RJDBC: A SIMPLE DATABASE REPLICATION ENGINE*

Javier Esparza Peidro, Francesc D. Muñoz-Escoí, Luis Irún-Briz, Josep M. Bernabéu-Aubán
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46071 Valencia, SPAIN
Email: {jesparza, fmunyoz, lirun, josep}@iti.upv.es

Key words: Replication, fault tolerance, reliability, distributed systems, databases, middleware, systems integration

Abstract: Providing fault tolerant services is a key question among many services manufacturers. Thus, enterprises usually acquire complex and expensive replication engines. This paper offers an interesting choice to organizations which can not afford such costs. RJDBC stands for a simple, easy to install middleware, placed between the application and the database management system, intercepting all database operations and forwarding them among all the replicas of the system. However, from the point of view of the application, the database management system is accessed directly, so that RJDBC is able to supply replication capabilities in a transparent way. Such solution provides acceptable results in clustered configurations. This paper describes the architecture of the solution and some significant results.

1 INTRODUCTION

Reliable systems are becoming more and more valuable nowadays. At present, customers demand quality services and unavailability or data losses due to unexpected failures are not accepted. On the other hand, the blowing up of service demands is causing annoying bottlenecks due to non-scalable systems. Thus, enterprises must spend substantial amounts of money and efforts in order to improve the availability and performance of their services. Common solutions consist on deploying complex and expensive systems which frequently imply the reengineering of the software products that use such systems. However, many companies can not afford such costs, and other low-cost solutions are required.

Enterprise applications implemented in Java, and also most of J2EE services, need to interact with a JDBC driver in order to access their data, commonly stored in a relational DBMS. The aim of our Replicated JDBC (RJDBC, hereafter) driver is to provide transparent support for database replication, without needing any modification in the underlying database schema, nor in the applications that have to access these replicated data. In order to achieve this, a second JDBC driver is needed, placed on top of the native one. This second driver intercepts all the operations requested by the user applications and, when needed, broadcasts them to the rest of replicas using total order (i.e., with a uniform atomic broadcast protocol (HT93)). This will ensure that all database replicas process all update operations in the same order, guaranteeing database consistency.

Although there are other kinds of database replication protocols with better performance features and that require less replica interaction (WSP$^+$00), the RJDBC solution offers other interesting features that are appealing for small and medium enterprises. First, it does not require any schema modification; usually, other database replication protocols require that some metadata are added to the database being managed, for instance, data item versions, or owner replica numbers (JAJ$^+$02; RMA$^+$02). Such metadata prevent replication transparency, since they force that all applications access the database using the replication support. Accesses made by-passing it, may get undesirable results. In RJDBC this problem is not present.

*This work has been partially supported by the EU grant IST-1999-20997 and the Spanish grant TIC2003-09420-C02-01.

The second feature is that the needed replication support can be easily and unexpensively implemented using a broadcast toolkit, and it does not need many resources. So, RJDBC can be deployed in computers with less memory and less powerful processors than most other database replication protocols.

The rest of the paper is structured as follows. Section 2 describes the structure of RJDBC. Section 3 exposes some results of the developed prototype, comparing it with other systems. Related work is described in section 4. Finally, section 5 concludes this paper.

2 ARCHITECTURE OF RJDBC

A database replicated by means of the RJDBC middleware is composed by multiple replicas accessible via JDBC. In each node where a client application exists, there will be a RJDBC driver on top of RJDBC core and the usual JDBC driver. As stated previously, this RJDBC core uses a uniform atomic broadcast protocol in order to intercommunicate the replicas.

Basically, each node pursues two main targets:

1. To spread all significant operations performed on each node among all the nodes of the system, in the same order. The operations are codified, packed in some ready-to-transmit way and finally applied on actual database objects.
2. To ensure database consistency among all the replicas. Notice that all the operations arrive to each node through an atomic broadcast protocol, so that the same operation delivery order is guaranteed in all nodes.

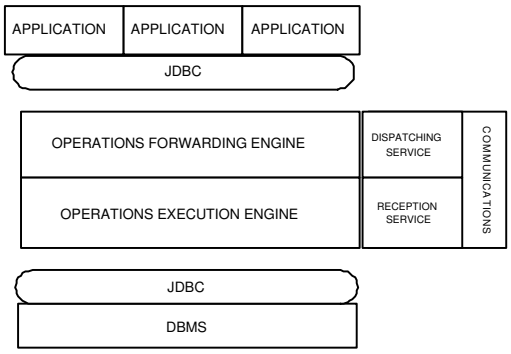


Figure 1: RJDBC global architecture.

