Proactive Byzantine Quorum Systems

Eduardo A. P. Alchieri[1], Alysson Neves Bessani[2], Fernando Carlos Pereira[1], and Joni da Silva Fraga[1]

[1] DAS, Federal University of Santa Catarina - Florianópolis - Brasil
[2] University of Lisbon, Faculty of Sciences, LaSIGE - Lisbon - Portugal

Abstract. Byzantine Quorum Systems is a replication technique used to ensure availability and consistency of replicates data even in presence of arbitrary faults. This paper presents a Byzantine Quorum Systems protocol that provides atomic semantics despite the existence of Byzantine clients and servers. Moreover, this protocol is integrated with a protocol for proactive recovery of servers. In that way, the system tolerates any number of failures during its lifetime, since no more than f out of n servers fail during a small interval of time between recoveries. All solutions proposed in this paper can be used on asynchronous systems, which requires no time assumptions. The proposed quorum system read and write protocols have been implemented and their efficiency is demonstrated through some experiments carried out in the Emulab platform.

1 Introduction

Quorum systems [7] are fundamental tools used to ensure consistency and availability of data stored in replicated servers. Appart from its use in the construction of synchronization protocols (e.g., consensus), quorum-based protocols for register implementation are appealing due to their scalability and possibility of load balancing, since most operations does not need to be executed in all servers, but only in a subset of them (a quorum). The consistency of the stored data is ensured by the intersection between every quorum of the system. Quorum systems can be used to implement registers that provide read and write operations with several possible semantics (safe, regular or atomic) [8].

The concept of quorum systems was initially proposed for environments in which servers could be subject to crash faults [7]. Later, the model was extended to tolerate Byzantine faults [11]. However, the biggest challenge in quorum systems is how to design efficient protocols that tolerate malicious clients. The problem is that clients can execute some malicious actions to hurt system properties, e.g., sending an update only to some servers and not to a complete quorum [10]. This possibility of misbehaviour should not be discarded since quorum systems were developed to be used mainly in open systems such as the Internet, where there is a high probability of at least some clients being malicious.

The first protocols that tolerate Byzantine clients required at least $4f + 1$ servers to tolerate f faults (on servers) [11, 12]. However, these protocols does not completely constraint faulty actions of malicious clients. There are some attacks that still can be executed, e.g., a malicious client prepare several writes to be executed by another malicious client (colluder) after its exclusion of the system. More recently, these weakness were mitigated by the BFT-BC protocol [10], which requires only $3f+1$ servers and allow clients to execute a write only after its previous write completes. The BFT-BC protocol relies on digital signatures (one of the biggest latency sources on Byzantine fault-tolerant protocols [4, 6]) to constraint the actions of malicious clients.

In this paper we extend BFT-BC and propose a new Byzantine quorum system protocol in which the actions of malicious clients are constrained through threshold cryptography. The main benefit of our approach is **(1)** the use of a single public key for the whole storage service (instead of one public key per server), which simplifies the management of the system and make it more affordable in dynamic environments in which clients come and go, and **(2)** a considerable reduction on the size of message certificates, which makes our protocol much more efficient than BFT-BC when the number of faults tolerated f increases.

One important property of fault tolerant replicated systems is the bound f on the number of faulty servers that can be tolerated. This property can be a problem for long lived systems, since given a sufficient amount of time, an adversary can manage to compromise more than f servers and then impair the correctness of the system. To overcome this limitation, proactive recovery schemes [4, 18] should be used. In a system in which this kind of technique is employed, each server is recovered periodically, in such a way that all servers are recovered in a bounded time interval. This mechanism allows an unlimited number of faults to be tolerated on the system lifetime, provided that no more than f faults occur during a recovery period. Another novel feature of our protocols is that the read/write algorithms were developed together with a proactive recovery scheme in order to make the register abstraction usefull in long lived systems. As far as we known, this is the first generic solution for register implementation that includes a proactive recovery scheme.

The contributions of this paper can be summarized as follows:

1. new quorum system read and write protocols that uses threshold cryptography to constraint the actions of malicious clients and ensure the consistency of the stored data;
2. a proactive recovery scheme for quorum systems that does not suffer from common weakness of previous efforts [18];
3. an experimental evaluation in which we compare our protocol with BFT-BC and shows the benefits (in terms of performance) of using threshold signatures instead of "common" public key criptography (e.g., RSA).

The paper is organized as follows. Section 2 presents our system model and the concept of Byzantine Quorum Systems, among other preliminary definitions used in this paper. Section 3 describes our proposal for Proactive Byzantine Quorum Systems. Section 4 presents an analytical analysis and some experiments realized with our protocols. Some related work are discussed in Section 5. Finally, Section 6 presents our final remarks and the next steps of this work.

2 Background

2.1 System Model

The system model consists of a set $C = \{c_1, c_2, ...\}$ of clients interacting with a set of n servers $S = \{s_1, ..., s_n\}$, which implement a quorum system with atomic semantic operations. Clients and servers have unique identifiers.

All the processes in the system (clients and servers) are prone to *Byzantine failures* [9]. A process that shows this type of failure may present any behavior: it may stop, omit transmission and delivery of messages, or arbitrarily deviate from its specification. A process that present this type of failure behavior is called faulty, otherwise it is called correct. However, faulty processes may be recovered, resuming a correct behavior again (proactive recovery). Furthermore, in this work we assume *fault independence* for processes, i.e., processes failures are uncorrelated. This assumption can be substantiated in practice using several types of diversity [14].

In terms of guarantees, the system remains correct while it presents a maximum of f faulty servers in a given time, being needed $n = 3f + 1$ servers in the system. Furthermore, an unlimited number of clients may be faulty.

We assume the asynchronous system model[3], where processes are connected by a network that may not send, delay, duplicate or corrupt messages. Moreover, time bounds for message transfers and local computations are unknown in the system. The only requirement for our protocols to terminate is that, if a process sends infinite times a message to another correct process, then this message will eventually be delivered in the receiver. To fulfill this requirement and to simplify the presentation of protocols, we consider that communications between processes are made through reliable and authenticated point-to-point channels.

