Fault Tolerance Logging-based Model for Deterministic Systems

Óscar Mortágua Pereira, David Simões and Rui L. Aguiar
Instituto de Telecomunicações, DETI, University of Aveiro, Aveiro, Portugal

Keywords: Fault Tolerance, Logging Mechanism, Software Architecture, Transactional System.

Abstract: Fault tolerance allows a system to remain operational to some degree when some of its components fail. One of the most common fault tolerance mechanisms consists on logging the system state periodically, and recovering the system to a consistent state in the event of a failure. This paper describes a general fault tolerance logging-based mechanism, which can be layered over deterministic systems. Our proposal describes how a logging mechanism can recover the underlying system to a consistent state, even if an action or set of actions were interrupted mid-way, due to a server crash. We also propose different methods of storing the logging information, and describe how to deploy a fault tolerant master-slave cluster for information replication. We adapt our model to a previously proposed framework, which provided common relational features, like transactions with atomic, consistent, isolated and durable properties, to NoSQL database management systems.

1 INTRODUCTION

Fault tolerance enables a system to continue its operation in the event of failure of some of its components (Randell et al., 1978). A fault tolerant system either maintains its operating quality in case of failure or decreases it proportionally to the severity of the failure. On the other hand, a fault intolerant system completely breaks down with a small failure. Fault tolerance is particularly valued in high-availability or life-critical systems.

Relational Database Management Systems (DBMS) are systems that usually enforce information consistency and provide atomic, consistent, isolated and durable (ACID) properties in transactions (Sumathi and Esakkirajan, 2007). However, without any sort of fault-tolerance mechanism, both atomicity and consistency are not guaranteed in case of failure (Gray and others, 1981).

We have previously proposed a framework named Database Feature Abstraction Framework (DFAF) (Pereira et al., 2015), based in Call Level Interfaces (CLI), that acts as an external layer and provides common relational features to NoSQL DBMS. These features included ACID transactions, but our framework lacked fault-tolerance mechanisms and, in case of failure, did not guarantee atomicity or consistency of information.

This paper presents a model that can be used to provide fault-tolerance to deterministic systems through external layers. We describe how to log the system state, so that it is possible to recover and restore it when the system crashes; possible ways to store the state, either remotely or locally; and how to revert the state after a crash.

We prove our concept by extending DFAF with the proposed logging mechanisms in order to provide fault-tolerant ACID transactions to NoSQL DBMS. DFAF acts the external layer over a deterministic system (a DBMS). We consider that non-deterministic events can happen in the deterministic systems and are either expected (e.g.: receiving a message), triggering deterministic behaviour, or unexpected (e.g.: crashing), leading to undefined behaviour.

The remainder of this paper is organized as follows. Section 2 describes common fault tolerance techniques and presents the state of the art. Section 3 provides some context about the DFAF and Section 4 formalizes our fault tolerance model, describing what information is stored and how to store it. Section 5 describes a fault-tolerant data replication cluster which can be used for performance enhancements and Section 6 shows our proof of concept and evaluates our results. Finally, and Section 7 presents our conclusions.

Pereira, Ó., Simões, D. and Aguiar, R.
Fault Tolerance Logging-based Model for Deterministic Systems.
DOI: 10.5220/0005979101190126
In *Proceedings of the 5th International Conference on Data Management Technologies and Applications (DATA 2016)*, pages 119-126
ISBN: 978-989-758-193-9
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

2 STATE OF THE ART

Fault tolerance is usually achieved by anticipating exceptional conditions and designing the system to cope with them. Randell et al. define an erroneous state as a state in which further processing, by the normal algorithms of the system, will lead to a failure (Randell et al., 1978). When failures leave the system in an erroneous state, a roll-back mechanism can be used to set the system back in a safe state. Systems rely on techniques like check-pointing, a popular and general technique that records the state of the system, to roll-back and resume from a safe point, instead of restarting completely. Log-based protocols (Johnson, 1989) are check-pointing techniques that require deterministic systems. Non-deterministic events, such as the contents and order of incoming messages, are recorded and used to replay events that occurred since the previous checkpoint. Other non-deterministic events, such as hardware failures, are meant to be recovered from. Indirectly, they are recorded as *lack of information*.

