Active Quorum Systems: Specification and Correctness Proof

Alysson Bessani

DI–FCUL–TR–2010–02

DOI:10455/6673

(http://hdl.handle.net/10455/6673)

July 2010

Published at Docs.DI (http://docs.di.fc.ul.pt/), the repository of the Department of Informatics of the University of Lisbon, Faculty of Sciences.

Active Quorum Systems: Specification and Correctness Proof

Alysson Neves Bessani
University of Lisbon, Faculty of Sciences – Portugal

July 23, 2010

Abstract

In this report we specify the Active Quorum System replication protocol for Byzantine fault tolerance, which uses a set of diverse algorithms for implementing different kinds of operations based on the semantics of the application, thus being efficient independently of the environmental conditions. The algorithms are specified and their correctness (linearizability and wait-freedom) are proved.

1 Introduction

In this report we show the complete algorithms and correctness proofs for implementing the AQS Byzantine fault-tolerant (BFT) protocol. The main feature of this protocol is the fact that it defines three classes of operations: read, write and rmw (read-modify-write) and use the weakest protocols that are necessary to implement these operations ensuring linearizability and wait-freedom even in face of malicious behavior. From the algorithmic point of view, AQS is build over previous works both for read/write Byzantine quorum systems [14] and BFT state machine replication [3], integrating these two approaches in a natural way to build a simple, elegant and efficient BFT replication protocol.

This report is organized as follows. Section 2 presents our system model and the properties that AQS aims to satisfy. Section 3 presents the core of this report: the AQS object specification; the three AQS protocols (read, write and rmw) and its detailed correctness proofs. Finally, in Section 4, we discuss some interesting features of the protocol.

2 Preliminaries

2.1 System Model

We assume a fully connected networked system with $n = 3f + 1$ servers in which at most f can fail in a Byzantine way [13]. An arbitrary number of clients interact with the system, and these clients can also be subject to Byzantine faults. We assume that the servers fail independently, i.e.,

the probability of a server being faulty is not affected by the existence of other faulty servers on the system. This can be implemented using several kinds of diversity [4, 19].

Since we rely on a modified version of CL-BFT [3] to execute some operations, we require a partially synchronous system to ensure liveness: there is an unknown instant after which all communications and computations are synchronous (have unknown time bounds) [6].

To make our algorithms simpler, we assume that all communications are made through authenticated reliable FIFO channels. These channels can be implemented in practice using MACs (Message Authentication Codes) and retransmissions over a *fair link*[1] [15]. This assumption is equivalent to the one used by previous BFT works [3, 1, 17, 5, 12, 21]: they assumed an unreliable authenticated channel that cannot drop the same message forever (i.e., a fair link) and make this channel reliable by resending messages.

To ensure secure communication, we assume there is a public-key infrastructure which provides to each process the public key of any other process of the system. The pairs of private and public keys are used to sign and verify messages (a message m correctly signed by a server s is denoted by m_{σ_s}). Moreover, we assume the existence of a collision-resistant hash function H. Finally, all messages contains nonces to avoid replay attacks.

2.2 Properties of AQS

Byzantine fault-tolerant (replicated) objects implemented using AQS satisfies the following properties as long as the assumptions described in the previous section hold:

- **Linearizability**: the system appears to be accessed sequentially, i.e., the operations are executed in some order which is in accordance with its sequential specification and respects their real-time precedence [11];
- **Wait-freedom**: all operations requested by correct *clients* complete independently of the behavior of other (correct or incorrect) *clients* of the system [10].

We remark that wait-freedom is ensured in respect to *client failures*. If more than f servers fail, it is not guaranteed that the system satisfies any property.

3 Active Quorum Systems

The *AQS model*, which is used to implement *AQS objects*, is an hybrid replication model in which some operations are executed using quorum-based protocols while other use agreement-based protocols. A key feature of this model is that object operations are divided in three classes that are implemented through different protocols:

- **write**: the state of the object is (over)written by the argument of the operation;
- **read**: the state of the object is read;
- **read-modify-write (rmw)**: the state of the object is modified according to the operation arguments and its current state.

