ADAPT
IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

BS Middleware Platform

Deliverable Identifier:	D13
Delivery Date:	3/1/2005
Classification:	Public Circulation
Authors:	A. Bartoli, R. Jiménez-Peris, B. Kemme, S. Patarin, M. Patiño-Martínez, F. Perez, M. Prica, J. Salas, J. Vuckovic, H. Wu, S. Wu
Document version:	Final, March 1, 2005
Contract Start Date:	1 September 2002
Duration:	36 months
Project Coordinator:	Universidad Politécnica de Madrid (Spain)
Partners:	Università di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Università di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd.(UK)

Project funded by the European Commission under the Information Society Technologies Programme of the 5th Framework (1998-2002)

Contents

Dependencies with other Deliverables

This deliverable describes the Basic Services (BS) Middleware. This integrated platform is made of several components that were detailed in previous deliverables. The ADAPT framework for application server replication, Enterprise Java Bean (EJB) and Web Service (WS) replication algorithms, and database replication have been presented in deliverable D3 (due on month 24). The reliable communication layer used by the various replication algorithms has been presented in deliverable D2 (due on month 18). While evaluations of these individual components have been carried out in their respective deliverable, the evaluation of the BS middleware as a whole will be part of the Evaluation deliverable (D19 due on month 36), using the Demonstrator basic service (D18 due on month 30).

1 Introduction

Tools and algorithms developed by the ADAPT project allow to perform replication at each of the three tiers of the J2EE (Java2 Enterprise Edition) architecture: the web tier with JMiramare [BPA05] on top of the framework [BBM$^+$04], the business tier with the Eager Replication algorithm [WKM04] also on top of the framework, and both Postgres-R(SI) [WK05] and Middle-R [JPKA02, MJPK04, LKPJ05] for the database tier. In this deliverable we describe the approach we have followed to integrate these various elements into a single unified middleware.

In the three tier architecture, two interfaces are to be considered: between the web tier and the business tier and between the business tier and the database tier. These interfaces correspond to interactions that may exist among the tiers. In particular, this means that there exist no direct interactions between the web tier and the database tier: they necessarily go through the business tier. At each of these interfaces two different integration approaches may be followed:

- loose coupling: the two contiguous tiers are independent from each other, they can fail independently and they are replicated independently. This approach gives more flexibility and adaptability but is much more complex to implement correctly given the number and the variety of failure cases.

- tight coupling: the two contiguous tiers are intimately bound to each other, so that they cannot fail independently. When it is possible to do so, this approach is easy to implement, but necessarily increases the number of failure cases as a failure in one tier implies to make the other tier fail even if it could have continued to work.

We have chosen to use the tight coupling approach to integrate the web and the business tier, while the business and the database tier remain loosely coupled. We will motivate and describe these choices in the rest of this deliverable (respectively in Section 2 and Section 3).

Another aspect to consider when discussing the Basic Service Middleware is how it interacts with the Composite Service Middleware. This interaction happens mainly by the means of web service invocations. However, the adaptability of the basic services is achieved transparently with respect to the upper layers (this was one of the key requirements). This type of interaction is thus carried out through the standard web service interfaces and transport mechanisms, and does not require any specific support from any side. There is still a non-standard interaction between BS and CS middleware: BS middleware is supposed to provide CS middleware with non-functional properties of the services it hosts. This includes, for example, performance measurements or availability statistics. It is meant to help the composition engine during the assembly stage to choose between equivalent services those that would suit best the constraints it has been given. This integration of the BS middleware with the CS middleware makes use of sensors that collect different metrics and make the results available through a standard web service interface. This part of the BS middleware will be described in Section 4.

2 Web Tier and Business Tier Integration

2.1 Approach

We have chosen to use tight coupling to integrate the web and the business tiers. Tight means, that there is now one replication algorithm that spans both the web tier and the business tier. This is easily possible since Enterprise Java Beans can be directly exported as web services, and the web tier itself, although existent through special Java objects, is only a forwarding mechanism. It does not contain any relevant state, and both tiers can be actually run in one environment. From a conceptual point of view, the changes to the existing EJB replication algorithm presented in deliverable D3 are actually very small. Furthermore, although it is possible to run the web tier independently (e.g. on distinct machines), in many real world settings, the web server and the application server are actually two threads within the same Java virtual machine. In this case, the failures of both servers are not entirely independent anymore and the loose coupling approach would not provide much additional adaptability.