In the fault tolerance context, logging mechanisms and their concepts and implementation techniques have been discussed and researched extensively (Gray and Reuter, 1992), with popular write-ahead logging approaches (Mohan et al., 1992) having become common in DBMS to guarantee both atomicity and durability in ACID transactions. There are also other approaches which do not rely on logging systems to provide fault tolerance, like Huang et al.'s method and schemes for error detection and correction in matrix operations (Huang et al., 1984); Rabin et al.'s algorithm to efficiently and reliable transmit information in a network (Rabin, 1989); or Hadoop's data replication approach for reliability in highly distributed file systems (Borthakur, 2007). Some relational DBMS use shadow paging techniques (Ylönen, 1992) to provide the ACID properties. However, the above described fault tolerance mechanisms are not suitable to be used in an external fault-tolerance layer, since they are very dependent on the architecture of the systems they were designed for.

The most general proposals fall in the category of data replication, where several algorithms and mechanisms have been proposed. These include Hadoop's data replication approach for reliability in highly distributed file systems (Borthakur, 2007); (Oki and Liskov, 1988), which is based on a primary copy technique; (Shih and Srinivasan, 2003), an LDAP-based replication mechanism; or (Wolfson et al., 1997), which provides an adaptive algorithm that replicates information based on its access pattern. Recently, proposals have also focused on byzantine failure tolerance (Castro and Liskov 1999; Cowling et al. 2006; Merideth and Iyengar 2005; Chun et al. 2008; Castro and Liskov 2002; Kotla and Dahlin 2004). Byzantine fault-tolerant algorithms have been considered increasingly important because malicious attacks and software errors can cause faulty nodes to exhibit arbitrary behaviour. However, the byzantine assumption requires a much more complex protocol with cryptographic authentication, an extra pre-prepare phase, and a different set of techniques to reach consensus.

To the best of our knowledge, there has not been work done with the goal of defining a general logging model that provides fault tolerance as an external layer to an underlying deterministic system. Some solutions provide fault tolerance, but are adapted to a specific context or system. Others are overly-abstract general models, like data replication, and do not cover how to generate the necessary said data from an external layer to provide fault-tolerance to the underlying system. Not only that, but many data replication systems also assume conditions we do not, such as the possibility of byzantine failures, or overly complex data access patterns. While byzantine failures are of enormous importance in distributed unsafe systems, such as in the BitCoin environment (Nakamoto, 2008), we consider their countermeasures to be complex and performance-hindering in the scope of our research. Not only that, but byzantine assumptions have been proven to allow only up to 1/3 of the nodes to be faulty. We intend to focus on fault-tolerance for underlying deterministic systems through a logging system, and while distributed data replication is used for reliability, expected DFAF use cases do not assume malicious attacks to tamper with the network. However, our model is general enough that it supports the use of any data replication techniques to replicate logging information across several machines.

3 CONTEXT

We have previously mentioned the DFAF, which allows a system architect to simulate non-existent features on the underlying DBMS for client applications to use, transparently to them. Our framework acts as a layer that interacts with the underlying DBMS and with clients, which do not access the DBMS directly. It allowed ACID transactions, among other features, on NoSQL

DBMS, but was not fault tolerant. Typically, NoSQL DBMS provide no support to ACID transactions. An ACID transaction allows a database system user to arrange a sequence of interactions with the database which will be treated as atomic, in order to maintain the desired consistency constraints. For reasons of performance, transactions are usually executed concurrently, so *atomicity*, *consistency* and *isolation* can be provided by file- or record-locking strategies. Transactions are also a way to prevent hardware failures from putting a database in an inconsistent state. Our framework must be adjusted to take hardware failures into account with multi-statement transactions. In a failure free execution, our framework registers what actions are being executed in the DBMS and how to reverse them, using a reverser mechanism (explained further below). Actions are executed in the DBMS immediately and are undone if the transaction is rolled-back.

However, during a DFAF server crash, the ACID properties are not enforced. As an example, consider a transaction with two *insert* statements. If the DFAF server crashed after the first insert, even though the client had not committed the transaction, the value would remain in the database, which would mean the *atomic* aspect of the transaction was not being enforced. To enforce it, we propose a logging mechanism, whose records are stored somewhere deemed safe from hardware crashes. That logging system will keep track of the transactions occurring at all times and what actions have been performed so far. When a hardware crash occurs, the logging system is verified and interrupted transactions are rolled-back before the system comes back on-line. Our logging system is an extension to DFAF and is a log-based protocol where the underlying DBMS acts as the deterministic system mentioned previously. Each action in a transaction represents a non-deterministic event and is, as such, recorded, so that the chain of events can be recreated and undone when the system is recovering from failure.

