A Replication Toolkit for J2EE Application Servers

Huaigu Wu Bettina Kemme
McGill University
hwu19, kemme@cs.mcgill.ca

Alberto Bartoli
Università di Trieste
bartolia@univ.trieste.it

Simon Patarin
Università di Bologna
patarin@cs.unibo.it

1 Introduction

Web service technology allows organizations to provide programmatic interfaces to the services they export. In most cases, these services are implemented with a multi-tier architecture consisting of a client external to the organization, a middle-tier and a back-end tier. The middle-tier typically uses the infrastructure of an application server (AS) whereas the back-end tier consists of a database system. In this demo we present our work for enhancing current AS technology with flexible and transparent failure management. We consider multi-tiered services based on the J2EE architecture and replicate the middle-tier for fault-tolerance. The novelty of our contribution consists in the guarantees we provide with respect to failures. Our work has been carried out as part of the ADAPT project (Middleware Technologies for Adaptive and Composable Distributed Components), whose results are summarized in [3].

In a multi-tier architecture execution across AS and the database is usually performed in the context of transactions. Transactions provide durability for the data stored in the database, atomicity (all-or-nothing) and isolation of concurrent transactions on both the AS and the database. Although all persistent data is stored in the database system, the AS can maintain data that can exist across several client requests in order to efficiently execute the business logic. A typical example is session information. We say that such application servers are *stateful*.

Our proposal provides fault-tolerance for stateful AS by making use of replication: an AS runs with several instances (replicas), and if one replica crashes, the others will take over its tasks without interrupting the service offered to clients. We enforce the following correctness requirement, that we state informally: the service as a whole behaves as its non-replicated counterpart. This notion of correctness encompasses both the AS layer and the database server layer: as long as the client does not crash, a transaction must be executed exactly-once (if the client crashes the semantics is at-most-once) [6]. Note that this correctness requirement implies that the set of ASs must maintain *state consistency* internally. That is, if a transaction commits at the database, each available AS replica must have the state changes performed by this transaction. If the transaction aborts at the database, all available AS replicas must know about the abort and be able to return to the previous state.

It is simple to realize that fulfilling our correctness requirement necessarily involves some form of coordination amongst ASs. Since ASs are stateful, any state change performed on a AS must be known by the other replicas. Otherwise the crash of an AS would lead to data loss, which might lead to unexpected results for the clients and violate the correctness requirement. Transaction management must be carefully coordinated as well. On the one hand, the crash of an AS might leave the database in an unexpected state because transaction execution spans over both the AS and the database. On the other hand, the set of ASs must maintain a consistent notion of which transactions have been committed at the database layer and which ones have been aborted (the state consistency issue mentioned above).

Many of the recently developed application server products follow the J2EE specification [8]. Although many of them use replication to tolerate server failures, e.g., WebLogic [5], WebSphere [9], and JBoss [7], we are not aware that any of them guarantees both exactly-once transactions and state consistency in all crash cases for stateful AS. This motivated us to develop a suite of replication algorithms providing these guarantees in [11, 10]. We implemented these algorithms as an open-source toolkit based on the JBoss application server [3].

Although good performance and strong consistency guarantees in the presence of failures are rather conflicting requirements, we obtained a satisfactory trade-off between the two. We run the ECperf benchmark and found that our system compares favorably against the JBoss clustering solution for high availability. This is a significant result, having considered that JBoss clustering provides neither exactly-once execution nor state consistency (see [11] for details). In this demonstration, however, we will focus on failure management and show how our toolkit handles both failure and recovery of AS replicas for various failure scenarios. In particular, we will show that the replication algorithm behaves as expected irrespective of when an

AS failure occurs, i.e., irrespective of which portion of the replication algorithm is being executed when an AS crashes. Giving a demonstration of this kind is not a trivial task because failover happens behind the scenes. Hence, we have developed a graphical user interface that illustrates the flow of execution and the actions upon failure and recovery at the various server replicas and allows forcing AS failures at selected points of the replication algorithm.