[1] A message transmitted infinitely many times is received infinitely many times.

Write and rmw operations are collectively called *update operations*. The main difference between these two types of operations is that in the first, the state of the object is updated to the value being written (independently of the previous value) while in the second the resulting state depends on the arguments of the operation and its previous state. For example, the operation "$x \leftarrow 2$" is a write while the operation "$x \leftarrow x + 2$" is a rmw (read x, update its value by adding 2 and store this new value in x). Notice that the modification done on the state is completely arbitrary and dependent of the semantics of the object being implemented.

Given these three classes of operations, AQS uses quorum-based protocols to implement read and write operations efficiently. Operations of the class rmw, on the other hand, require more expensive consensus-based protocols that are less efficient in at least two aspects: *(i.)* they usually require synchrony assumptions to terminate (read and write quorum protocols can be implemented in completely asynchronous systems), and *(ii.)* they usually have $O(n^2)$ message complexity instead of the usual $O(n)$ exhibited by read and write quorum protocols. This means that if the replicated object supports only operations with read and write semantics, AQS behaves like an atomic register protocol for f-dissemination Byzantine quorum systems [16], while if the object supports only general rmw semantics, the system operates like CL-BFT state machine replication algorithm.

As expected, read and write quorum protocols do not fit directly with a state machine replication algorithm, so, we had to develop some techniques to combine these two approaches in a single replication algorithm. There are two challenges that must be addressed when these two techniques are integrated to be used together. First, there should be some mechanism that allows quorum-based read protocols to obtain a response when state machine replication update protocols are being executed. The main problem here is that read protocols are developed to work concurrently with write protocols since both are basic quorum-based abstractions [7, 16], and now we must augment these protocols to be able to operate concurrently with agreement-based update algorithms. Second, we must extend the state machine replication protocol used in the rmw operations to be able to execute operations even in replicas with different states, e.g., when a write is being executed concurrently with a rmw. To cope with the first challenge, we use timestamps and some properties of quorum systems to be sure that reads concurrent with rmws do not impair the replicated object linearizability. The second challenge is addressed though the modification of the CL-BFT protocol to make the primary send the current state of the object and the proposed result of the rmw operation together with the operation ordering it proposes.

There are other interesting features of AQS when compared with other replication models. First, AQS is not a classic state machine replication protocol [20, 3], it is a quorum protocol in which rmw operations are ordered and executed by a primary server, as defined in the CL-BFT protocol. The immediate consequence of this fact is that there is no need for replica determinism as long as other replicas are able to verify the legality of the rmw result proposed by the primary[2]. Another consequence is that the state of a service should be partitioned on as many AQS objects as possible. Second, AQS does not advocate any optimization for well behaved executions (failure/contention-free or synchronous): the protocols are as efficient as the semantics of the operation allows them to be. Regarding this second point, AQS protocols do not preclude the

[2]In practice, this new state can be the difference between the previous and the new state or the operation to be executed plus a set of data items (like timestamps and random numbers) that should be used for non-deterministic operations [4].

use of optimizations, but we do not require specific execution properties to be significantly more efficient. In fact, the three AQS operations protocols require only two extra communication steps (a round-trip) in non-favorable executions (e.g., presence of concurrency or faulty non-primary servers).

Finally, a "subjective" advantage of the protocols employed by AQS is their simplicity when compared with other efforts on the area (e.g., [5, 12]: there is no need to use complex mechanisms to ensure safety when the optimistic assumptions do not hold and servers do not need to store different versions of the object state to make rollbacks when needed.

3.1 AQS Object Specification

To prove the linearizability of AQS protocol we should first define what is the correctness condition of the objects implemented by the protocol. Since we will not prove that all operations are executed by all replicas at the same order (which directly implies linearizability) as is the case with most state machine replication protocols, we have to define what are the ordering constraints considering interactions between the operations that can be executed on AQS objects.