4 LOGGING SYSTEM

Logging systems for fault-tolerance mechanisms have several different aspects that need to be defined: firstly, the logging system must be designed in a way that the logging is not affected by hardware failures. In other words, if the server crashes while a database state was being logged, the system must be able to handle an incomplete log and must be able to recover its previous state. Secondly, logging an action is not done at the same time as that action is executed. Taking an insertion in a database as an example, the system logs that a value is going to be inserted, the value is inserted and the system logs that the insertion is over. However, if the system crashes between both log commands, there is no record of whether the insert took place or not. To solve this, the underlying system must be analysed to check if it matches the state prior to the insertion or not. Thirdly, while recovering from a failure, the server can crash again, which means the recovery process itself must also be fault tolerant. Finally, cascading actions imply multiple states of the underlying system, all of which must be logged so that they can all be rolled-back. In other words, if an insert in a database triggers an update, then the database has three states to be logged: the initial state, the state with the insertion and the state with the insertion and the update. Because the server can crash at any of these states, they all need to be logged so that the recovery process rolls-back all the states and nothing more than those states.

4.1 Logging Information

In order to provide fault tolerance, there are two choices to compensate for the failure (Garcia-Molina and Salem, 1987): backward recovery, or executing the remainder of the transaction (forward recovery). For forward recovery, it is necessary to know *a priori* the entire execution flow of the transaction, which is not always possible. DFAF uses the backward recovery model to avoid leaving the system in an inconsistent state when a rollback is issued by a client. To do so, along with the actions performed, DFAF registers how to undo them. In other words, when a client issues a command, the command to revert it, referred to as the reverser, is calculated. In a SQL database, for example, an *insert*'s reverser is a *delete*. Reversers are executed backwards in a recovery process to keep the underlying system in a consistent state. However, logging actions and performing them cannot be done at the same time. It is also not adequate to log an action after it has already been performed, since the server could crash between both stages, and there would be no record that anything had happened. Therefore, actions (and their reversers) must be logged before they are executed on the underlying system. However, if the server crashes between the log and the execution, the recovery process would try to reverse an action that had not been executed. Because we have no assumptions regarding when

the system can crash, the only way to solve this problem is to directly assess the underlying system's state to figure out whether the action has been performed or not. Since we have access to the underlying system's state prior to the action being executed, we can find a condition that describes whether the action has been executed or not. This condition will be referred to as verifier from now on.

For example, after the insertion of value *A*, it is trivial to verify if the value has been inserted or not by the amount of rows with value *A* that existed prior to the insertion. If there were two *A*s and the transaction crashed during the insertion of a third, by counting how many exist in the database, we can infer whether we need to reverse this action in the transaction (if we now have three *A*s) or if the action did not get completed (if we still have two *A*s). The concept is extended to cascading actions. A reverser is determined for each cascading action in DFAF, which means a verifier must also be calculated to determine whether that effect happened and needs to be rolled-back or not. If the server crashes during these triggered actions or during a rollback, each verifier must be checked before applying the corresponding reverser, to ensure that 1) we are not reverting the same action twice, and that 2) we are not reverting an action that was not executed. During the recovery process, reversers are executed backwards. If a verifier shows that an action has not been completed, or after an action has been reversed, its record (along with the reverser and verifier) is removed from the log. If the server crashes during a recovery, due the verifier system, there is no risk of reverting actions that need not be reverted or that have not yet been executed.

4.2 Logging Information Storage

We have implemented two possible information storage mechanisms: a local and a remote one. These can be used with regular hardware and standard computational resources nowadays. Other storage mechanisms are supported, such as using a relational DBMS to store and retrieve the logs. The only requirement is that the mechanisms are fault-tolerant.

The local mechanism relies on writing the logging information to disk: fault tolerance is supported even in a complete system crash, but with heavy performance costs. It does not require any additional software, other than file system calls. The remote mechanism tries to leverage both performance and fault tolerance and relies on a remote machine to keep the logging information in memory. I/O operations are not as heavy on performance as writing to disk, but fault tolerance is only guaranteed if the logging server does not crash.

