Byzantine Fault-Tolerant Deferred Update Replication

Conference Paper *in* Journal of the Brazilian Computer Society · April 2011

DOI: 10.1109/LADC.2011.10 · Source: DBLP

3 authors:

Fernando Pedone
University of Lugano
207 PUBLICATIONS **4,535** CITATIONS
SEE PROFILE

Nicolas Schiper
Cornell University
23 PUBLICATIONS **512** CITATIONS
SEE PROFILE

José Enrique Armendáriz-Iñigo
Universidad Pública de Navarra
108 PUBLICATIONS **587** CITATIONS
SEE PROFILE

Byzantine Fault-Tolerant Deferred Update Replication

Fernando Pedone
University of Lugano (USI)
Lugano, Switzerland
fernando.pedone@usi.ch

Nicolas Schiper
University of Lugano (USI)
Lugano, Switzerland
nicolas.schiper@usi.ch

José Enrique Armendáriz-Iñigo
Universidad Pública de Navarra
Navarra, Spain
enrique.armendariz@unavarra.es

***Abstract*—Replication is a well-established approach to increasing database availability. Many database replication protocols have been proposed for the crash-stop failure model, in which servers fail silently. Fewer database replication protocols have been proposed for the byzantine failure model, in which servers may fail arbitrarily. This paper considers deferred update replication, a popular database replication technique, under byzantine failures. The paper makes two main contributions. First, it shows that making deferred update replication tolerate byzantine failures is quite simple. Second, the paper presents a byzantine-tolerant mechanism to execute read-only transactions at a single server.**

***Keywords*-Database replication, byzantine fault-tolerance, dependable systems.**

I. Introduction

Replication is a well-established approach to increasing database availability. By replicating data items in multiple servers, the failure of some servers does not prevent clients from executing transactions against the system. Database replication in the context of crash-stop failures has been largely studied in the past years (e.g., [4], [10], [13], [16], [17]). When a crash-stop server fails, it silently stops its execution. More recently, a few works have considered database replication under byzantine failures (e.g., [18], [21]). Byzantine failures are more severe than crash-stop failures since failed servers can present arbitrary behavior.

Several protocols for the crash-stop failure model are based on deferred update replication. According to deferred update replication, to execute a transaction, a client first picks a server and submits to this server its transaction commands. The execution of a transaction does not cause any communication among servers until after the client requests the transaction's commit, at which point the transaction enters the termination phase and is propagated to all servers. As part of termination, each server certifies the transaction and commits it, if doing so induces a *serializable* execution, i.e., one in which transactions appear to have been executed in some serial order.

Deferred update replication scales better than state-machine replication and primary-backup replication. With state-machine replication, every update transaction must be executed by all servers. Thus, adding servers does not increase the throughput of update transactions. With primary-backup replication, the primary first executes update transactions and then propagates the database changes to the backups, which apply them without re-executing the transactions. The throughput of update transactions is limited by the capacity of the primary, not by the number of replicas. Deferred update replication scales better because it allows all servers to act as "primaries", locally executing transactions and then propagating the modifications to the other servers. As applying transaction modifications to the database is usually cheaper than executing transactions, the technique provides better throughput and scalability.

Ensuring strong consistency despite multiple co-existing primaries requires servers to synchronize. This is typically done by means of an atomic broadcast protocol to order transactions and a certification test to ensure the consistency criterion of interest. One of the key properties of deferred update replication is that read-only transactions can be executed by a single server, without communication across servers. This property has two implications. First, in geographically distributed networks it can substantially reduce the latency of read-only transactions. Second, it enables read-only transactions to scale perfectly with the number of servers in the system.

This paper considers deferred update replication under byzantine failures. It proposes the first deferred update replication protocol that is faithful to its crash-stop counterpart: (i) the execution of a transaction does not require communication across servers, only its termination does, and (ii) only one server executes the transaction commands, but all correct servers apply the updates of a committing transaction. Our protocol is surprisingly simple and similar to a typical crash-stop deferred update replication protocol, although based on a more strict certification procedure to guarantee that transactions only commit if they do not violate consistency and read valid data (i.e., data that was not fabricated by a byzantine server).

Our most significant result is a mechanism to execute read-only transactions at a single server under byzantine failures. Some protocols in the crash-stop model achieve this property by carefully scheduling transactions so that they observe a consistent database view. In the byzantine failure model, however, clients may inadvertently execute a read-only transaction against a byzantine server that fabricates a

bogus database view. In brief, our solution to the problem consists in providing enough information for clients to efficiently tell whether the data items read form a valid and consistent view of the database. Clients are still subject to malicious servers executing read-only transactions against old, but consistent, database views. We discuss in the paper the extent of the problem and remedies to such attacks.

The remainder of this paper is organized as follows. Section II describes the system model. Sections III and IV discuss deferred update replication in the crash-stop failure model and in the byzantine failure models respectively. Section V discusses related work. Section VI concludes the paper.

II. System model and definitions

In this section, we detail the system model and assumptions common to both the crash-stop failure model and the byzantine failure model. Further assumptions, specific to each model, are detailed in Sections III and IV.

