Byzantine Fault-Tolerant Transaction Processing for Replicated Databases

Aldelir Fernando Luiz*†, Lau Cheuk Lung‡, Miguel Correia§
*Campus of Blumenau, Federal Institute Catarinense - Brazil
†Department of Automation and Systems, Federal University of Santa Catarina - Brazil
‡Department of Informatics and Statistics, Federal University of Santa Catarina - Brazil
§Instituto Superior Técnico, Technical University of Lisbon, INESC-ID - Portugal
aldelir@das.ufsc.br, lau.lung@inf.ufsc.br, miguel.p.correia@ist.utl.pt

***Abstract*—Transaction commit is a problem much investigated, both in the databases and systems communities, from the theoretical and practical sides. We present a modular approach to solve this problem in the context of database replication on environments that are subject to Byzantine faults. Our protocol builds on a total order multicast abstraction and is proven to satisfy a set of safety and liveness properties. On the contrary of previous solutions in the literature, it assures strong consistency for transactions, tolerates Byzantine clients and does not need centralized control or multi-version databases. We present an evaluation of a prototype of the system.**

I. Introduction

The notion of *transaction* was first introduced in database systems, with the objective of supporting the consistent execution of concurrent operations over shared data [1]. However, since then transactions have been applied much more broadly, e.g., in distributed systems in many application scenarios, in which they improve reliability and guarantee data consistency. This paper is about transaction commit for *replicated databases*. There is work in this area for some years, with authors suggesting the utilization of abstractions commonly used to specify reliable distributed systems (e.g., consensus, total order multicast) to support crash fault-tolerant database replication or, more generically, transaction processing [2], [3].

Protocols for database replication face many challenges. The main ones are: (*i*) to ensure the consistency of the replicas while allowing a high level of concurrency; (*ii*) to ensure the execution of the transactions in the same or an equivalent order/sequence in all replicas; and (*iii*) the transaction commitment protocol itself, as the global atomicity has to be ensured. Transaction commitment is a problem much studied in the literature, often under the designation of non-blocking atomic commit (NBAC) [4]. In systems with only crash faults, NBAC is reducible to consensus, as it is essentially an agreement between a set of processes on committing or aborting a transaction [5].

There is much literature about crash fault-tolerant database replication and transactions [6], [7], [3], [5]. On the contrary, the lack of solutions to tolerate Byzantine or arbitrary faults [8] has been pointed out in a 2007 work that shows the existence of many bugs in database management systems (DBMS) [9]. These bugs are Byzantine faults that typically cannot be tolerated by crash fault-tolerant replication protocols. It is tempting to use Byzantine fault-tolerant (BFT) state machine replication protocols [10], [11] to implement BFT transactions by ordering all operations, but this would create a considerable overhead and would lead to deadlocks when there were conflicting transactions. Optionally such protocols might be used to order static transactions instead of ordering operations, but we want to support *dynamic transactions*, i.e., transactions in which the operations to be executed inside the transaction can be defined during the execution (e.g., because they depend of the result of previous operations of the transaction).

Furthermore, NBAC is concerned only with fault-tolerant transaction committing, but to achieve a correct decision on the commitment, it is imperative that transaction processing (the execution of the operations) is done correctly. This issue is directly related to the agreement involved in atomic commitment, but it is neglected by the classical definition of NBAC [4]. Therefore, we need to investigate the means to preserve not only the consistency (the main requirement of transactions), but also integrity of replicated data, despite of replica faults.

This paper presents a solution to tolerate Byzantine faults in databases replicated using the *database state machine approach* (DBSM) [3]. We use the same replication technique as DBSM to propagate data modifications executed in the context of a transaction, but we tolerate Byzantine faults both in the clients and replicas, on the contrary of DBSM that tolerates only crashes. This technique, known as *deferred update*, is much used to implement efficient data replication protocols, because it scales well. In our protocol we exploit the use of a BFT total order multicast algorithm [12] together with cryptographic mechanisms, in order to obtain consistency and data integrity despite the existence of Byzantine faults.

The main contribution of this paper is a robust and safe protocol for both *one-copy serializable* transaction commitment and processing despite Byzantine faults. Our protocol is based on distributed system abstractions and tolerates Byzantine faults in any client or server (up to a limit). To the best of our knowledge, there are only three previous works that support Byzantine fault-tolerant database replication: HRDB is based on a hybrid fault model, in the sense that it uses a centralized controller that can not fail in a Byzantine way [13]; Bizantium is really Byzantine fault-tolerant, but provides

only *snapshot isolation*, a consistency criterion weaker than one-copy serializability [14]; BFT Deferred Update is more similar to ours, but does not tolerate Byzantine clients and requires multi-version databases (instead of single-version like ours) [15]. Our work is, therefore, a considerable advance in relation to those previous works. A more detailed comparison is provided in the following section.

II. Related Work

Data replication in transaction processing systems has been much studied considering only crash faults [7], [3]. Byzantine faults have been mostly forgotten, even though a seminal paper studied the problem more than two decades ago [16].

For the case of crash faults, Agrawal et al. were the first to exploit multicast primitives to support data replication in transaction systems [7]. Their approach was to employ a total order multicast primitive to simplify replication management, to provide strong consistency, and to reduce the number of deadlocks. More recently, Pedone et al. proposed the database state machine approach (DBSM), in which they replaced the atomic commit protocol by a total order multicast protocol, providing a better support to consistency of the replicas under heavy concurrency [3]. Unfortunately, none of these works can be used directly to tolerate Byzantine faults on transaction processing systems, because these systems assume that the primary replica processes the statements correctly, so the results of the operations it executes are simply propagated to the other replicas. If the primary replica was Byzantine, it might corrupt the state of the system arbitrarily.