We have designed a fault-tolerant master-slave architecture, deemed a Cluster Network (CN), to allow several machines to coordinate and replicate information among them. This system can be used to store the logs from the remote mechanism, which allows some machines to crash without loss of information. In a CN, the only case where the logs would be lost would be a scenario where all machines crashed, which is unlikely if the machines are geographically spread. We expect the performance of this mechanism to be superior in comparison with the local mechanism. The remote mechanism uses TCP sockets to exchange information between the servers. Because TCP provides reliability and error control, both machines know when a message has been properly delivered and the system server can perform the requested actions while the logging server keeps the information in memory. Both servers can detect if the network failed or the remaining server has crashed. In these cases, the recovery process can be initiated until connectivity is re-established.

The local mechanism, as previously stated, was designed to store the information in the file system. We assume that the hardware crashes will not be so severe that they render the hard drive contents unrecoverable, or that a back-up system is deployed to allow the recovery of a defective file system. Most file systems do not provide fault-tolerant atomic file creation, removal, copy, movement, appending or writing operations, so we need to first address this issue and prevent the logging system from entering an inconsistent state, if there is a crash during a logging operation. We start by creating a file for each transaction occurring in the system. The file is created as soon as a transaction is started and deleted just before it is complete. If the server crashes when the transaction is starting and creating the file, the file can either exist and be empty, or not exist. There are no actions to be rolled-back, so either case is fine and the file is ignored. If the server crashes when deleting the file and closing the transaction, the file can either exist with its contents still intact, or not exist. If it does not exist, the transaction was already over. If it still exists, then it is possible to read it and rollback the database. The *log* file update must be done in a way that the logging system's last state must be recoverable. As such, to prevent file corruption, a copy of the old state is kept until the new one is completely defined. Firstly, we create a file, *temp*, that signals we were

updating the log and whose existence means that the original *log* file is valid. After we create it, we copy the log to a *copy* file. When all of the contents have been copied, we delete *temp*. If the server crashes at any point and *temp* exists, *log* is still valid and the server ignores *copy*. If *temp* does not exist, but *copy* exists, then *copy* is valid and the server ignores the original *log*. After *temp* has been deleted, *log* is updated with the new information (a new state in the database, for example). After *log* has been fully updated, *copy* can be deleted, since it is no longer necessary. Table 1 shows the several stages described above.

Table 1: A log-update cycle, with the several stages of the update, the state of each of the files, and what file is chosen on each stage.

Stage:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
log (L)	✓	✓	✓	✓	✓	?	✓	✓
temp	×	✓	✓	✓	×	×	×	×
copy (C)	×	×	?	✓	✓	✓	✓	×
File:	L	L	L	L	C	C	C	L

With the two proposed mechanisms, a recorded log of executed actions on the database can be safely stored and used to return the underlying system to a consistent state.

5 CLUSTER NETWORK

Our remote logging mechanism can rely on a cluster-based system to store the needed information. This allows for fast interactions, reliability and consistency. Data replication techniques such as byzantine tolerant approaches are a valid option, but have an associated performance decay due to the byzantine assumption and a low threshold for the amount of faulty machines. As such, we designed a fault-tolerance master-slave network that replicates information across all the slaves and better fits DFAF's requirements.

We require our Cluster Network to be able to grow as needed, without having to interrupt service or without having maintenance downtime. We considered that nodes should be symmetrical to avoid the human error factor present in id-based systems. We also want a stable algorithm (a master stays as master until it crashes) to avoid unnecessary operations when an ex-master is turned back on. Finally, we consider that an IP network is not perfect and that network elements (switches, routers) and well as network links can crash at any time. We therefore allow a set of any number of nodes that communicate through IP where any of the nodes can crash and be restarted at any given time. The master node is contacted by clients and it forwards the information to the slave nodes. Clients can find the master node through any number of methods, like DNS requests, manual configuration, broadcast inquiries, etc. If the master crashes, one of the slaves is nominated to be master and, because all the information was replicated among the slaves, it can resume the master's process.