A. Clients, servers and communication

Let $C = \{c_1, c_2, ...\}$ be the set of *client* processes and $S = \{s_1, s_2, ..., s_n\}$ the set of *server* processes. Processes are either *correct*, if they follow their specification and never fail, or *faulty*, otherwise. We distinguish two classes of faulty processes: *crash-stop* and *byzantine*. Crash-stop processes eventually stop their execution but never misbehave; byzantine processes may present arbitrary behavior.

We study deferred update replication in two models: in one model faulty servers are *crash-stop*; in the other model faulty servers are *byzantine*. In either case, there are at most f faulty servers, and an unbounded number of crash-stop-faulty clients.

Processes communicate by message passing, using either one-to-one or one-to-many communication. One-to-one communication is through primitives send(m) and receive(m), where m is a message. If sender and receiver are correct, then every message sent is eventually received. One-to-many communication is based on atomic broadcast, through the primitives abcast(m) and deliver(m), and used by clients to propagate messages to the group of servers.

In the crash-stop model, atomic broadcast ensures that (i) if one server delivers a broadcast message, then all correct servers also deliver the message; and (ii) no two servers deliver any two messages in different orders. In the byzantine model, atomic broadcast ensures that (i) if one correct server delivers a broadcast message, then all correct servers also deliver the message; and (ii) no two correct servers deliver any two messages in different orders.

B. Transactions and serializability

Let $X = \{x_1, x_2, ...\}$ be the set of data items, i.e., the *database*, $Cmd = \{\mathsf{commit}, \mathsf{abort}\} \cup (\{\mathsf{r}, \mathsf{w}\} \times X \times V)$ the set of *commands*, where V is the set of possible values of a data item, and $S = C \times Cmd$ the set of *statements*. Statement $(c, (\mathsf{r}, x, v))$ means that client c has read item x with value v; statement $(c, (\mathsf{w}, x, v))$ means that c has modified the state of x to v.

We define a *history* h as a finite sequence of statements in S. We define the projection $h|_c$ of history h on $c \in C$ as the longest subsequence h' of h such that every statement in h' is in $c \times Cmd$. In a projection $h|_c = \sigma_0 ... \sigma_m$, statement σ_i is *finishing* in $h|_c$ if it is a commit or an abort; σ_i is *initiating* if it is the first statement in $h|_c$ or the previous statement σ_{i-1} is a finishing statement.

A sequence of commands $t = \sigma_0 ... \sigma_m$ in $h|_c$ is a *transaction* issued by c if (i) σ_0 is initiating in $h|_c$ and (ii) σ_m is either finishing in $h|_c$ or it is the last statement in $h|_c$. Transaction t is *committing* if σ_m is a commit statement. We denote as $com(h)$ the longest subsequence h' of h such that every statement in h' is part of a committing transaction in h. In other words, $com(h)$ is the committed projection of h, with all statements of all committed transactions in h.

Let t and u be transactions in a history h. We say that t precedes u in h, $t <_h u$, if the finishing statement of t occurs before the initiating statement of u in h. A history h is *serial* if for every pair (t, u) of transactions in h, either $t <_h u$ or $u <_h t$. History h is *legal* if in h (i) every read statement $(c_i, (\mathsf{r}, x, v_j))$ is preceded by a write statement $(c_j, (\mathsf{w}, x, v_j))$ and (ii) in between the two there is no statement $(c_k, (\mathsf{w}, x, v_k)), v_j \neq v_k$.

History h is *serializable* if there is a serial permutation h' of $com(h)$ such that for each data item x, $h'|_x$ is legal, where $h'|_x$ is the projection of h' on x. *Serializability* is the set of all serializable histories.

III. Deferred Update Replication

In this section, we review deferred update replication in the crash-stop failure model. In this model, some atomic broadcast protocols require a majority of correct processes [11]. Thus, we assume the existence of $2f+1$ correct servers in the system. The database is fully replicated, that is, every server has a complete copy of the database.

A. Overview

In deferred update replication, transactions pass through two phases in their lifetime: the *execution phase* and the *termination phase*. The execution phase starts when the client issues the first transaction command; it finishes with a client's request to commit or abort the transaction, when the termination phase starts. The termination phase finishes when the transaction is committed or aborted.

Before starting a transaction t, a client c must select the server s that will receive and execute t's commands; other servers will not be involved in t's execution. Each data item in the server's database is a tuple (x, v, i), where x is the item's unique identifier, v is x's value and i is the value's version. We assume that read and write commands

on database tuples are atomic operations. When s receives a read command for x from c, it returns the current value of x (or the most up-to-date value if the database is multiversion) and its corresponding version. Write commands are locally stored by c. It is only during transaction termination that updates are propagated to the servers.

In the termination phase, the client atomically broadcasts t's *readset* and *writeset*, denoted respectively by $t.rs$ and $t.ws$—for simplicity, we say that "c broadcasts t". The readset of t is the set of all tuples (x, i), where x is a data item read by t and i the version of the value read; the writeset of t is the set of all tuples (x, v), where x is a data item written by t and v is x's new value. Notice that the readset does not contain the values read.

Upon delivering t's termination request, s *certifies* t. Certification ensures a serializable execution; it essentially checks whether t's read commands have seen values that are still up-to-date when t is certified. If t passes certification, then s executes t's writes against the database and assigns each new value the same version number k, reflecting the fact that t is the k-th committed transaction at s.

