Replication Tools in the MADIS Middleware *

L. Irún-Briz[1], J. E. Armendáriz[2], H. Decker[1], J. R. González de Mendívil[2], F. D. Muñoz-Escoí[1]

[1]Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
46022 Valencia, SPAIN
{lirun, hendrik, fmunyoz}@iti.upv.es

[2]Depto. de Matemática e Informática
Universidad Pública de Navarra
31006 Pamplona, SPAIN
{enrique.armendariz, mendivil}@unavarra.es

Abstract

To deal with consistency in replicated database systems, particularities of the chosen target environment and applications must be considered. To this end, several replication protocols have been discussed in the literature, each one requiring a different set of data to be maintained for each replicated object or for each transaction being executed. For instance, many protocols need to collect the writeset of each transaction.

In this paper, we describe the MADIS middleware architecture and its support for transaction management provided to its replication protocols.

1 Introduction

There are two approaches for building a replication support for databases in a *share-nothing* case. The first one consists in adding or modifying some components of the DBMS core at each replica. Its main advantages are a good performance, since the DBMS may provide direct access to the information needed by replication protocols, and that the solutions to be provided require the addition of a minimal amount of code. On the other hand, this approach also implies that the internal architecture of the target DBMS has to be carefully studied and the resulting solution would be DBMS-dependant, i.e., it is difficult to port such a solution to other DBMSes.

The second approach tries to implement the replication support in a middleware. In this case, some performance will be lost, since the information needed by the replication protocol regarding transactions and accessed data items has to be obtained using the standard API provided by the DBMS; i.e., using SQL. Thus, many optimisations that are available in the previous approach cannot be used in a middleware. However, this second case also provides some advantages, being portability the most important one.

There are many examples of systems that have provided replication support using one of these approaches. For instance, the Dragon [4] and Escada [20] projects required both some changes to the DBMS core in order to provide its replication support. The middleware alternative has also been used by many research groups, like the Distributed Systems Laboratory of the Technical University of Madrid [12], and projects, like C-JDBC [13], or GlobData [8]. Additionally, projects like GORDA [21] are developing replication support for these two environments. This is very interesting since it will allow a direct comparison between both approaches.

This paper describes some components of the replication support provided in the MADIS middleware [9], a replication solution based on the second approach described above. This middleware is being implemented in Java and provides a JDBC interface to its client applications. Its current release works on top of PostgreSQL, but uses only a JDBC interface and some stored procedures and triggers to access the database. So, it will be easily portable to other DBMSes. Some parts of our architecture are described in the following sections, particularly the support needed for collecting transaction writesets and for notifying the middleware when a transaction gets blocked and for allowing the termination of ongoing transactions.

The rest of the paper is structured as follows. Section 2 describes the structure and functionality of MADIS. Section 3 describes the schema modification that MADIS proposes to aid a local consistency manager (CM). Section 4 outlines a Java implementation of the CM, in the form of a standard JDBC driver. In section 5 a performance analysis is included, presenting a comparative study of a PostgreSQL database with the MADIS schema modification. Section 6 compares our approach with other systems and section 7 summarises the paper.

2 The MADIS Architecture

The MADIS architecture is composed by two main layers. The bottom one (or *MADIS DBlayer*) generates some extensions to the relational database schema, adding some fields in some relations and also some tables to maintain the collected writesets and (optionally) readsets of each transaction. These columns and tables are automatically filled by some triggers and stored procedures that must be installed, but they only use standard SQL-99 features and

*This work has been partially supported by the Spanish grant TIC2003-09420-C02 and EU grant FP6-2003-IST-2-004152.

can be easily ported to different DBMSes. Thus, the application layer will see no difference between the MADIS JDBC driver and the native JDBC driver.

The top layer is the *MADIS Consistency Manager* (CM) and is composed by a set of Java classes that provide a JDBC-compliant interface. These classes implement the following JDBC interfaces: Driver, Connection, Statement, CallableStatement, ResultSet, and ResultSetMetaData. They are used to intercept all invocations that could be relevant for a database replication protocol. The invocations made on other interfaces or operations are directly forwarded to the native JDBC driver (the PostgreSQL one, in our case). Besides these classes there exists a *Core* class (or *RepositoryMgr*) that is also able to provide a skeleton for this layer that maintains the rest of classes and gives also support for parsing the SQL sentences in order to modify them in some cases.

