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

Aldelir Fernando Luiz
Federal University of Santa Catarina - Brazil
aldelir@das.ufsc.br

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

Miguel Correia
INESC/ID, IST
Lisbon - Portugal
miguel.p.correia@ist.utl.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 make 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 transactional database management systems (DBMSs) have been a key component of applications from a wide-range of business areas. These applications depend on the reliability and availability of the data stored in the databases, therefore DBMSs have to be fault-tolerant. Replication is a well-known approach to make services fault-tolerant, which has already been applied to DBMSs [9, 19, 4]. 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 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 context of databases and distributed systems. Although Gray commented that it is typically hard to achieve strong consistency in replicated databases [9], Schiper and Raynal have shown that transactions on replicated databases have common properties with group communication primitives such as ordering and atomicity [16]. After that result, several researchers studied the use of group communication systems and other middleware to replicate databases and make them fault-tolerant [13, 3, 11, 18, 7, 14].

Most of the solutions for database replication tolerate only crash faults [13, 3, 11]. 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 disks 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 [8].

Some solutions to replicate databases and make them Byzantine fault-tolerant (BFT) have already been presented [8, 18, 7, 14]. However, they are either focused on a specific problem, based on assumptions hard to substantiate in practice, or simply not the best for certain applications. Specifically, [8] does not allow concurrent transactions, HRDB [18] depends on a centralized controller, Byzantium [7] assumes a consistency criterion weaker than serializability (snapshot isolation), and BFT-DUR [14] does not handle the case of Byzantine clients.

This paper presents the design of MITRA (Middleware for Interoperability on TRAnsactional replicated databases), a middleware for Byzantine fault-tolerant database replication. MITRA supports design diversity [15], i.e., different replicas running different DBMSs. This is an important mechanism to avoid common mode failures caused, e.g., by a bug common to 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 [12], but on the design and architectural aspects of MITRA.

MITRA is modular in the sense that it does not require any change to the DBMSs and hides the complexity of BFT replication from both client applications and DBMSs. The middleware is written in Java and modularity is achieved by following the Java Database Connectivity (JDBC) specification [6].

The rest of this paper is organized as follows. Section 2 present background on replication of database systems. Section 3 introduces MITRA. Section 4 describes some implementation details and experimental results. Section 5 concludes the paper.

2. DATABASE REPLICATION

There are several taxonomies of database replication schemes in the literature [9, 20]. A particularly interesting taxonomy classifies protocols in terms of their update propagation strategy. This taxonomy classifies database replication schemes in three classes: synchronous (or eager replication), asynchronous (or lazy replication), and certification-based. The differences between these strategies are related to the way in which they make the state of the replicas converge.

A *synchronous* or *eager replication* protocol propagates the updates of a transaction applying them on the replicas before the transaction commits [9]. 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 works by executing and committing each transaction at a single replica, delaying the propagation of updates until the transaction has committed [9]. On the positive side, lazy replication tends to perform better because it avoids the communication overhead of eager replication during normal execution. On the negative, lazy replication can let replicas diverge and lose the effects of some transactions [19].

A *certification-based* protocol uses group communication primitives such as total order multicast for ordering transactions and propagating write sets / read sets to the group of replicas [13, 19]. This approach is optimistic in the sense that it updates a single replica without synchronization with the rest, and at the end the transaction aborts it if there were conflicting updates. MITRA follows this approach, which has been shown to be able to provide good performance [13, 11].

3. THE MITRA MIDDLEWARE

MITRA is a middleware for Byzantine fault-tolerant certification-based database replication. It implements the JDBC API specification, in order to provide a heterogeneous, replicated, DBMS that appears to the clients as a single virtual DBMS. The ability to support DBMS diversity is important because different systems are not likely to have the same common bugs or vulnerabilities [8].