2 Replication of Stateful J2EE Server

J2EE architecture In the J2EE architecture the business logic is implemented by means of objects called *Enterprise JavaBeans (EJB)* and hosted by the AS. Two kinds of EJB can maintain state. *Stateful session beans* (SFSB) maintain internal state for the lifetime of a caller session. *Entity beans* (EB) represent persistent data in the database. State changes take place in the context of transactions. State changes in the database are persistent while changes on AS components remain volatile. If a transaction aborts, any state changes performed on the database are undone by the database system whereas state changes on AS components ramain. Some ASs, however, allow specifying rollback methods for SFSBs. In this case, the AS will invoke these methods automatically if the transaction aborts.

Transactions can be demarcated in several different ways. The very basic execution pattern provides all-or-nothing execution for each client request: each client request initiates a single transaction T and accesses only one database, and the entire request execution on the AS and the database is performed within T. We presented a replication algorithm for this pattern in [11]. More complex execution patterns are supported: the client may begin and terminate a transaction explicitly, in which case more than one client request may be involved in a transaction; a client request may initiate more than one transaction; a transaction could also access more than one database, in which case a 2-phase-commit protocol among the participating databases and AS is needed. We present replication algorithms for these advanced patterns in [10].

Due to the time restrictions, our demonstration will focus on the basic execution pattern and assume that the AS supports rollback methods for SFSBs. We will also consider only transaction aborts caused by AS crashes and ignore aborts caused by application semantics or database errors. We refer to the system presented in this demo as the ADAPT-SIB toolkit.

Assumptions The replication algorithm considered here is based on the following assumptions. Communication is asynchronous and reliable. Individual components within a server do not fail. An AS may fail entirely by crashing. Databases and clients do not crash. Clients and components are single-threaded that block when waiting for the response of a request. Execution does not need to be deterministic, though. When an AS replica crashes, the database will automatically abort all active transactions (not yet committed) that have been submitted by this replica. The algorithm is built above a group communication system (GCS) providing uniform reliable multicast delivery and membership management (failure detection and notification).

We remark that our more complex replication solution [2] handles network partitions to the client, allows clients to fail, and is able to work with a replicated database (that provides fault-tolerance of the database tier).

Replication Algorithm The replication algorithm can only be outlined here due to space constraints. Full details can be found in [11]. We use a primary/backup model where state changes to SFSBs are propagated to the backups. Changes to EBs are not propagated (they are always written to the database at commit time and are read from the database at failover).

The client attaches a unique identifier to each request and sends the request to the current primary. If the client receives a failure exception instead of a normal response, it resends the very same request to the new primary (after having determined who is the new primary).

The primary executes a client request within a transaction T. At commit time, it multicasts a *committing* message containing the request/response pair and a description of all EJBs changed by T (the full state changes for SFBSs, the identifiers for EBs). Moreover, it stores the request identifier into the database as part of the transaction T. When the primary receives confirmation from all available backups that they have received the multicast (i.e. uniform reliable message delivery), it commits T, returns the response to the user, and multicasts a *committed* message.

When the primary crashes, the backups are informed by the GCS. One backup becomes the new primary and performs failover. It applies the state changes for SFSBs included in each *committing* message for which it has received the *committed* message (this can be done already during normal processing to speed up failover), retrieves the state of the EBs from the database and constructs the set of all request/response pairs. For *committing* messages for which no corresponding *committed* message was received, the new primary can determine whether the corresponding transaction has committed at the database by checking whether the request id exists in the database or not. If yes, the database transaction successfully committed and the new primary performs the same actions as if the *committed* message had been received. Otherwise, it means that the database transaction was aborted upon crash of the old primary. Hence, the new primary ignores the *committing* message. When the new primary receives a client request,

it checks whether the request id identifies one of the existing request/response pairs, and if yes, returns immediately the response. Otherwise it executes the request as usual.

To show that both state consistency and exactly once execution are guaranteed, we discuss the most significant crash cases, i.e., when the primary crashes before returning a response to the client. There are four cases to consider. (1) The primary crashes in the middle of execution of the request. Hence, the database aborts the corresponding transaction T upon the crash, and the new primary has not received any message. The request resubmitted by the client is simply executed. (2) The primary crashes after sending the committing message, but before the database commits T. Hence, the database aborts T upon the crash, the new primary checks at failover in the database for the request id, does not find it, and hence, does not apply the changes in the committing message. The resubmitted request is simply executed. (3) The primary crashes after committing T but before sending the committed message. The new primary checks at failover in the database, finds the request id, and hence applies the changes in the committing message and stores the corresponding request/response pair. When the client resubmits the request, the new primary immediately returns the response. (4) The primary crashes after sending the committed message. The new primary applies the changes in the committing message and stores the request/response pair. When the client resubmits the request, the new primary immediately returns the response.