In the field of Byzantine faults, the seminal work of Molina et al. employed Byzantine agreement [8] and state machine replication [17] with databases, in order to preserve integrity and consistency of data despite Byzantine faults [16]. However, their solution imposed a sequential execution of transactions, thus limiting concurrency. A more recent paper studied a large number of bugs in DBMSs, which have to be classified as Byzantine faults because they corrupt data [9]. To tolerate these faults, the authors proposed a middleware solution to handle transaction execution and to determine the consistency of the results by doing voting. Unfortunately, similarly to [16], their solution does not allow the execution of transactions concurrently.

Three more recent studies proposed new protocols for combining concurrency and consistency aspects on transaction processing with Byzantine faults: HRDB [13], Byzantium [14], and BFT Deferred Update [15]. All these work are solutions for database replication.

HRDB is a commit barrier scheduling protocol that allows concurrency between transactions, ensuring one-copy serializability consistency [13]. However, clients access the system through a centralized coordinator that cannot fail in a Byzantine way. The coordinator elects one replica to act as the primary, leaving the rest as backups. This coordinator is also responsible for detecting and solving conflicts between transactions, in order to avoid the occurrence of deadlocks. Even though the solution is interesting, in practice it is difficult to justify the assumption of an honest coordinator on a Byzantine environment.

On the other hand, Byzantium is a protocol that supports the execution of concurrent transactions in Byzantine environments, assuming a weak consistency criterion, snapshot isolation [14]. However, although snapshot isolation is much used, the literature shows that it is weaker than one-copy serializability. There is also evidence that this semantics is susceptible to anomalies that can affect data integrity [18], which in a Byzantine environment might be exploited by the adversary to cause data corruptions intentionally. Byzantium uses as a building block a BFT state machine replication library, PBFT [10].

Very recently a new protocol in the area appeared, BFT deferred update [15]. It is closer to ours and to DBSM than HRDB or Byzantium and it scales better because multiple transactions can execute simultaneously at different replicas, leaving ordering only to the transaction commitment process. This work is very similar to our solution, in the sense that we use the same update propagation technique. There are however several differences: (1) our solution tolerates Byzantine clients, while theirs does not; (2) we consider a single-version database to support more DBMSs, while they require a multi-version database, a feature not supported by some commercial DBMSs; (3) we do both reads and writes in the primary's copy of the database, while they keep writes in a buffer and execute them only when the transaction is committed.

III. The Protocol

The goal of this work is to propose and implement a Byzantine fault-tolerant protocol for database replication that uses a total order multicast primitive and the deferred update propagation principle. This work can be viewed as an adaptation of the DBSM [3] for Byzantine faults. DBSM, which tolerates only crash faults, has the following interesting properties: (*i*) it provides speculative execution of transactions by the replicas; (*ii*) despite replication, there is no need for distributed locks over data; and (*iii*) it ensures strong consistency in a fully distributed way. Moreover, DBSM seems to be fast in the crash-fault case [3].

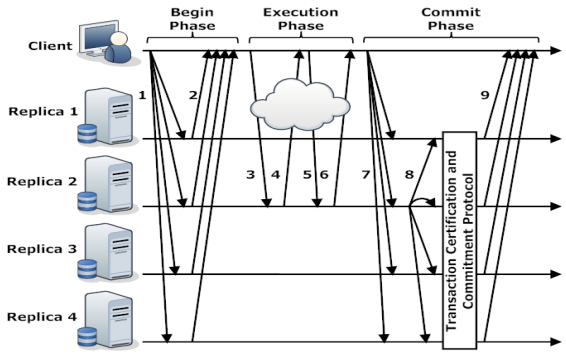


Fig. 1. Protocol execution

Our protocol is designed to preserve the ACID properties (Atomicity, Consistency, Isolation, Durability) even if

up to $f \leq \lfloor \frac{n-1}{3} \rfloor$ replicas deviate arbitrarily from their specification (n is the number of replicas). Furthermore, the protocol seeks good throughput and scalability by using an optimistic/speculative approach in the transaction operations execution, in accordance to the deferred update replication technique [19], [20]. Therefore, during the course of a transaction, a client interacts only with a primary replica, and the propagation of updates happens only when the transaction is committed. Together with the updates, the primary propagates also the *Read-Set* (RS) and the *Write-Set* (WS), which are the sets of the identifiers that were respectively read and written in the transaction.

Briefly, the protocol works as follows (see Figure 1). In step 1, the client begins a transaction by using the total order multicast primitive to send a message requesting this beginning to all replicas. When replicas deliver this message, they choose a primary for that transaction and reply the primary id to the client (step 2). Then, the client sends the transaction operations (reads and writes) directly to the primary replica, which executes and replies to each operation (steps 3-6). When the client requests the commitment of the transaction (step 7), it total order multicasts a request commit message to all replicas. In step 8, the primary replica attaches RS and the WS for that transaction to the request commit message (delivered on previous step) and total order multicasts it on a commit message to all replicas. Upon delivery of the commit message, every replica (except the primary) executes the transaction operations and tries to commit the transaction verifying if it is serializable. If the transaction is serializable, it is committed, otherwise, it is aborted. Finally, in step 9 every replica replies the outcome of the transaction to the client. Needless to say, the total order multicast primitive used in the protocol has to be Byzantine fault-tolerant [10], [11].