Also, our protocols use threshold cryptography to constraint the actions of Byzantine clients and to ensure the integrity of stored data. Thus, we assume that in the startup of the system each server receives its partial key (a share of a service secret key [16]) that will be used in the preparation of partial signatures. A correct server never reveals its partial key. These keys are generated and distributed by a correct administrator that is only needed in the initialization of the system. The public key of the service, used to verify the signatures generated by the combination of signature shares (partial signatures) in this mechanism, is stored by the servers and is available to any process of the system.

We also assume the existence of a cryptographic hash function h resistant to collisions, so that any process is able to calculate the hash $h(v)$ from the value v. It is computationally infeasible to obtain two distinct values v e v' such that $h(v) = h(v')$.

Finally, to avoid replay attacks, some messages are tagged with *nonces*. We assume that clients do not choose repeated *nonces*, i.e., *nonces* already used.

[3] However, the proactive recovery procedure uses some mechanisms to guarantee that it starts and ends (Section 3.3).

2.2 Byzantine Quorum Systems

Byzantine Quorum Sytems [11], hereafter called simply as quorum systems, can be used to implement replicated data storage systems, ensuring consistency and availability of the stored data even in the presence of Byzantine faults in some replicas. Quorum systems algorithms are recognized for their good performance and scalability, once clients of these systems in fact access only a particular subset instead of all servers.

Servers in a quorum system are organized into subsets called quorums. Any two quorums have a nonempty intersection that contains a sufficient number of correct servers (ensuring consistency). Also, there is at least one quorum in the system that is formed only by correct servers (ensuring availability) [11]. Quorum systems are used to build register abstractions that provide read and write operations. The data stored on the register is replicated on the servers of the system. Each register stores a pair $\langle v, t \rangle$ with a value v of the stored data and an associated timestamp t.

The protocol presented in this paper is an extension of the BFT-BC [10], allowing proactive recovery of servers of the system. We choose this protocol due to its optimal resilience (it requires $n = 3f + 1$ to tolerate up to f faulty servers), strong semantics (implements an atomic register [8]) and tolerance to malicious clients.

In order to use a Byzantine quorum system with only $3f + 1$ servers it is required the stored data to be self-verifiable [11]. This is a direct consequence of the fact that the intersection between any two quorums on this system would have at least $f+1$ servers, and thus can contain only a single correct and updated server. Thus, clients correctly obtain the stored data, from this correct server, only if the register data is self-verifiable, i.e., it is possible to verify the integrity of the data stored on a single server without consulting other servers. In this sense, BFT-BC introduces a new way to make data self-verifiable: verification of a set of servers signatures. Other protocols, such as *f-dissemination quorum* system [11, 12], are based on client signatures and therefore, they do not tolerate malicious clients[4].

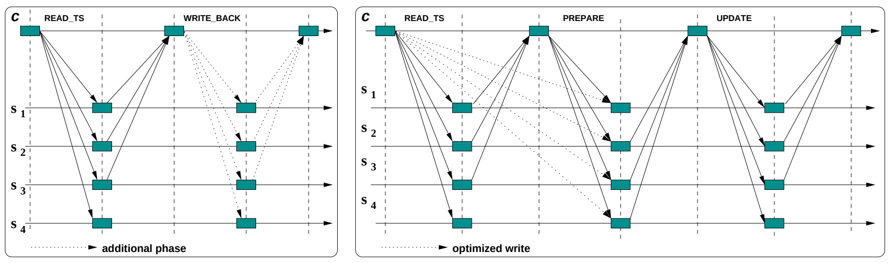
Thus, to maintain its consistency semantics, BFT-BC uses a mechanism of completion proofs signed by servers at all phases of the protocol. To a client be able to enter in a new phase of the protocol, it is necessary that it presents a proof that it completed the previous phase. This proof, called certificate, comprises a set of signed replies collected from a quorum of $n - f = 2f + 1$ servers in the previous phase. For example, to a client write some value in the system it needs to have completed its previous write. By using this technique, BFT-BC employs 2/4 communication steps to execute read operations and 4/6 steps to execute write operations (Figure 1). The BFT-BC algorithm works as follows:

– **Read operations** (Figure 1(a)) – the client requests a set of valid pairs $\langle v, t \rangle$ from a quorum of servers and selects that one with the highest timestamp

[4] The *f-masking quorum system* protocol [11, 12] also tolerates malicious clients, but requires replication in $4f + 1$ servers, because it does not store self-verifying data.

$\langle v_h, t_h \rangle$. The operation ends if all returned pairs are equals, i.e., they have the same timestamp t_h and the same value v_h (this happens in executions without concurrency and failures). Otherwise, the client performs an additional phase of *write back* in the system and waits for confirmations until it can be assured that a quorum of servers has the most recent value v_h;

- **Write operations** (Figure 1(b)) – the client obtains a set of timestamps from a quorum of servers (as in read operations) and then performs the preparation for its writing. In this phase it tries to obtain from a quorum of servers a set of proofs necessary to complete its current write operation. In case of success in the preparation, the client writes on a quorum of servers, waiting for confirmations. In an alternative scenario, the client may run an optimized protocol, performing in a single phase the both timestamp definition and write preparation (dotted lines in Figure 1(b)).



(a) Read (b) Write

Fig. 1. BFT-BC Read/Write Operations.

2.3 Threshold Cryptography

The main mechanism used in this work is a threshold signature scheme (TSS) [16] by which it is possible to control actions of clients and to ensure the integrity of the data stored by servers. Its flexibility also facilitates the servers recovery procedure. In a scheme (n, k)-TSS, a trusted **dealer** initially generates n secret key shares (partial keys) $(SK_1, ..., SK_n)$, n verification keys $(VK_1, ..., VK_n)$, the group verification key VK and the public key PK used to validate signatures. Moreover, the dealer sends these keys to n different players, called **share holders**. Thus, each share holder i receives its partial key SK_i and its verification key VK_i. The public key and all verification keys are available for any part that composes the system.