Recovery Algorithm Our system is able to integrate previously failed or completely new replicas into the system. In this case the joining replica has to receive first the current state and then will become a backup. To this end, one of the existing replicas, referred to as the *peer*, must send its current state to the joining replica, which occurs as follows.

The new replica joins the group maintained by the GCS. The existing group members are informed via a group change message, and the new replica receives all multicasts issued by the current primary from then on. The peer node is decided on by means of a simple agreement protocol. The peer sends a *recovery* message to the joining replica via point-to-point communication. This message basically contains the committing and committed messages that are needed to reconstruct the state in case the new replica has to perform failover. The joining replica might receive committing and committed messages from the primary before receiving the recovery message from the peer. It puts those messages in a queue Q. Upon receiving the recovery message, the joining replica processes the included committing and committed messages, and then processes the messages in Q. The *recovery* message might contain messages also enqueued in Q. These messages are removed from Q before the backup algorithm can start processing messages from Q.

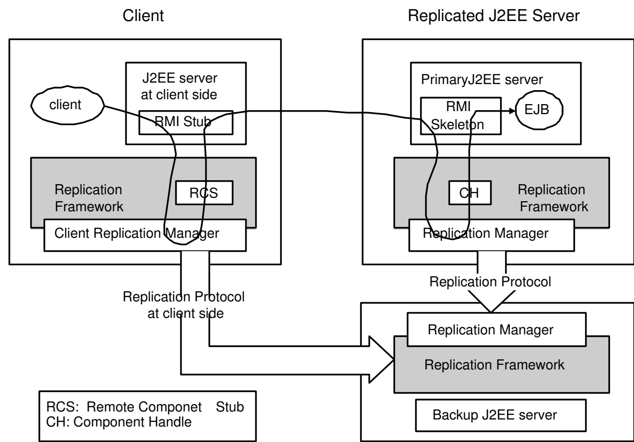

Figure 1. The ADAPT framework separates the replication algorithm from the J2EE server.

3 Implementation

The implementation is based on a J2EE replication framework that we designed and implemented [1]. The framework eases the prototyping of replication algorithms at the application server layer, on the grounds that to design and evaluate a replication algorithm for J2EE (or any practical component architecture) requires a substantial investment in development. With this approach, the task of developing a replication algorithm is factored into two portions: (i) the framework itself, which handles all the detailed interactions with the underlying application server code, and (ii) the specific replication algorithm. When a component is invoked, the framework transfers control to the replication algorithm that may perform any actions, such as setting component state or communicating with other replicas, before continuing the invocation. Through the framework, the replication algorithm sees an highly abstracted view of the component, the invocation, and the other elements of the server. The developer codes a replication algorithm by implementing a simple API, and deploys it by moving the classes into the server's deployment directory. Fig. 1 shows how the framework separates the J2EE server from the replication algorithm implemented within the replication managers. The framework has been implemented for the JBoss open-source application server and the Axis SOAP engine (in case of web services). We have developed several replication algorithms on top of this framework, including the ADAPT-SIB toolkit that will be shown in this demo. We use Spread (`http://www.spread.org`) as GCS, as it provides the required functionality efficiently. We access Spread through JBora, a thin middleware layer that we have developed (see [4] for its features).

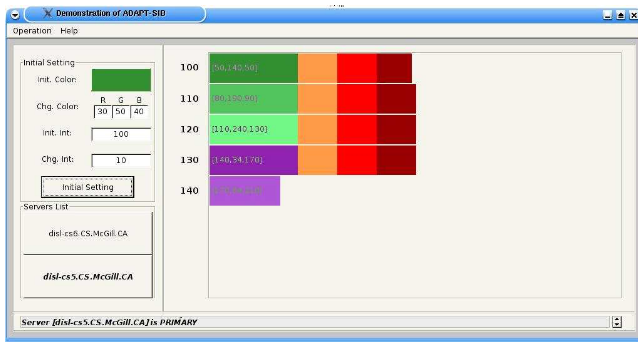

Figure 2. Demonstration user Interface

4 Demonstration

The demonstration will show how our ADAPT-SIB replication algorithm running over the ADAPT framework and JBoss, tolerates different crash scenarios and recovery of the application server. Two replicas of JBoss 3.2.3 will be deployed on two Linux machines which can communicate through JBora/Spread. One will act as the primary, and the other as the backup. The backend database is Postgres SQL 7.0, which is running in a Linux machine. We use a java program with GUI to simulate the client, which is also running in a Linux machine. Figure 2 shows the client interface. For simplicity, the database, the client and the backup will all run on the same machine.