A. System Model

Processes are divided into two sets, $C = \{c_1, c_2, ..., c_m\}$ and $R = \{r_1, r_2, ..., r_n\}$, that represents clients and replicas, respectively. The C set has an arbitrary (but not infinite) number of clients, and the R set has cardinality $|R| \geq 3f + 1$. We assume that an unlimited number of clients and up to $f \leq \lfloor \frac{|R|-1}{3} \rfloor$ replicas may present Byzantine behavior. If a process deviates from its specification it is said faulty, otherwise it is said correct. A faulty process can deviate from its specification arbitrarily, e.g., by stopping, by omitting to send, receive or deliver messages, by replying incorrect results to the clients and by colluding with other faulty processes with some malicious purpose. Nevertheless, we assume that replicas fail independently due to the use of diversity [9]. Every replica has a complete copy of the database, and every database copy is single-version.

The termination of transactions cannot be ensured in an asynchronous environment, i.e., in a system in which the delay of communication and processing is unbounded [21]. Thus, we assume that system is eventually synchronous, i.e., that bounds for those delays eventually hold [22]. This model is quite realistic because periods of asynchrony tend to be followed by stable periods in which delay bounds do exist. Processes communicate by message passing, using point-to-point authenticated reliable FIFO channels. We assume that adversaries do not have power to subvert cryptographic mechanisms (e.g., signatures and message digests). We use a collision-resistant message digest and message authentication codes to ensure data integrity and authenticity.

We assume that the replicas are deterministic, in the sense that they support a common subset of data read/write operations (e.g., ANSI SQL) for which the result of the execution of an operation is the same in all replicas. We assume that clients only submit operations within that subset (or that replicas discard any operations outside that subset). A client submits a transaction only if it does not have any pending transaction and, in the context of a transaction an operation is send only if there are no pending operations.

B. Protocol Overview

As in the classical transaction model [21], in our protocol a transaction is a block of commands (or operations) which is initiated by the client using the *BEGIN* statement, followed by read and/or write statements, and terminated by the *COMMIT* or *ABORT* statement. Our protocol satisfies the following two properties:

- **Consistency** (safety): correct replicas have equivalent logical state, and clients always get correct answers to transactions that commit; in other words, the protocol provides one-copy serializability (1-SR) consistency;
- **Termination** (liveness): a transaction that reaches the *preparing* state eventually ends, even if there are conflicting and/or pending transactions blocking the same data items, i.e., a transaction in commit phase always ends.

As already pointed out, an obvious solution to make transactions Byzantine fault-tolerant would be to use a BFT state machine replication protocol. However, this approach is not adequate because of the overhead of ordering all operations and the possibility of causing a deadlock between two transactions. However, considering that a transaction takes effect in the system only when it is committed, we do not need to order all operations, but only the transaction commitment. This idea matches well the notion of deferred updates (or deferred writes), in which statements are executed speculatively in only one of the replicas, and propagated to the other replicas only when the transaction is committed. Similarly to [3], to ensure serializability we use a total order multicast to send transaction commit messages along with transaction statements propagation to the replicas. So, at the end of a transaction, the commit will happen in every replica in the same order, according to total order multicast properties [12], thus producing serializable executions.

A total order multicast is a protocol that guarantees that correct processes agree on the messages to be delivered and on the order of delivery [12]. Formally, a total order multicast algorithm ensures that: (*i*) if some process delivers a message m, then all correct processes deliver m (atomicity); (*ii*) every correct process delivers a message m in the same order (total

order); (*iii*) the correct processes deliver the message m only if m was previously multicast by some process (integrity); (*iv*) every correct process eventually delivers m (termination). Note that the problem is specified in terms of delivery of messages to the high level protocol, not of reception of messages at the process.

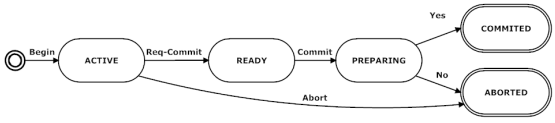


Fig. 2. Finite state machine that models the transaction processing protocol

Our protocol can be modeled as a finite state machine (see Figure 2). The state transitions to the intermediate states happen according to the type of messages delivered by the total order multicast protocol, and transitions to the final states happen according to the final outcome of the transaction certification test (details in Section III-D).

The normal case operation of the protocol is as follows. The transaction starts when a client c_i sends a *BEGIN* statement and message using the total order multicast. This primitive ensures that every correct replica delivers this message/statement, so the transaction moves to the *active* state. While in *active* state, the client sends the operations that compose the transaction to the primary replica only. After all operations are completed, the client requests the commitment of the transaction by means of a total order multicast *REQCOMMIT* message. Upon delivering *REQCOMMIT*, all correct replicas become aware of the transaction operations and their results (the transaction propagation to all replicas), and then every correct replica changes the transaction to the *ready* state. At this point, only the primary replica does something: it sends a *COMMIT* message with the operations and their results (to confirm the client's request) through the total order multicast protocol. Upon delivery of the *COMMIT* message, every correct replica changes the state to *preparing*. Then, they start a commitment process where the transaction is submitted to a certification test, in order to verify if it is serializable. If so, the transaction results received from the primary replica are compared to the local results and, if they match, the transaction is committed in the system (and database) and the state changed to *committed*. Otherwise, the transaction is aborted. Notice again that the *BEGIN*, *REQCOMMIT*, *COMMIT* and *ABORT* statements are delivered by all replicas in the same order due to the total order multicast, so all replicas follow the same sequence of states.