The web service specification [IBM03] states that web services might be implemented by session beans. The tight coupling we are aiming at uses this features extensively. However, the specification does not consider the issue of persistence across invocations: it is only interested in the different ways a web service interface (defined in WSDL) might be implemented on the server side. Consequently, only request-scope web services and their corresponding stateless session bean implementation are addressed and supported by compliant application servers.

Nonetheless, our target platform, JBoss [JBo03], supports the implementation of session-scope web services by stateful session beans. The only requirement is that the web service transport protocol is HTTP (which is by far the most common transport protocol used for web services). To this end, JBoss maintains a mapping between HTTP session names and specific bean instances in a standard hash table. Given this infrastructure, it is also possible, respectively, to expose a session bean (stateless or stateful) as a web service. Only the WSDL definition file that describes the bean interface needs to be provided. The rest of the machinery, that makes the service available and route requests from the web service to the bean, is provided by Axis and JBoss. One should note, however, that in this case the bean must provide an *ejbCreate* method with no arguments: this is the only limitation to the full EJB specification [DeM03]. But this is not too constraining as, for session beans, it is always possible to perform initialization in auxiliary methods.

2.2 Framework Support

In this document, we will only discuss the modifications that were required from the framework to support Enterprise Java Beans exposed as web services. These modifications are limited and do not impact the behavior of the framework in its other use cases. Hence, the description of the framework made in deliverable D3 remains accurate.

The first modification deals with the visibility of invocations (requests and their corresponding responses) targeted to EJB exposed as WS. Such invocations go through both the standard web service stack and the standard EJB stack. As such, they were originally intercepted by the framework twice and the component monitor was first notified to handle a web service invocation and then to handle an

EJB invocation. To remove those duplicates, such special invocations are now tracked by the framework. When it detects one, it skips its own web service invocation specific processing. Consequently, these invocations are seen (only) as plain EJB invocations by the component monitor.

The second issue is related to "fail over". It has always been a requirement that a client could fail over to another replica when the server it was connected to crashes (or becomes unreachable). This has been made possible by making session information part of the state of the components (EJB or web service). With the previous version of the framework, this information was transparently replicated by the replication algorithm together with the components themselves. This scheme, however, was not sufficient to deal with the case of beans exposed as web services. In this case, the session information used by the client is the HTTP session, while the session information replicated by the algorithm is the EJB session. We have mentioned the fact that the application server was maintaining a mapping between HTTP and EJB sessions, but this mapping was lost from one server to its replica. To circumvent this limitation, we have introduced a new mapping mechanism, overriding the server original one. Then, we have added this additional mapping information to the state of beans exposed as web services and have had the mapping transparently restored remotely when the bean state is replicated. This way, a client can safely contact a server replica using the same HTTP session it was given by the original server.

The error management mechanisms of the framework were also subject to some modifications. Those were necessary to correctly translate errors generated on the EJB side back to the web service side. They were also needed to have these errors duly reported to the client component monitor.

2.3 Unified Replication Algorithm

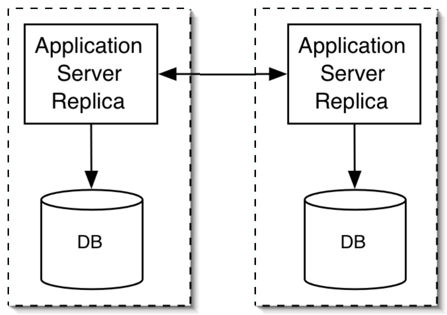
With the minor framework adjustments we have just described, the eager replication algorithm (presented in deliverable D3), originally designed to work with plain Enterprise Java Beans only, was able to work unmodified for beans exposed as web services. Client-side logic, which enables client to fail over to other replicas in particular, is derived from JMiramare (also described in D3). This completes the integration of the web tier with the business tier.

3 Business Tier and Database Tier Integration

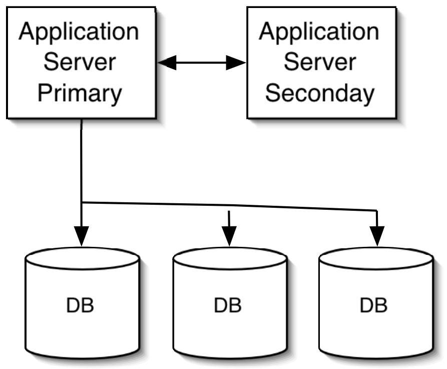
3.1 Coupling Alternatives

In the previous sections we have seen a tight integration of web-server and application-server replication. Coupling business and database tiers in such a tight way is more complex and raises some challenges, because the two tiers have very different tasks, and data is by definition distributed among both tiers. A database system represents a very independent software system that provides a clear but restricted interface. One can choose a tight coupling between business and database tier by writing application semantics as stored procedures and triggers within the database system. That is, the application semantics is moved to the database tier. The solution is very cumbersome and doesn't allow taking advantage of many of the features of current business tier technology as provided by J2EE. The second solution is to not use a database system but use simple persistence within the business tier for atomicity. With this, however, the many optimizations provided by databases systems for transaction management (persistence and concurrency control), and fast query execution (indexing etc.) cannot be exploited. Hence, this approach is also unattractive.