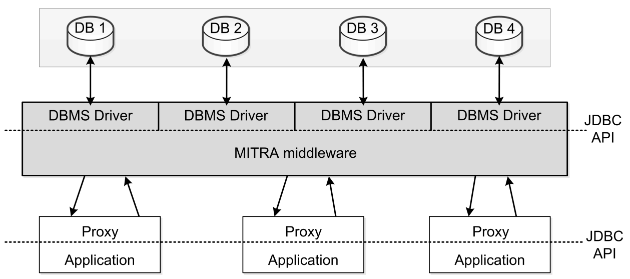


Figure 1: Basic architecture of MITRA

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 has a DBMS driver that hides the specifics of the DBMSs from the replication process. These drivers are also JDBC drivers, so they are readily available for a wide range of DBMSs[1]. 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 / write statements over these items ending with a commit or an abort statement. We make two assumptions related to certification-based replication. First, the back-end DBMSs must support the rollback of operations, so they have to be transactional. Second, statements have to modify every DBMS atomically, without side effects.

[1] `http://devapp.sun.com/product/jdbc/drivers`

As mentioned, MITRA is based on a group communication primitive. Specifically, MITRA runs a BFT total order multicast based on a consensus protocol called Byzantine Paxos [21]. 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 assume the existence of collision-resistant cryptographic hash functions and message authentication functions 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 [19] to handle Byzantine faults. This section briefly presents the protocol, which was reported in detail elsewhere [12].

A certification-based replication protocol operates by letting transactions execute optimistically in a single replica (the leader) and during the commitment phase sending the updates to the rest of the replicas (followers) using a total order multicast primitive [2, 19]. This primitive makes all replicas execute the commitment phases of different transactions in the same order, letting them take the same decision about each transaction: commit or abort. This avoids the need of a voting phase at the end of transactions, in opposition to other replication strategies that need it.

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.

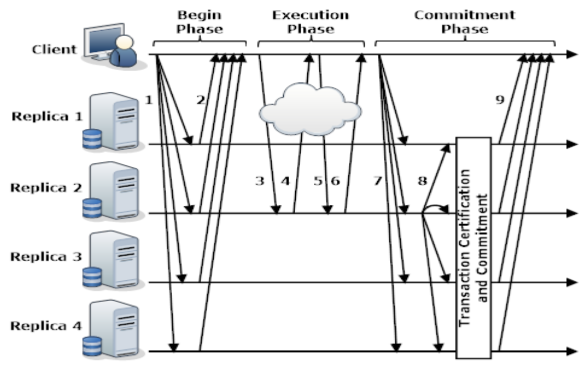


Figure 2: MITRA's protocol with its three phases.

Beginning. As the name suggests, this is the first phase of a transaction. It starts when a client tries to open a connection within the database system. The client's proxy sends a BEGIN message to all replicas using the total-order multicast primitive. Upon delivering this message, the replicas apply a deterministic criterion to select a single leader. This will be the replica that will execute speculatively the statements for that transaction. Afterwards, every replica sends an acknowledgment with the identifier of the leader to the client. This phase of the protocol is shown in steps 1 and 2 of Figure 2.

Execution. When the client receives the acknowledgment, it becomes aware of which replica is the leader. Then, the client starts issuing statements to the leader that executes them speculatively, and returns results to the client. The rest of the replicas do not even receive the statements. This phase is represented in the steps 3-6 of the figure.

Commitment. The commitment phase of a transaction is shown in the steps 7-9 of the figure. This phase starts when the client requests the commitment of the transaction by sending a REQ-COMMIT message using total-order multicast. This message takes the statements issued by the client during the transaction and a cryptographic hash of their results. When the leader receives this message, it also uses the total-order multicast primitive to send a COMMIT message to all the replicas containing: (i) the sequence of transaction's statements that it executed; (ii) a digest summarizing the actions for that transaction (reflecting the state after its execution); (iii) the read set and write set of the transaction (i.e., the data items read/written). When the followers receive these two messages, they become aware of the operations executed during the transaction.

The REQ-COMMIT and COMMIT messages must be delivered in the same order to all replicas, so that all execute the certification in the same order. This order is imposed by having the leader of each transaction disseminating these messages using the total-order multicast primitive. If a transaction t_j is committed after the request to commit t_i and before the commitment of t_i, t_j is concurrent with t_i and t_i has to be certified against t_j in order to preserve serializability. These messages (and events) also prevent a Byzantine client from forging a commit request or committing a spurious transaction, in the sense that all replicas only commit transactions that matches the contents of the REQ-COMMIT and COMMIT messages. Therefore, when a replica delivers a COMMIT message it first checks its integrity. If the check fails the transaction is simply aborted. Oth-