After this initial setup, the system is able to generate signatures. To obtain a signature A to some data d, each share holder i generates its partial signature a_i (also called signature share) of d. Later, a **combiner** obtains at least k valid partial signatures $(a_1, ..., a_k)$ and builds the signature A through the combination of these k valid partial signatures. An important feature of this scheme is the impossibility of generating valid signatures with less than k valid partial signatures. This scheme is based on the following primitives:

– *Thresh_Sign*$(SK_i, VK_i, VK, data)$: function used by the share holder i to generate its partial signature a_i to $data$ and the proofs v_i of validity of a_i, i.e.,$\langle a_i, v_i \rangle$;
– *Thresh_VerifyS*$(data, \langle a_i, v_i \rangle, PK, VK, VK_i)$: function used to verify if the partial signature a_i, obtained from the share holder i, is valid for $data$;
– *Thresh_CombineS*$(a_1, ..., a_k, PK)$: function used by the combiner to compose signature A from k valid partial signatures;
– *verify*$(data, A, PK)$: function used to verify if A is a valid signature of $data$.

In this work, we use the protocol proposed in [16], where it is proved that such a scheme is secure in the random oracles model [2], not being possible to forge signatures. This protocol represents a RSA [15] threshold signature scheme, i.e., the combination of partial signatures generates a RSA signature. In this model, the generation and verification of partial signatures is fully non-interactive, not being necessary to exchange messages to perform these operations.

3 Proactive Byzantine Quorum Systems

This section presents our approach to build a register that offers read and write operations, implemented through Byzantine quorum replication. Our protocol is based on BFT-BC [10] and therefore is able to tolerate malicious clients. The main feature introduced in our protocol is the possibility of proactive recovery of servers (Section 3.3), and thus it is called PBFT-BC (*proactive Byzantine fault-tolerance for Byzantine clients*). Moreover, PBFT-BC outperforms BFT-BC (Section 4) and also presents optimal resilience ($n = 3f + 1$).

A Byzantine quorum system should tolerate the presence of malicious clients, since they can perform the following actions to corrupt the system [10]: *(i)* write different values associated with the same timestamp; *(ii)* execute the protocol partially, updating the value stored by only some (few) servers; *(iii)* choose a very large timestamp and exhaust the timestamp space; and *(iv)* issue a large number of write requests and hand them off to a colluder who will run them after the bad client has been removed from the system.

Thus, protocols developed for this purpose should make impossible for malicious clients to perform these actions (or mask them). Our protocol uses a threshold signature scheme to control the actions performed by clients and to ensure the integrity of stored data. We consider the existence of a correct administrator who will perform the distribution function (dealer) of the scheme, where servers act as share holders and clients as combiners. The administrator generates the partial keys and distributes them to the servers. Also, all public information (public and verification keys) are sent to all servers. Thereafter, each client gets these public information from the servers by sending a request and waiting for $f+1$ identical replies. Notice that the administrator is only necessary in the initialization of the system, during the setup phase.

Our protocol uses quorums of $2f + 1$ replicas and a $(n, 2f + 1)$-TSS scheme, i.e., it is required a quorum of servers in order to generate a valid signature. Thus, all actions performed by clients must be authorized by a quorum of servers.

Clients need to prove that they are acting properly to move from one phase to another phase of the protocol. They do it by using certificates, which contain data indicating the validity of the actions that they are trying to execute and a signature that ensures the integrity of these data. This signature is generated by the $(n, 2f+1)$-TSS scheme ensuring that a quorum of servers approved its actions. Our protocols use two kinds of certificates:

Prepare Certificate: A client uses this certificate to prove that its writing was approved by at least a quorum of servers. On the other hand, servers use it to prove the integrity of the stored values. A prepare certificate pc has three fields: $pc.ts$ – timestamp of the proposed write; $pc.hash$ – hash of the value v proposed in the write; $pc.A$ – service signature, proving that at least a quorum of servers approve the write of v with timestamp $pc.ts$. A prepare certificate pc is valid only if its signature $pc.A$, for the tuple $\langle pc.ts, pc.hash\rangle$, is valid. This is determined by the operation $verify(\langle pc.ts, pc.hash\rangle, pc.A, PK)$.

Write Certificate: A client uses this certificate to prove that its last write has been completed. A write certificate wc has two fields: $wc.ts$ – timestamp of the completed write; $wc.A$ – service signature, proving that the client has done the write with timestamp $wc.ts$ in at least a quorum of servers. A write certificate wc is valid only if its signature $wc.A$, for $wc.ts$, is valid. This is determined by the operation $verify(wc.ts, wc.A, PK)$.

PBFT-BC can deal with multiple objects, since each of them has a distinct identifier. However, to simplify the presentation, we consider that servers store only a single register. Thus, each server i stores the following variables: **(1)** $data$ – value of the object; **(2)** P_{cert} – a prepare certificate valid for $data$; **(3)** P_{list} – a set of tuples $\langle c, ts, hash\rangle$ containing the client identifier c, the timestamp ts and the hash $hash$ of the value of a proposed write; **(4)** max_{ts} – timestamp of the latest write that i knows that was completed in at least a quorum of servers; **(5)** SK_i – partial key of i, that is used by the threshold signature scheme; **(6)** VK_i e VK – verification keys, that are used to generate proofs of validity of partial signatures; and **(7)** PK – service public key, that is used to validate certificates. Moreover, each client c uses the following variables: **(1)** W_{cert} – write certificate of the last write of c; **(2)** PK – service public key, that is used to validate certificates; and **(3)** VK e $VK_1, ..., VK_n$ – verification keys, that are used to validate partial signatures.

3.1 Read and Write Protocols

This section presents PBFT-BC's read and write protocols. We present the pseudocodes for each one of the operations and discuss their main features.

Clients should choose timestamps from different subsets in order to these protocols work properly. Thus, each client concatenates its unique identifier with a sequence number, i.e., $ts = \langle seq, id\rangle$. Timestamps are compared through their sequence number. If two timestamps have the same sequence number, then they are compared through their client identifier. Timestamps are incremented through the function $succ(ts, c) = \langle ts.seq + 1, c\rangle$.