A third tight coupling approach is depicted in Figures 1(a) and 1(b). In here, there exist several application server replicas and several database replicas. Only the business tier is responsible to coordinate



(a) one-to-one



(b) primary – secondary

Figure 1: Coupling AS and DBS

replication and the database replicas do not know anything about the fact that there exist several database copies. The responsibility of keeping the database copies consistent relies completely on the business tier. We present two alternative architectures. In the first (Fig. 1(a)), each application server replica is connected to one database replica: this is “vertical replication”. If one application server replica updates data within the database copy it is connected to, the other application server replicas have to perform the same updates on their database copies. An according coordination has to run between the application server replicas. To guarantee correctness and consistency in the case of application server failures, such a coordination has to take place before transaction commit. Either the application server replicas can guarantee that they all act deterministically (submit operations in the same order to their corresponding database replicas) or a special form of 2-phase commit protocol has to be executed to guarantee that all or none of the database replicas commit a transaction. Furthermore, if all application server replicas are allowed to execute transactions (e.g., each is primary for some clients), then an even more complex coordination of the entity beans of the different application server replicas and their access to the database replicas will be necessary. If either a application server replica or a database replica fails, the corresponding database (resp. application server) replica has to be forced to fail, too. When an application server replica recovers, the corresponding database replica has to recover and receive the current database at the same time. This by itself is a challenging task.

The second approach (Fig. 1(b)) is possible if there is one primary application server replica that performs all execution, and the other application server replicas are only used for failover. Then, the primary can be connected to all database replicas and coordinate the execution on these replicas. There is not a one-to-one relationship between application server and database replicas, and the number of replicas might differ per tier. If the primary application server replica fails, a backup takes over, establishes connections to the database replicas, and continues from there. However, such failover might be quite complicated because in case of failures the connections between the primary and the database replicas are broken. Typically, upon connection loss, database systems abort the active transaction on the connection. At the time the primary crashes, a given transaction T_i might have committed at some database replicas, being active at others, and/or not even started at some. The backup has to make sure that such transactions are eventually committed or aborted at all replicas.

In both cases, the replication of the business tier state and replica control in regard to the database replicas has to be coordinated to guarantee exactly-once execution, and transaction atomicity at both tiers. This means, we basically have to design and implement an integrated algorithm that takes care of state both at the business tier and the database system. This is different to the tight integration between web tier and business tier where we were able to find a solution that only had to do minor adjustments on the existing replication algorithm of the business tier.

In fact, we believe that an integrated solution for application and database replication would probably use similar techniques for database replica control as proposed in deliverable D3, but has to completely reimplement them to run within the business tier and together with the application server replication protocol. Instead of such a complete reimplementation, we propose a loose-coupling approach where the business tier uses an existing replicated database infrastructure. In the ideal case, the replicated business tier would actually not even be aware of the fact that the database system is replicated but view it as a single, highly reliable and highly scalable database system. But of course, this depends on the interface and functionality provided by the replicated database infrastructure.

In the following, we will outline one solution, namely how to integrate our replicated business tier infrastructure with Middle-R. We will see that the replication algorithm at the business tier has to be modified in a very minimal way. Apart of this, the existing Middle-R infrastructure can be completely used by the replicated application server with its full functionality.

3.2 Middle-R Overview

In this section we want to shortly recall the architecture of Middle-R and its interfaces that are of interest for the integration. Since the last deliverable D3, some changes have been performed [LKPJ05], and we want to shortly outline them.

In Middle-R, a middleware layer is installed between the database replicas and the application. This middleware infrastructure itself is replicated. That is, there is a middleware replica in front of each database replica. The middleware replicas build a group and communicate with each other via group communication. A pair of middleware/database replica behaves as a unit, i.e., either both or none fails. If a database replica fails, the middleware replica automatically shuts down. If the middleware replica fails, the database replica is unreachable. When the middleware replica restarts, it will reconnect to the database replica and they will rejoin the system as an entity. The middleware provides a standard JDBC interface to the application. The application can submit transactions by either using autocommit on in which each SQL statement represents a transaction, or with autocommit off in which the application has to submit explicit commit requests and several operations can be bundled into one transaction.