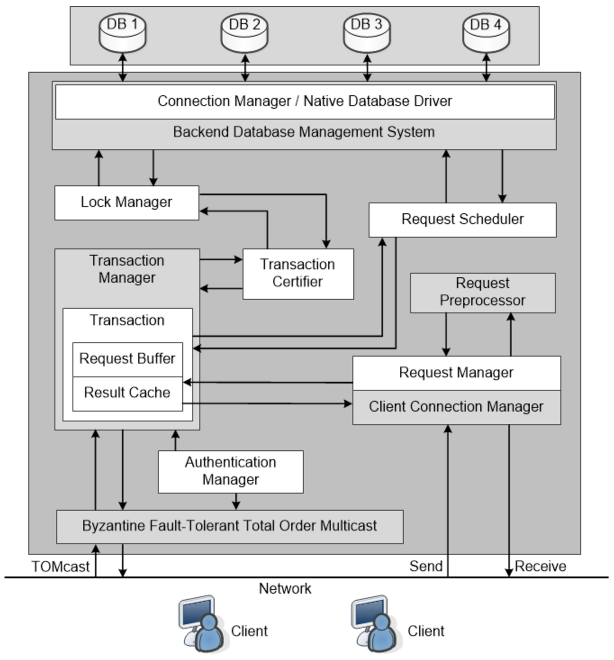


Figure 3: MITRA's components.

erwise, every follower keeps locked the data items in the read and write sets in their local database. So, every replica, including the leader starts a certification test that verifies if reads and writes of committing transactions do not conflict with previously committed concurrent transactions to preserve serializability. This certification is very similar to the backward-validation of classical optimistic concurrency control [10]: if no conflicts occur, the transaction is committed; otherwise, it is aborted.

The last step of the protocol consists in the replicas replying to the client with the outcome of the transaction. The client accepts an outcome if it receives $f+1$ matching replies from that same number of replicas.

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.

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 sends them messages. It works at the level of communication abstractions such as sockets and channels. Therefore, this module simply receives messages with statements and forwards them to the request manager for appropriated treatment.

Request Manager. The request manager has the role of linking a statement with its transaction, as there can be several concurrent transactions being executed using the middleware. For each statement received, the module first performs basic syntax checking and preprocessing of the statement (next module). Once the request passes this check, it is directed to the transaction manager for execution.

Request Preprocessor. This component is responsible for the just mentioned syntax checking and preprocessing. The second aspect is the most interesting. 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 do it for a large number of SQL dialects, e.g., the SwisSQL API [17].

A related issue is the lack of determinism on the order of the results returned by SQL queries. Since relations are unordered sets, a result set may be returned in any order unless an order is explicitly specified. In our case, different replicas may return differently ordered results to the same SQL query. We solve this in MITRA by adding an *order by* clause to all SELECT queries.

Transaction Manager. When the request manager receives a transaction control message, it forwards it to the transaction manager module. There are three types of control messages: (*i*) BEGIN, sent by clients to start a transaction; (*ii*) REQ-COMMIT, sent by clients to request committing a transaction; and (*iii*) COMMIT, sent by the leader when it receives a REQ-COMMIT from a client. When a replica receives one of these messages, the transaction manager performs the according action for the relevant transaction.

Transaction Certifier. This module performs the certification step that gives its name to certification-based database replication. This operation is done during the commitment phase of the protocol and consists in verifying if there is some conflict between the transaction being committed and other already commited concurrent transactions. This verification is based on Kung-Robinson's certification [10], which verifies if there are some conflict between the transaction that is being committed (t_i) and concurrent transactions that were already committed (t_j). More specifically, t_j is considered concurrent with t_i if the replicas delivered the COMMIT message corresponding to t_j both after the delivery of the REQ-COMMIT and before delivery of the COMMIT for transaction t_i (i.e. in the middle of these two events). It is noteworthy that these messages are delivered in the same order by all replicas, therefore all do the same verifications.

Lock Manager. This module is responsible for acquiring the read and write locks on the data items of a given transaction, according to the read and write sets for that transaction. The lock manager acts as a scheduler for requesting the locks. All lock requests are done in the same order in all replicas because they are started when replicas receive the COMMIT message for a transaction, which they all do in the order imposed by the total order multicast protocol.

