MITRA: Byzantine Fault-Tolerant Middleware for Transaction Processing on Replicated Databases

Aldelir Fernando Luiz
Federal University of Santa Catarina – Brazil
aldelir.luiz@posgrad.ufsc.br

Lau Cheuk Lung
Federal University of Santa Catarina – Brazil
lau.lung@ufsc.br

Miguel Correia
University of Lisboa – IST, INESC-ID – Portugal
miguel.p.correia@tecnico.ulisboa.pt

ABSTRACT

Replication is often considered a cost-effective solution for building dependable systems with off-the-shelf hardware. Replication software is usually designed to tolerate crash faults, but Byzantine (or arbitrary) faults such as software bugs are well-known to affect transactional database management systems (DBMSs) as many other classes of software. Despite the maturity of replication technology, Byzantine fault-tolerant replication of databases remains a challenging problem. The paper presents MITRA, a middleware for replicating DBMSs and making them tolerant to Byzantine faults. MITRA is designed to offer transparent replication of off-the-shelf DBMSs with replicas from different vendors.

1. INTRODUCTION

For many years, relational database management systems, often called simply *database management systems* (DBMSs), have been a key component of applications of a wide-range of business areas. We believe this will continue to be true for many applications for years ahead. Applications based on DBMSs usually rely on the reliability of transaction processing and on the availability of the data stored in the database. Therefore, fault tolerance is a desirable property for DBMSs.

Replication is a well-known approach to make services fault-tolerant, which has already been applied to DBMSs [8, 18]. The idea is that the service is executed in a set of servers in such a way that if some of them fail, the service as a whole stays operational and clients continue to be able to execute transactions. However, DBMS vendors usually do not provide native support for replication or hooks for third-party replication protocols. This puts on third-parties the burden of either modifying DBMS source code (if available) or to develop middleware that intercepts client requests and delivers them to the servers. The latter is the approach followed in this paper.

The replication of databases has been studied both in the databases and distributed systems research communities. Although Gray commented that it can be hard to achieve strong consistency in replicated databases [8], Schiper and Raynal have shown that transactions on replicated databases have common properties with group communication primitives such as ordering and atomicity [15]. After that result, several researchers studied the use of group communication systems and middleware to support database replication [12, 4, 10, 16, 6, 13]. Most of the solutions for database replication tolerate only crash faults [12, 4, 10]. Although these are arguably the most common faults, Byzantine or arbitrary faults are also common in today's systems. Faults such as data corruption in disk or RAM due to physical effects or in software due to bugs are Byzantine faults, not crashes. Interestingly, many bugs have historically been found in DBMSs [7].

In the literature there are a few proposals of Byzantine fault-tolerant (BFT) protocols to replicate databases [7, 16, 6, 13]. However, they are focused on a specific problem, based on assumptions hard to substantiate in practice, or simply not the best for certain applications. Specifically, [7] does not allow concurrent transactions; HRDB [16] depends on a centralized controller; Byzantium [6] adopts a consistency criterion that may cause anomalies violating the semantics of some applications (snapshot isolation); and BFT-DUR [13] does neither handle relational databases nor Byzantine clients.

This paper presents the design of a middleware for BFT database replication, MITRA (Middleware for Interoperability on TRAnsactional replicated databases). MITRA supports design diversity [14], i.e., different replicas can run different DBMSs. This is an important mechanism to avoid common mode failures caused, e.g., by a bug existing in all replicas. The middleware supports concurrent transactions, has no centralized components, and provides serializability. The paper does not delve into the details of the protocol at the core of the middleware, which has already been presented elsewhere [11], but on its

design and architectural aspects.

MITRA is modular in the sense that it does not require changes on the DBMSs and it encapsulates all complexity of the BFT replication. The middleware is written in Java and modularity is achieved by following the Java Database Connectivity (JDBC) specification. The rest of this paper is organized as follows: Section 2 discusses some concepts on database replication; Section 3 introduces MITRA and its design; in Section 4, we describe some implementation details and an experimental evaluation; Section 5 concludes the paper.