Pseudocodes 1 and 2 present the write protocol executed by clients and servers, respectively. These pseudocodes represent the write version without optimizations, that demands three phases to complete a write operation.

Pseudocode 1 Protocol used by client c to write *value*.

```
w1.1 Client $c$ sends a message $\langle \text{READ\_TS}, nonce \rangle$ to all servers.
w1.2 $c$ waits for a quorum $(2f+1)$ of valid replies from different servers. A reply $m_i$ from
     server $i$ is valid if it is well-formed, i.e., $m_i = \langle \text{READ\_TS\_REPLY}, p, nonce \rangle$ where $p$ is
     a valid prepare certificate (well-formed and the service signature is valid). Moreover, $m_i$
     should be correctly authenticated, i.e., its $nonce$ matches the $nonce$ used in step $w1.1$.
w1.3 Among the prepare certificates received in step $w1.2$, $c$ selects the certificate containing
     the highest timestamp, called $P_{max}$.
w2.1 $c$ sends a message $\langle \text{PREPARE}, P_{max}, ts, h(value), W_{cert} \rangle$ to all servers. Here $ts \leftarrow$
     $succ(P_{max}.ts, c)$, $h$ is a hash function and $W_{cert}$ is a write certificate of $c$'s last write
     or $null$ if this is $c$'s first write.
w2.2 $c$ waits for a quorum $(2f+1)$ of valid replies from different servers. A reply $m_i$ from
     server $i$ is valid if it is well-formed, i.e., $m_i = \langle \text{PREPARE\_REPLY}, \langle ts_i, hash_i \rangle, \langle a_i, v_i \rangle \rangle$
     where $ts_i$ and $hash_i$ match the values sent in step $w2.1$ ($ts$ and $h(value)$, respectively).
     Moreover, $m_i$ is valid if $Thresh\_verifyS(\langle ts_i, hash_i \rangle, \langle a_i, v_i \rangle, PK, VK, VK_i)$ is $true$.
w2.3 $c$ combines the $2f+1$ correct signature shares received in step $w2.2$, invoking
     $Thresh\_combineS(a_1, ..., a_{2f+1}, PK)$, and obtains the service signature $A$ for the pair
     $\langle ts, h(value) \rangle$. Then, $c$ forms a prepare certificate $P_{new}$ for $ts$ and $h(value)$ by using $A$.
w3.1 $c$ sends a message $\langle \text{WRITE}, value, P_{new} \rangle$ to all replicas.
w3.2 $c$ waits for a quorum $(2f+1)$ of valid replies from different servers. A reply $m_i$ from server
     $i$ is valid if it is well-formed, i.e., $m_i = \langle \text{WRITE\_REPLY}, ts_i, \langle a_i, v_i \rangle \rangle$ where $ts_i$ matches
     the value $ts$ defined in step $w2.1$ and $Thresh\_verifyS(ts_i, \langle a_i, v_i \rangle, PK, VK, VK_i)$ is $true$.
w3.3 $c$ combines the $2f+1$ correct signature shares received in step $w3.2$, invoking
     $Thresh\_combineS(a_1, ..., a_{2f+1}, PK)$, and obtains the service signature $A$ for the times-
     tamp $ts$. Then, $c$ forms a write certificate $W_{cert}$ for $ts$ by using $A$. This certificate is
     used in $c$'s next write.
```

In the first phase of the protocol the client defines the write timestamp and in the second phase it obtains a prepare certificate for this write operation. Afterwards, the write operation is definitely executed in the third phase. The progress of the protocol (i.e., for a client moves from one phase to another) is based on the use of certificates. The processing related with these certificates is one of the differences between PBFT-BC and BFT-BC (also, PBFT-BC provides protocols to recover servers periodically – Section 3.3). On PBFT-BC, a client obtains a certificate by waiting for a quorum of valid partial signatures (steps $w2.2$ e $w3.2$) and, then, combining them (steps $w2.3$ e $w3.3$), what results in a signature that is used to prove the validity of this certificate. Notice that to validate a certificate it is necessary to verify only one signature (service signature), differing from BFT-BC where a full quorum of signatures must be verified.

The most important phase of the write protocol is the second one. In this phase each server (Pseudocode 2) checks if: **(1)** the timestamp being proposed is correct; **(2)** the client is preparing just one write; **(3)** the value being prepared does not differ from a (possible) previous prepared value with the same timestamp; and **(4)** the client has completed its previous write. The item (1) is checked in the step $w2.1$. The items (2), (3) and (4) are checked in the steps $w2.2$ and $w2.3$, where each server uses its list of prepared writes (P_{list}). An important feature related with the use of this list is that a client is not able to prepare many write requests. Thus, a malicious client m is not able to prepare multiple write requests and hand them off to a colluder, that could execute them after m is removed from the system, what limits the damage caused by malicious clients.