Request Scheduler. When a request with a read / write statement is received and validated, it is passed 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 then sends it to the client. 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 – and 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.

4. EVALUATION

We implemented a prototype of MITRA in Java. The BFT total order multicast was provided by BFT-SMaRt [1], a BFT replication library written in Java that implements a variation of Byzantine Paxos [21]. Although the code was implemented carefully, no attempt was made to make it efficient to the point of being usable in real systems at this stage.

4.1 Analytical Evaluation

The first part of the evaluation is analytical. We compare MITRA with three well-known BFT replication protocols in the literature, HRDB [18], Byzantium [7], and BFT-DUR [14]. HRDB is based on a concurrency control protocol called commit barrier scheduling and requires a coordinator that can be a single-point of failure. Byzantium, on the contrary, is distributed and has no single-points of failure, but provides only snapshot isolation, not serializability. Both protocols tolerate f faulty servers and any number of faulty clients, similarly to MITRA. BFT-DUR is distributed like Byzantium and MITRA, but considers that clients do not fail.

The evaluation is shown in Table 1. Most lines are self-explainable. The communication steps line shows the number of communication steps for committing a transaction. This line is simplified by including a value *TOMcast* that represents the number of communication steps of the total order multicast protocol (which can vary), and a δ that represents the number of statements for a transaction. The table shows that HRDB has lower number of communication steps (it uses no total order multicast) and message complexity ($O(n)$ instead of $O(n^2)$), benefiting from the centralized controller. In the case of BFT-DUR, only read statements take part in the equation, as write statements are disseminated just at commit time. This improvement on communication steps is mainly due to the type of database assumed, once key/value database transactions are less complex than relational ones.

4.2 Experimental Evaluation

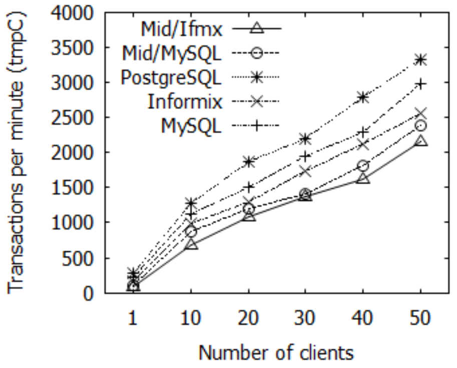
The experiments were based on the industry standard OnLine Transaction Processing TPC-C benchmark[2]. The experimental evaluation compares MITRA, HRDB (with CBS configuration), Byzantium (with multi-master configuration) and stand-alone DBMSs. BFT-DUR was not considered because it assumes a key/value database and TPC benchmarks are not compatible with it.

We did our experiments in a LAN with nine machines running CentOS 5.8 and IBM's JDK 6.0. In order to employ diversity, we used MySQL 5.5.8, Informix Innovator-C Edition 11.70 and PostgreSQL 8.4 as DBMSs. We loaded the databases with 10 warehouses and 10 districts per warehouse. We considered only the case of $f = 1$, which is the typical value used in the evaluation of BFT protocols, as replicas are expensive. This means that we used 4 replicas in MITRA and 3 in HRDB. The other 5 machines were used to run up to 50 clients issuing transactions with an interval of 200 ms between them. The values presented are averages of 25 experiments.

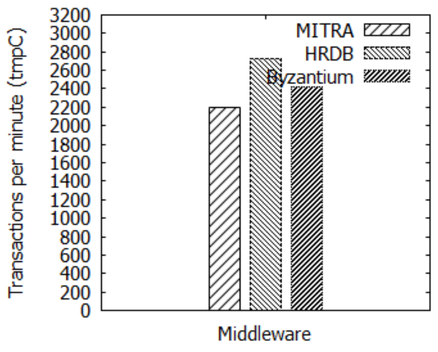
[2]http://www.tpc.org/tpcc

Table 1: Analytical evaluation.