Currently, the system is designed for LANs where we assume no network partitions. Addresses of individual replicas do not need to be known in advance. Instead, the middleware only has a fixed IP multicast address. Upon a connection request, the JDBC driver (running in the context of the application program) multicasts a discovery message to this multicast address (with possible retries). Replicas that are able to handle additional workload respond with their IP address/port. The driver collects all addresses and connects to one of them, keeping the others for failure cases. We will discuss these failure cases later.

When a client (application program) is connected to one of the middleware replicas, it can submit SQL statements embedded in transactions. The middleware replica first executes all requests (except commit) at the local database replica. Upon an abort, the transaction aborts locally without any communication with other database replicas. Upon the commit request from the application program, the middleware replica retrieves the writeset from the database (as explained in deliverable D3) and multicasts it in total order to the other middleware replicas. All middleware replicas now serialize conflicting transactions according to this total order. In particular, if there are two transactions T1 and T2 (that have executed at different replicas), T1 and T2 conflict and both send their writesets concurrently, then if T1 is delivered before T2 all replicas will commit T1 and abort T2. If T1 and T2 do not conflict they will both commit in the order of delivery. The middleware layer performs a version check to detect such conflicts. We will not explain in detail this version check since it is out of the scope of this deliverable (see [LKPJ05] for more details). However, we want to point out that our approach uses the semantics of snapshot isolation [BBG$^+$95] as level of isolation provided to concurrent transactions. This is the standard isolation level provided in Oracle and PostgreSQL.

Our approach requires one message exchange between the middleware replicas per transaction, namely the writeset. The writeset is sent with uniform reliable delivery guaranteeing that if one database replica commits the transaction then for each other replica it either also commits the transaction or fails. From the client perspective, the client receives the confirmation of commit once the local middleware replica has received the writeset in uniform reliable total order, the version check has not detected any conflicts with concurrent transactions that have executed on other replicas, and the transaction has been locally committed. For the client itself, the coordination among the different middleware replicas is completely transparent except of the fact that the commit might now take longer than in the centralized case since it involves one message delay.

If the middleware replica to which a client is connected crashes, the connection is lost. The driver will detect this and automatically connect to another replica. At the time of crash the connection might have been in one of the following states.

1. There was currently no transaction active on the connection. In this case, failover is completely transparent.

2. A transaction T was active and the client has not yet submitted the commit request. In this case, T was still local on the middleware/DB replica that crashed, and the other replicas do not know about the existence of T. Hence, it is lost. The JDBC driver returns an appropriate exception to the client program. But the connection is not declared lost, and the client can restart T.

3. A transaction T was active and the client has already submitted the commit request which was forwarded to the middleware replica but the client has not yet received the commit confirmation. In this case, the state at the different replicas might be as follows.

   (a) The crashed middleware has not multicast T's writeset before the crash. Hence, the remaining replicas do not know about the existence of T, and T must be considered aborted.

   (b) The other replicas have received T's writeset. If validation succeeds, they will commit T.

Let's have a closer look at case 3. If clients are directly connected to the database and the database crashes after a commit request but before returning the confirmation, clients do not know whether the transaction aborted or committed. In our case, however, we will be able to provide such feature. When a new transaction starts at a middleware replica, the replica assigns a unique transaction identifier and returns it to the driver. Furthermore, the identifier is forwarded to the remote middleware replicas together with the writeset. Each replica keeps these identifiers together with the outcome of the transaction (commit/abort determined at validation) for a certain time. If now a middleware/DB replica crashes during a commit request, the JDBC driver connects to a new replica and retries the commit of the in-doubt transaction (with the same transaction identifier attached). If the new replica had not received the writeset, it does not know about the identifier, and hence, we can be sure that the transaction did not commit. The JDBC

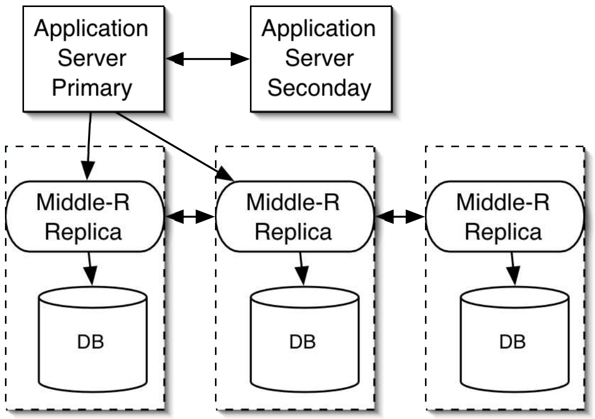


Figure 2: Coupling AS and Middle-R

driver is informed about this and it returns the same exception to the client as if the commit was not yet submitted at the time of crash. If the new replica has the identifier, it checks for the outcome and returns the outcome to the driver which forwards it to the client program. In this case, failover was completely transparent.