To certify t, s maintains a set CT of tuples (i, up), where up is a set with the data items written by the i-th committed transaction at s. We state the certification test of t, $C_{cs}(t.rs, CT)$, more formally with the predicate below.

$$C_{cs}(t.rs, CT) \equiv \forall (x, i) \in t.rs : \nexists (j, up) \in CT \text{ s.t. } (x \in up) \text{ and } (j > i) \quad (1)$$

If t passes certification, then s updates the database and CT. Certifying a transaction and creating new database items is an atomic operation. When a new version of x is created, the server can decide to keep older versions of x or not. If multiple versions of a data item exist, then we say the database is *multi-version*; if not it is *single-version*.

B. Algorithm in detail

Algorithms 1 and 2 are high level descriptions of the client's and server's protocol. Notice that to determine the outcome of a commit request, the client waits for a reply from a single server. For brevity, we do not show in Algorithm 1 the case in which the client decides to abort a transaction. In Algorithm 2, $items(t.ws)$ returns the set of data items in $t.ws$, without the values written by t, that is, $items(t.ws) = \{x \mid (x, v) \in t.ws\}$.

C. Read-only transactions

We describe two mechanisms to allow read-only transactions to be executed by a single server only. One mechanism is based on multi-version databases and does not require updates from committing transactions to be synchronized with on-going read-only transactions at a server; the other mechanism assumes a single-version database but synchronizes the updates of committing transactions with read-only transactions at servers.

With multi-version databases, each server stores multiple versions of each data item (limited by a system parameter). When a transaction t issues its first read command, the client takes the version of the value returned as a reference for future read commands. This version number specifies the *view* of the database that the transaction will see, $t.view$. Every future read command must contain $t.view$. Upon receiving a read command with $t.view$, the server returns the most recent value of the item read whose version is equal to or smaller than $t.view$. If no such a value is available, the server tells the client that t must be aborted. This technique is sometimes called *multiversion timestamps* [8].

If a single version of each data item exists, then read commands must be synchronized (e.g., through two-phase locking [2]) with the updates of committing transactions. During the execution of a transaction t, each read command

Algorithm 1: Deferred update replication, client c's code:

```
Command (c, (r, x, v)) occurs as follows:
  if first command for t then
    t.rs ← ∅
    t.ws ← ∅
    choose some server s ∈ S
  if (x, v) ∉ t.ws then
    send (c, (r, x)) to s
    wait until receive (c, (x, v, i)) from s
    t.rs ← t.rs ∪ {(x, i)}
    return v
  else
    return v such that (x, v) ∈ t.ws
Command (c, (w, x, v)) occurs as follows:
  t.ws ← t.ws ∪ {(x, v)}
A commit command is executed as follows:
  if t.ws ≠ ∅ then
    abcast(c, commit-req, t.rs, t.ws) to all servers
    wait until receive (c, outcome) from s
```

Algorithm 2: Deferred update replication, server s's code:

```
Initialization
  CT ← ∅
  lastCommitted ← 0
upon receiving (c, (r, x)) from c
  retrieve (x, v, i) from database
  send (c, (x, v, i)) to c
upon delivering (c, commit-req, t.rs, t.ws)
  if C_cs(t.rs, CT) then
    lastCommitted ← lastCommitted + 1
    CT ← CT ∪ {(lastCommitted, items(t.ws))}
    for each (x, v) ∈ t.ws do
      create database entry (x, v, lastCommitted)
    send (c, commit) to c
  else
    send (c, abort) to c
```

of t must first acquire a read lock on the data item. If a transaction u passes certification, then the server must acquire write locks on all data items in u's writeset and then commit u. Since read and write locks cannot be acquired simultaneously, this technique may block transactions. This mechanism has been used by other protocols based on the deferred update replication model (e.g., [15]).

D. Correctness

To reason about correctness, we must define when read, write and commit commands of a transaction t take place. For read and write commands this is simple: a read happens when the client receives the corresponding reply from the server that executes t; a write happens after the client updates t's writeset. It seems natural to assume that the commit of t also happens when the client receives the first commit reply from a server. In our model, however, clients are allowed to crash and so, the commit may never take place, despite the fact that some databases consider the transaction committed. To avoid such cases, without assuming correct clients, we define the commit event of t as taking place when the first server applies t's updates to its database.

1) Update transactions: We argue now that Algorithm 1 is correct for update transactions only. We extend our argument to include read-only transactions in the next section.

Let h_0 be a history created by Algorithm 1. We must show that we can permutate the statements of committed transactions in h_0 such that the resulting history, h_s, is serial and legal. Our strategy is to create a series of histories $h_0, ..., h_i, ..., h_s$ where h_i is created after swapping two statements in h_{i-1} .

Let t and u be two committing transactions in h_i such that t commits before u, which implies that t's commit request was delivered before u's. We show that all of t's statements that succeed u's statements in h_i can be placed before u's statements. There are two cases to consider.