Pseudocode 2 Write protocol executed at server i.

```
Upon receipt of $\langle \text{READ\_TS}, nonce \rangle$ from client $c$

  w1.1  $i$ sends a reply $\langle \text{READ\_TS\_REPLY}, P_{cert}, nonce \rangle$ to $c$.

Upon receipt of $\langle \text{PREPARE}, P_c, ts, hash, W_c \rangle$ from client $c$

  w2.1  if the request is invalid or $ts \neq succ(P_c.ts, c)$, discard request without replying to $c$. A
        PREPARE request is invalid if either certificate $P_c$ or $W_c$ is invalid (not well-formed
        or the service signature is not valid).
  w2.2  if $W_c$ is not $null$, set $max_{ts} \leftarrow max(max_{ts}, W_c.ts)$, and remove from $P_{list}$ all entries
        $e$ such that $e.ts \leq max_{ts}$.
  w2.3  if $P_{list}$ contains an entry for $c$ with a different $ts$ or $hash$, discard the request without
        replying to $c$.
  w2.4  if $\langle c, ts, hash \rangle \notin P_{list}$ and $ts > max_{ts}$, add $\langle c, ts, hash \rangle$ to $P_{list}$.
  w2.5  $i$ generates its signature share (partial signature):
        $\langle a_i, v_i \rangle \leftarrow Thresh\_sign(SK_i, VK_i, VK, \langle ts, hash \rangle)$.
  w2.6  $i$ sends a reply $\langle \text{PREPARE\_REPLY}, \langle ts, hash \rangle, \langle a_i, v_i \rangle \rangle$ to $c$.

Upon receipt of $\langle \text{WRITE}, value, P_{new} \rangle$ from client $c$

  w3.1  if request is invalid or $P_{new}.hash \neq h(value)$, discard request without replying to $c$.
        A $write$ request is invalid if the prepare certificate $P_{new}$ is invalid (not well-formed or
        the service signature is not valid).
  w3.2  if $P_{new}.ts > P_{cert}.ts$, set $data \leftarrow value$ and $P_{cert} \leftarrow P_{new}$.
  w3.3  $i$ generates its signature share (partial signature):
        $\langle a_i, v_i \rangle \leftarrow Thresh\_sign(SK_i, , VK_i, VK, P_{new}.ts)$.
  w3.4  $i$ sends a reply $\langle \text{WRITE\_REPLY}, P_{new}.ts, \langle a_i, v_i \rangle \rangle$ to $c$.
```

Pseudocode 3 Read protocol executed at client c.

```
r1.1  Client $c$ sends a message $\langle \text{READ}, nonce \rangle$ to all servers.
r1.2  $c$ waits for a quorum $(2f + 1)$ of valid replies from different servers. A reply $m_i$ from
      server $i$ is valid if it is well-formed, i.e., $m_i = \langle \text{READ\_REPLY}, value, p, nonce \rangle$ where
      $p$ is a valid prepare certificate (well-formed and the service signature is valid) and
      $p.hash = h(value)$. Moreover, $m_i$ should be correctly authenticated, i.e., its $nonce$
      matches the $nonce$ used in step $r1.1$.
r1.3  Among all replies received in step $r1.2$, $c$ selects the reply with the prepare certificate
      containing the highest timestamp and returns the $value$ related with this reply. Also, if
      all timestamps obtained in step $r1.2$ are equals the read protocol ends.
r2.1  Otherwise the client performs the write back phase for the highest timestamp. This is
      identical to phase 3 of writing (steps $w3.1$, $w3.2$ and $w3.3$), except that the client needs
      to send only to servers that are out of date, and it must wait only for enough responses
      to ensure that a quorum $(2f + 1)$ of servers now have the updated information.
```

Pseudocodes 3 and 4 present the read protocol executed by clients and servers, respectively. Readings are completed in only one phase when there are no faults and concurrency on the system. However, an additional *writeback* phase [12] may be need, where clients write back the reading value in an enough number of servers, ensuring that the most recent information is stored in at least one quorum of servers.

Pseudocode 4 Read protocol executed at server i.

```
Upon receipt of $\langle \text{READ}, nonce \rangle$ from client $c$

  r1.1  $i$ sends a reply $\langle \text{READ\_REPLY}, data, P_{cert}, nonce \rangle$ to $c$.
```

The read protocol executed at servers is very simple. In these operations, servers send a reply containing the stored value and the certificate that proves the integrity of this data.

Correctness. The correctness conditions of PBFT-CUP, as the proofs that PBFT-CUP meets these conditions, are equals to those presented in [10], since these conditions are based on the validity of certificates and on the properties of quorum intersections. The atomicity of the register [8] is ensured by the write back phase of the read, i.e., writing back the read value ensures that all subsequent reads would read this value or a newer one. Wait-freedom is satisified due to the fact that all phases of the protocols require replies of only $n - f$ servers (a quorum), which are, by definition, always available on the system.

3.2 Optimizations

There are two optimizations that can be used to make the protocols more efficient in contention- and fault-free scenarios.

Avoiding the prepare phase of a write. As in BFT-BC (see Section 2.2), it is possible to agglutinate the functions of the phases 1 and 2 of the write protocol of PBFT-BC, reducing the number of communication steps from 6 to 4 in writes in which there are no faulty servers and no write contention. The idea is simple: if all timestamps read by a client c on the first phase of the write protocol are equal to ts, the obtained READ_TS_REPLY messages can be used as the prepare certificate for a timestamp $succ(ts, c)$. In order for this optimization to be used, the hash of the value to be written must be included both in READ_TS and READ_TS_REPLY messages.

Avoiding verification of partial signatures. Two of the most costly steps of the threshold signature scheme are: **(1)** verification of partial signatures (steps $w2.2$ and $w3.2$ of Pseudocode 1), executed by clients; and **(2)** generation of proofs of partial signatures validity (steps $w2.5$ and $w3.3$ of Pseudocode 2), executed by servers. In these steps, if there are no malicious servers in the system, the first quorum of replies received by clients will contain correct partial signatures that suffice to generate the service signature. So, we can change the algorithm to make the client first try to combine the first quorum of partial signatures received without verifying them. If some invalid partial signature is used in the combination, the service signature generated will be invalid for the data that the client is trying to obtain a signature. In this case, the client must wait for new partial signatures and make all possible combinations (using one quorum of partial signatures) until that it obtains a valid signature. Notice that, in the worst case, the client receives f invalid partial signatures in the first quorum of replies and must wait for more f replies in order to obtains a valid service signature. Moreover, as the system always has at least a quorum of correct servers, it is always possible to get a valid signature. In the fault-free case, this optimization drastically reduces the cryptographic processing time required in the write operations. Another advantage of this optimization is related with the proactive recovery of servers (Section 3.3), where their keys (verification keys and partial keys) are updated. Since clients do not verify the received partial signatures, they also do not need update the server verification key after the server recovery. Alternatively, to avoid many combinations in scenarios with failures, the client could execute the normal protocol when it does not obtain the correct service signature from the first quorum of replies.

3.3 Proactive Recovery

Most Byzantine fault-tolerant protocols consider that only a limited number of servers can fail during the system lifetime. However, many systems are designed to remain in operation for a long time (long lived systems), making this assumption problematic: given a sufficient amount of time, an adversary can manage to compromise more than f servers and corrupt the system.