2. DATABASE REPLICATION

There are several taxonomies of database replication schemes in the literature [8, 17]. A particularly interesting one classifies protocols in terms of their update propagation strategy, i.e., of the way in which they make the state of the replicas converge. This taxonomy classifies database replication schemes in three classes: synchronous (or eager replication), asynchronous (or lazy replication), and certification-based.

A *synchronous* or *eager replication* protocol propagates the updates of a transaction by applying them on the replicas before the transaction commits [8]. More specifically, such a scheme provides strong consistency and fault tolerance by ensuring that updates are stable at multiple replicas before replying to the clients.

An *asynchronous* or *lazy replication* protocol executes and commits each transaction at a single replica, delaying the propagation of updates until the transaction has committed [8]. Lazy replication tends to perform better than eager replication because it avoids the communication overhead of the later during normal execution. However, lazy replication can let replicas diverge and lose the effects of some transactions [18].

A *certification-based* protocol uses group communication primitives such as total order multicast for ordering transactions and propagating write and read sets to the group of replicas [12, 18]. This approach is optimistic since reads and updates are first executed on a single replica without any synchronization with the rest, being a transaction committed only if there are no conflicting updates. MITRA follows this approach since it has shown good performance in the past [12, 10].

3. THE MITRA MIDDLEWARE

As explained, MITRA is a middleware that aims to tolerate Byzantine faults on database systems following the certification-based replication approach. It encapsulates the fault tolerance mechanisms by implementing the JDBC API specification, in order to provide a heterogeneous and replicated environment that appears to the clients as a single virtual DBMS. The ability to support DBMS diversity is important because different systems are unlikely to share the same bugs or vulnerabilities [7].

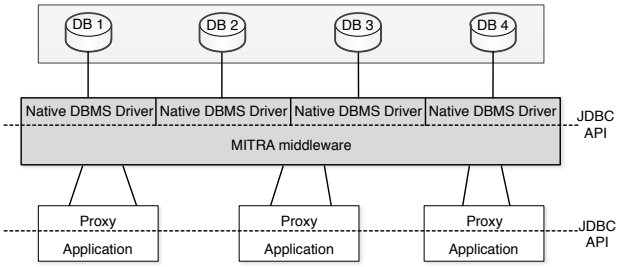


Figure 1: Basic architecture of MITRA

Figure 1 presents the basic architecture of MITRA. The client application interacts with the database through a proxy that exports the JDBC API and replaces what would normally be a DBMS-specific JDBC driver (e.g., a MySQL or a PostgreSQL JDBC driver). The middleware is implemented essentially on the server-side, meaning that the protocol is mostly executed by the servers where the database is replicated. The figure represents it abstractly in the form of a mid-layer, but there are server-side replication processes running in all the servers and communicating through the network. The middleware at each server makes calls to a DBMS driver – again a JDBC driver – that hides the specifics of the DBMS from the replication process. These drivers are readily available for a wide range of DBMSs (http://devapp.sun.com/product/jdbc/drivers). The use of different DBMSs in different servers provides diversity.

3.1 Assumptions

We consider a system composed of an arbitrary, finite, set of clients $C = \{c_1, c_2, ..., c_n\}$ and a set of n replicas $S = \{r_1, r_2, ..., r_n\}$. These entities communicate by message passing, through the network. We assume that an unlimited number of clients and up to $f = \lfloor \frac{n-1}{3} \rfloor$ replicas can be faulty, i.e., can deviate arbitrarily from their specification (Byzantine faults).