C. Begin, Execute, Request Commitment of Transactions

As stated in Section III-A, we assume that every message is signed and authenticated, so every correct replica receives and delivers only messages that come from authenticated clients. To prevent unauthorized entities from accessing objects of the database, this access is controlled using access control lists (ACL). The client sends the operations to the primary replica using a point-to-point reliable FIFO channel, so the operations are executed by the primary replica in the order they were sent by the client. To simplify understanding the protocol, we split the code executed by replicas in two parts, presented in Figures 3 and 4. In Figure 3, lines 12-17 correspond to the beginning of the transaction, lines 18-31 the speculative execution of the operations (done by primary replica), and lines 32-41 the request to commit the transaction. Due to space constraints, the client-side algorithm was omitted.

Variables:
1: $\prod^c = \emptyset$ {set of committed transactions}
2: $\prod^a = \emptyset$ {set of active transactions}
3: $t_i = \perp$ {transaction identifier}
4: $t_i^{ops} = \emptyset$ {list of transaction operations}
5: $t_i^{ans} = \emptyset$ {list of result for transaction operations}
6: $t_i^l = \perp$ {primary replica of a transaction}
7: $t_i^{tsb} = \perp$ {timestamp of beginning}
8: $t_i^{tsc} = \perp$ {timestamp of request commit}
9: $can\_commit = false$ {flag for commit}
10: $has\_exec = false$ {flag for re-execution}
11: $has\_redo = false$ {flag for redo}

upon: *TO-deliver*$(c_i, \langle BEGIN \rangle)$ {beginning of transaction}
12: $i \leftarrow last\_transaction\_id() + 1$
13: $t_i^{tsb} \leftarrow delivery\_order()$
14: $t_i^l \leftarrow i \bmod |R|$
15: $state(t_i) \leftarrow active$
16: $\prod^a \leftarrow \prod^a \cup \{\langle t_i, t_i^{tsb}, \perp, t_i^l, t_i^{ops}, t_i^{ans} \rangle\}$
17: $send(r_k, \langle ACTIVE, t_i, t_i^l \rangle)$ *to* c_i

upon: *receive*$(c_i, \langle r_i, t_j, op^{order}, op \rangle)$ *from* c_i {speculative execution of operations}
18: **if** $\exists t_j \in \prod^a \wedge state(t_j) = active$ **then**
19: $\langle t_i, t_i^l, t_i^{ops}, t_i^{ans} \rangle \leftarrow get\_context(\prod^a, t_j)$
20: **if** $r_k = t_i^l$ **then**
21: **if** $op^{order} = (max(t_i^{ops}) + 1)$ **then**
22: $result \leftarrow execute(op)$
23: $t_i^{ops} \leftarrow t_i^{ops} \cup \{op\}$
24: $t_i^{ans} \leftarrow t_i^{ans} \cup \{result\}$
25: $send(r_k, \langle result \rangle)$ *to* c_i
26: **else if** $op^{order} = max(t_i^{ops})$ **then**
27: $result \leftarrow t_i^{ans}[order]$
28: $send(r_k, \langle result \rangle)$ *to* c_i
29: **end if**
30: **end if**
31: **end if**

upon: *TO-deliver*$(c_i, \langle REQCOMMIT, t_j, t_j^{ops}, H(t_j^{ans}) \rangle)$ {process request commitment}
32: **if** $\exists t_j \in \prod^a \wedge state(t_j) = active$ **then**
33: $\langle t_i, t_i^{tsb}, \perp, t_i^l, t_i^{ops}, t_i^{ans} \rangle \leftarrow get\_context(\prod^a, t_j)$
34: $state(t_i) \leftarrow ready$
35: $t_i^{tsc} \leftarrow delivery\_order()$
36: $commit\_data \leftarrow commit\_data \cup \{\langle t_j, t_i^{tsc}, t_j^{ops}, H(t_j^{ans}) \rangle\}$
37: **if** $r_k = t_i^l$ **then**
38: $\langle RS, WS \rangle \leftarrow get\_readwrite\_Sets(t_i)$
39: *TO-multicast*$(r_k, \langle COMMIT, t_i, t_i^{tsc}, RS, WS, t_i^{ops}, H(t_i^{ans}) \rangle)$
40: **end if**
41: **end if**

Fig. 3. Transaction begin, execution and request commit protocol executed by replica r_k

The transaction starts when every correct replica delivers the *BEGIN* message sent by the client (lines 12-17). Upon this delivery, replicas assign an id and a *timestamp* to the new transaction (lines 12-13). Function $last\_transaction\_id()$ returns the id of the last transaction and function $delivery\_order()$ returns the sequence number of the last message delivered by the total order multicast primitive. Line 14 defines the primary replica for that transaction. Then, replicas set their state to *active* and include data about the transaction in the set of active transactions $\prod^a$ (lines 15-16). Finally, the replicas send the client acknowledgements that the transaction started, together with its id and the primary replica identifier t_i^l. A client accepts that information when it receives $f+1$ acknowledgements from different replicas.