To overcome this, we have developed a recovery protocol[5] that makes faulty replicas behave correctly again. Thus, PBFT-BC can tolerate any number of failures provided that only a limited number of faults f occurs concurrently within a small time interval, called **window of vulnerability**.

To recover a server, it is necessary execute the following actions: *(1)* reboot the system (both, hardware and configurations); *(2)* reload the code from a safe place (e.g., read-only memory); *(3)* recover the server state, that might have been corrupted; and *(4)* make obsolete any information that an attacker might have obtained (e.g., keys and vulnerabilities) from compromissed servers.

Additional Assumptions

To implement automatic recovery (without administrators), some additional assumptions are necessary. We use the same assumptions of [4]:

Key Pairs. Each process handles a pair of keys (public and private from an asymmetric cryptosystem). Private keys are known only to each owner, however all processes know all public keys (through certificates). These keys are only used to reestablish authenticated channels between processes, i.e., to share a secret.

[5] As a Byzantine-faulty replica may appear to behave properly even when compromised, the recovery must be proactive, i.e., even correct servers must execute the recovery procedure.

Secure Cryptography. Each server has a secure cryptographic coprocessor that stores its private key. Thus, it can sign and decrypt messages without exposing the private key. This coprocessor also contains a counter that never goes backwards. The value of this counter is appended in each signed messages in order to avoid replay attacks. An example of co-processor that can be used to provide this service is the TPM (Trusted Platform Module) [19], already available in many commodity PCs.

Read-Only Memory. Each server stores the public keys of other servers, as well as the service public key PK, in some memory that survives failures without being corrupted. Moreover, a hash of the server code is also stored in this memory.

Watchdog Timer. Each server has a watchdog timer that periodically interrupts processing and hands control to a recovery monitor, which is stored in the read-only memory. An attacker is not able to change the rate of watchdog interruptions without having physical access to the machine.

Diversity in Time. The set of vulnerabilities of each replica is modified after each recovery. This ensures that an adversary will not use the same attack to compromise a server imediatelly after its recovery. This can be implemented through a combination of techniques [3] such as changing the operating system of the replica (to modify the set of vulnerabilities), using memory randomization (to modify the address layout of the server) and changing configuration parameters of the system.

Modified Protocol

The main change in the protocol is related with the windows of vulnerability. Each window has a number that is defined in increasing order. Servers append the number of the window of vulnerability in the replies, i.e., there is an additional parameter indicating in which window of vulnerability is the server. A client knows that servers have moved to the next window when it receives at least $f+1$ replies indicating this change in the system. Moreover, prepare certificates also have an additinal attribute to inform in which window it was created.

Sometimes, a client may restart one phase of a write operation concurrent with a recovery procedure. This can happen in phases 2 and 3, when a client is trying to obtain the service signature for a certificate. In fact, servers partial keys are updated by the recovery procedure, but a service signature is correctly generated only if all partial signatures are performed with partial keys of the same window of vulnerability. On the other hand, recovery does not affect read operations, unless, of course, while the server is rebooting and becomes unavailable for a short time.

The time to recover a server can be divided into three parts (Figure 2): T_r – time to restart the system; T_k – time to update keys (patial keys and session keys); and T_s – time to update the server state. A window of vulnerability starts at the begining of a recovery procedure and ends when the next recovery procedure ends (Figure 2). In this period, up to f out of $3f+1$ servers can fail.

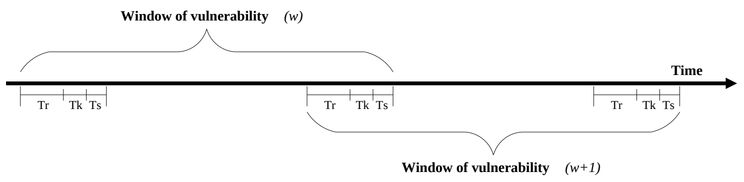


Fig. 2. Relationship between windows of vulnerability and recovery procedures.

It is impossible to develop protocols for proactive recovery in completely asynchronous systems [18], where we can not make any time assumption. Then, there is the problem of ensuring the periodic execution of the recovery procedure. Also, we must find ways to guarantee that this protocol ends.

To solve the first problem we use watchdogs timers that generate periodic interrupts. The second problem is much more complicated and should be solved in one of three ways: *(1)* assume (and justify this assumption!) that an adversary can not control the communication channels connecting two correct processes [22]; *(2)* use an hybrid distributed system, where the recovery procedure is performed in a synchronous and secure subsystem [17, 5]; or *(3)* whether the recovery timeout expires, the server can contact an administrator that can take actions to allow the recovery to terminate [4]. Any of these methods can be incorporated in our system. The steps to recover a server are given bellow:

System Reboot. Each server restarts periodically when the watchdog timer goes off. Before rebooting, each server sends a message to inform other servers that it will perform the recovery, i.e., it is going to the next window of vulnerability. Any correct server that receives $f + 1$ of these messages also restarts, even if its watchdog timer is not expired. This improves availability because the servers do not have to wait for their times to expire before changing to the next window of vulnerability. Moreover, the recovery monitor saves the state of the server ($data$ and P_{cert}) and its keys (partial key and verification key). Then, it reboots the system with correct code and restarts the server from the saved state. The integrity of the operating system and service code can be verified through the hashes of them stored in the read-only memory. If the copy of the code stored by the server is corrupt, the recovery monitor can fetch the correct code from other servers. Thus, it is guaranteed that the server code is correct and it did not lose its state. This state may be corrupted but the server must use it to process requests during the recovery phase in order to ensure the system properties when the recovering server is correct. Otherwise, the recovery procedure could cause more faults than the maximum tolerated by the system. But the recovering server could be faulty. So, the state should be updated, together with all confidential data that an attacker might have obtained (partial and session keys).

Keys Update. Session keys are updated as in [4]: the process i updates its session key used to authenticate a channel to other process j, by sending to j a secret that must be encrypted with the public key of j (thus, only j is able to access this secret) and signed with the private key of i (ensuring authenticity).