The database replication protocol (or *consistency protocol*, on the sequel) has to be plugged into this CM, and it has to provide a *ConsistencyProtocol* interface to the CM, and it may implement some *Listener* interfaces in order to be notified about several events related to the execution of a given transaction. This functionality will be described in section 4.

Figure 1 shows the overall layout of the MADIS architecture.

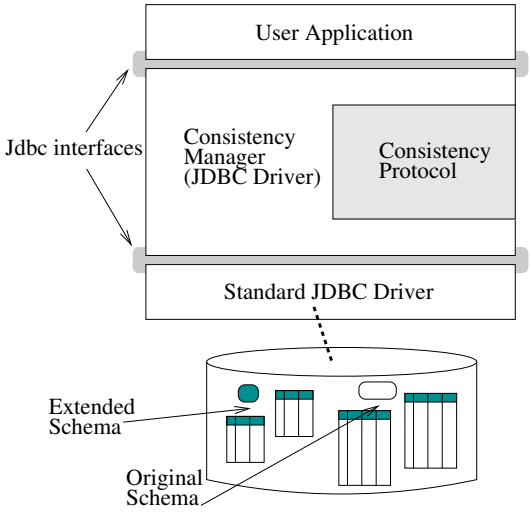


Figure 1: MADIS architecture.

Take into account that the consistency protocol can also gain access to the incremented schema of the underlying database to obtain information about transactions, thus performing the actions needed to provide the required consistency guarantees. The consistency protocol can also manipulate the incremented schema, making use of the provided database procedures when needed.

Finally, this protocol is also responsible of managing the communication among the database replicas. To this end, it has to use some group communication toolkit that provides several kinds of multicast operations. Our current protocols need at least a FIFO reliable multicast, plus a FIFO atomic, and also uniform variants of them, according to the multicast descriptions given in [6].

3 The MADIS DBlayer

The MADIS DBlayer is an extension of the original schema of a given database that provides the following items:

- Metadata information is collected in a second table for each one of the original database tables. This additional table maintains a global identifier for its associated original record, a version number, the identifier of the transaction that has generated the latest version of such an item, and the timestamp for this latest update. This additional table is called *MADIS_Meta_*T_j if the original table was T_j.

- A global *TrReport* table (or per-transaction, depending on the overall load) is also needed to collect the writeset of each transaction. The contents of this table are automatically filled by a set of database triggers that are executed when an *update*, *delete*, or *insert* operation is made by the target transaction. These triggers are disabled when the transaction being managed corresponds to the application of a remote update on the local database replica. To this end, the consistency protocol has to set a flag when a transaction is initiated in order to avoid the use of such triggers.

A detailed description of these two kinds of tables is provided in [10], we only give now a minimal description needed to explain the overall helping mechanisms provided by our middleware for developing replication protocols. These mechanisms are the writeset collection, the detection of conflicts between transactions, and the mechanisms needed for cancelling ongoing transactions; i.e., rolling them back.

3.1 Writeset Collection

As stated above, MADIS introduces a set of new triggers in the database schema definition. Some of these triggers are devoted to the generation of metadata information as, for instance: (i) version numbers that are increased each time an update has been made, (ii) setting timestamps, or (iii) writing the identifier of the latest updating transaction. However, these metadata updating triggers are quite trivial, and the interested reader could refer again to [10] to get a thorough description of them.

The writeset collection is performed defining three triggers for each table T_i in the original schema. They insert in the *TrReport table* the information related to any write-access to the table performed by the executing transactions.

The writeset collector (WSC) triggers are named WSC_I_T_i, WSC_D_T_i, and WSC_U_T_i, and its definition allows to intercept any write access (insert, delete or update respectively) to the T_i table, recording the event in the transaction report table (*TrReport*).

The following example shows the definition of a basic WSC_I trigger, related to the insertion of a new object. Note that the trigger executes the procedure getTrid() to obtain the current transaction identifier. The example inserts a single row in the *TrReport* table for each insertion in the table *mytable*. The execution of the invoked procedure causes the DBMS to insert in the *TrReport* table the adequate rows, in order to keep track of the transaction activities.