1) Some u's statement σ_u precedes t's read statement $(c_t, (\mathsf{r}, x, v))$. If σ_u is a read on x, or a read or a write on an item different than x, then we can trivially swap the two. Assume now that σ_u is a write on x. Since t is delivered before u and only values of a delivered transaction can be read by other transactions, we know that t's read statement did not read the value written by σ_u, and thus, the two can be swapped.
2) Some u's statement σ_u precedes t's write statement $(c_t, (\mathsf{w}, x, v))$. If σ_u is a statement on an item different than x, then the two can be obviously swapped. Assume then that σ_u is a statement on x. If σ_u is a write, then we can swap the two as no read statement from other transactions have seen them; they only take effect after a transaction is delivered and t is delivered before u. Finally, let σ_u be a read. We show by way of contradiction that $(c_u, (\mathsf{r}, x, v_u))$ cannot precede $(c_t, (\mathsf{w}, x, v_t))$. Since u is certified after t and u passes certification, from the certification test, it must be that version read by u is still up-to-date. But since t modifies x, it updates x's version, a contradiction that concludes our argument.

2) Read-only transactions: We consider first read-only transactions in the presence of multi-version databases. Let t be a read-only transaction in some history h_i of the system. We extend the argument presented in the previous section to show that all of t's read commands can be placed in between two update transactions, namely, after the transaction u that created the first item read by t and before any other update transaction. From Section III-C, every future read of t will return a version that is equal to or precedes $t.view$. Therefore, every read command issued by t can be placed before any command that succeeds u's write commands.

We claim now that read-only transactions are also serializable with single-version databases. In this case, the local execution at a server follows the *two-phase locking* protocol, which is serializable [2]. Notice that although read and write commands are synchronized in a server, certification of update transactions is still needed since two update transactions executing on different servers may see inconsistent reads. For example, transaction t may read data item x and write y, while transaction u, executing at a different server, may read y and write x. In such a case, certification will abort one of the transactions.

IV. BFT Deferred Update Replication

To adapt our protocol to the byzantine failure model, we make a few extra assumptions. We first assume that the number of servers is at least $3f+1$. This is the minimum to solve atomic broadcast with malicious faults [12]. We make use of message digests produced by collision-resistant hash functions to ensure data integrity [19], and public-key signatures to ensure communication authenticity [20]. We follow the common practice of signing message digests instead of signing messages directly. We also assume that each client-server pair and each pair of servers communicate using private channels that can be implemented using symmetric keys [5]. Finally, clients are authenticated and servers enforce access control. This forbids unauthorized clients from accessing the database, and prevents potentially byzantine servers from issuing transactions that may compromise the integrity of the database. Obviously, a byzantine server can compromise its local copy of the database, but this behaviour is handled by our protocols.

A. Overview

A byzantine server could easily corrupt the execution of the algorithm presented in the previous section. For example, it could hamper the execution of the atomic broadcast protocol or answer client commands with incorrect data. While the first problem can be solved by replacing the atomic broadcast algorithm for crash-stop failures with one

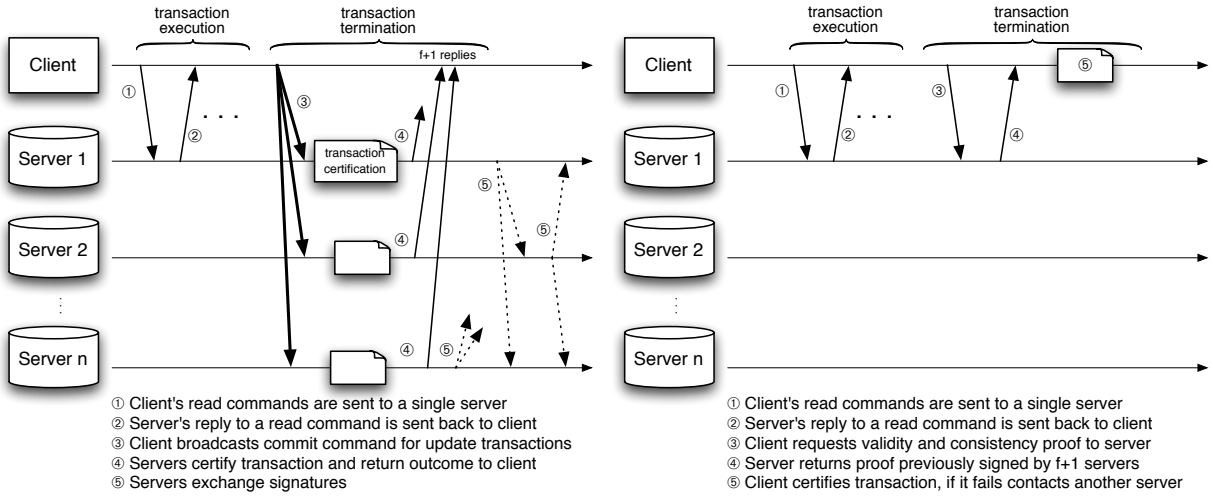

Figure 1. BFT deferred update replication: update (left) and read-only transactions (right)

that tolerates byzantine failures [14], the second problem is less obvious to address.

A byzantine server may return two types of "incorrect data": *invalid* or *stale*. Invalid data, as opposed to *valid* data, is fabricated by the server and does not correspond to any value created by a committed transaction. Stale data is too old, although it may be valid. We address the problem of stale data with the certification test by checking that the values read are still up-to-date. To guarantee that transactions read valid data, each database tuple is redefined as (x, v, i, d), where in addition to x's value v and version i, it contains a digest d of v. A read command returns the value read and its digest, and readsets include the digest of values.