So, a new session key is established to authenticate messages that i sends to j. These messages are signed by the secure coprocessor that appends the value of the counter (*count*) to avoid replay attacks, i.e., the *count* must be larger than the value of the last message that j has received from i. Moreover, the servers update its partial and verification keys through a proactive update protocol such as APSS [22]. In this procedure, the asymmetric keys of the servers are used to exchange confidential data.

State Update. Each server recovers its state by performing a normal read operation on a quorum of servers (considering himself). The value read is used to update the variables *data* and P_{cert}. Other variables are reset to the same value that they had in the boot of the system, i.e., P_{list} to a empty set and max_{ts} to null. These variables are used to avoid that malicious clients prepare many write operations. As these variables are "lost" in the recovery procedure, we use the following mechanism to control the actions of clients: as each prepare certificate contains the value w of the window of vulnerability, servers do not consider correct (in the third phase of the write) prepare certificates generated in previous windows of vulnerability. Thus, if a client receives replies from $f+1$ servers indicating that they have advanced to the next window of vulnerability, such client must restart the second phase of the write protocol and wait for responses from a quorum of servers that are in the new window. We could relax this requirement by accepting certificates generated in the previous window of vulnerability. In that case, a malicious client is able to prepare at most two (without optimization – 3 phases) or four (with optimization – 2 phases) write operations.

4 Evaluation

In this section, we analyze the PBFT-BC performance by comparing it with its precursor, the BFT-BC [10].

Analytical Evaluation. Both PBFT-BC and BFT-BC have read and write protocols with linear communication complexity ($O(n)$) and the same number of communication steps: 2 for reads (4 under write-contention) and 4 for writes (6 under write-contention). However, the messages of PBFT-BC are much smaller than the messages in BFT-BC due to the size of its certificates. A certificate of BFT-BC requires $2f+1$ signed messages from a quorum of servers, while in PBFT-BC, the certificate requires a single threshold signature. This represents an improvement by a a factor of $2f+1$ in terms of message size for PBFT-BC.

Even more important than lowering message sizes, are the bennefits of PBFT-BC in terms of the number cryptographic operations required by the protocols. Table 1 presents an analysis of the cryprographic costs to write a value in the system. As shown in the table, BFT-BC executes verifications in quadratic order ($O(f^2)$), while for PBFT-BC this cost is linear ($O(f)$). This happens because both protocols verify a quorum of certificates (first phase), where in BFT-BC each certificate is validated by a quorum of signatures and in PBFT-BC by only one signature. This cost is also reflected in read operations, where, discarding a possible write back phase, BFT-BC executes $4f^2+4f+1$ verifications while PBFT-BC executes only $2f+1$ verifications at client side.

Phase	BFT-BC		PBFT-BC	
	client	server	client	server
1[a]	$4f^2 + 4f + 1$ verify	—	$2f + 1$ verify	—
2[a]	$2f + 1$ verify	$4f + 2$ verify + 1 sign	1 comb of $2f + 1$ partial signatures	2 verify + 1 partial sign
3[a]	$2f + 1$ verify	$2f + 1$ verify + 1 sign	1 comb of $2f + 1$ partial signatures	1 verify + 1 partial sign
Total Costs	$4f^2 + 8f + 3$ verify	$6f + 3$ verify + 2 sign	$2f + 1$ verify + 2 comb	3 verify + 2 partial sign
	$4f^2 + 14f + 6$ verify + 2 sign		$2f + 4$ verify + 2 partial sign + 2 comb	

Table 1. Write protocol costs for a scenario without failures.

Experimental Evaluation. To better ilustrate the bennefits of PBFT-BC when compared with BFT-BC we run some experiments to observe the latency of their read and write protocols. In particular, we are interested in observing the latency to execute read/write operations, since the effects caused by concurrency and failures is basically the same in both protocols.

We implemented both protocols using the Java programming language[6]. We evaluate the protocols without proactive recovery, since they can only impact the latency of operations executed during the recovery procedures.

In order to quantify this latency, some experiments were conducted in Emulab [20], where we allocate 11 *pc3000* machines (3.0 GHz 64-bit Pentium Xeon with 2GB of RAM and gigabit network cards) and a 100Mbs switched network. The network is emulated as a VLAN configured in a Cisco 4509 switch where we add a non-negligible latency of $10ms$ in the communications. The software installed on the machines was Red Hat Linux 6 with kernel 2.4.20 and Sun's 32-bit JRE version 1.6.0_02. All experiments were done with the Just-In-Time (JIT) compiler enabled, and run a warm-up phase to transform the bytecode into native code.

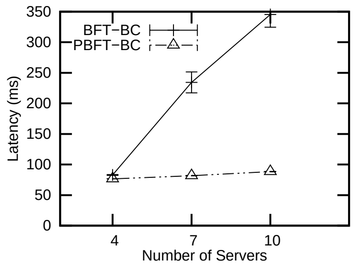
In the experiments, we use the standard 512-bits RSA signature in the BFT-BC and we configure the threshold signature scheme to generate RSA signatures of 512 bits in the PBFT-BC. The size of the objects, that were written and read from the system, was fixed in 1024 bytes. Then, we set the system with 4 ($f = 1$), 7 ($f = 2$) or 10 ($f = 3$) servers to analyze its scalability. We executed each operation 1000 times and obtained the mean time discarding the 5% values with greater variance.

Figure 3 presents the latency observed in the execution of each operation. We can see in this figure that PBFT-BC outperforms the performance of the BFT-BC. This happens mainly because the optimization in PBFT-BC write operations (verifications of partial signatures are avoided – Section 3.1) and due to the fact that PBFT-BC certificates have constant small size (only one signature), while BFT-BC certificates contains a quorum of signatures.