Our election algorithm is inspired in Gusella et al.'s election algorithm (Gusella and Zatti, 1985). While many other leader election algorithms would be supported, this one suits the DFAF requirements the best. The authors have developed a Leader Election algorithm that is dynamic (nodes can crash and restart at any time), symmetric (randomization is used to differ between nodes), stable (no leader is elected unless there is no leader in the cluster) and that uses User Datagram Protocol (UDP) communication (non-reliable, non-ordered). It supports dynamic topology changes to some degree, but it is not self-stabilizing (nodes start in a defined state, not in an arbitrary one). When a master is defined, the master is the one receiving requests from clients. In order to guarantee consistency among all the nodes, the master forwards any incoming requests to the slaves before answering the client with the corresponding response. This guarantees that all the slaves will have the same information as the master. If the master crashes during this process, because the client still has not been answered, he will retry the request to the new master, which will store it (while avoiding request duplication) and forward it to the slaves. When a slave joins the network, he contacts the master and requests the current system information (in this case, the current log). A mutual exclusion mechanism is necessary to avoid information inconsistency when information is being relayed to a new slave. To avoid request duplication from clients when the master node crashes, a request identification number is used. Using this approach means that up to *N-1* nodes in the CN can crash without information being lost or corrupted. Using other approaches for data replication, such as (Castro and Liskov, 1999) only allows up to *N/3* nodes to be faulty and is expected to have worse performance. However, byzantine-tolerant approaches are more robust and, as previously stated, our logging model is general enough that any data replication mechanism can be used to safe-keep the logging information.

6 PROOF OF CONCEPT

We extended the previously mentioned DFAF with our proposed logging mechanism, in order to guarantee the *atomic* and *consistent* properties of transactions. This way, even if the DFAF server crashed during multiple concurrent transactions, those transactions will all be rolled-back and the underlying database will be on a consistent state when the recovery process has finished. The reverser and verifier system in DFAF depends on the underlying DBMS schema and query language. Different schemas can imply different cascading actions, if, for example, different triggers are defined in each schema. However, NoSQL DBMS don't usually support cascading actions such as triggers, and they do not fall under the expected use cases of DFAF. Different query languages also imply different reversers and verifiers, since an *insert* in SQL has a very different syntax from a NoSQL DBMS's custom query language. However, the reverser and verifier creation mechanism is trivial for most SQL and SQL-like languages. Verifiers are *select* statements related with the values being inserted, deleted or updated. Reversers are *delete* statements for *insert* statements, *insert* statements for *delete* statements, and *update* statements for *update* statements. Having multiple transactions occurring at the same time implies having either multiple log files or a single log file with information from all transactions. This could lead to problems during the recovery process, if the order of actions in separate transactions was not being logged. However, the fact that transactions guarantee the isolation property means that each of their actions will not affect other transactions. Therefore, the order in which each transaction is rolled-back is irrelevant, as long as the statements in each transaction are executed backwards. To prove our concept, we tested the local logging mechanism using DFAF with a single client connecting to the database. The client starts a transaction, inserts a value and updates that value, finishing the transaction. During this process, the logging information is stored in a local file. We crashed the transaction on several stages (shown in Table 1) and verified that the recovery process could correctly interpret the correct log file and set the database in a correct state, the one previous to the transaction. In order to interrupt the process on particular stages, exceptions were purposely induced in the code, which were thrown at the appropriate moments. The recovery process was then started and tested as to whether it could successfully recover and interpret logged information and, if needed, rollback the database to a previous state. Results showed that the system was always able to recover from a failed transaction and returned the database to a safe state. To prove our concept with the remote mechanism, we deployed a network with a client connected to a DBMS and to a CN, as shown in Figure 1.

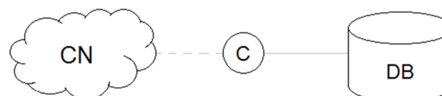

Figure 1: The deployed network for tests with the remote mechanism and a single client.

We used the same transaction used to test the local mechanism. In our first test, we checked whether the CN could detect and roll-back failed transactions. We crashed the client after the first insertion and the CN immediately detected the crash and rolled-back the transaction. In our second test, we checked if a correct rollback was ensued with crashes on different stages of the transaction. We crashed the client at several stages of the transaction (before logging the action, after logging but before performing the action, after performing but before logging that it has been performed and after logging that the action had been done) and monitored the roll-back procedure to guarantee the database was in the correct state after the recovery process had finished. Finally, we checked whether several concurrent transactions occurring in a DFAF server could all be rolled-back without concurrency issues. We used a DFAF server to handle several clients while connected to a CN, as can be seen in Figure 2, and crashed the server during the client's transactions. The CN detected the crash and rolled-back all transactions, leaving the database once more in a consistent state.