There are two further, special cases that we have to consider. Firstly, due to the asynchrony of message exchange it might be possible that the middleware receives the inquiry about a transaction from a driver and after that it receives the writeset for the transaction. In order to handle this correctly, the replica does not immediately return to the JDBC driver if its does not find the transaction identifier. Instead, it waits until the group communication system (GCS) informs it about the crash of the old replica. According to the properties of the GCS, the new replica can be sure that it either receives the writeset before being informed about the crash (and then tell the driver the outcome) or not at all (and then the transaction did not succeed). The second issue to consider is that the crashed replica might have started the multicast before the crash. In this case, however, uniform reliable delivery provides us with a nice property: Either all or none of the remote replicas have received the writeset. Thanks to this property, if the local replica received the writeset and committed the transaction before the crash, then all (available) remote replicas will receive the writeset and hence, also will commit the transaction.

Garbage collection is quite simple because for each connection we only have to keep track of the last transaction that terminated (commit/abort) over this connection. Only for this transaction the JDBC driver might ask the outcome. Hence, transaction identifiers should implicitly identify the connection object.

3.3 Coupling the Application Server Replication Algorithm with Middle-R

During normal processing, only the primary application server replica is connected to Middle-R. If it has several connections open they can be linked to different middleware replicas. Fig. 2 shows this architecture.

As long as there are no crashes, the database replicas are guaranteed to be all completely synchronized, and database replication is transparent to the application server.

We will now discuss what happens in the various failure cases.

3.3.1 Database replica fails, application server primary does not fail

Assume the application server primary does not fail but one of the middleware/database pairs it is connected to fails. In this case the application server primary is the simple client as described above. In the following we will refer to the middleware/database replica pair as database replica for simplicity.

1. If the database replica fails while there was no transaction active on any connection, the JDBC driver automatically reconnects to a different middleware replica without the application server primary even noticing.
2. For each active transaction for which no commit was yet submitted, the JDBC driver will provide the application server primary with an abort exception informing that this abort was due to a database replica crash. The replication algorithm of the application server can intercept this abort exception and perform according actions. Recall that a transaction is always associated with one client request. The application server primary can abort the activities on the application server primary related to this request and simply restart the execution of the request. This reexecution will initiate a new transaction at the database system. This time the application server primary is connected to a different middleware replica but the application server primary does not really care to which one. Reexecuting the client request is fine since all effects in regard to the first execution have been undone at the application server and database, and no response has yet been returned to the client.
3. The database replica fails after the application server primary has submitted the commit request but before it has received the response. In this case, the automatic failover of the JDBC driver is able to detect whether the transaction was committed or aborted at the remaining database replicas and provide the corresponding response to the application server primary. The application server primary does not need to perform any special actions in case of commit but will abort and reexecute the client request in case of database abort as above.

Hence, the actions upon the crash of a database replica are very simple to integrate and allow for a nearly seamless failover to another database replica. The only action that has to be performed is the following: whenever the application server primary receives an abort exception indicating that the abort was due to a database crash, it has to also abort the transaction at the application server level, and restart the execution of the corresponding client request.

3.3.2 Database replicas do not fail, application server primary fails

Before going into the coupling we want to recall the application server replication algorithm and the steps taken at failover by the new primary. Whenever the primary application server replica intercepts a commit request for transaction T to the transaction manager, it first sends to the backups (i) the state changes on stateful session beans triggered by T and (ii) the client request associated with T, and (iii) the response that will be sent to the client. Then it writes a marker into the database as part of the transaction T. Then it commits the database transaction. After that it sends a commit message to the backups. Finally it returns the response to the client. Now assume the primary fails and a new application server primary takes over. The new primary will connect to the database. Assume a transaction T that changed state at the application server and within the database.

1. If the primary fails before sending the state changes of T that occurred within the application server, the database transaction is aborted, and the backup will restart the execution of the request.

2. If the primary fails after sending the state but before submitting the commit request to the database, the transaction is aborted at the database replica. The new application server primary will not find the marker, discard the application state received from the old primary for T, and reexecute the associated client request.

3. If the primary fails after submitting the commit request to the database, the new primary will find the marker in the database, install the application server state changes for T and not reexecute the request but return the response immediately in case the client resubmits the request.