Let a database be a collection of data items $D = \{x_1, x_2, ..., x_n\}$. A transaction is a sequence of read and/or write operations on these data items, initiating by a *begin transaction* operation and ending with a *commit* or an *abort* operation. To support certification-based database replication, we make some assumptions about the replica DBMSs: (i) they are relational and transactional; (ii) they support rollback of operations; (iii) they implement

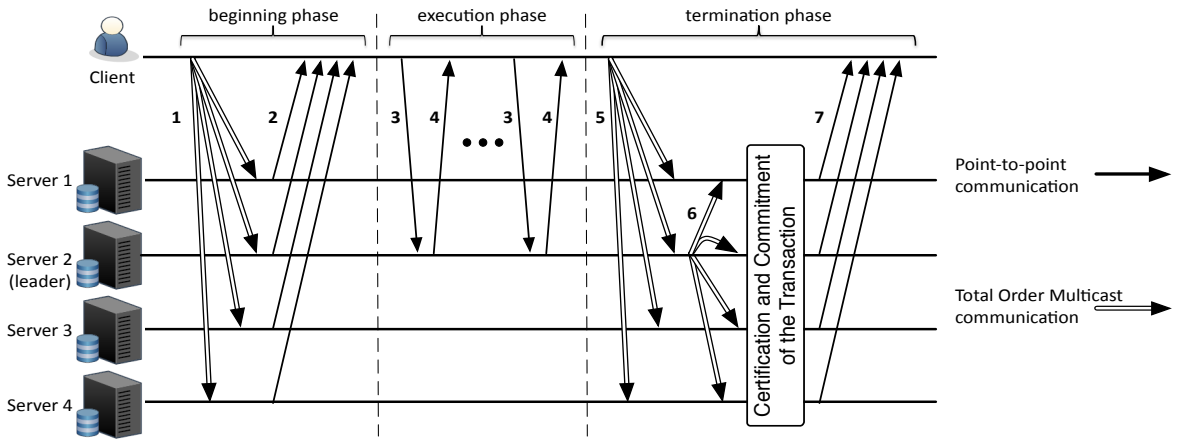


Figure 2: MITRA's replication protocol with its three phases.

strict two-phase locking and serializable isolation; (*iv*) statements modify the database atomically, without side effects.

As mentioned, MITRA is based on a group communication primitive. Specifically, MITRA uses a BFT *total order multicast* primitive based on a consensus protocol [3]. We make a weak assumption about the synchrony of the system to ensure the termination of this primitive: communication delays do not grow exponentially. This is required by the impossibility of solving consensus deterministically in asynchronous systems [5]. We also assume the existence of a collision-resistant cryptographic hash function and a message authentication function to ensure the integrity and authenticity of messages.

3.2 Replication Protocol

The replication strategy adopted in MITRA is an extension of the original certification-based replication scheme [18] to handle Byzantine faults. In this sense, MITRA operates by letting transactions execute first on a single replica in an optimistic way; when the termination of the transaction is requested, that replica total order multicasts the transaction's reads and writes to the rest of the replicas for certification and commitment. The use of this multicast primitive guarantees that all replicas execute these operations in the same order despite the existence of concurrency [1]. This section briefly presents the protocol (details in [11]).

The execution of MITRA's protocol for a single transaction is represented in the time diagram of Figure 2. For each transaction, a replica is selected to be the leader and the rest are followers. The protocol has three phases that we explain next: beginning, execution, and commitment.

Beginning. A transaction begins with a client opening a connection to the database. The client-side part of the middleware sends a *begin transaction* message to all replicas using the total order multicast primitive (step 1). Upon delivering this message, every replica applies a deterministic criterion to select a single replica to be the leader and execute the transaction's operations in a speculative way. Then, every replica sends an acknowledgement to the client with the identifiers of the leader and the transaction, concluding this phase (step 2).

Execution. Figure 2 illustrates this phase in steps 3 and 4, which are repeated for all the statements of the transaction. On receiving the begining-phase acknowledgment from at least $f + 1$ replicas, the client becomes aware of which replica is the transaction's leader. The client connects to the leader and submits its read and write operations to that replica (step 3). After receiving a statement, the leader executes it and returns the result to the client (step 4). Note that during this phase, there is no interaction among replicas since our protocol relies on optimistic concurrency control [9]; the other replicas do not even receive the statements. This phase ends when the client requests the commitment of the transaction.