CREATE TRIGGER WSC_I_mytable
BEFORE INSERT ON mytable
FOR EACH ROW EXECUTE
PROCEDURE tr_insert(mytable,
getTrid(), NEW.l_mytable_oid);

Deletions and updates must also be intercepted by means of similar triggers. Note also that the actual insertion of the data into the *TrReport* table is made by a stored procedure called tr_insert().

Finally, when an object is deleted, the corresponding metadata row must be also deleted. To this end, an additional trigger is also included for each table in the original schema.

3.2 Detecting Transaction Conflicts

In many database replication protocols we may need to apply the updates propagated by a remote transaction. If several local transactions are accessing the same data items that this remote update, such remote update will remain blocked until those local transactions terminate. Moreover, if the underlying DBMS uses a multiversion concurrency control combined with a *snapshot isolation* level [2], such a remote update is commonly aborted, and it has to be reattempted until no conflicts arise with any local transaction. Additionally, in most cases, the replication protocol will end aborting also such conflicting local transactions once they try to commit. As a result of this, it seems appropriate to design a mechanism that notifies to the replication protocol about conflicts among transactions, at least when the replication protocol requires so. Once notified, the replication protocol will be able to decide which of the conflicting transactions must be aborted and, once again, a mechanism has to be provided to make possible such abortion.

To this end, we have included in the MADIS DBlayer some support for detecting transaction conflicts that have produced a transaction blocking. It consists of the following elements:

- A stored procedure named getBlocked() that looks for blocked transactions in the *pg_locks* view placed in the PostgreSQL system catalog. It returns a set of pairs composed by the identifier of a blocked transaction and the identifier of the transaction that has caused such a block.

- An execution thread per transaction that is used each time its associated transaction begins any operation that might be blocked due to the concurrency control policy of the underlying DBMS. Take into account that in multiversion DBMSes the read-only operations cannot be blocked.

Thus, once a database connection is created, a thread is also created and associated to it. Each time the current transaction in a given connection initiates an updating operation, its associated thread is temporarily suspended, with a given timeout. If such an updating operation terminates before that timeout has expired, the thread is awakened and nothing else needs to be done. On the other hand, if the timeout is exhausted and the operation has not been concluded, the thread is reactivated and then makes a call to the *getBlocked* procedure. As a result, the replication protocol is able to know if the transaction associated to this thread is actually blocked and which other transaction has caused its stop.

This mechanism can be combined with a transaction priority scheme in the replication protocol. The O2PL [3] BULLY variation described in [1] uses this priority scheme as follows. Three priority classes are defined, with values 0, 1, and 2. Class 0 is assigned to local transactions that have not started their commit phase. Class 1 is for transactions that have started their commit, but whose updates have not yet been delivered in the local node. Finally, class 2 is assigned for those transactions associated to delivered writesets that have to be locally applied. Once a conflict is detected, if the transactions have different priorities, then that with the lowest priority is aborted. Otherwise, i.e., when both transactions have the same priority, both of them are allowed to proceed. This replication protocol [1] is an update everywhere, constant interaction, and voting protocol, following the classification given by [22]. Similar approaches may be followed in other replication protocols that belong to the UE-CI (*update everywhere* with *constant interaction*) class.

3.3 Transaction Termination

A replication protocol may abort an ongoing transaction cancelling all its statements. This implicitly rollbacks such a transaction, and may be requested using standard JDBC operations. If the transaction is currently executing a statement, it may be aborted using another thread to request such a cancellation.

4 Consistency Manager

The current Java implementation of the MADIS consistency manager allows a pluggable consistency protocol to intercept any access to the underlying database, in order to coordinate both local accesses, and update propagation of committed local transactions (and, consequently, the local application of remotely initiated transactions).