The state of an AQS object O is denoted by $S_o = \langle v,t \rangle$, where v is the current value of the object and t is the timestamp associated with this value. Regarding timestamps, we assume that each process produces timestamps from different domains. This can be easily done making each timestamp be composed by its value concatenated with the id of the process that produced it in its lower bits

Given an operation o performed on an object O, $S(O,o)$ denotes the state of the object o after operation o completes. Given $S(O,o) = \langle v,t \rangle$, we say that an operation o' on O is *subsequent* to o if $S(O,o') = \langle v',t' \rangle$, with $t' > t$. We use $S(O,o).t$ and $S(O,o).v$ to access the timestamp and value associated with the state $S(O,o)$. Moreover, the *base state* of a rmw operation is the state on which the rmw operation being invoked will be applied.

An AQS object O that supports atomic and non-concurrent read, write and rmw operations should satisfy the following constraints:

1. a write w_2 that executes after some write w_1, writes a timestamp $S(O,w_2).t > S(O,w_1).t$;
2. a read r that executes after some write w, returns $S(O,w)$ or the state of a subsequent operation executed before r;
3. a read r_2 that executes after some read r_1, reads the state read by r_1 or the state of a subsequent operation executed before r_2;
4. a rmw m_2 that executes after some rmw m_1, uses $S(O,m_1)$ or the state of a subsequent operation as its base state and produces a timestamp $S(O,m_2).t > S(O,m_1).t$;
5. a read r that executes after some rmw m, returns $S(O,m)$ or the state of a subsequent operation executed before r;
6. a rmw m that executes after some read r, will use the state read by r or the state of a subsequent operation executed before m as its base state;
7. a write w that executes after some rmw m, produces a timestamp $S(O,w).t > S(O,m).t$;

8. a rmw m that executes after some write w, uses $S(O,w)$ or the state written by a subsequent operation executed before m as its base state and produces a timestamp $S(O,m).t > S(O,w).t$;

The first three conditions are the ones usually employed to prove atomicity of read/write registers implemented over quorum systems [14, 18]. The fourth, fifth and sixth conditions are the ones that would be proved for a SMR system which employs the optimization to avoid ordering read-only requests [3]. The last two are specific to AQS objects and define the relationship between writes and rmws.

3.2 Protocols

In this section we present the AQS protocols. In order to simplify the presentation, this description considers a single replicated object. To support multiple objects it is necessary to associate object identifiers to each variable and message handled in the protocols.

The protocols on this section use the function $succ(t,c)$ to calculate the timestamp that will succeed t produced by client c. This function corresponds to the increment of t by one unit and the appending of c id on the lowest order bits of this value.

Before we delve into the protocols, we first present the information that comprises the server state and describe the certificates used in the protocols.

3.2.1 Server State

Each server s stores its copy of the state of the object $\langle v_s, t_s \rangle$ plus an update certificate C_s associated with this state (see Section 3.2.2). Additionally, for each client c that can write on the object, each server s stores the value and timestamp of the last write it prepared on $prepared_s[c]$ or $optimized_s[c]$ (if an optimized write was executed, see Section 3.2.3). Finally, each server also stores, for each client c, the type of the next message it is expecting from this client in $pending_s[c]$.

3.2.2 Certificates

The AQS protocols use the concept of certificate [2, 3, 14] to make clients and servers justify their actions to other clients and servers. More specifically, our protocols use two types of certificates.

An *update certificate* for v and t (produced by client c) is a set of $n-f$ correctly signed messages of the form $\langle \text{PREPARED}, t, \mathsf{H}(v) \rangle_{\sigma_s}$ from different servers, which proves that the client commits to write v with the timestamp t. Alternatively, a set of $n-f$ correctly signed messages of the form $\langle \text{TS}, \mathsf{H}(v), t' \rangle_{\sigma_s}$ ($t = succ(t',c)$) from different servers can be used as an update certificate for v and t, which we call *fast update certificate*. Finally, an update certificate for the pair v and t resulting from a rmw operation on the object can be the set of COMMIT messages produced in the last phase of the corresponding execution of the CL-BFT protocol [3]. A certificate is said to be *valid* if it satisfies one of these three definitions. when a certificate C is valid for the pair $\langle v,t \rangle$, we say that C *justifies* the association of v with t. It is important to mention that all messages on the same certificate should have the same nonce to avoid replay attacks.