Our example application consists of a single SFSB. This bean has a single attribute that stores color information. The database maintains a table with one integer attribute and a single record. The SFSB provides two methods. An initialization method takes as input an initial color, a parameter that indicates how subsequent requests should change the color, and a parameter that indicates by how much the integer value stored in the database should be increased by each request. The other method, that we call *doStep()*, has no input parameters. Executing this method changes the color of the SFSB and increases the value of the record stored in the database according to the parameters given by the initialization. It returns the newest color and provides the latest value of the record in the database.

The client application will invoke the initialization method at the beginning of the demo and then it will invoke *doStep()* repeatedly. The client application, thus, will gradually change the color attribute stored in the SFSB and increase the integer value stored in the record in the database.

The client application has a graphical interface that shows execution of each method invocation as a progress bar. The replication algorithm is such that each execution can be divided into four periods: normal execution, committing message sent, transaction commit, and committed message sent. The graphical interface draws the progress bar with different colors, one for each period. In particular, normal execution is shown with the color stored in the SFSB, whereas the three other periods are always shown in the bar as orange, red, and brown, respectively (for the purpose of this demonstration, the AS has been instrumented to send a message to the client application at the beginning of each period). Furthermore, the value stored in the database record will be shown besides the bar. The graphical interface keeps the result of the last eight executions.

It will be possible to force the crash of the primary by double clicking the progress bar. To show all possible failure scenarios, we enforce an artificial delay within each period. The demonstration will show that color and data keeps changing as in the failure-free case irrespective of when we crash the primary, i.e., during which period. This is a subtle but key property of our proposal. In a sense, it is this property what makes the difference between “reliability guarantees” and “cross-the-fingers” reliability.

The demonstration also shows recovery. Each server is represented by a button. When the current primary crashes, the backup becomes the new primary. Afterwards, we can click the button of the crashed server to recover it. After it is recovered it is a backup. If we crash the current primary, the recovered server will become the primary again.

References

[1] O. Babaoglu, A. Bartoli, V. Maverick, S. Patarin, J. Vuckovic, and H. Wu. A Framework for Prototyping J2EE Replication Algorithms. In *Int. Symp. on Distrib. Objects and Applications (DOA)*, 2004.

[2] A. Bartoli, R. Jiménez-Peris, B. Kemme, S. Patarin, M. Patiño-Matinez, F. Perez, M. Prica, J. Salas, J. Vuckovic, H. Wu, and S. Wu. Bs middleware platform, http://adapt.ls.fi.upm.es/Downloads.htm.

[3] A. Bartoli, R. Jiménez-Peris, B. Kemme, C. Pautasso, S. Patarin, S. Wheater, and S. Woodman. The adapt framework for adaptable and composable web services. In *IEEE Distributed System ONLINE*, 2005.

[4] A. Bartoli, M. Prica, and E. Antoniutti Di Muro. A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container. In *Concurrency and Computation: Practice and Experience (to appear)*.

[5] BEA Systems Inc. *BEA WebLogic Server, release 7.0: Programming WebLogic Enterprise JavaBeans*, 2002.

[6] S. Frølund and R. Guerraoui. X-ability: a theory of replication. In *Symp. on Princ. of Distrib. Comp. (PODC)*, 2000.

[7] The JBoss Group. *JBoss Clustering*, 2002.

[8] SUN Microsystems Inc. *JAVA 2 Platform Enterprise Edition Specification, V1.3*.

[9] H. Wang and M. Bransford. *Server Clusters For High Availability in WebSphere Application Server Network Deployment Edition 5.0*. Software Group, IBM Corporation, release 5.0 edition, April 2003.

[10] H. Wu and B. Kemme. Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In *Symp. on Reliable Distributed Systems (SRDS)*, 2005.

[11] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Int. Symp. on Distributed Objects and Applications (DOA)*, 2004.