Termination. To request commitment of a transaction, the client totally order multicasts a *request commit* message to all replicas (step 5). This message carries the statements issued in the transaction and a hash of the results received by the client. Upon delivering the *request commit* message, every replica changes the transaction to a state indicating that it is ready to commit. Then, the leader totally order multicasts to all replicas a message with the following data (step 6): a hash of the transaction's statements received and processed; a hash of the transaction's state; the transaction's readset and writeset. Upon delivering the *commit* message, all replicas verify the data in the *request commit*

and *commit* messages. If they match, every replica starts a *certification* test where it checks the validity of transaction's readset in the database. These checks are needed to guarantee the transaction integrity and validity, and to preserve the serializability. These checks also prevent a Byzantine client from either forging a transaction or committing a spurious transaction. If any of the checks fails, the transaction is aborted.

If the transaction passes these checks and there is a concurrent transaction (or more), the termination still involves another *certification* step. Consider two transactions, T_i and T_j. We say that T_i precedes T_j if the *commit* message for T_i is delivered by the replicas before they deliver the *request commit* message for T_j. We say that these transactions are concurrent if neither T_i precedes T_j nor T_j precedes T_i. The transaction T_i being committed passes the certification test iff for any concurrent transaction T_j, $readset(T_i) \cap writeset(T_j) = \emptyset$. When the transaction passes this test, we can be assured that it does not violate serializability, so its writes are applied in the databases at all replicas. The certifications test is deterministic so every replica will reach the same outcome for T_i.

The protocol and the transaction end when all non-faulty replicas reply to the client with the outcome of the transaction (step 7). The client accepts the outcome if it receives the same reply from at least $f+1$ distinct replicas. No more than f replicas are faulty, so this avoids that the client accept the outcome of a faulty replica that unilaterally commits a non-serializable transaction.

A final remark. In a serializable execution where T_j is executed before a concurrent transaction T_i, T_i would see all of T_j's writes. Our approach is conservative in the sense that T_i and T_j could be executed optimistically, concurrently, in different replicas, but the certification test allows T_i to commit only if T_i did not read any item written/updated by T_j (a read-write conflict and dirty read).

3.3 Middleware Components

Figure 3 presents a detailed architecture of the server-side of the middleware. Recall that the middleware runs at each server, so the representation of the middleware as a single box is an abstraction of reality. The client-side is not detailed as it is much simpler, e.g., it encapsulates calls to the database as messages to the replication protocol.

Client Connection Manager. The *client connection manager* is the interface with the clients, i.e., it is the component that receives messages from the clients and forwards them messages. It works at the

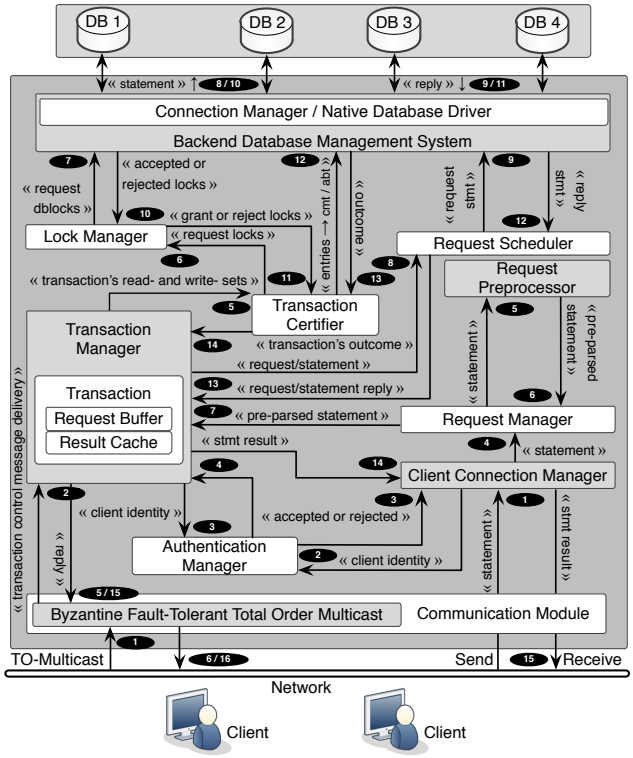


Figure 3: MITRA's components.