A *completeness certificate* for client c and write operation with timestamp t is a set of $n-f$ correctly signed messages of the form $\langle \text{ACK}, t \rangle_{\sigma_s}$ proving that client c completed this operation. This certificate is used to *bound to two* the number of incomplete writes a malicious client can execute, as described in [14].

3.2.3 Write

To execute a *write*(v) operation, a client c with a completeness certificate C_c executes the following protocol with AQS servers:

1. The client sends a message $\langle$GET-TS$, \mathsf{H}(v), C_c\rangle$ to all servers and waits for valid replies $\langle\langle \text{TS}, \mathsf{H}(v), t_s\rangle_{\sigma_s}, C_s\rangle$ from $n-f$ different servers. A server s only replies $\langle\langle \text{TS}, t_s, \mathsf{H}(v)\rangle_{\sigma_s}, C_s\rangle$ to a GET-TS message if $pending_s[c] = \emptyset$, there is no RMW request from this client being processed (received but reply not sent) and C_c is a valid *completeness certificate* showing that t_c is the timestamp of the last write[3] by client c (stored in $optimized_s[c]$ or $prepared_s[c]$). After replaying, the server sets $pending_s[c] = PREPAREW$ and $optimized_s[c] = succ(t_s, c)$.

2. A TS reply is valid iff C_s is a certificate that proves that the object timestamp is t_s, i.e., the maximum timestamp found in the update certificate C_s is the successor of t_s. The client waits for $n-f$ valid replies from different servers and verifies if the write will be optimized or not:

   Optimized write. If all accessed servers replied TS messages with the same timestamp t, these replies define a fast update certificate C_v and the client can proceed to step 3;

   Normal write. If there are different timestamps among the valid replies received by the client, it chooses the one with greater value t_{max}, justified by C_{max}, and sends a message $\langle$PREPAREW$, \mathsf{H}(v), succ(t_{max}, c), C_{max}\rangle$ to all servers. When a server s receives this message, it verifies if $pending_s[c] = PREPAREW$, $succ(t_{max}, c) > prepared_s[c]$ and if the proposed timestamp is the successor of t_{max} for c, which should be validated by C_{max}. If any of the these conditions are not satisfied, the message is discarded, otherwise a reply $\langle$PREPARED$, succ(t_{max}, c), \mathsf{H}(v)\rangle_{\sigma_s}$ is sent to the client. After replaying, the server sets $pending_s[c] = WRITE$ and $prepared_s[c] = succ(t_{max}, c)$. The client collects valid PREPARED replies from $n-f$ servers and uses them as an update certificate C_v for the next step.

3. The client defines t_v as $succ(t, c)$, being t the timestamp read from $n-f$ servers in step 1 (if an optimized write is being executed), or as the timestamp prepared in step 2 and sends $\langle$WRITE$, t_v, v, C_v\rangle$ to all servers. A server s accepts this message only if $pending_s[c] \in \{PREPAREW, WRITE\}$ and C_v is an update certificate for v and t_v. If $t_v > t_s$, the server changes t_s, v_s and C_s to t_v, v and C_v, respectively, otherwise the write is ignored. In any case, if a message is accepted, s updates $pending_s[c] = \emptyset$ and sends a message $\langle$ACK$, t_v\rangle_{\sigma_s}$ to c. The client collects $n-f$ valid ACK replies to form the completeness certificate C_c to be used on the next write.

Figure 1 illustrates the message pattern of the write protocol. The second phase is presented in a gray area representing that it is not mandatory for all executions.