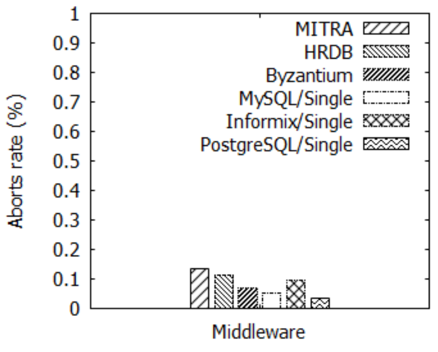
	MITRA	**HRDB** [18]	**Byzantium** [7]	**BFT-DUR** [14]
Number of replicas	$3f + 1$	$2f + 1 + controller$	$3f + 1$	$3f + 1$
Communication steps	$2\delta + 3(TOMcast) + 2$	$4\delta + 3$	$2\delta + 2(TOMcast) + 2$	$2\delta + (TOMCast) + 1$
Message complexity	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n^2)$
Consistency	serializability	serializability	snapshot isolation	serializability
Control	distributed	centralized	distributed	distributed
Database Type	relational	relational	relational	key/value
Byzantine clients	yes	yes	yes	no



(a) Throughput without replication.



(b) Throughput with replication.



(c) General abort rate.

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

The experimental results are shown in Figure 4. Figure 4(a) assesses the overhead introduced by the middleware. The label Mid/MySQL corresponds to MITRA with the MySQL DBMS and Mid/Ifmx to MITRA with Informix. In both cases the interaction between clients and servers was done through MITRA but there was a single server, no replication. It is noteworthy that we use the Mid/* only as a reference point, because they represent the best case of what our prototype is expected to achieve. The experiments labeled MySQL, Informix and PostgreSQL were obtained with these DBMSs accessed directly through JDBC, without the middleware being involved. A comparison of the lines for the same DBMS and number of clients shows that the overhead introduced by MITRA varies from 0% to near 35%. We believe this overhead can be reduced by better engineering the source code of the middleware, but our purpose was to have only a proof-of-concept prototype.

Figure 4(b) presents the results of running TPC-C with MITRA, HRDB and our own implementation of Byzantium. These experiments were done used 50 clients, MITRA and HRDB used MySQL as DBMS, and Byzantium used PostgreSQL for the same purpose. The figure shows that MITRA has worse performance than the rest, which is coherent with the analytical evaluation of the previous section. The fact that HRDB relies on a coordinator removes the need of using multicast between the replicas or run a certification in every replica. However, the benefit is clear: the failure of up to any f servers does not preclude the system from continuing to process transactions, on the contrary to what happens with HRDB that can be stopped by a single failure. Byzantium scales better due to two reasons: it uses snapshot isolation that needs only write sets to certify transaction; and just one atomic multicast to accomplish commit, another advantage of using snapshot isolation.

Figure 4(c) shows the abort rate of each evaluated system. As the experiments of Figure 4(b) the environment had 50 clients running MITRA and HRDB with MySQL and Byzantium with PostgreSQL. Again MITRA has a higher abort rate than the rest. This was to be expected as an increase in the abort rate is a well-known side-effect of speculative/optimistic executions. Nevertheless, this abort rate seemed to be not too high with TPC-C.

5. FINAL REMARKS

The paper presents a flexible middleware for the Byzantine fault-tolerant replication of databases using heterogeneous DBMSs. Due to use of the JDBC interface, MITRA is transparent to the DBMSs, except for issues related to the dialects of SQL. The middleware is also compatible with applications that use JDBC to interact with the database. 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 made 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.

6. REFERENCES

[1] BFT-SMaRt: High-performance Byzantine fault-tolerant state machine replication. *http://code.google.com/p/bft-smart*, 2010.

[2] 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.

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

[4] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *LNCS*. Springer-Verlag, Berlin, Heidelberg, 2010.

[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, Apr. 1985.

[6] M. Fisher, J. Ellis, and J. Bruce. *JDBC API Tutorial and Reference*. Addison Wesley, 3 edition, jun 2003.

[7] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault-tolerant database replication. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2011.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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*. SBC, 2011.

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

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

[17] SwisSQL. Swissql api 5.5. *http://www.swissql.com/products/sql-translator/sql-converter.html*, 2012.

[18] 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*, Oct. 2007.

[19] 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 - ICDCS'00*, pages 464–474, Washington, DC, USA. IEEE Computer Society.

[20] 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 - SRDS'00*, pages 206–215, Washington, DC, USA, 2000. IEEE Computer Society.

[21] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, Univ. of Cambridge Computer Lab, Cambridge, UK, June 2004.