* allows the server to suppress any duplicate requests, insuring the request is not executed multiple times. We follow this approach.

The second one, *the request message to the client to the server is lost*, is treated using *timeout* in the client–side, following the EJB *spec*. The third one, *the response from the server to the client is lost*, is treated using *timeout* in the client–side. In this case the client reissues the request to the same server. The message identifiers allow the server to suppress any duplicate requests. The forth one, *the server crashes after receiving a request*, we also treat using message identifiers to assure the *at-most-once semantic*. Finally, the fifth one, *the client crashes after sending a request*, could potentially generate *orphan bean instances*. The EJB *spec* treats *orphan bean instances* through the container. It periodically removes all *bean instances* from the server memory if they are not current used. We need extend this approach to remove all correspondent distributed state established to that client.

6 Preliminary results

Initially our implementing effort was focused in changing the GenIC compiler provided by the JOnAS EJB server. The programmer builds its *beans* and uses the GenIC compiler to mainly generate the container classes. However, the GenIC acts over the *home* and the *remote interface*. We need acts over the *bean class* that actually holds the *bean state*. So we need include *hooks* in the *bean class* not in the *home* and *remote interface* as GenIC does.

Presently, we are testing our implementation for automatic re–routing of clients' requests (client side) and the approach to establishing the distributed state to *stateful session beans* (server side). A performance study will take into account different replica number and failure scenarios. We expect that replication does not considerably disturbs the application response time, when compared with non–replicated application servers, by allowing requests to be handled by several nodes rather than one besides eliminating a single point–of–failure.

7 Related works

Some EJB application server providers implement high availability to *session bean*s. They mainly use the *in–memory replication technique*. In–memory replication has two different variations. The first approach writes information to a centralized server (all servers in the cluster use the same centralized server). In the second approach, each server chooses an arbitrary backup. The BEA WebLogic 6.0 uses this last approach and the HP Bluestone Total–e–server uses the first one. However, these and other solutions [6] are implemented using *cluster concepts*. We are using a more non–restrictive model. Our system runs over a local network and could support asynchronous communication.

There is no accurate information available about how state propagation is applied in other high available EJB servers. The majority of the implementations are proprietary solutions. However, these implementations seem forwarding distributed states using centralized approaches as *n–phase commit protocols*. Two–phase commit protocols are mainly used to assure consistent states in distributed transactional systems. In these protocols, a process called *coordinator* applies distributed state to all replicas. They could accept human intervention to solve abnormal behavior because they may block if the coordinator crashes. Three–phase commit protocols provide higher availability, but they assume a restricted fail–stop model. Group communication abstractions allow non–blocking protocols and support a wider failure model. It has proven to be a convenient abstraction to improve distributed application availability.

Some related work is showed in *de Sousa* [8]. That work also differs from our work mainly because we use a group communication system to provide reliable communication through EJB servers. It assures that our *multicast protocols* are already extensively validate avoiding implementing our own protocols and minimizing programming errors. The solution described in *de Sousa* [8] does not mention several aspects as enabling multiple primaries simultaneously to avoid bottlenecks. Replication (and high availability – our aim) is not the main aim of that work: its main aim is explore the reflexive features of the EJB architecture.

Our work also differs from CORBA solutions implemented around both the `CosNaming` service and a centralized JNDI tree. The Sybase Enterprise Application Server uses this approach in a cluster. Name servers house the centralized JNDI tree for the cluster and keep track of which servers are on–line. If there is a failure in between EJB method invocation, the CORBA stub retrieves another home or remote interface from an alternate server returned from the name server. The name server is the drawback of this solution because it needs to remain active in one node at least. If all nodes that hold JNDI servers instances are down, the system will be unavailable. Finally, centralized JNDI tree clusters suffer from an increased time to convergence (the time the cluster takes to know all its server active instances) as the cluster grows in size. That is, scaling requires adding more name servers. Our solution does not require time to convergence because we are using the independent JNDI tree approach.

8 Concluding remarks

Transactional systems could benefit from group communication to achieve availability [11]. Nevertheless group communication introduces the membership management overhead that can be reduced using suitable design decisions. Here, these decisions include excluding clients from the replication group (to avoid frequent membership changing) and propagating to backups only updates or committed results. It allows eliminating additional communication rounds and improves performance. We also expecting reducing the overhead generated by the replication protocol enabling several servers working as primary servers.

References

[1] Amir, Y.; Dolev, D.; Melliar–Smith, P. M.; Moser, L. E. *Robust and efficient replication using group communication.* Technical Report CS9420, Institute of Computer Science, Hebrew University of Jerusalem, Nov. 1994.
[2] Ban, B. *JavaGroups user's guide*. Department of Computer Science, Cornell University. August 1999. 73p.
[3] Frolund, S. and Guerraoui, R.; *Implementing e–transactions with asynchronous replication*. Proceedings of the International Conference on Dependable Systems and Networks 2000, New York, IEEE, June 2000.
[4] Guerraoui, R.; Schiper, A. *Fault tolerance by replication in distributed systems*. In Proc. Conference on Reliable Software Technologies (invited paper), p. 38–57. Springer Verlag, LNCS 1088, June 1996.
[5] *JOnAS – Java Open Application Server* – http://www.objectweb.org/~jonas
[6] Kang, Abraham. *J2EE clustering, Part 1. Clustering technology is crucial to good Website design; do you know the basics?* JavaWorld. Feb. 2001. http://www.javaworld.com/javaworld/jw–02–2001/jw–0223–extremescale.html
[7] Schneider, F. B. *Replication management using the state machine approach*. In: Mullender, Sape (Ed.). Distributed Systems. 2. ed., New York: ACM Press, 1993. p. 169–198.
[8] de Sousa, C.V.P.B.; Maziero, Carlos Alberto. *Uma abordagem reflexiva para replicação de componentes servidores da plataforma Java para corporações*. WTF 2000. Curitiba – PR, pp.106–111.
[9] Sun Microsystems, Inc. *EJB specification* 2.0.
[10] Tanembaum. A.S. *Communication in distributed systems*. In: Modern Operating Systems. Prentice Hall, New Jersey 1992. pp.395–462.
[11] Wiesmann, M.; Pedone, F.; Schiper, A.; Kemme, B. and Alonso, G. *Understanding replication in databases and distributed system*s. Proc. ICDCS 2000, pp.264–274, Taipei, Taiwan, R.O.C., April 2000.