The first two phases of the protocol have two main purposes: reading the current timestamp of the object and building an update certificate for a timestamp-value pair. The second phase of the protocol is only needed if different servers reply different timestamps in the protocol's first phase. This can happen if there are concurrent updates being executed or if some server is faulty and replies an old timestamp. This certificate makes the state of the object *self-verifying* and allow us to use the minimal number of servers $(3f+1)$ even with faulty clients [16].

[3] If this is the first write by c the certificate can be empty.

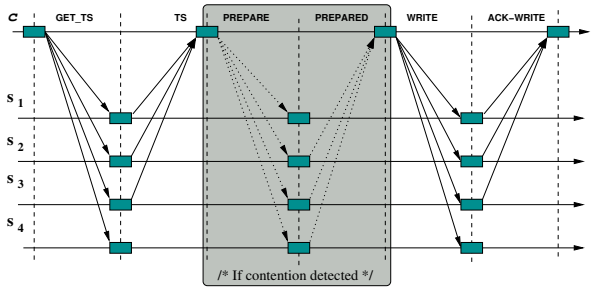

Figure 1: The AQS' write protocol.

3.2.4 Read

In the read operation, besides obtaining the current state of the object, the client must ensure that all subsequent reads that happen before some update is executed will read this same value observed by it. This is done by writing back the read value in the servers that report to have an old value for the object.

To execute a $read()$, a client c has to execute the following protocol with the servers:

1. The client sends a $\langle \text{READ} \rangle$ operation to all servers and waits for $\langle \text{READ-REPLY}, t_s, v_s, C_s \rangle$ replies from $n-f$ different servers with with pairs $\langle t_s, v_s \rangle$ justified by C_s. When a server receives the READ message, it sends the READ-REPLY message to the client informing its current timestamp, value and update certificate.

2. After receiving $n-f$ READ-REPLY messages from different servers, two situations can arise:

   - if all messages have the same $\langle v,t \rangle$ pair, the client returns v as the operation result and finishes the protocol;
   - otherwise, the tuple $\langle v,t,C \rangle$ with the highest t value is written back to the system: the client sends a message $\langle \text{WRITE-BACK}, t, v, C \rangle$ to all servers that not reported $\langle v,t,C \rangle$. When a server receives a writeback, it updates its object state (if its current t_s is less than the one being written) and sends an (unsigned) ACK-WB message to the client. Let $n_{outdated} < n$ be the number of servers that not reported $\langle v,t,C \rangle$. The client waits for ACK-WB messages from $n_{outdated} - f$ different servers to finish the operation returning v.

Figure 2 presents the message pattern of AQS' read protocol. Notice that write-backs are employed in a fault-free read execution only if they are concurrent with a write or a rmw. If there is no contending update, the read finishes in two communication steps.

3.2.5 Read-Modify-Write

The rmw protocol is build upon the CL-BFT state machine replication protocol [3]. More specifically, AQS rmw extends this protocol taking advantage of two of its features: *(i.)* the existence of a primary and the capability to elect a new one in case it is faulty and *(ii.)* the ordering of operations,

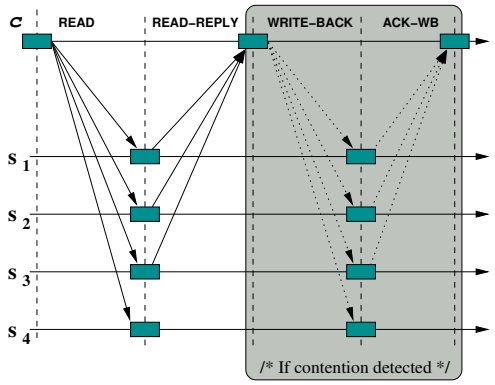

Figure 2: The AQS' read protocol.

required for correct execution of rmw operations. Before presenting the AQS protocol in detail, let us briefly recall how CL-BFT works.