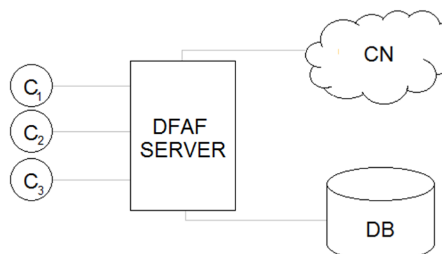

Figure 2: The deployed network for tests with the remote mechanism and multiple clients.

To demonstrate the soundness of our approach in a practical environment, we examined the performance of our logging mechanism's implementation and of our CN using a 64-bit Linux

Mint 17.1 with an Intel i5-4210U @ 1.70GHz, 8GB of RAM and a Solid State Drive. For tests involving a CN, a second machine was used, running 64-bit Windows 7 with an Intel i7 Q720 @ 1.60GHz, 8GB of RAM and a Hard Disk Drive. A 100Mbit cable network was used as an underlying communication system between both nodes. Figure 3 shows how the local (green) and remote (red) logging mechanisms, using as a basis for comparison a transaction with up to 1000 statements on a SQLite table. This number of statements was based on previous DFAF evaluations. Tests were repeated several times to get an average of the values, the 95% confidence interval was calculated, and the base time for operations was removed to allow for a more intuitive graph analysis. The CN used for the remote mechanism was a local single-node, which removed most of the network interference with the tests.

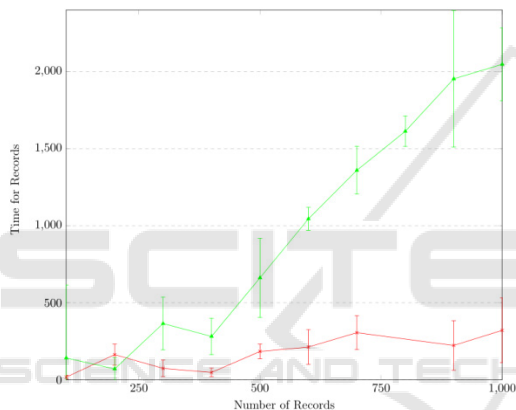

Figure 3: Performance (in milliseconds) of the different logging mechanisms.

As expected, the most performant mechanism is the remote mechanism, where a sub-second performance decay is noticed (around 321±209 milliseconds for 1000 operations). The baseline time for 1000 operations was 10295±1142 milliseconds, which means remote mechanism has a performance decay of approximately 3.1%. The local mechanism is the least performant, due to the high amount of disk operations, with around 2047±237 milliseconds for 1000 operations, a 19.8% performance decay. The performance difference of an order of magnitude between both mechanisms is due to the fact that, as the logging file gets bigger, it takes longer to read, copy and write it. This means that, with a transaction of 1000 insertions, for example, the 1000th insertion will take a lot longer than the 1st insertion, while the remote mechanism takes the same amount of time for any insertion.

We tested Cluster Networks to find how long it takes to find a master and make the information consistent among them. These values have a direct correlation to the defined time-outs on each state of the network, as defined by Gusella et al.'s algorithm. We created two-node networks (1 master, 1 slave) and measured the times taken for each node to become a master/slave (with a confidence interval of 95%) and to guarantee the consistency of information among them. Tests with more nodes were not feasible, due to hardware restraints. Tests show an average of 5±1 milliseconds to get a node from any given phase of the election algorithm to the next, excluding the defined time-outs. The time taken to exchange all the information from a master to a slave depends on the current information state, but in our tests, any new slave took approximately 8±1 milliseconds to check whether information was consistent with the master. Transferring the log with 1000 records from the first test took approximately 20±4 milliseconds.

7 CONCLUSIONS

We have previously proposed DFAF, a CLI-based framework that implements common relational features on any underlying DBMS. These features include ACID transactions, local memory structure operations and database-stored functions, like Stored Procedures. However, the proposal lacked a fault tolerance mechanism to ensure the atomic property of transactions in case of failure. We now propose a fault tolerance model, general enough to work with several underlying deterministic systems, but adapted to DFAF.