level of communication abstractions such as sockets and channels. Upon receiving a client's message it forwards it to the *authentication manager*, to verify if its content is valid and belongs to some active transaction. If that is the case, it forwards the message to the *request manager* for proper treatment; otherwise, it discards the message and sends an exception to the client.

Request Manager. The *request manager* manages all requests received by the server. When a statement is received, the module does basic syntax checking to verify if it is well-formed, then invokes the next module to preprocess the statement. Then, it invokes the *transaction manager* to link the statement with its transaction, as there can be several concurrent transactions being executed through the middleware, and process it.

Request Preprocessor. This component is responsible for the just mentioned syntax checking and preprocessing. MITRA has to support diversity of DBMSs, so there are compatibility issues that must be solved. The middleware supports the standard way of interaction with databases using SQL (Structured Query Language), so the use of different dialects of this language is a problem when diversity is needed. A solution would be to restrict statements to ANSI SQL, but our experience shows that this is too limitative. Therefore, our middleware solves

this problem by translating the SQL statements issued by the clients into the native SQL dialect of the back-end database replica running at the server. This translation is a complex task, but there are software packages that are able to do it for most SQL dialects, e.g., the SwisSQL API (http://www.swissql.com/products/sql-translator/sql-converter.html) or the SQL-Translator (http://search.cpan.org/~frew/SQL-Translator-0.11018).

Transaction Manager. This module controls transaction execution and keeps data concerning active and committed transactions. This module is responsible for: (*i*) scheduling the statements' executions in the database (by invoking the *request scheduler*); and (*ii*) executing the actions requested by control messages. When the total order multicast protocol delivers a control message, this message is passed to the transaction manager that does the following: *begin transaction* – starts a new transaction on the database; *request commit* – changes the state of the transaction and sends a *commit* message if it is the leader; *commit* – starts the transaction certification test in order to either commit or abort the transaction.

Transaction Certifier. The *transaction certifier* executes two tasks: (*i*) checks the validity of every data item read by a transaction; and (*ii*) checks whether the data items read in an optimistic way are up to date when commit is requested. These tasks are done during the termination phase of the protocol to ensure consistency and serializability. In this way, the *transaction certifier* guarantees that a transaction only commits if it is consistent, it has read valid data items, and its reads do not conflict with writes of any concurrent transaction already committed (see Section 3.2). Note that both tasks are necessary because a Byzantine replica may send spurious data items to the client, or may reply to statements with obsolete data. It is noteworthy that due to the use of total order multicast all replicas deliver the *commit* message to the same transaction in the same order, thus they can certify and commit the transaction in the same way. Lastly, since the *transaction certifier* does transaction certification and commitment in a serialized way, this ensures determinism and that the replica's states do not diverge.

Lock Manager. The *lock manager* is responsible for acquiring the read and write locks on the data items of a given transaction against the database, according to the read and write sets for that transaction. This module acts as a scheduler for requesting the locks in the database. All locks for a transaction are requested and acquired in the same order in all replicas, since this happens when these replicas deliver the *commit* message for that transaction. In this way, the order in which all replicas do it is the order imposed by the total order multicast protocol.

Request Scheduler. When the leader replica receives a valid read or write statement from a client, it passes that statement to the *request manager*. Next, the statement goes to the *transaction manager* and, finally, to the *request scheduler* that executes it in the local database. The scheduler is responsible for dealing with concurrency control issues in cooperation with the local DBMS. All operations are executed synchronously, in the sense that the scheduler waits for a result from the DBMS to send it to the client (by replying to the *transaction manager* that then sends it to the *client connection manager*). The scheduler waits for a response for a given interval of time and returns an exception to the client if that time expires without receiving it.