The protocol begins with a client sending a signed request m to all servers. One of the servers, the *primary*, is responsible for establishing the sequence number of of the messages sent by clients. The primary then sends a PRE-PREPARE message to other servers (the backups) giving a sequence number i to m. A server accepts a PRE-PREPARE message if the proposal of the primary is *good*: the signature of m is valid and no other PRE-PREPARE message was accepted for sequence number i. When a server accepts a PRE-PREPARE message, it sends a PREPARE message with m and i to all servers. When a server receives $2f$ PREPARE messages from other servers with the same m and i, it marks m as prepared and sends a COMMIT message with m and i to all servers. When a server receives $2f$ COMMIT messages from other servers with the same m and i, it commits m, i.e., it accepts that message m is the i-th message to be delivered. While the PREPARE phase of the protocol ensures that there cannot be two prepared messages for the same sequence number i (which is sufficient to order messages when the primary is correct), the COMMIT phase ensures that a message committed with sequence number i will have this sequence number even if the primary is faulty.

When the primary is detected to be faulty, a view change protocol is started to freeze the execution and elect a new primary. When a new primary is elected, it collects the protocol state from $2f+1$ servers (including itself). The protocol state comprises information about accepted, prepared and committed messages. This information is signed and allows the new primary to verify if some message was already committed with some sequence number. Then, the new primary continues to order messages. For a complete description of CL-BFT protocol and its many subtleties, we refer the reader to [3].

In order to execute a rmw operation op on the system, a client c has to send a message $\langle \text{RMW}, op \rangle_{\sigma_c}$ to the servers and wait for $n-f$ reply messages $\langle \text{RMW-REPLY}, r \rangle$ from different servers with the same result r.

Upon receiving the RMW message from a client c, each correct server s verifies if the client still has pending operations to be executed, i.e., $pending_s[c] \in \{PREPAREW, WRITE\}$ or there is some RMW for this client still being processed (request received but reply not sent). If this is the case, the message is discarded, otherwise the RMW operation starts.

After that, each backup server s sends a message $\langle \text{STATE}, \mathsf{H}(op), t_s, v_s, C_s \rangle_{\sigma_s}$ to the current

primary of the CL-BFT protocol. After receiving the RMW message, the primary waits for $n-f$ STATE messages from different servers (including itself), in which the C_s justifies the pair $\langle v_s, t_s \rangle$, and stores them on a C_l set. The primary then chooses the STATE message with greater timestamp from C_l, stores its value and timestamp on $base\_state = \langle v_{max}, t_{max} \rangle$ and begins the execution of the CL-BFT protocol with the following modifications:

1. Before ordering the rmw request, the primary l executes op on the value v_{max} and generates the updated state v_l (v_{max} from *base_state*) and the reply r_l for op. The *execution info* $\langle base\_state, C_l, v_l, r_l \rangle$ for the request op is then appended on the PRE-PREPARE message sent by the primary.

2. Each server s only accepts a PRE-PREPARE message for op with sequence number i and execution info $\langle base\_state, C_l, v_l, r_l \rangle$ from a primary l in the first step of the CL-BFT protocol if:

   (a) the usual conditions required for preparing m with sequence number i in CL-BFT are meet [3];

   (b) $base\_state = \langle v_{max}, t_{max} \rangle$ is validated by C_l, i.e., the state in which the primary applied the rmw operation is the most recent state observed (the one with greatest timestamp) on the first step of the rmw protocol;

   (c) the outcome of op proposed by l is valid, i.e., v_l and r_l are *possible* resulting state and reply, respectively, after executing op on v_{max}.

   If the PRE-PREPARE message is not accepted, a new view is started to change the primary.

3. COMMIT messages must be signed in order to be used to build update certificates.

4. After committing operation op, server s generates $t_l = succ(t_{max}, l)$ (t_{max} from the *base_state* and l is the primary that defined the sequence number for this operation[4]) and, if $t_l > t_s$, the server defines t_s as t_l, v_s as v_l and C_s as the set of $n-f$ signed COMMIT messages just received. After that, s returns $\langle \text{RMW-REPLY}, r \rangle$ to c.