In our basic implementation of MADIS, we implement the consistency manager as a JDBC driver that encapsulates an existing PostgreSQL driver, intercepting the requests performed by the user applications. The requests

are transformed, and a new request is elaborated in order to obtain additional information (as metadata). The user perception of the result produced by the requests is also manipulated, in order to hide to the user applications the additionally recovered information. This mechanism allows the plugged replication protocol to be notified about any access performed by the application to the database, including query execution, row recovery, transaction termination requests (i.e. commit/rollback), etc. The protocol then has a chance to take specific actions during the transaction execution, in order to accomplish its tasks. To this end, our consistency manager has a set of classes that implement the following JDBC standard interfaces: *Driver*, *Connection*, *Statement*, *CallableStatement*, *PreparedStatement*, *ResultSet* and *ResultSetMetaData*. All these classes provide the support needed by the replication protocol. Some of the transformations that may request the protocol may imply a modification of the sentence to be sent to the database. This is accomplished using a parsing tree that can be easily modified using a special interface.

4.1 Protocol Interface

The interaction between our consistency manager and the plugged replication protocol is ruled by an interface with operations to complete the following tasks:

- **Protocol registration**. The protocol has to be plugged into the consistency manager using a registration method. In this registration procedure it has to specify with a parameter the set of events it is interested in. Some of these events depend on the information that has been put into the *TrReport* that was described in section 3. The available events are:

  1. RECOVERED: Some objects have been recovered in a ResultSet. The protocol will receive an extended ResultSet that also contains the OIDs of the objects being recovered, and may use this information for building the transaction readset, if needed.
  2. UPDATED: This event is similar to the previous one, but reports the objects that have been updated, instead of those that were read.
  3. UPDATE_PRE: The protocol will be notified when the current transaction is going to initiate an updating operation on the database. Thus, the protocol may modify the update sentence at will, if needed.
  4. UPDATE_POST: The protocol will be notified after an update sentence has been executed. Thus, it may read the current transaction report for obtaining the set of updated objects. This is an alternative way of doing the same as in the event number 2 described above.
  5. QUERY_PRE: The protocol will be notified before a *select* operation is initiated in the database. It may modify the query, if needed.
  6. QUERY_POST: The protocol will be notified once a query has been completed. It may access then the transaction report, if needed.
  7. ACCESSED: The protocol will get all the objects accessed by the latest SQL sentence, instead of the objects being recovered in its ResultSet.
  8. TREE: The protocol requests that the consistency manager builds a parsing tree for each sentence being executed. Later, the protocol may ask for such a tree, modifying it when needed.

- **Event requesting**. There are also a set of explicit operations that the protocol may use for requesting those events that were not set at protocol registration time.

- **Event cancellation**. A set of operations for eliminating the notification of a given event to the currently plugged-in protocol.

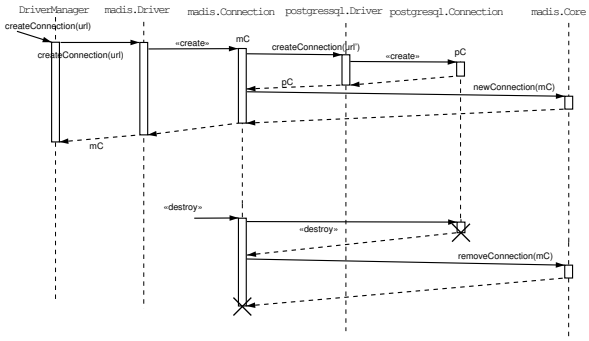
- **Access to transaction writeset and metadata**. A set of operations that allow the full or partial recovery of the current writeset or metadata for a given transaction. Most of the protocols will need the transaction writeset only at commit time in its master node, for its propagation to the rest of replicas, but others may need such data before and these operations allow this earlier recovery, too.

This interface is general enough to implement most of the replication protocols currently developed for databases.

4.2 Connection Establishment

In the figure 2, a UML sequence diagram is shown describing how a new MADIS connection is obtained.