4. If the primary fails after having sent the commit message, the transaction is guaranteed to have committed at the database, and the new primary will not reexecute the request in any case.

Let's now have a look when the old primary is connected to Middle-R instead of a centralized database system. When the primary fails its connections to Middle-R are lost. Middle-R will abort each active transaction for which it has not yet received the commit request from the application server primary. This is the same behavior as that of a centralized database system. If Middle-R has received the commit request it will have committed the transaction (unless it will abort due to conflict or application semantics). Again, this is exactly the same behavior as that of a centralized database system. As a result, it seems that the failover actions described above seem to be the same independently whether the business tier is connected to a single database system or to Middle-R.

However, we have to be a bit more careful. Assume the old primary had a connection to a middleware replica M and had an active transaction T over this connection. When a new primary takes over it will establish a connection to Middle-R for each in-doubt transaction (for those for which it has received the state but not the commit message). Assume T is such an in-doubt transaction. The new primary might now connect to different middleware replica, i.e., not M but, e.g., M'. One has to be aware that the database replicas do not run a 2-phase commit protocol. As a consequence, a T might be committed already at M but still running at M' or might not even have started at M'. Hence, if the new primary now checks for the marker for T through M it might not find the marker because M' has not yet started T. The problem is the asynchrony of business tier and database tier. Since the database commit is not instantaneous, the new primary can be faster performing its checks at failover than the database system is able to abort or commit the transaction system-wide, and the new primary might draw wrong conclusions. As such we have to provide mechanisms to ensure that business and database tiers are synchronized.

In order to address this issue we suggest two solutions that are implemented by Middle-R to correctly handle replicated clients (as the application server is).

The first solution enhances the JDBC interface with a special ”quiet” request method. Assume again the old primary executes T at middleware replica M and crashes after sending the state but before sending the commit confirmation to the backup application servers (T is in-doubt). The new primary will first create a new JDBC connection object which connects to any middleware replica M'. Then, the new primary will submit the ”quiet” request method over this connection. Upon receiving the quiet request, M' will multicast a corresponding message to the middleware replica group. Each middleware replica will respond with a total order message once there is no client transaction active anymore at this middleware replica (a transaction that is local at this middleware replica). Once M' has received all responses it is guaranteed that it has also received the writeset of T because the local middleware replica M of T will send $T's$ writeset before sending the response. Now, when M' has terminated all local transactions and applied all writesets of remote transaction, it returns a ”quiet successful” to the new primary. The new primary now knows that all transactions that were active at the time the old primary crashed were either aborted at the database, or were received and committed at M'. It can now check for the marker of T (and any other in-doubt transactions) using the connection to M'.

The second solution uses a quite different approach and uses two concepts. (i) The JDBC connection objects become ”state” objects of the application server that have to be replicated to the backups just like other state changes at the application server. (ii) The submission of the commit request becomes an idempotent operation at Middle-R. With these features the business tier has to perform the following actions. During normal processing the application server primary sends a copy of the JDBC connection object used for a transaction T together with the state changes performed by the transaction T to the backups at commit time. This connection object keeps track of the transaction identifier of T as maintained by Middle-R. This transaction identifer also implicitly identifies the middleware replica M to which the transaction T was submitted. Now assume again the old primary executes T at middleware replica M and crashes after sending the state but before sending the commit confirmation to the backup application servers. When now the new application server primary takes over it performs the following action for T. It sends the commit request over the connection object copy that belongs to T. The connection object copy, of course, is not really connected to any middleware replica. Hence, it first connects to any middleware replica M' ($M = M'$ or $M \neq M'$) and submits the commit request with the transaction identifier stored in the connection object copy. If $M = M'$, M' can immediately detect whether this is a resubmission or not. If yes, then it can confirm the commit to the driver. If not, it had aborted T when the connection to the old primary was lost. It can inform the new primary about this accordingly, and the new primary can abort T at the application server level and reexecute the corresponding request. If the new primary connects to $M' \neq M$, and M' had already received the writeset, it can immediately respond. Otherwise M' sends a request to M (it can determine M because the transaction identifier for T contains the identifier of M). If M had not received the commit request for T from the original primary and hence, aborted T, it informs M'. Otherwise, M' will eventually receive the writeset and be able to inform the new primary about the outcome. Note that with this mechanism, there is actually no need for the marker mechanism. Instead of looking for the marker, the new primary simply submits the commit request over the connection object copy. Hence, this extended functionality of Middle-R – allowing a resubmission of a commit request (with idempotent characteristics) – provides additional functionality over a centralized system. As a result, the application server replication algorithm can be simplified.