5. When a new primary is elected (to substitute some previous primary that was unable to order some operation op), each server s sends a message $\langle \text{STATE}, \mathsf{H}(op), t_s, v_s, C_s \rangle_{\sigma_s}$ with its current object state to the new primary (this message can be sent together with the VIEW-CHANGE message of CL-BFT [3]). The new primary then can process op to update its current state choosing from the received states the valid one with greater t_s.

Notice that the first phase of the RMW protocol (in which the servers send their current state to the primary) plus modification 5 described above are implicit reads executed by the primary in order to get the most recent state to apply the rmw operation.

[4]This must be preserved across views.

Optimization. One optimization that we can make in this protocol is to exploit the "expected" common case in which there is no write concurrent with the rmw operation. In this case, the first phase of the protocol, in which the primary collect STATE messages from other servers does not need to happen. Instead, when the primary receive the RMW message from the client, it can send the PRE-PREPARE message using its current state $\langle v_l, t_l \rangle$ as the operation base state and the update certificate of its state as C_l. Other server s will accept the proposed current state if it is valid and its timestamp $t_l \geq t_s$ and will continue to execute the CL-BFT protocol as usual. If the message is not accepted by some server it will send a STATE message to the primary that, after collecting $n - f$ of these messages, resumes the execution using the obtained most recent state. Moreover, s will send a message $\langle$UPDATE-PRIMARY$\rangle$ to other servers. When a server receives $f + 1$ of these messages it send a STATE message to the primary (if it did not sent it yet) and waits for a new PRE-PREPARE.

This optimization makes the protocol execute in one step less in the optimistic case (5 communication steps - the same number of steps of CL-BFT) and one step more when this is not the case (7 communication steps), when compared with the version without the optimization (6 communication steps). Moreover, it avoids costly STATE messages and the use of a large C_l when write contention does not exists. Figure 3 illustrates a fault-free execution of the rmw protocol with the proposed optimization.

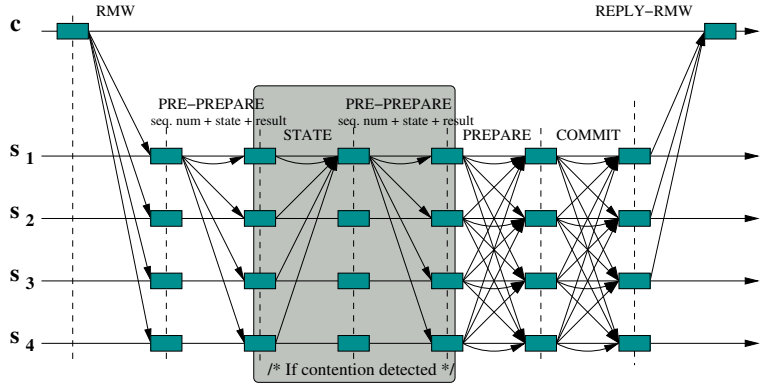

Figure 3: The optimized AQS' rmw protocol.

3.3 Correctness

In this section we present the correctness proof of the AQS protocols, i.e., we prove that they satisfy safety (linearizability) and liveness (wait-freedom).

The proofs presented in this section make use of the following terminology:

- when an operation o reads (resp. writes) a value v with timestamp t, justified by an update certificate C, we say that o reads (resp. writes) $\langle v, t, C \rangle$, denoted by $o(v, t, C)$.
- a *rmw* operation *uses* $\langle v, t, C \rangle$ *as its base state* when the primary l executes the rmw operation on v and defines the timestamp for the new value as $succ(t, s)$.
- we say that an operation o_2 *directly follows* an operation o_1 if and only if o_1 completes before o_2 starts and no other update operation starts or completes before o_2 completes.

3.3.1 Informal Correctness Proof

To prove that the system satisfies linearizability we must show that all clients see the same sequence of operations in the system and this sequence satisfies the constraints defined in Section 3.1. This implies in showing two main properties of the protocols:

- *(l-i.)* let u_1 be an update operation that writes the pair $\langle v_1, t_1 \rangle$ and u_2 be the update operation that writes the pair $\langle v_2, t_2 \rangle$. If there are no writes or rmws between u_1 and u_2, all reads (resp. rmws) that take effect after u_1 takes effect and before u_2 takes effect return (resp. use as its base state) $\langle v_1, t_1 \rangle$;
- *(l-ii.)* if some read r_1 returns $\langle v_1, t_1 \rangle$, then all reads that take effect after r_1 take effect and before any update takes effect will return $\langle v_1, t_1 \rangle$.