Figure 2: Connection Establishment



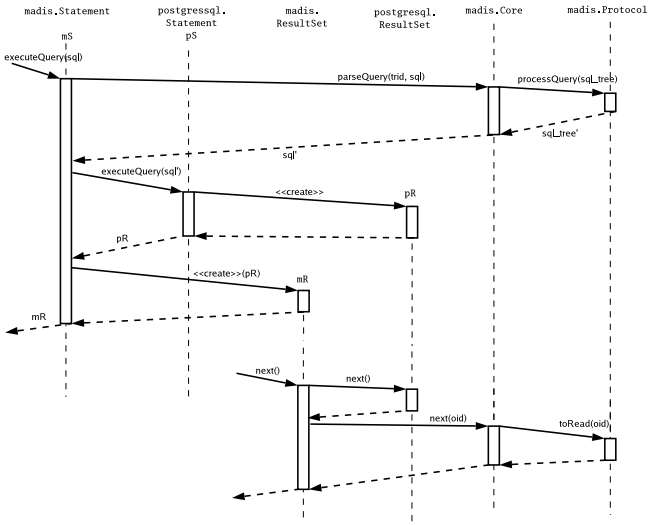
The sequence starts with a request to the DriverManager, and the selection of the MADIS JDBC Driver. Then, the MADIS Driver invokes the MADIS Connection to be built, indicating the underlying PostgreSQL connection URL to be used. The constructor of the MADIS Connection builds a PostgreSQL Connection, and includes it as an attribute. Finally, the MADIS Driver returns the new MADIS Connection.

4.3 Common Query Execution

Application query executions are also intercepted by MADIS, by means of the encapsulation of the Statement class. As response of user invocations to "createStatement" or "prepareStatement" the MADIS Connection generates Statements that manage user query execution. When the user application requests a query execution, the request is sent to the consistency manager, which may call the processStatement() operation of the plugged consistency protocol if it previously requested any of the *_PRE or TREE events.

Now, the consistency protocol may modify the statement, adding to it the patches needed to retrieve some metadata, or collect additional information into the transaction report. However, this statement modification is only needed by a few consistency protocols, which also have the opportunity to retrieve these metadata using additional operations once the original query has been completed. Optimistic consistency protocols do not need such metadata (like current object versions, or the latest update timestamps for each accessed object) until the transaction has requested its commit operation. So, they do not need these statement modifications on each query. The process for queries is depicted in figure 3.

Figure 3: Query Execution



We recommend to access the metadata using a separate query. Otherwise, the following additional steps are needed:

1. The resulting SQL statement is executed, performing a common invocation to the encapsulated JDBC Statement instance, and a ResultSet is obtained as a response. The obtained ResultSet is also encapsulated by MADIS, returning to the user application an instance of a MADIS ResultSet. This MADIS ResultSet contains the ResultSet returned by the JDBC Statement.

2. When the application tries to obtain a new record from the ResultSet, MADIS intercepts the request, and notifies about the new obtained object to the Core class. This allows MADIS to notify the plugged protocol about the row recovery. Consequently, in order to keep the required guarantees, the protocol may modify the database, the state of the MADIS ResultSet, or even abort the current transaction. In addition, the MADIS ResultSet tasks also include the "hidding" of the metadata (included in the query) when the application requests the different fields of the current row.

4.4 Commit/Rollback Requests

The termination of a transaction is also requested by the user application. Either when the application requests a *commit* or when a *rollback* is invoked, MADIS must intercept the invocation, and take additional actions.

When the user application requests a commit operation (see Figure 4), the MADIS Connection redirects the request to the MADIS Core instance. Then, the plugged protocol is notified, having then the chance to perform any action involving other nodes, access to the local database, etc.

If the protocol concludes this activity with a positive result, then the transaction is suitable to commit in the local database, and the MADIS Core responds affirmatively to the Connection request. Finally, the MADIS Connection completes locally the commit, and returns the completion to the user application after the notification to the MADIS Core using the *doneCommit()* operation. On the other hand, a negative result obtained from the protocol activity will be notified directly to the application, after the abortion of the local transaction.

Take into account that the *doneCommit()* method is also able to notify a unilateral abort, generated by the underlying database, and that this may allow that the plugged protocols were able to manage such unilateral aborts, too. This is the case of the BULLY protocol described in [1].