We decompose the certification test into a component that checks the validity of data read, $C_b^v(t.rs, CT)$, and another component that checks whether the items read are up to date, $C_b^u(t.rs, CT)$.[1] Moreover, tuples (i, up) in CT contain in set up elements (x, d), that is, the items written and the digest of the values written.

$$C_b^u(t.rs, CT) \equiv \forall (x, i, d) \in t.rs : \quad (2)$$
$$\nexists (j, up) \in CT \text{ s.t. } ((x, *) \in up) \text{ and } (j > i)$$
$$C_b^v(t.rs, CT) \equiv \forall (x, i, d) \in t.rs : \quad (3)$$
$$\exists (j, up) \in CT \text{ s.t. } ((x, d') \in up) \text{ and } (d = d')$$

Additionally, to shield clients from byzantine servers that would commit a transaction t that violates serializability, clients wait for more than one server's reply before concluding the outcome of transactions. It turns out that at commit time, by waiting for a set of $f + 1$ identical replies, t's outcome can be safely determined by the clients since only up to f servers can be compromised.

The left side of Figure 1 illustrates the protocol for update transactions. Note that step 5 in the illustration will be explained as part of the protocol for read-only transactions (cf. Section IV-C).

B. Algorithm in detail

Algorithms 3 and 4 present the client's and server's protocol. To execute a read command, a client c contacts a server s and stores the version, the value, and the digest in $t.rs$. Write operations are buffered in $t.ws$ as before. When c wishes to commit t, c atomically broadcasts a signed message to all servers. This is denoted by $\langle m \rangle_{\sigma_c}$, where m is a message and σ_c is c's signature. Signing messages guarantees that only authenticated clients issue commit requests. After receiving $f+1$ identical replies from servers, c can determine t's outcome.

Besides a change in the certification test and in the data sent back to the client when answering read requests, the server's code is similar to the crash-stop case. In Algorithm 4, we must instantiate $items(t.ws)$ (see line 11) as $items(t.ws) = \{(x, v\text{'s digest}) \mid (x, v) \in t.ws\}$.

C. Read-Only Transactions

A simple way to handle read-only transactions is to execute and terminate them in the same way as update transactions. This leads to a simple solution but increases the latency of read-only transactions since they need to be atomically broadcast and certified by servers. In the following, we describe a mechanism that allows read-only transactions to be executed locally to a server only, just like deferred update replication in the crash-stop failure model.

[1] The decomposition of the certification test into two components is done for clarity purposes. An implementation would probably combine the two to speed up the execution.

Algorithm 3: Deferred update replication, client c's code:

Command $(c, (\mathsf{r}, x, v))$ occurs as follows:
  if first command for t **then**
    $t.rs \leftarrow \emptyset$
    $t.ws \leftarrow \emptyset$
    choose some server $s \in S$
  if $(x, v) \notin t.ws$ **then**
    send $(c, (\mathsf{r}, x))$ to s
    wait until receive $(c, (x, v, i, d))$ from s
    $t.rs \leftarrow t.rs \cup \{(x, i, d)\}$
    return v
  else
    return v such that $(x, v) \in t.ws$
Command $(c, (\mathsf{w}, x, v))$ occurs as follows:
  $t.ws \leftarrow t.ws \cup \{(x, v)\}$
A commit command is executed as follows:
  if $t.ws \neq \emptyset$ **then**
    abcast $\langle c, \mathsf{commit\text{-}req}, t.rs, t.ws\rangle_{\sigma_c}$ to all servers
    wait for identical $(c, outcome)$ from $f+1$ servers

x: the data item's key v: the data item's value
i: the data item's version d: the data item's digest

Algorithm 4: Deferred update replication, server s's code:

Initialization
  $CT \leftarrow \emptyset$
  $lastCommitted \leftarrow 0$
upon receiving $(c, (\mathsf{r}, x))$ from c
  retrieve (x, v, i, d) from database
  send $(c, (x, v, i, d))$ to c
upon delivering $\langle c, \mathsf{commit\text{-}req}, t.rs, t.ws\rangle_{\sigma_c}$
  if $C_b^u(t.rs, CT)$ **and** $C_b^v(t.rs, CT)$ **then**
    $lastCommitted \leftarrow lastCommitted + 1$
    $CT \leftarrow CT \cup \{(lastCommitted, items(t.ws))\}$
    for each $(x, v) \in t.ws$ **do**
      create database entry...
                ...$(x, v, lastCommitted, v$'s digest$)$
    send (c, commit) to c
  else
    send (c, abort) to c

x: the data item's key v: the data item's value
i: the data item's version d: the data item's digest

To allow read-only transactions to execute at a single server, without inter-server communication, clients must be able to tell unilaterally whether (i) a value returned by a server as a response to a read command is valid (cf. Section IV-A) and (ii) any set of valid values read by the client belongs to a consistent view of the database. If the client determines that a value returned by the server is invalid or inconsistent, it aborts the transaction and retries using another server.

A set of values read by a client is a *consistent view* of the database if the values could be the result of a serial execution of the committed transactions. For example, assume that transactions t and u modify the values of data items x and y. Any transaction that reads x and y must see either the values created by t or the ones created by u or none, but not a mix of the two (e.g., x from t and y from u).