Once a transaction t_j is started, the client can execute read and write operations. They are executed in a speculatively way by the primary replica r_k (lines 18-31). Remember that in this phase the client interacts only with the primary replica. When the primary replica receives an operation, it checks whether transaction t_j is *valid* – a transaction is valid if it

was previously started by every correct replica – and if the transaction state is *active*, because this is the only state in which a transaction can receive statements (cf. Section III-B). If some of these conditions are false, the operation is simply discarded. If these conditions are true, the primary replica gets data about transaction t_j from $\prod^a$ (function $get\_context()$ does this for transaction id) and checks if it is in fact the primary replica for that transaction in line 20 (a malicious client can send operations to a replica that is not the primary). Then, the primary checks whether the operation identifier is an unit greater than the last executed operation and, if so, it executes the operation, saves the statement and its results in retention buffers for future use, and sends the reply to the client (lines 21-25). Otherwise, if the number of the operation is the same as the last executed, the replica treats it as a resend to the last executed statement, so it sends the last result to the client (lines 26-28).

The code in lines 32-41 deals with the delivery of a *REQCOMMIT* message, which is sent by the client to request the commitment of the transaction. Upon delivery of *REQCOMMIT*, every replica verifies if transaction t_j is valid and if its state is *active*. Then, correct replicas retrieve data about the transaction from $\prod^a$ and set the transaction state to *ready* (lines 33-34). From this point, transaction t_j is no longer able to receive and execute new statements, but just waits for its commitment. In the next step, replicas mark the order in which *REQCOMMIT* message was delivered as the *timestamp* for the request commit. These *timestamps* are used to verify which transactions precede t_j. Then, the replicas save data in set $commit\_data$, used later to verify if data provided by the primary replica is correct. The steps of lines 38-39 are done just by the primary replica, the only one that has the results of the operations. It gets the data items affected by read and write operations (the read and write sets), and sends this information to all replicas in a *COMMIT* message. This is when the transaction is propagated to the group of replicas. The transaction commit protocol is presented in the next section.

upon: $TO\text{-}deliver(r_k, \langle COMMIT, t_j, t_j^{tsc}, RS, WS, t_j^{ops}, H(t_j^{ans})\rangle)$

```
1: if ∃t_j ∈ ∏^a ∧ state(t_j) = ready then
2:     if ⟨t_j, t_j^tsc, t_j^ops, H(t_j^ans)⟩ ∈ commit_data then
3:         ⟨t_i, i, t_i^r, t_i^l, t_i^ops, t_i^ans⟩ ← get_context(∏^a, t_j)
4:         state(t_i) ← preparing
5:         can_certify ← true
6:         if t_i^l = r_k then
7:             if has_redo = true then
8:                 undo_transaction(t_i)
9:                 has_exec ← true
10:            else
11:                has_exec ← false
12:            end if
13:        else
14:            has_exec ← true
15:        end if
16:        if has_exec = true then
17:            t_i^ops ← ⊥
18:            request_locks(⟨RS, WS, t_j^ops⟩, delivery_order())
19:            for all t_k ∈ ∏^a : {RS(t_k) ∩ WS ≠ ∅ ∨ WS(t_k) ∩ WS ≠ ∅} do
20:                if state(t_k) = active then
21:                    abort_transaction(t_k)
22:                else if state(t_k) = ready then
23:                    [ undo_transaction(t_k) ; has_redo(t_k) ← true ]
24:                end if
25:                ∏^a ← ∏^a \{t_k}
26:            end for
27:            if acquire_locks() = true then
28:                for all op_j ∈ t_j^ops do
29:                    result ← execute(op_j)
30:                    [ t_i^ops ← t_i^ops ∪ {op_j} ; t_i^ans ← t_i^ans ∪ {result} ]
31:                    t_j^ops ← t_j^ops \ {op_j}
32:                end for
33:            else
34:                can_certify ← false
35:            end if
36:        end if
37:        if
           (¬(∃T_j ∈ ∏^c : {T_j ⇝ t_i ∧ (WS(T_j) ∩ RS ≠ ∅)}) ∧ (can_certify = true)
           then
38:            can_commit ← true
39:        else
40:            can_commit ← false
41:        end if
42:        if can_commit = true ∧
           matches(⟨H(t_i^ops), H(t_i^ans)⟩, ⟨H(t_j^ops), H(t_j^ans)⟩) then
43:            [ outcome ← COMMITED ; T_i ← t_i ; ∏^c ← ∏^c ∪{T_i} ; ∏^a ←
               ∏^a \{t_i} ]
44:            [ state(t_i) ← commited ; commit_transaction(T_k) ]
45:        else
46:            ∏^a ← ∏^a \{t_i}
47:            [ outcome ← ABORTED ; abort_transaction(t_i) ]
48:        end if
49:        send(r_k, ⟨outcome, t_i⟩) to c_i
50:    else
51:        ∏^a ← ∏^a \{t_i}
52:        [ state(t_j) ← aborted ; abort_transaction(t_j) ]
53:        send(r_k, ⟨ABORTED, t_i⟩) to c_i
54:    end if
55: end if
```

Fig. 4. Transaction commit protocol

D. Transaction Commit Protocol

This part of the protocol is in Figure 4. Its purpose is to finish the transaction – committing or aborting it – while ensuring one-copy serializability consistency [21]. When a client wishes to commit its transaction, it atomically multicasts a signed *REQCOMMIT* message to all replicas. When replicas deliver this message, just the primary extracts the identifiers of data records that were affected by that transaction (RS and WS) and atomically multicasts a signed *COMMIT* message to all replicas, in order to confirm the commit request from the client. The protocol uses signatures as a mean to guarantee that clients issue commit requests only to their own transactions.