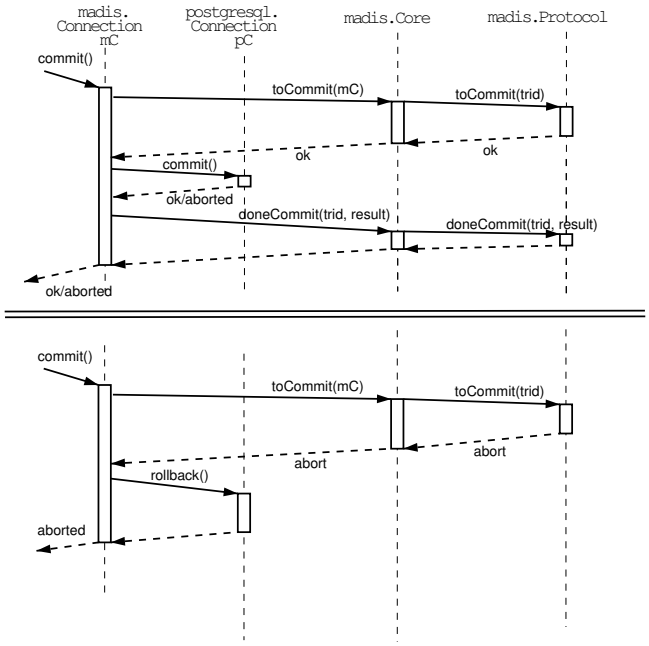
Finally, *rollback()* requests received from the user application must be also intercepted, redirected to the MADIS Core, and notified to the plugged protocol.

5 Experimental Results

As presented above, the proposed architecture is based on the modification of the database schema of an existing information system. With this technique, the database manager is the main responsible for generating and maintaining the information needed by any pluggable replication protocol to accomplish the tasks of consistency maintenance, concurrency control, and update propagation.

However, an important question to be discussed is the cost to be paid by the system from obtaining such benefits. This question, for our architecture, corresponds to the degree of performance degradation when we consider different types of accesses.

Figure 4: Commit suceeded vs aborted



5.1 Overhead Description

In spatial terms, the overhead introduced by the schema modification may be easily determined. Considering the trigger and procedure definitions as negligible, the main overload in space is produced by the *MADIS_Meta_*T_j tables. These tables contain at least two identifiers (local and global object identifier) and the rest of fields are used by each one of the pluggable protocols. We consider that many protocols can be implemented with the support of a transaction identifier, a timestamp, and a sequential version number. Finally, the transaction report maintains the information regarding the executed transactions just during the lifetime of such transactions. Thus, in global terms, this does not constitute a spatial overhead by itself.

Regarding computational overhead, our architecture introduces a number of additional SQL sentences and computations for each access to the database.

This overhead can be classified into four main categories:

- **Insertion**. The overhead is mainly caused by the insertion of a row into the *TrReport* table for registering such insertion. An additional row is also inserted in the *MADIS_Meta_*T_j. Thus, for each row inserted in the original schema, two additional rows are inserted by the schema extension.

- **Update**. When updating a row of the original schema, there will be inserted an additional row in the *TrReport* table. However, in this case there will not be needed to insert into the *MADIS_Meta_*T_j table any row, but just an update.

- **Deletion**. In this case, an additional row must be inserted in the *TrReport* table to register the deletion, and the deletion of the corresponding row in *MADIS_Meta_*T_j should be also deleted (although in a deferred mode).

- **Selection**. When selecting a row from the original schema, there is no need to alter the *MADIS_Meta_*T_j table at all. In addition, depending on the particular replication protocol plugged in the system[1], it can also be avoided any insertion in the *TrReport* table.

Summarizing, *Insertion, Update* and *Deletion* need additional insertions on the *TrReport* table, and other operations with the corresponding *MADIS_Meta_*T_j table. In contrast *Selection* overhead varies depending on the plugged protocol. Since many database replication protocols do not need the transaction readsets, readset collection will not be analysed here.

5.2 Performance Results

The experiments consisted in the execution of a Java programme, performing database accesses via JDBC. The schema used by the programme contains four tables (CUSTOMER, SUPPLIER, ARTICLE, and ORDER). Each article references a row in the SUPPLIER table, and each ORDER references a CUSTOMER row, as well as an ARTICLE row. Each table contains additional fields as item description (a varchar[30]).

A programme execution starts with the database connection, and schema creation. Then, a number of "training" transactions are executed, ensuring that all Java classes are loaded, and then three measurements are done. Each measurement calculates the time taken by *numtr* sequential transactions (performing a number of INSERTIONS, UPDATES or DELETIONS depending on the required measurement).