We ensure proper transaction execution by letting the client ask the server, at the end of the transaction execution, for a proof that the values read are valid and consistent. A *validity and consistency proof* for a transaction t, denoted as $vcp(t)$, consists of all elements of CT whose version lies between the lowest and highest data item version read by t. Moreover, to ensure that byzantine servers do not fabricate data, every element (i, up) in $vcp(t)$ must be signed by $f + 1$ servers, denoted $\langle i, up\rangle_{\Sigma_{f+1}}$. Given a validity and consistency proof $vcp(t)$, the client decides to commit t if the following conditions hold:

1) The proof $vcp(t)$ is valid: If i_{min} and i_{max} are, respectively, the minimum and maximum data item versions read by t, then $vcp(t)$ contains all tuples (i, up) such that $i_{min} \leq i \leq i_{max}$. Moreover, each element of $vcp(t)$ is signed by $f + 1$ servers, which guarantees that at least one correct server abides by this element.
2) The values read by t are valid: Each data item with version i read by t matches its corresponding digest in $vcp(t)$ with version i.
3) The values read by t are consistent: For each item x read with version i, no newer version $i' > i$ of x exists in $vcp(t)$.

Conditions 2 and 3 can be stated more precisely with predicates (4) and (5), respectively. Note that these predicates are similar to those used to certify update transactions (cf. Section IV-A). The main difference is that read-only transactions do not need to be certified against elements in CT whose version is newer than the highest data item version t read.

$$C_r^c(t.rs, vcp(t)) \equiv \forall (x, i, d) \in t.rs : \nexists \langle j, up\rangle_{\Sigma_{f+1}} \in vcp(t) \text{ s.t. } ((x, *) \in up) \text{ and } (j > i) \quad (4)$$

$$C_r^v(t.rs, vcp(t)) \equiv \forall (x, i, d) \in t.rs : \exists \langle i, up\rangle_{\Sigma_{f+1}} \in vcp(t) \text{ s.t. } ((x, d') \in up) \text{ and } (d = d') \quad (5)$$

To build a validity and consistency proof from CT, we add to each CT entry a certificate of $f+1$ server signatures. Database servers build certificates asynchronously. When the i-th update transaction t commits on s, the value, version, and digest of each data item x written by t are updated. Periodically, server s signs new tuples (i, up) and sends this information to all servers. When s gathers $f+1$ signatures of the tuple (i, up), s inserts this new element in CT. This asynchronous scheme does not add communication overhead to update transactions. However, a read-only transaction t may stall until the server answering t's requests gathers enough signatures to provide a validity and consistency proof for t. The protocol for read-only transactions is illustrated on the right side of Figure 1.

Algorithms 5 and 6 present the client and server protocols to execute read-only transactions. In the algorithms, given a transaction t and a validity and consistency proof $vcp(t)$ for t, read validity and consistency are expressed by predicates $C_r^v(t.rs, vcp(t))$ and $C_r^c(t.rs, vcp(t))$ respectively.

Algorithm 5: Read-only transactions, client c's code:

```
Command (c, (r, x, v)) occurs as follows:
    if first command for t then
        t.rs ← ∅
        choose some server s ∈ S
    send (c, (r, x)) to s
    wait until receive (c, (x, v, i, d)) from s
    t.rs ← t.rs ∪ {(x, i, d)}
    return v
Command (c, commit)/(c, abort) occurs as follows:
    i_max ← max({i | (x, i, *) ∈ t.rs})
    i_min ← min({i | (x, i, *) ∈ t.rs})
    send(c, commit-req, i_min, i_max) to s
    wait until receive(c, vcp(t))
    if vcp(t) is valid and
       C_r^c(t.rs, vcp(t)) and C_r^v(t.rs, vcp(t)) then
        outcome ← commit
    else
        outcome ← abort
```

x: the data item's key v: the data item's value
i: the data item's version d: the data item's digest

Algorithm 6: Read-only transactions, server s's code:

```
Initialization
    forwarded ← ∅
upon receiving (c, (r, x)) from c
    retrieve (x, v, i, d) from database
    send (c, (x, v, i, d)) to c
upon receiving (c, commit-req, i_min, i_max)
    vcp ← ∅
    for i in i_min to i_max do
        wait until element e = ⟨i, up⟩_{Σ_{f+1}} is in CT
        vcp ← vcp ∪ {e}
    send (c, vcp) to c
periodically do
    for each (i, up) ∈ CT \ forwarded do
        send ⟨i, up⟩_{σ_s} to all servers
        forwarded ← forwarded ∪ {(i, up)}
upon receiving ⟨i, up⟩_* from f + 1 servers
    CT ← CT ∪ {⟨i, up⟩_{Σ_{f+1}}}
```

x: the data item's key v: the data item's value
i: the data item's version d: the data item's digest

Table I summarizes the costs of the proposed protocols for the crash-stop and byzantine failure models. To compute these costs, we consider a transaction t that performs r reads and present the latency and number of messages sent for the execution and termination phases, when t is an update and a read-only transaction.