So, in order to reach these goals, each RJDBC node can be roughly split up into two main components, connected by means of a third one, the communications subsystem (figure 1):

1. The operations forwarding engine has the job of broadcasting all the operations performed over the local node among all the nodes of the RJDBC environment, including itself. It provides a JDBC interface to the application layer and we refer to it as RJDBC driver.
2. The operations execution engine feeds on the operations delivered by the communications subsystem. It sequentially applies all received operations to the underlying DBMS.
3. The communications subsystem consists of two different elements, the dispatching service, which broadcasts all the operations provided by the forwarding engine, and the reception service, which admits the broadcast operations and delivers them to the operations execution engine.

The latter two components constitute the RJDBC core.

Going into further details, the applications interact with the RJDBC node through standard JDBC invocations, using our RJDBC driver. Such driver complies with the JDBC specification and provides all the objects required for database accessing. These objects wrap JDBC invocations and forward them to the RJDBC core. These objects are named *wrappers*. They stand for proxies of the actual JDBC database objects. Thus, all database operations requested by the client applications are immediately packed and forwarded by the wrappers to the RJDBC core. This communication takes place using Java RMI.

When the RJDBC core is notified about a new operation, it promptly sends it towards all the system nodes, including itself, using a uniform atomic broadcast (this implies total order). Not all operations are required to be broadcast but only those affecting the database management system state. For instance, if the underlying database uses multi-version concurrency control, read-only accesses may not be transmitted.

When a RJDBC node receives a new operation from the communications protocol, it is enqueued for subsequent processing. Each RJDBC node contains an Operations Execution Thread (OET), in charge of executing sequentially all enqueued operations. This feature, plus the total order delivery, ensure the database inter-replica consistency.

On the other hand, the OET does not apply operations over the database directly either. Instead, it executes them over exact copies of the final JDBC database objects, called *adapters*. These elements allow fine grained control of operation execution results. They maintain a reference to the actual JDBC database objects and execute the requested operations on them, collecting the results and communicating them to the OET.

In essence, each node consists of five components arranged as figure 2 states. These components are: the wrappers layer, the manager, the communications layer, the operations execution thread and the adapters layer.

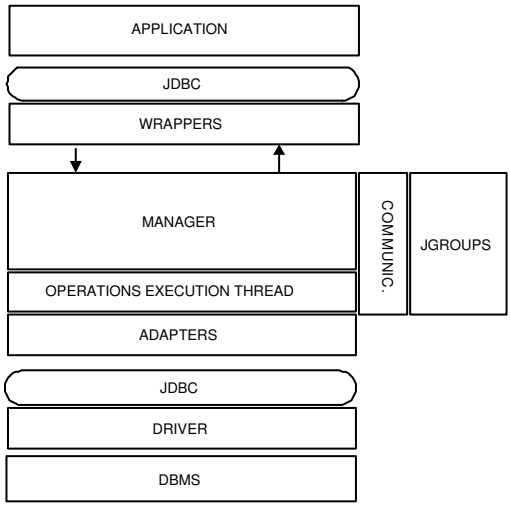


Figure 2: RJDBC architecture.

- The **wrappers layer** provides access to the underlying database management system, hiding the RJDBC replication engine. Thus, applications access to regular JDBC objects, but these objects do not operate directly on the database. Instead, they act as proxies of the actual JDBC objects, placed on the adapters layer.
- The **manager** is the core of the RJDBC middleware. When the wrappers layer forwards an operation, the manager determines if it must be broadcast or not. To this end, the manager contains a *forwards table*. For instance, in multi-version based DBMSs (PostgreSQL is a sample), read-only queries may not lead to any transaction lock, and they are not forwarded.

  If the forwards table states that the operation must be broadcast, it is transferred to the communications layer. But if the operation does not need to be broadcast, it is immediately applied on the local database.
- The **communications layer** is the component that implements the atomic broadcast services needed in RJDBC. Currently, this component has been implemented using an open-source toolkit called JGroups (Jav04), although other similar toolkits could be easily integrated in our system.
- The **operations execution thread** serves sequentially all the operations received via atomic broadcast, once they have been delivered by the communications layer. Note that these operations are not applied directly to the underlying database, but to the RJDBC adpters layer.

  Concurrent transactions may get locked if they try to access the same data items in conflicting modes (write-write, for instance). Since only one operations execution thread exists in RJDBC, a lock of this kind could imply a system deadlock. To prevent this, a timeout-based mechanism exists that, once a given time has ellapsed, starts a new thread that will serve the remaining enqueued operations.
- The **adapters layer** is the lowest layer of RJDBC and directly uses the services provided by the "native" JDBC driver.

  Whilst the wrappers provide an image of the JDBC object to the application, the adapters represent the JDBC object to the RJDBC replication engine.