Authentication Manager. The authentication manager maps the authentication credentials provided by the user – login and password in the current version – with the credentials used to access the local DBMS in the server. The login/password provided by the user are not the ones used in the local databases, but a form of access to the virtual database provided by the middleware. This module also verifies the authenticity of the messages received from the clients.

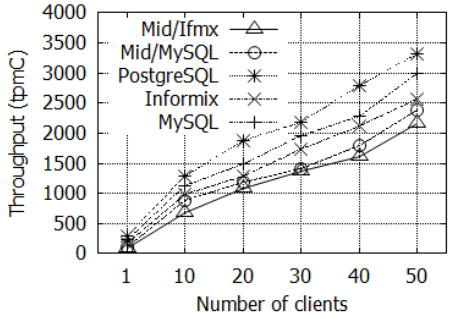
4. PROTOTYPE AND EVALUATION

We implemented the MITRA prototype fully in Java, as a replicated middleware on top of database replicas. As we said in Section 3, our implementation provides a standard JDBC interface, in order to be transparent to client applications. MITRA's JDBC driver is the client-side part of the middleware. It encapsulates the database requests in protocol messages that it forwards to the server-side part of the middleware, the replicas. The replicas access the databases through their native Type IV JDBC drivers. As BFT total order multicast protocol we used BFT-SMaRt, a stable and efficient BFT replication library written in Java (http://code.google.com/p/bft-smart) [3]. Although the code was implemented carefully, at this stage no attempt was made to make it efficient to the point of being usable in real systems.

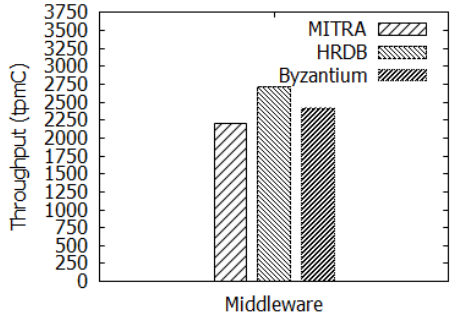
We present some experimental results of the execution of MITRA. Other evaluation aspects such as message complexity can be found in [11]. We evaluate MITRA with the industry standard Online

Transaction Processing TPC-C benchmark (http://www.tpc.org/tpcc). In our evaluation, we compare MITRA, HRDB (with CBS configuration), Byzantium (our own implementation with single-master configuration) and standalone DBMSs. We did not run BFT-DUR because it relies on a key/value database, and the TPC benchmarks are not compatible with this type of databases.

All experiments were done in a LAN with nine machines running CentOS 5.8 and IBM's JDK 6.0, each with 4GB of RAM, an Intel Core 2 Duo E8400 2.67GHz CPU and an Intel 82567LM-3 Gigabit Ethernet interface. The replicas and the clients were attached to the same network switch (Gigabit Ethernet). In order to have database diversity, we used MySQL 5.5.8 (InnoDB) and Informix Innovator-C Edition 11.70-UC1 with MITRA and PostgreSQL 8.4 with Byzantium. For the TPC-C experiments, we loaded the databases (every replica) with 10 warehouses and 10 districts per warehouse. We consider only the case of $f = 1$, which is the typical value used in the evaluation of BFT protocols, as replicas are expensive. Therefore, we executed both MITRA and Byzantium with 4 replicas, and HRDB with 3. The remaining 5 machines were used to run up 50 clients issuing transactions with an interval of 200 ms. The values reported in Figure 4 are the averages of 25 experiments.



(a) Throughput without replication.



(b) Throughput with replication.

Figure 4: Experimental results for standard TPC-C workload (with no batches).