D. Liveness issues

Byzantine servers may compromise the progress of the above protocol by being non-responsive or slow. Besides attacks that would slow down the delivery of atomic broadcast messages [1], byzantine servers may also not answer client read requests or slow down their execution. The first case can be treated as in the crash-stop case, that is, the client may simply consider that the server has crashed and restart executing the transaction on another server. The second case is more problematic since it may not be possible to distinguish between a slow *honest* server and a *malicious* one. To avoid such an attack, the client can execute the transaction on two (or more) servers and abort the transaction on the slower server as soon as the faster server is ready to commit.

A more subtle attack is for a byzantine server to provide read-only transactions with old, but valid and consistent, database views. Although serializability allows old database views to be seen by transactions (i.e., strictly speaking it is not an attack), useful implementations try to reduce the staleness of the views provided to transactions. There are (at least) two ways to confront such server misbehaviour. First, clients can broadcast read-only transactions and ask servers to certify them, just like update transactions. If the transaction fails certification, the client can retry using a different server. Second, clients may submit a read command to more than one server and compare their versions. Submitting read commands to $f+1$ servers ensures the "freshness" of reads, but may be an overkill. More appropriate policies would be to send multiple read commands when suspecting a server misbehavior, and possibly try first with a small subset of servers.

E. Optimizations

Our protocols can be optimized in many ways. In the following, we briefly present three optimizations.

Client caches: To increase scalability, clients can cache data item values, versions, digests, and elements of CT. In doing so, clients can execute queries without contacting any server, provided that the necessary items are in the cache. Before inserting a tuple (x, v, i) in the cache, where v and i are x's value and version respectively, we verify the validity of v and make sure that version i of x has value v by using the appropriate element of CT. We proceed similarly with elements of CT by verifying their signatures before inserting them in the cache. At the end of the execution of a read-only transaction t, the consistency of the values read can be checked using cached elements of CT. If some elements of CT are missing, they are retrieved from a server. If t is an update transaction, the consistency of the values read by t are performed by the certification test. To avoid reading arbitrary old values, cache entries are evicted after some time (e.g., a few hundreds of milliseconds).

Failure model	Number of servers	Transaction type	Execution		Termination	
			latency	messages	latency	messages
crash-stop	$2f+1$	update	$r \times 2$	$O(r)$	$\delta(abcast_{cs})+1$	msgs($abcast_{cs}$) + $O(n)$
		read-only	$r \times 2$	$O(r)$	-	-
byzantine	$3f+1$	update	$r \times 2$	$O(r)$	$\delta(abcast_{byz})+1$	msgs($abcast_{byz}$) + $O(n^2)$
		read-only	$(r+1) \times 2$	$O(r)$	-	-

Table I
THE COST OF THE PROPOSED PROTOCOLS (n IS THE NUMBER OF SERVERS, f IS THE MAXIMUM NUMBER OF FAULTY SERVERS, r DENOTES THE NUMBER OF ITEMS THE TRANSACTION READS, $abcast_{cs}$ DENOTES AN ATOMIC BROADCAST ALGORITHM FOR CRASH-STOP FAILURES, AND $abcast_{byz}$ IS AN ATOMIC BROADCAST ALGORITHM FOR BYZANTINE FAILURES).

Limiting the size of CT: We limit the number of element in CT by some value K to reduce the space overhead of the set of committed transactions on the servers. After the k-th transaction commits and server s inserts tuple $\langle k, up \rangle_{\Sigma_{f+1}}$ into CT, s checks whether CT contains more than K elements. If so, the element of CT with the lowest timestamp is removed. This scheme may force servers to unnecessarily abort transactions due to missing versions in CT. Choosing K must thus be done carefully.

Efficient message authentication: In the above protocol, public-key signatures are used by servers to sign elements of the set of committed transactions and by clients to authenticate their commit messages. Public-key signatures are expensive however. In particular, it is orders of magnitude slower than message authentication codes (MACs). In contrast to public-key signatures, a MAC cannot prove the authenticity of a message to a third party. We thus replace signatures by vectors of MACs [3]. This vector contains one entry per machine of the system, that is, clients and servers. In doing so, any vector of MACs can be verified by any client or server.

F. Correctness

Correctness of the protocol for update transactions relies on the fact that at certification, we use value digests to check for data integrity and versions to check for data staleness. The rest of the correctness argument is based on the same principles as the crash-stop case and we thus omit it.

Proving the correctness of read-only transactions is more subtle. To be serializable, read-only transactions must read values from a consistent view of the database that is the result of a serial execution of some finite sequence of transactions. Let i_{max} and i_{min} respectively be the highest and lowest data item versions read by a transaction t. We claim that if t commits, then the view of the database t observes is the result of the execution, in version order, of all transactions whose corresponding tuple in CT has a version that is smaller than, or equal to i_{max}. Let h_t be the sequence of such transactions sorted in ascending version order.

We first note that since t commits, the server s on which t executed behaved as prescribed by the protocol. This is because each element of the validity and consistency proof $vcp(t)$ of t is signed by $f+1$ servers, and thus ensures that the values and versions read by t are those that would be returned by a correct server.

Since the client checks that (i) $vcp(t)$ contains all versions between i_{min} and i_{max} and (ii) for any data item x read by t, no newer version of x exists in $vcp(t)$, t reads a consistent view of the database that is the result of the serial execution of transactions in h_t.