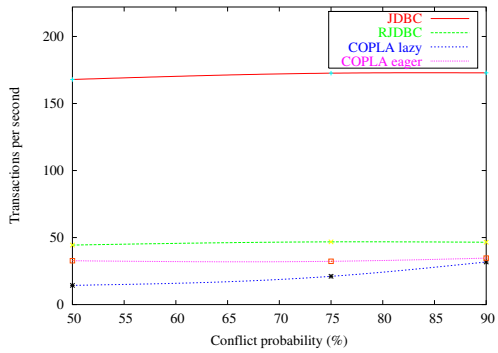
3 RESULTS

One of the advantages of RJDBC when it is compared to other database replication mechanisms is its simplicity, both in its design and in its resource use. So, in our tests, we have used a COPLA (JAJ$^+$02; IMDB03) prototype, a replication engine that uses *constant interaction*, instead of the *linear interaction* approach used in RJDBC, but COPLA requires that some metadata are written in the replicated database. So, we have tested there the differences between a full-featured replication engine and RJDBC. Additionally, another system used in the tests is a non-replicated database, accessed directly using JDBC.

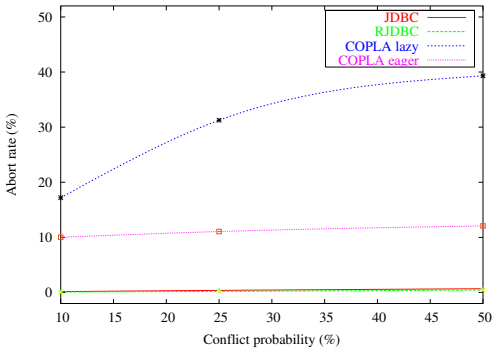
Tests results are shown in figure 3. The first subfigure shows the productivity rate of each system, measured in transactions per second. The best results correspond to directly accessing the DBMS via JDBC, executing up to four times more transactions than the RJDBC system. The overload of RJDBC is evident, although not excessive, since two communications layers (RMI and JavaGroups) are being used. However, RJDBC throughput is substantially higher than COPLA, standing for an alternative in clustered systems.

The second subfigure shows the abort rate of each system. As it can be found the RJDBC engine beats COPLA, providing a much lower abort rate. As expected, RJDBC and JDBC throw similar abort rates, since in both systems operations should be executing in a similar order.

In conclusion, RJDBC provides higher performance than more complex systems in not very big controlled environments, with negligible deployment effort. It also supplies similar abort rate than accessing the database directly, via JDBC, obviously with lower performance.



(a) Transactions per second results



(b) Abort rate results

Figure 3: Systems performace results

4 RELATED WORK

Much work about database replication is based on the use of efficient atomic broadcast primitives (GHOS96; PGS98; PJKA00). However, most of this work uses a constant interaction approach, only needing update propagation at the transaction commit phase. This is a very good approach to enhance the replication protocol performance, but usually it also implies a more ellaborated protocol which may require more resources. RJDBC uses linear interaction to propagate the updates, but it does not require any metadata information on disk, nor in memory, in order to decide about transaction completion. Thus, RJDBC may be an adequate solution when the available memory and processor are limited.

5 CONCLUSIONS

RJDBC claims to be a simple solution for enterprises which require database replication capabilities but do not want to invest much money and time in deploying complex, full-featured systems. On the contrary, RJDBC is extremely easy to install and setup, and fits comfortably into the Java database accessing architecture.

RJDBC provides an extensible lightweight platform, with tolerable performance results in clustered environments. Besides, depending on the underlying database management system, RJDBC fine-tunning could improve such results, resolving every instant if an operation must be spread over all the nodes or not.

The current RJDBC prototype does not integrate fault tolerance, but only replication capabilities. However, it contains all required services for easy integration with a recovery protocol. In fact, an alpha version of it is currently being developed.

REFERENCES

J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, Canada, 1996.

V. Hadzilacos and S. Toueg. *Fault-tolerant broadcasts and related problems*, chapter 5, pages 97–145. Addison Wesley, 2nd edition, 1993.

L. Irún, F. Muñoz, H. Decker, and J. M. Bernabéu. Copla: A platform for eager and lazy replication in networked databases. In *5th Int. Conf. Enterprise Information Systems (ICEIS'03)*, volume 1, pages 273–278, April 2003.

J.Esparza Peidro, A.Calero, J.Bataller, F.Muñoz, H.Decker, and J.Bernabéu. Copla - a middleware for distributed databases. In *3rd Asian Workshop on Programming Languages and Systems*, pages 102–113, 2002.

JavaGroups. JGroups web site. Accessible in URL: *http://www.javagroups.com*, 2004.

F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of EuroPar'98*, 1998.

M. Patiño, R. Jiménez, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of 14th IEEE DISC*, pages 315–329, 2000.

L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *Proc. of the First Eurasian Conference on Advances in Information and Communication Technology, Teheran, Iran*, October 2002.

M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE SRDS*, pages 206–217, October 2000.