Figure 4(a) aims to show the overhead introduced by the middleware without replication. The label Mid/MySQL corresponds to MITRA with MySQL as DBMS and Mid/Ifmx to MITRA with Informix. In both cases, the clients interact with a single database server through MITRA. These two configurations are not fault-tolerant so they aim simply to provide a lower bound on the performance that our prototype can achieve. The experiments labeled MySQL, Informix and PostgreSQL were obtained through the native JDBC drivers for these DBMSs without involving the middleware. A brief comparison of MITRA's JDBC driver and native DBMSs JDBC drivers shows that the overhead introduced by the middleware is at most 35%, depending on the DBMS. We believe this overhead can be reduced by better engineering the source code, but at this stage our objective was only to have a proof-of-concept prototype.

Figure 4(b) presents the results of running TPC-C with MITRA, HRDB, and our own implementation of Byzantium, with 50 clients producing transactions. In these experiments, both MITRA and HRDB used MySQL as DBMS, while Byzantium used PostgreSQL. We used different DMBSs for MITRA and Byzantium because they have different requirements. MITRA needs DBMSs that supports serializable isolation, whereas Byzantium needs a DBMS that supports snapshot isolation. The figure shows that MITRA has a slightly worse performance than the rest, which was expected for the following reasons. HRDB relies on a centralized coordinator, so it does not need to use a multicast between replicas or to run a certification in every replica, reducing much the overhead involved. However, the benefits of MITRA in relation to HRDB are clear: the failure of up to any f servers does not preclude our middleware from continuing to process transactions; in HRDB the failure of the coordinator stops the system. Byzantium scales better for three reasons, all related to snapshot isolation: it needs only write sets to certify transactions; it needs just one atomic multicast to do commit; it does not need to acquire locks on read operations. However, snapshot isolation is weaker than serializability and under certain circumstances, it can produce incorrect results by violating integrity constraints [2], which not happen in MITRA.

5. FINAL REMARKS

The paper presented a flexible middleware for Byzantine fault-tolerant replication of databases using heterogeneous DBMSs. Due to the use of JDBC, MITRA is compatible with applications that use this interface to interact with the database, being

transparent to the DBMSs, except for issues related to dialects of SQL. Although MITRA is not the only solution for BFT database replication, none of the alternatives provides simultaneously serializability and full distribution. The performance of MITRA is slightly worse than others, which was expected due to the characteristics of that protocol and the fact that we did not make a strong effort to make the prototype efficient. Nevertheless, the results are promising and the costs seem to provide an adequate tradeoff with the benefits, at least for some applications. We believe that these overheads are not overly onerous, especially given the increased robustness and fault tolerance that MITRA offers for an application.

Acknowledgments. This work was partially supported by the CAPES through project Lead Clouds (A039_2013) and by the CNPq through PDI 560258/2010-0 and by the FCT through contract PEst-OE/EEI/LA0021/2013 (INESC-ID).

6. REFERENCES

[1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of the 3rd International European Conference on Parallel Processing*, pages 496–503, 1997.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[3] A. Bessani, J. Sousa, and E. Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of the USENIX Annual Technical Conference*, pages 9–18, 2004.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[6] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for Byzantine fault-tolerant database replication. In *Proceedings of the 6th European Conference on Computer Systems*, pages 107–122, 2011.

[7] I. Gashi, P. T. Popov, and L. Strigini. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing*, 4(4):280–294, 2007.

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

[9] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6:213–226, 1981.

[10] Y. Lin, B. Kemme, M. P. no Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2005.

[11] A. F. Luiz, L. C. Lung, and M. Correia. Byzantine fault-tolerant transaction processing for replicated databases. In *Proceedings of the 10th IEEE International Symposium on Network Computing and Applications*, pages 83–90, 2011.

[12] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[13] F. Pedone, N. Schiper, and J. Armendáriz-Iñigo. Byzantine fault-tolerant deferred update replication. In *Proceedings of the 5th Latin-American Symposium on Dependable Computing*, pages 7–16, 2011.

[14] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):221–232, 1975.

[15] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39:84–87, 1996.

[16] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, pages 59–72, 2007.

[17] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 206–215, 2000.

[18] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, pages 464–474, 2000.