V. RELATED WORK

Database replication and deferred update replication have been largely studied under benign faults (e.g., non-byzantine servers subject to crash-stop failures). Few works have considered the effects of byzantine servers on database replication. The first paper to consider the problem is [9], written more than two decades ago. This paper investigates the use of byzantine agreement and state machine replication in the context of databases. It proposes to execute transactions atop serializable databases at the expense of limiting transaction concurrency. The deferred update replication protocol we propose allows concurrency among transactions in that multiple transactions can be simultaneously executed by different servers. Only transaction termination needs to be serialized.

More recently, Vandiver et al. [21] proposed a system that allows more concurrency between transactions. Clients communicate through a central coordinator that chooses a replica as primary and the rest as secondaries. Transactions are first executed on the primary to determine which transactions can be executed in parallel. When the result of a query is returned to the coordinator, the latter ships this query to the secondaries. A commit barrier, maintained by the coordinator and incremented whenever a transaction commits, is used to determine which transactions can be executed in parallel at the secondaries. That is, two transactions that executed in parallel on the primary will be executed in parallel on the secondaries, provided that no transaction commits in the mean time. This is, roughly speaking, the basics of the Commit Barrier Scheduling protocol proposed in [21]. This approach is similar to ours in the sense that it allows the concurrent execution of transactions, although our protocol allows the execution of transactions at *any replica*.

Moreover, in contrast to [21], our protocol does not require a trusted coordinator, a strong assumption.

Byzantium [18] considers byzantine failures of servers that guarantee snapshot isolation, as opposed to serializability. Under *snapshot isolation*, transactions observe a committed instant of the database, which may correspond to the database state when the transaction started or, more likely in a distributed environment, to some earlier state of the database [7]. Transactions that execute concurrently can only commit if they do not modify the same data items, a policy sometimes referred to as *first-commiter-wins rule*. In Byzantium, a client first selects a replica that will act as the coordinator for its transaction and then atomically broadcast the *begin* operation to all replicas so that they all use the same database snapshot for the transaction. The transaction is entirely executed by the coordinator. At commit time, the operations along with their results are atomically broadcast to all replicas. If the transaction was executed in a correct coordinator, then a quorum of servers will obtain the same results and the transaction can commit; otherwise, the transaction is aborted and the client is notified about the byzantine coordinator. Both update and read-only transactions are atomically broadcast.

VI. Final remarks

This paper has considered the deferred update replication technique under the byzantine failure model. Deferred update replication has been largely used to implement database replication in the crash-stop failure model. It is more scalable than other replication techniques such as state machine replication and primary-backup since transactions may be executed at any server. Moreover, it allows read-only transactions to be executed at a single replica. The paper makes two contributions: First, it shows that it is surprisingly simple to use deferred update replication under byzantine failures—in fact, our protocol only requires a small modification of the certification procedure and an additional check, performed by clients, to filter out transaction outcomes sent by byzantine servers. Second, the paper shows that even though some servers may behave maliciously, read-only transactions can be executed at a single server—the execution must be certified by the client at the end of the transaction however.

Our protocols ensure serializable execution. Some database replication protocols are based on snapshot isolation (e.g., [6], [13]). It turns out that it would not be difficult to change our algorithms to ensure snapshot isolation instead. As we have described, read-only transactions see a view of the database that corresponds to its committed state when the first read command is issued. This mechanism can be used to provide transactions, both read-only and update, with a consistent and valid database snapshot. The first-commiter-wins rule can be easily implemented by having servers check against write-write conflicts, as opposed to write-read conflicts, as currently done for serializability.

Acknowledgements

The authors wish to thank Antonio Carzaniga, Rui Oliveira, Ricardo Padilha, José Orlando Pereira and the anonymous reviewers for the insightful discussions and comments about this work.

This work was partially funded by the Hasler Foundation, project number 2316.

References

[1] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *DSN*, pages 197–206, 2008.

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

[3] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[4] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, 2004.

[5] W. Diffie and M. E. Hellman. Multiuser cryptographic techniques. In *AFIPS '76: Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 109–112, New York, NY, USA, 1976. ACM.

[6] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *Proceedings of EuroSys*, 2006.

[7] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Symposium on Reliable Distributed Systems (SRDS'2005)*, Orlando, USA, 2005.

[8] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008.

[9] Hector Garcia-Molina, Frank M. Pittelli, and Susan B. Davidson. Applications of byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, 1986.

[10] B. Kemme and G. Alonso. A new approach ro developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3), September 2000.

[11] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[12] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[13] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *International Conference on Management of Data (SIGMOD)*, Baltimore, Maryland, USA, 2005.

[14] J.-P. Martin and L. Alvisi. Fast byzantine consensus. In *DSN'05*, pages 402–411, 2005.

[15] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1999. Number 2090.

[16] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, Durham (USA), 1997.

[17] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, 2004.

[18] Nuno M. Preguiça, Rodrigo Rodrigues, Cristóvão Honorato, and João Lourenço. Byzantium: Byzantine-fault-tolerant database replication providing snapshot isolation. In *HotDep*, 2008.

[19] R. Rivest. The md5 message-digest algorithm. internet rfc-1321. 1992.

[20] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and Public-Key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[21] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Samuel Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, pages 59–72, 2007.