For both solutions, the application server has to be aware of the replication at the database level – at least to some degree. In the first case, it has to call a special method to synchronize the actions at the business tier with the actions at the database tier. In the second case, it must replicate the JDBC connection objects as objects that contain state. In both cases, however, the extensions to the business tier replication algorithm are straightforward. In the same way, Middle-R has to be aware of the replication of the business tier. The additional functionality provided in Middle-R is relatively complex to implement. However, it follows very clear and simple semantics: a general ”quiet” procedure, or an idempotent commit and a mechanism to allow for the replication of connection objects.

3.3.3 Both the application server primary and the middleware replica to which the primary is connected to fail

In this case, the new primary cannot connect to the same middleware replica M as the old primary since M also fails. However, both solutions described in the previous section can be used in this case, since they do not require the new primary to connect to the same middleware replica the old primary was connected to.

Let’s first look at the solution using the quiet request method. If there is an in-doubt transaction T

(originally executed at M), the new primary connects to a middleware replica M' and submits the quiet request. M' multicasts the request to the group, and waits for all responses. However, it will not receive a response from M since M failed. Instead, the group communication system will deliver a view change message excluding M from the view. Furthermore, the group communication system will either deliver any write set sent by M before the view change or not at all. Hence, when receiving this view change message, M' knows that it will not receive any further write sets from M. Hence, it terminates all local transactions and transactions for which it has received the writesets and then returns from the quiet call. This guarantees that the business and database tier are synchronized and the new primary can safely check for the markers of in-doubt transactions.

In the same way, when the connection objects are replicated and the new primary submits the commit request for an in-doubt transaction T to middleware replica M', then M' will forward this request to the old middleware replica M. Since M crashed, M' will not receive any response but a view change message. It will either receive the T's writeset before the view change and can respond to the new primary accordingly, or not receive the writeset at all. In the latter case, in can return the according abort exception to the new primary.

4 Adapt Web Services Performance Monitoring Tool

4.1 Overview

The monitoring tool consists of a set of sensors for the server-side performance monitoring of JAX-RPC web services.

Each sensor performs two basic operations: calculates the mean latency of the service and counts the invocations of the service (per service, not per method). Counting is activated both by successful invocations of the service and by fault-generating invocations and the two counts are kept separated. Latency is measured for successful invocations only. Sensors come in two flavors: A continuous sensor monitors a given service since the monitoring has been activated. A periodic sensor monitors a service over a given time interval, that is, it tracks the number of invocations and the mean latency in the last N milliseconds, N being a configurable parameter.

A sensor can monitor a single web service or a group of services. In the latter case, the group may be composed from any number of services and, for monitoring purposes, it appears as a single service. A service may be part of any number of groups. A service may be monitored as a part of a group even if it is not being monitored individually.

All data collected by the monitoring tool are exposed through a single web service. Configuration of the monitoring tool is performed through that web service as well. Set-up tasks include activating or stopping a specific sensor; choosing which sensors should be activated for a specific service (continuous, periodic or both); setting the time interval for a periodic sensor; grouping services for monitoring purposes. All configuration tasks can be performed dynamically, without interfering in any way with the web services being monitored.

Monitoring tool thus allows for the composite services middleware composition engine to easily set-up the monitoring or to access the performance and availability data of the basic services it is interested in. That information is valuable in choosing among equivalent services during the assembly process of a composite service and also, during the execution of composite services, it allows for smart load balancing. Other intended use of the monitoring tool is to help system administration.

4.2 Implementation Details

The monitoring is fully transparent to the web-service users as is implemented with an Axis handler.

In order to allow monitoring, each service must add our specific *SensorHandler* to its request and response paths. Once the monitoring is enabled, this handler intercepts every request (call to the service) and starts a high precision timer for it. Once the request is executed and the response is generated, the handler intercepts the response and, if not a fault, the service latency is calculated.

The timer utility used is Vladimir Robutsov's *com.vladium.utils.timing.Itimer,* which allows submillisecond timing precision in Java (including J2SE1.4) [Rou03].

The main tasks of the monitoring tool are implemented as a singleton object, which holds references to all sensors and all configured service groups. Our *SensorHandler* stores the collected data by calling methods of this object. Periodic sensors are implemented as follows. For each periodic sensor there is a pair of FIFO stacks, one for successful invocations and one for faults. Whenever a response in intercepted, a new entry is inserted into the corresponding stack as follows. If the invocation is successful, then the entry will contain the latency measurement and the time at which the entry has been inserted into the stack. If the invocation generated a fault, then the entry will contain just a timestamp. Moreover, entries associated with timestamps older than the current period are purged.