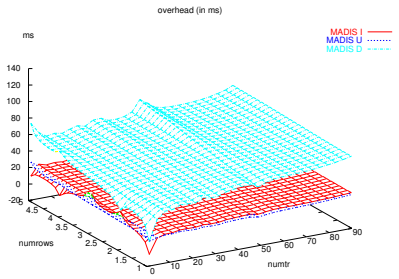
For each measurement, the experiment provides three values: the total cost of the *numtr* transactions of type I, U and D respectively, each one acting with *numrows* rows per table. These performance tests have been taken in a system with an Intel Pentium 4 processor at 2.8 GHz, with 1 GB of RAM, and a hard disk of 7200 rpm with an average seek time of 8.5 ms, running a Fedora Core 2 operating system. The DBMS is PostgreSQL 7.4.1.

We observed that deletions are the most overheaded operations in our core implementation. To determine with a more descriptive sense such overhead, we calculated the times per transaction (figures 5(a) and 5(b)).

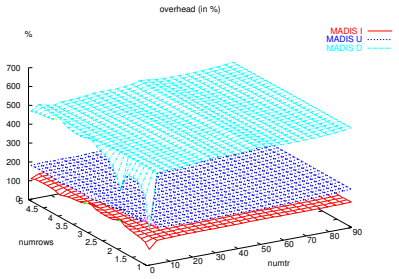
The results stabilised with a few of transactions, which indicates that the system does not suffer appreciable performance degradation along time. In addition, it is shown in figure 5(a) that the overhead per transaction is always lower than 80 ms in our experiments. In addition, figure

[1]Replication protocols just based on the writeset will not need records about the objects read by a transaction.

Figure 5: Mean Overhead



(a) Absolute (ms) Overhead



(b) Relative (%) Cost

5(b) shows that the sensibility for *numrows* is unappreciable (the system scales well in respect to managed rows) for any of the transaction types (I,U, and D).

We conclude that our implementation of the MADIS database core introduces bounded overheads for Insertion and Update operations. On the other hand, Delete operations imply a cost that is 6 times higher than in a native JDBC driver.

6 Related Work

As already outlined in section 1 there are multiple approaches to implement a support for database replication. The best option for getting a good performance is to modify the DBMS core, like in the Postgres-R [11], Dragon [4] and Escada [20] projects. However, such solutions cannot be seamlessly ported to other DBMSes.

Several examples of middleware approaches can be found in the literature:

- GlobData [17, 8] is a middleware providing a subset of the standard ODMG API for Java applications. The system also included a heavy Relational-Objectual transformation. This allows the applications to make use of an object-oriented database schema, and the system translates this schema to a relational database. The system, although allows multiple consistency protocols to be plugged into, provides a propietary API for the applications to gain access to distributed databases, reducing the generality of the solution.

- Other specific solutions for Java, implemented as a JDBC driver: like C-JDBC [13] and RJDBC [5]. The former emphasizes load balancing issues, whilst the latter puts special attention to reliability. The implementation of these approaches are centred in Java, and porting the solution to other platforms has a high complexity, due to the characteristics of the specific techniques.

- PeerDirect [14] uses a technique based on database triggers and procedures to replicate a database. However, the system only includes one consistency protocol, providing particular guarantees, well fitted for a limited kind of applications.

- Other papers [12] have focused on replication protocols that could be easily implemented in a middleware.

Besides this, another good characteristic of these middleware solutions is that they provide some interface for replication protocols, and multiple protocols can be designed and tested on them. A future work in the MADIS project will be the design and implementation of replication protocols for mobile databases, or the implementation and testing of some well-established solutions in this research area [15, 16, 7]. These protocols are specially appropriate for partitionable environments, and they could be compared with the hybrid replication and reconciliation protocols being designed in the DeDiSys project [19, 18].

DeDiSys is a research project focused on the trade-off between availability and consistency in partitionable distributed systems[18]. It uses a synchronous replication model in a healthy system and an asynchronous one when failures arise, so its replication protocols could be considered as *hybrid*. Additionally, when partitions are merged a reconciliation protocol is needed to bring the system to a consistent state. MADIS could be used as a persistent storage layer for a DeDiSys system if special replication protocols were implemented on it. At least, we will be able to compare the DeDiSys-specific replication protocols (specially tailored for a consistency model based on constraints, and dealing with object replication instead of data replication), with those designed for mobile databases in MADIS.