The *COMMIT* message contains: the identifier of the transaction that is being committed (t_j); a list of statements executed in the transaction (t_j^{ops}); a cumulative hash (message digest) based on the results received for each statement executed by the primary replica ($H(t_j^{ans})$); the identifiers for each data item affected by read and write operations of the transaction (RS and WS); and the *timestamp* that was marked at delivery of *REQCOMMIT* message (t_j^{tsc}), which is important to identify concurrent/conflicting transactions. Upon the delivery of *COMMIT*, every replica checks if the transaction is valid and whether it is in the *ready* state, the only state in which commit is allowed. The condition of line 2 verifies if the transaction that the client requested the commitment is the same that was performed by the primary replica for that transaction. This step allows to identity spurious transactions that were created by Byzantine clients or by a Byzantine primary replica. From this point, the transaction goes to the *preparing* state, where it acquires priority on grant locks over the other active transactions. The flag of line 5 is explained later. Lines 6-12 are executed only by the primary replica. In this case, the primary replica firstly checks if transaction must be executed again (condition from line 7), despite its operations already performed in speculative way. This can happen if a transaction in *ready* state locks some data item that conflicts with other transaction that are in *preparing* state (line 23). Thus, the primary replica undoes the entire transaction (line 8) and marks it for re-execution (line 9). This means that the

transaction will be executed again by the primary replica before trying to commit it.

Lines 16-36 deal with the execution of the transaction operations (reads/writes in t_j^{ops}) by the replicas other than the primary. The primary replica executed them before speculatively, so it executes these statements again only if the condition of line 7 is true. Then, the replicas initialize their list of received statements for local execution and, in line 18, they request the appropriated locks for each transaction data item, as well as for each statement in the received list (t_j^{ops}). The lock grant is done purposely over RS and WS together in order to verify if the statements affect no more data items that those that are in RS and WS (a Byzantine entity could send invalid data items or statements). Thus, the locks will be granted only if the statements and the RS and WS are tuned. Note that transactions are allowed to lock their data items in the order in which they are delivered. Afterwards (lines 19-26), the replicas check if there exists some active transaction running locally that holds the locks for some data items that will be affected by the transaction that is being committed, because if so, the former is preventing the latter from getting the locks. If $\prod^a$ is not empty in line 19, the local running transaction is: (*i*) simply aborted, if its state is *active*; or (*ii*) undone and flagged for running again, if its state is *ready* (note that, at this point it is not possible to abort the transaction due to the client having requested its commitment). After that, if the transaction operations, RS and WS are tuned, the locks are granted and it starts the execution for statement list delivered along with *COMMIT* message (lines 27-32). On the contrary, if the locks are not granted for some reason, the flag for certification will be turned off (line 34) and the transaction will be aborted.

If locks are successfully granted, the statements are executed, and the replicas have to *certify* the transaction prior to its commitment. This procedure is done through a certification test that will checks if the transaction is consistent and can be serialized with other committed transactions (i.e., if it is serializable, line 37). This test is based on Kung-Robinson's certification [19], that simply verifies if there are some conflict between the transaction that is being committed (t_i) and concurrent transactions that were already committed (t_j). A transaction t_j is said to be *concurrent* with (i.e., does not proceed) transaction t_i if it was committed on the interval between the *timestamps* of delivery of *REQCOMMIT* and *COMMIT* messages of transaction t_i. Any transaction t_j that does not verify this condition is said to precede t_i, and is not considered by the certification test. To help understanding the certification test, Figure 5 presents a conflict situation. It shows two transactions, T_1 and T_2. In this case, the request commit for T_2 is delivered before the request commit for T_1, due to *REQCOMMIT* message for T2 be delivered at *timestamp* 6, while *REQCOMMIT* message for T1 is delivered at *timestamp* 8. As long as T_2 delivers its *COMMIT* at *timestamp* 11, it is possible to verify that no other transaction was committed between *timestamps* 6 and 11, only a request to commit T_1 was delivered. Thus, T_2 can be committed safely because there are no conflict or concurrent transactions that will be considered by the certification test. As opposed to T_2, T_1 needs to be certified against T_2, due to T_2 having been committed after its request commit, but before its commit (between *timestamps* 8 and 19). This means that T_2 is concurrent with T_1, so, it will be considered by the certification test on T_1 commitment. In this case, if T_1 has been read some data written by T_2, T_1 will abort because it will hurt the serializability (a phenomena called dirty-read); otherwise T_1 will be committed. Once transactions are delivered by correct replicas in the same order, only read/write conflicts must be avoided, and write/write conflicts are solved directly by the order of applying updates.

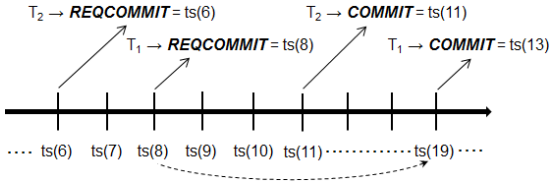


Fig. 5. Sample of concurrent transactions

Finally, if the transaction passes the certification test, it remains to verify if the results obtained for each statement on the local execution are the same as those received from the primary replica (already sent in a speculative way to the client). This comparison is done through function *matches* (line 42), which takes as arguments the hashes that are calculated over both list of statements and list of results received from primary and the same hashes calculated over lists gotten from local execution. If these hashes match, the transaction is committed and terminated (lines 43-44), otherwise it is aborted (lines 46-47). Once the transaction is committed, it is included in the set of committed transactions ($\prod^c$), withdrawn from the set of active transactions ($\prod^a$), and made persistent in the database (line 44, $commit\_transaction$ function). If the transaction is aborted, it is removed from the set of active transactions ($\prod^a$) and its statements are undone in the database (line 47, $abort\_transaction$ function). At the end, replicas send a reply to the client (line 49) and, after receiving at least $f+1$ identical replies from replicas, the client can determine the transaction's outcome.