A separate class, *MonitorConsole* that is exposed as a JAX-RPC web service acts as a front-end to this machinery. MonitorConsole controls all client-server interactions for accessing the collected data and for configuring the monitoring system. Methods of this class may be grouped into four logical categories: global status methods, service set-up methods, sensor-configuration methods and data access methods. The first group consists of only three methods whose names are self-explanatory: *startMonitoring(), stopMonitoring()* and the boolean *monitoringActive*.

There is a second group of methods that deals with service set-up. These methods are: *createServiceGroup(String name)* which creates a new service group with the specified name, and its opposite *removeServiceGroup(String name)* which destroys it. A service group is initially empty, i.e. no service belongs to the service group. A service may be added to or removed from a service group with *addToServiceGroup (String groupName, String serviceName)* and *removeFromServiceGroup (String groupName, String serviceName)*. The *serviceName* identifies a service as it appears in the *AxisServlet* class. Finally, method *listServiceGroups()* lists all service groups while method *listServiceGroupMembers(name)* lists all the services that belong to the named service group. None of these methods start or stop monitoring on any service. To do that, appropriate sensor-configuration methods must be called as described below.

Sensor-configuration methods include *addSensor(String name)* for starting and *removeSensor(String name)* for stopping the monitoring sensor for the named service or the group; *listSensors()* lists all active sensors; *resetAllSensors()* resets all sensors back to its initial values while *resetSensor(String name)* resets just the named sensor. All these methods operate on continuous sensors and there are their accordingly named counterparts that operate on periodic sensors. Finally, there are two methods for configuring the time interval of periodic sensors only: *setInterval(String name, long interval)* and *getInterval(String name)*.

The final group of methods deals with fetching sensor data from the monitoring tool. Names of the methods present here should be quite self-explanatory: *getMeanLatency(String name), getInvocations(String name)* and *getFaults(String name)*. There is a periodic sensor double for each of these methods as well.

A Software Distribution

The software distribution of this deliverable consists of:

- a zip file, `jboss-adapt-framework-1.0-src.zip`, which contains the source distribution of the ADAPT J2EE framework;
- a zip file, `jboss-adapt-replication-sib-1.0.zip`, which contains the source distribution of the eager replication algorithm, as a component monitor of the framework;
- a zip file, `adapt-jbora-2.0.zip`, which contains the source distribution of the group communication layer used by the replication algorithm;
- a zip file, `adapt-sensors-0.9.zip`, which contains the source distribution of the sensor infrastructure to use within the framework.

All these files are accompanied by their respective JavaDoc documentation and can be download from the project SourceForge page:

`http://j2ee-adapt.sourceforge.net/`

The JMiramare software distribution may also be downloaded from the same site.

References

[BBG$^+$95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):1–10, 1995.

[BBM$^+$04] Ozalp Babaoglu, Alberto Bartoli, Vance Maverick, Simon Patarin, Jaksa Vuckovic, and Huaigu Wu. A Framework for Prototyping J2EE Replication Algorithms. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2004)*, pages 1413–1426, Larnaca, Cyprus, October 2004.

[BPA05] Alberto Bartoli, Milan Prica, and Etienne Antoniutti di Muro. A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container. *Concurrency and Computation: Practice and Experience*, 2005. To appear.

[DeM03] Linda G. DeMichiel. *Enterprise JavaBeans^{TM} Specification, Version 2.1*, November 2003. `http://java.sun.com/products/ejb/docs.html`.

[IBM03] IBM Corporation. *Web Services for J2EE Specification*, 2003. `http://jcp.org/aboutJava/communityprocess/final/jsr921/index.html`.

[JBo03] JBoss Group. JBoss 3.2.3, November 2003. `http://www.jboss.org/`.

[JPKA02] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002. IEEE Computer Society.

[LKPJ05] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, June 2005. To appear.

[MJPK04] J. M. Milan-Franco, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Adaptive middleware for data replication. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 175–194. Springer-Verlag New York, Inc., 2004.

[Rou03] Vladimir Roubtsov. My kingdom for a good timer! JavaWorld, January 2003. `http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html`.

[WK05] Shuqing Wu and Bettina Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *IEEE International Conference on Data Engineering (ICDE)*, Tokyo, Japan, April 2005. To appear.

[WKM04] Huaigu Wu, Bettina Kemme, and Vance Maverick. Eager replication for stateful J2EE servers. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2004)*, pages 1395–1412, Larnaca, Cyprus, October 2004.