7 Conclusions

MADIS is a middleware designed to give support to a wide range of replication protocols, using a minimal database schema extension and some triggers, stored procedures and rules in order to collect the metadata needed by such protocols.

The MADIS consistency manager makes use of the automatically collected information in the database, notifying

such accesses to a plugged replication protocol. It is possible to include a wide range of protocols in the system, each one providing different guarantees and behaviours to the user transactions. The implementation of this upper layer is simple enough to be ported from one platform to another with a minimal cost.

In this paper, we have described the MADIS architecture and its current implementation. This implementation allows user applications to access in a standard way a replicated database without needing to include changes in their code.

In the future we plan to use the MADIS middleware for testing several replication protocols, particularly some simplifications of the object replication and reconciliation protocols designed in the DeDiSys project, and other protocols for mobile databases.

References

[1] J. E. Armendáriz, J. R. Juárez, I. Unzueta, J. R. Garitagoitia, F. D. Muñoz-Escoí, and L. Irún-Briz. Implementing replication protocols in the MADIS architecture. In *Proc. of the XIII Jornadas de Concurrencia y Sistemas Distribuidos*, Granada, Spain, September 2005. *Accepted for publication*.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–10, San Jose, CA, USA, May 1995.

[3] M. J. Carey and M. Livny. Conflict detection tradeoffs for replicated data. *ACM Trans. Database Syst.*, 16(4):703–746, 1991.

[4] École Polytechnique Fédérale de Lausanne. Dragon project web page, 2003. Accessible in URL: *http://lsrwww.epfl.ch/~dragon*.

[5] J. Esparza-Peidro, F. D. Muñoz-Escoí, L. Irún-Briz, and J. M. Bernabéu-Aubán. RJDBC: A simple database replication engine. In *6th Int. Conf. Enterprise Information Systems (ICEIS'04)*, April 2004.

[6] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press, 2nd edition, 1993. ISBN 0-201-62427-3.

[7] J. Holliday, D. Agrawal, and A. El Abbadi. Disconnection modes for mobile databases. *Wireless Networks*, 8(4):391–402, July 2002.

[8] Instituto Tecnológico de Informática. GlobData web site. Accessible in URL: *http://globdata.iti.es*, 2002.

[9] Instituto Tecnológico de Informática. MADIS web site. Accessible in URL: *http://www.iti.es/madis*, 2005.

[10] L. Irún-Briz, H. Decker, R. de Juan-Marín, F. Castro-Company, J. E. Armendáriz, and F. D. Muñoz-Escoí. MADIS: a slim middleware for database replication. In *Proc. of the 11th Intnl. Euro-Par Conf.*, Monte de Caparica (Lisbon), Portugal, September 2005. Springer.

[11] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2000.

[12] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware-based data replication providing snapshot isolation. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Baltimore, Maryland, USA, June 2005.

[13] ObjectWeb. C-JDBC web site. Accessible in URL: *http://c-jdbc.objectweb.org*, 2004.

[14] PeerDirect. Overview & comparison of data replication architectures (white paper), November 2002.

[15] S H. Phatak and B. R. Badrinath. Multiversion reconciliation for mobile databases. In *Proc. of the 15th International Conference on Data Engineering*, pages 582–589, March 1999.

[16] N. Preguiça, C. Baquero, J. L. Martins, F. Moura, H. Domingos, R. Oliveira, J. O. Pereira, and S. Duarte. Mobile transaction management in Mobisnap. *Lecture Notes in Computer Science*, 1884:379–386, 2000.

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

[18] R. Smeikal and K. M. Göschka. Trading constraint consistency for availability of replicated objects. In *Proc. of 16th Intl. Conf. on Parallel and Distributed Computing and Systems*, pages 232–237, 2004.

[19] Technical University of Vienna. DeDiSys project web page, 2005. Accessible in URL: *http://www.dedisys.org/*.

[20] Universidade do Minho. Escada project web page, 2003. Accessible in URL: *http://escada.lsd.di.uminho.pt/*.

[21] Universidade do Minho. GORDA project web page, 2005. Accessible in URL: *http://gorda.di.uminho.pt/*.

[22] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 206–217, October 2000.