Our model is a logging mechanism which requires the performed action, its verifier (that checks whether it has been executed or not) and its reverser (to undo it, in case of failure). We describe two ways of storing the information: either locally in the file system, or remotely in a dedicated server. Because operating systems do not usually provide atomic operations, to prevent the logging information from becoming corrupted, we also describe how to update the information. In order to guarantee that the remote server is also fault tolerant and the information is not lost in case of failure, we describe a master-slave network that can be used to replicate the information. Clients contact the master, which replicates the information to slaves without consistency issues. Our performance results show that the use of our logging mechanism can be suitable for a real-life scenario. There is an expected performance degradation, but a fault tolerant system

provides several advantages over a slightly more performant fault intolerant system. Not only that, but the performance decay using the remote mechanism is nearly negligible.

In the future, we intend to improve both the local and remote mechanisms. Regarding the file system, we intend to develop a highly performant algorithm, that does not rely on copying the previous log on each update. Regarding the remote mechanism, we intend to adapt the CN for other requirements, in order to improve performance. This can be done by allowing priority nodes and removing the symmetry factor. This way, servers can preferentially become masters, if they have better hardware or conditions. The CN can also be improved by changing the underlying communication protocol, which at the moment is assumed to be unreliable. We also intend to develop a master look-up mechanism, like DNS registration. At the moment, there is no such mechanism, and clients resort to finding masters manually.

In conclusion, we extended DFAF with a log-based fault-tolerance model, this way guaranteeing ACID properties on the underlying DBMS transactions. We describe two ways of storing the information, to leverage performance and reliability, but support other models. We also propose a master-slave fault tolerant network which can be used as a remote server to keep information replicated and consistent. Both the logging model and the CN can be used for other applications as well; we have for example adapted the CN to act as a concurrency handler in another module of DFAF.

ACKNOWLEDGEMENTS

This work is funded by National Funds through FCT - Fundação para a Ciência e a Tecnologia under the project UID/EEA/50008/2013.

REFERENCES

Borthakur, D., 2007. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007), p.21.

Castro, M. and Liskov, B., 1999. Practical Byzantine fault tolerance. *OSDI*.

Castro, M. and Liskov, B., 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*.

Chun, B., Maniatis, P. and Shenker, S., 2008. Diverse Replication for Single-Machine Byzantine-Fault Tolerance. *USENIX Annual Technical Conference*.

Cowling, J., Myers, D. and Liskov, B., 2006. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. *Proceedings of the 7th …*.

Garcia-Molina, H. and Salem, K., 1987. *Sagas*, ACM.

Gray, J. and others, 1981. The transaction concept: Virtues and limitations. In *VLDB*. pp. 144–154.

Gray, J. and Reuter, A., 1992. *Transaction Processing: Concepts and Techniques* 1st ed., San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Gusella, R. and Zatti, S., 1985. *An election algorithm for a distributed clock synchronization program*,

Huang, K.-H., Abraham, J. and others, 1984. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6), pp.518–528.

Johnson, D.B., 1989. Distributed System Fault Tolerance Using Message Logging and Checkpointing by. *Sciences-New York*, 1892(December).

Kotla, R. and Dahlin, M., 2004. High throughput Byzantine fault tolerance. *Dependable Systems and Networks, 2004 …*.

Merideth, M. and Iyengar, A., 2005. Thema: Byzantine-fault-tolerant middleware for web-service applications. *… , 2005. SRDS 2005. …*.

Mohan, C. et al., 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1), pp.94–162.

Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system. Available at: http://www.cryptovest.co.uk/resources/Bitcoin paper Original.pdf [Accessed February 15, 2016].

Oki, B.M. and Liskov, B.H., 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. pp. 8–17.

Pereira, Ó.M., Simões, D.A. and Aguiar, R.L., 2015. Endowing NoSQL DBMS with SQL Features Through Standard Call Level Interfaces. In *SEKE 2015 - Intl. Conf. on Software Engineering and Knowledge Engineering*. pp. 201–207.

Rabin, M.O., 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2), pp.335–348.

Randell, B., Lee, P. and Treleaven, P.C., 1978. Reliability Issues in Computing System Design. *ACM Computing Surveys*, 10(2), pp.123–165.

Shih, K.-Y. and Srinivasan, U., 2003. Method and system for data replication.

Sumathi, S. and Esakkirajan, S., 2007. *Fundamentals of relational database management systems*, Springer.

Wolfson, O., Jajodia, S. and Huang, Y., 1997. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(2), pp.255–314.

Ylönen, T., 1992. Concurrent Shadow Paging: A New Direction for Database Research.