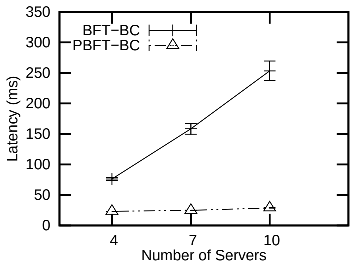
In fact, the total number of bytes exchanged by the client and each PBFT-BC server was approximately 2756 to write an object and 1466 to read an object,

[6] For threshold cryptography, we adapt the library found at `http://threshsig.sf.net/`, which is an implementation of the protocol described in [16].

for all configurations of the system. On the other hand, using the BFT-BC protocol, these numbers increase to 5884/2229, 8120/2824 e 10292/3419 bytes for write/read operations in the system composed by 4, 7 or 10 servers, respectively. Other point to highlight is the low time need to produce a partial signature (aprox. 4.5ms) and to combine a quorum of these signatures (0.45ms for quorum of 3 servers). The time necessary to produce a standard RSA signature was approx. 1.5ms and each verification spent approx. 0.26ms, in both protocols.



(a) Write



(b) Read

Fig. 3. PBFT-BC X BFT-BC.

These results clearly shows the benefits of using threshold signatures instead of signature sets to build certificate in quorum systems.

5 Related Work

There are many works that propose register implementation using quorum system protocols for different fault models (crash or malicious) and consistency semantics (safe, regular or atomic). The first protocols that tolerate malicious behavior of processes are presented in [11, 12]. These works address two types of quorum systems: *(1) f-dissemination quorum system*, which does not allow the presence of malicious clients and requires $3f + 1$ servers; and *(2) f-masking quorum system*, which allows the presence of malicious clients but requires $4f + 1$ servers. Moreover, the *writeback* phase of the read protocol was introduced in [12]. This phase guarantees the atomic semantics of the operations performed on the system.

The quorum system more similar to PBFT-BC is the BFT-BC [10]. This system also tolerates malicious clients, requiring only $3f + 1$ servers. BFT-BC uses server signatures to guarantee the integrity of the stored data. The main difference of our work is the use of threshold cryptography to make data self-verifiable. This approach improves significantly the performance of the system, makes possible the update of the shares (partial keys) stored on the servers and facilitates the development of proactive servers recovery protocols since the clients keys do not need be changed.

Some protocols for proactive recovery of servers were proposed in [4, 21, 13]. Our protocol uses the same assumptions adopted in [4], that presents a protocol for active replication, but does not use threshold cryptography for signatures. Other works that use threshold cryptography are COCA [21], that implements a fault-tolerant online certification authority, and CODEX [13], that implements a distributed service for storage and dissemination of secrets. These works employ the APSS [22] proactive secret sharing protocol to update the shares of a service private key. Our protocol also employ the APSS protocol. However, the architectures of COCA and CODEX systems are significantly different from the adopted in our work. These systems use a server, called delegate, to presides over the processing of each client request. The use of delegates decrease the performance of the system since that are necessary more communication steps to perform an operation. Moreover, the additional assumptions for proactive recovery, assumed by COCA and CODEX, seem to be insufficient to guarantee the progress of the system [18]. The architecture Steward [1] also utilizes threshold cryptography, but it is used to guarantee the authenticity of a decision made by a set of processes.

6 Conclusions and Future Works

In this work we proposed a new protocol for Byzantine Quorum systems, the PBFT-BC, which tolerates malicious clients, presents optimal resilience and supplies a register with atomic semantics of operations. Moreover, we developed a protocol for proactive recovery of servers that implement the PBFT-BC. Also, we showed that PBFT-BC outperforms the performance of BFT-BC protocol, which implements a service with the same characteristics.

The next steps of this work will focus on extending this model to operate in a dynamic environment, where processes can join and leave the system at any time. In this direction, the use of threshold cryptography provides the necessary flexibility to make the PBFT-BC protocol adaptable to the changes that occur in the composition of the servers group.

Acknowledgments. Alchieri, Pereira and Fraga are supported by CNPq (Brazilian National Research Council). This work was partially supported by the FCT through the Multiannual and the CMU-Portugal Programmes.

References

1. Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling Byzantine fault-tolerant replication to wide area networks. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 105–114, 2006.
2. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, November 1993.
3. Alysson Bessani, Alessandro Daidone, Ilir Gashi, Rafael Obelheiro, Paulo Sousa, and Vladimir Stankovic. Enhancing fault/intrusion tolerance through design and configuration diversity. In *Proc. of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*, June 2009.
4. Miguel Castro and Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

5. Rogerio Correia and Paulo Sousa. WEST: Wormhole-enhanced state transfer. In *Proc. of the DSN'09 Workshop on Proactive Failure Avoidance, Recovery and Maintenance (PFARM)*, June 2009.
6. Wagner Saback Dantas, Alysson Neves Bessani, Joni da Silva Fraga, and Miguel Correia. Evaluating Byzantine quorum systems. In *Proc. of the 26th IEEE International Symposium on Reliable Distributed Systems*, 2007.
7. David Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 1979.
8. Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, January 1986.
9. Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.
10. Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of the 26th IEEE International Conference on Distributed Computing Systems*, June 2006.
11. Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.
12. Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proc. of 17th Symposium on Reliable Distributed Systems*, pages 51–60, 1998.
13. Michael A. Marsh and Fred B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable Secure Computing*, 1(1):34–47, 2004.
14. Rafael Rodrigues Obelheiro, Alysson Neves Bessani, Lau Cheuk Lung, and Miguel Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, University of Lisbon, September 2006.
15. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
16. Victor Shoup. Practical threshold signatures. In *Advances in Cryptology: EUROCRYPT 2000, Lecture Notes in Computer Science*, volume 1807, pages 207–222. Springer-Verlag, 2000.
17. Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems.* to appear.
18. Paulo Sousa, Nuno Ferreira Neves, and Paulo Verissimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2005*, 2005.
19. Trusted Computing Group. Trusted platform module web page. `https://www.trustedcomputinggroup.org/groups/tpm/`, 2009.
20. Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of 5th Symposium on Operating Systems Design and Implementations*, December 2002.
21. Lidong Zhou, Fred Schneider, and Robbert Van Rennesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.
22. Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. APSS: proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security*, 8(3):259–286, 2005.