IV. Implementation and Evaluation

This section presents an analytical and an experimental evaluation of our protocol, which is compared with some of the other solutions in the literature. Our protocol is simply designated by OP (from "our protocol").

A. Analytical Analysis

A first evaluation is a comparison of some of the characteristics of our protocol with the main related work: HRDB [13], Byzantium [14] and BFT-Deferred Update (BFT-DU) [15]. The results reported on Table I come from the transaction commit protocol of the respective protocols, which is the heaviest part of the transaction processing. The communication related to the beginning of the transactions and the operations is not considered.

TABLE I
COST AND PROPERTIES OF TRANSACTION COMMIT PROTOCOLS (*TOMCast* IS THE NUMBER OF COMMUNICATION STEPS OF A TOTAL ORDER MULTICAST)

Protocol Name	Features and Properties					
	# Replicas	# Communication steps	Message complexity	Consistency	Centralized	Byzantine
OP	$3f+1$	$2(TOMCast)+1$	$O(n^2)$	Serializable (Strong)	No	Servers and Clients
HRDB	$2f+1+controller$	4	$O(n)$	Serializable (Strong)	Yes	Servers and Clients
Byzantium	$3f+1$	$(TOMCast)+1$	$O(n^2)$	Snapshot (Weaker)	No	Servers and Clients
BFT-DU	$3f+1$	$(TOMCast)+1$	$O(n^2)$	Serializable (Strong)	No	Only Servers

Table I shows that in terms of replicas needed, HRDB is the best solution for $f > 1$, due to it centralized control. HRDB is also the best in number of communication steps and message complexity. Byzantium and BFT-DU have the same number of steps, because they resort to a single total order multicast on their transaction commit protocol. Our protocol (OP) uses more steps because we have two total order multicasts. However, considering that these protocols are not appropriate for large scale environments, the relative cost of message transmissions and number of communication steps is fair. The message complexity is equal for OP, Byzantium and BFT-DU. The costs for HRDB are better, which was expected once this protocol has a centralized control (a weakness, in our opinion), so it is the coordinator who decides the order to execute the transaction statements and no agreement is needed.

The good results on number messages of Byzantium and BFT-DU come from the fact that they invokes only once the total order multicast protocol, but in Byzantium, its weaker consistency model (snapshot isolation) requires only to check write conflicts on data items, because concurrent transactions can only commit if they do not update the same data items (the first-committer-wins rule). Although this is favorable to Byzantium, the snapshot isolation model may be problematic in some cases, as already pointed out. In the case of BFT-DU, just one invoke on total order multicast is enough to achieve commitment, because in this protocol the clients are reliable (i.e., they are assumed not to suffer Byzantine faults). So, the transaction commitment becomes simpler. Even though it needs a higher number of communication steps than other protocols, ours is the only one that ensures the strong consistency of data on fully Byzantine environments, tolerating both Byzantine servers and clients.

B. Experimental Results

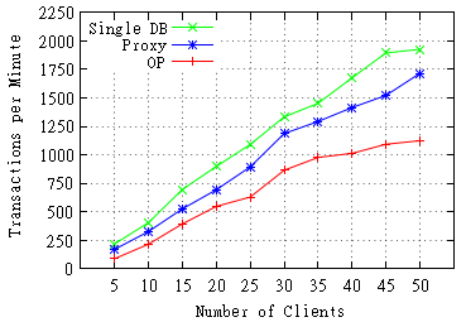
We implemented a prototype of our protocol in the context of a replicated SQL database. The prototype was developed in order to run with unmodified versions of several commercial and open source DBMSs as replicas. The implementation was done in Java due to its underlying security features (sandbox, memory safety, etc.) and the simple connection with DBMSs (JDBC). Our experiments involved executing OP and a single DBMS as a baseline. We did not compare with the other protocols as we did not have access to their implementations. Our experiments were conducted using the workload of the industry standard on-line transaction processing TPC-C benchmark (http://www.tpc.org/tpcc), which represents a generic wholesale supplier workload. TPC-C was used not only to measure the overhead of our protocol, but also to assess its scalability, as this benchmark produces a high concurrency workload. TPC-C defines five transaction types: New Order (NO), Payment (P), Delivery (D), Order Status (OS) and Stock Level (SL). It is noteworthy that only update transactions may cause conflicts, i.e., only D, P and NO. These three transaction types compose 92% of the TPC-C workload, i.e., 92% are read-write transactions and 8% are read-only transactions.

The experiments were done in a LAN environment with Dell Optiplex 755 machines, with the following configuration: one Intel Core 2 Duo 2.33GHz processor, 2GB of RAM and one NIC/Ethernet Gigabit Interface Intel 82566DM-2. As operating system we adopted Debian GNU/Linux 6.0 with kernel 2.6.32-5-686 #1 SMP. The database system used in experiments was MySQL 5.5.8. In Figure 6(a) the single database results were taken from MySQL. The JVM we used was Sun 1.6.0_24. Our experiments were done with four replicas ($n = 3f + 1 = 4$), thus at most one faulty replica ($f = 1$). The servers run in four machines, but that was not always the case for the clients that shared other ten machines (i.e. five clients per machine).

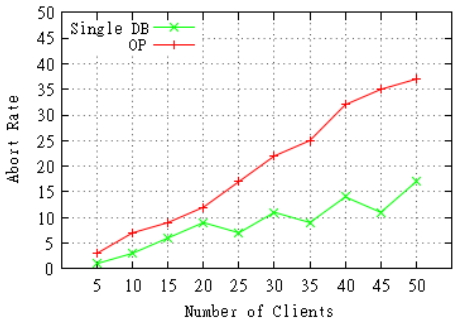
The experiments compared the performance of TPC-C with: a single non-replicated database accessed through standard JDBC interface (Single DB); a replicated database using our protocol (OP); a single non-replicated database accessed through our own JDBC interface (used in our protocol), using neither replication nor total order multicast (Proxy).

Figure 6 shows the experimental results. The values were taken from the average of 25 executions of each experiment. As we can see in Figure 6(a), our replication protocol apparently scales fairly despite the weight of the commit protocol and the total order multicasts. This is probably due to the fact that the protocol rotates the primary for each transaction started and it is the primary that first, speculatively, executes transaction operations, so the rotation provides a form of load balancing. The throughput for a single database and the proxy configuration are shown for reference, only with one replica because they do not support replication. The overhead of our protocol comes mostly from the commit protocol. When a transaction is requested to commit, every replica starts a distributed certification process due to the propagation technique that was adopted: deferred updates. With this technique, only when the client requests the transaction commitment the statements executed are propagated to the other replicas, so that their states can converge. This adds several communication steps, including two total order multicasts, and processing in the end of each transaction.

Figure 6(b) shows a second aspect of the performance of the protocol: it aborts more transactions than the non-replicated



(a) Scalability of TPC-C transactions.



(b) Abort rate (%) of TPC-C transactions.

Fig. 6. Measurements on standard TPC-C workload with no batches

database. This higher abort rate is a known side effect of the use of speculative execution. It is important to notice that the aborts are also due to the workload of TPC-C. Specifically, we used a setup that induces some conflicts: 8 warehouses and 10 districts per warehouse.

We believe that these preliminary experiments show that our protocol has a performance that is competitive with a non-replicated database, while providing stronger guarantees. They also show that there are several tradeoffs that lead to better or worse results in terms of some of the metrics considered, so it is up to the system designer to decide which is more adequate for a specific application.

V. Conclusion

The paper presents a Byzantine fault-tolerant transaction processing and commit protocol based on the *deferred update* technique, where an arbitrary (but finite) number of clients and a limited number of servers can fail in Byzantine way. The design of our protocol is based on the total order multicast abstraction, a widely used building block for the design of reliable distributed systems. The deferred update technique is a largely used database replication strategy, because it is considered to be more scalable than other replication strategies. Also, our protocol ensures strong consistency through serializability, and it is the first that tolerates Byzantine clients and servers in a fully distributed way in this consistency model.

Acknowledgments: This work was partially supported by CNPq (Brazilian National Research Council) through processes 482175/2010-9 and 560258/2010-0 and FCT through the PIDDAC Program funds (INESC-ID multiannual funding) and project PTDC/EIA-IA/100581/2008 (REGENESYS)

References

[1] B. W. Lampson, "Atomic transactions," in *Distributed Systems - Architecture and Implementation, An Advanced Course*. Springer-Verlag, 1981, pp. 246–265.

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

[3] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[4] O. Babaoğlu and S. Toueg, "Non-blocking atomic commitment," in *Distributed systems*, 2nd ed. ACM Press/Addison-Wesley, 1993, pp. 147–168.

[5] R. Guerraoui and A. Schiper, "The generic consensus service," *IEEE Transactions on Software Engineering*, vol. 27, pp. 29–41, January 2001.

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *SIGMOD '96: Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1996, pp. 173–182.

[7] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases (extended abstract)," in *Euro-Par'97: Proceedings of the 3rd International Euro-Par Conference on Parallel Processing*, 1997, pp. 496–503.

[8] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programing Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[9] 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*, vol. 4, no. 4, pp. 280–294, 2007.

[10] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *OSDI '99: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, Feb. 1999, pp. 173–186.

[11] P. Zielinski, "Paxos at war," University of Cambridge Computer Laboratory, Cambridge, UK, Tech. Rep. UCAM-CL-TR-593, Jun. 2004.

[12] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[13] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling," in *SOSP'07: Proceedings of 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[14] 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 - EuroSys'11*. ACM, 2011.

[15] 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 - LADC'11*. SBC, 2011.

[16] H. G. Molina, F. Pittelli, and S. Davidson, "Applications of byzantine agreement in database systems," *ACM Transactions on Database Systems*, vol. 11, no. 1, pp. 27–47, 1986.

[17] F. B. Schneider, "Implementing fault-tolerant service using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[18] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi SQL isolation levels," in *SIGMOD'95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995, pp. 1–10.

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

[20] B. W. Lampson, "Lazy and speculative execution in computer systems," in *OPODIS'06: Proceedings of the 10th International Conference on Principles of Distributed Systems*, ser. Lecture Notes in Computer Science, A. A. Shvartsman, Ed., vol. 4305. Springer-Verlag, 2006, pp. 1–2.

[21] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[22] C. Dwork, N. A. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of ACM*, vol. 35, no. 2, pp. 288–322, Apr. 1988.