Condition *(l-i.)* holds because there is always at least one correct server in the quorum intersection between reads and updates (two sets of $n-f$ servers intersect in at least $f+1$ servers), and it suffices for reading self-verifying data [16]. Condition *(l-ii.)* holds because after a read that results in $\langle v,t \rangle$, there is always a set of $n-2f$ correct servers that have $\langle v,t \rangle$ and at least one of these servers will be on every possible quorum of the system. This happens due to a read finishing only if the same value v appears on $n-f$ servers or a write-back is used to disseminate this value to a quorum with this size.

To prove wait-freedom we must show that the three protocols always terminate as long as the requesting client is correct, at most f servers are faulty and, for rmw operations, the partial synchrony assumption holds. The CL-BFT always terminates in our system model, and the modifications introduced do not change that, so rmw operations always terminate. The write protocol always terminate because at most $n-f$ servers responses are wait in every one of its phases. The same holds for the read protocol.

3.3.2 Safety - Linearizability

In this section, we formally prove the linearizability of the objects implemented by AQS. Since rmws and writes take effect exactly in the same way (when $f+1$ correct servers store the new value, timestamp and certificate of the object), Lemmas 1-2 consider updates that put the object on a state $\langle v,t,C \rangle$.

Lemma 1 *If a write $w(v',t',C')$ by c directly follows an update $u(v,t,C)$, then $t' = succ(t,c)$.*

Proof: After u completes, at least $f+1$ correct servers will have $\langle v,t,C \rangle$. The first step of w will read the justified timestamp stored on $n-f = 2f+1$ servers, which intersect in at least one correct server with the $f+1$ servers that processed u. This server will reply $\langle TS,...,t \rangle_\sigma$ on the first step of the write and thus the timestamp for the write will be $t' = succ(t,c)$, which can be certified by $2f+1$ TS messages with t received on the first step of the write or the set of $2f+1$ PREPARED messages that will be sent in response to a justified PREPAREW message sent for t. Therefore, c will be able to write $\langle v',t',C' \rangle$ on the system. ■

Lemma 2 *If a rmw $m(v',t',C')$ directly follows an update $u(v,t,C)$, then m uses $\langle v,t,C \rangle$ as its base state.*

Proof: After u completes, at least $f+1$ correct servers will have $\langle v,t,C\rangle$. When m is started, all servers will send a STATE message to l, that waits for $n-f=2f+1$ messages from different servers, which contains at least one correct server that processed u and will return $\langle v,t,C\rangle$. Knowing that the base state of m will be defined by the primary l, we have to consider two cases:

1. if l *is correct*, it will choose the greatest justifiable timestamp received on STATE messages, which will be $\langle v,t,C\rangle$ and use it as the base state for m;

2. if l *is faulty*, it can do one of four things:

   (a) select one of the received tuples, but not the one with the greatest timestamp. l will try to send this base state together with C_l in the PRE-PREPARE message to the other servers. However, the C_l set must contain $n-f=2f+1$ STATE messages bounded to the rmw operation op from the different servers, and the base state proposed by the primary should on the one associated the greatest timestamp among these messages. Since at least $f+1$ correct processes have $\langle v,t,C\rangle$ as their state, it is impossible for the primary to build C_l without at least one STATE message containing $\langle v,t,C\rangle$. Consequently, the base state proposed by the primary will not be accepted by correct servers since it does not contain the greatest timestamp reported in C_l or C_l is not valid.

   (b) l creates a new value that was not reported by other servers in STATE messages. In this case the primary will not be able to assemble a valid C_l, and thus its PROPOSE message will not be accepted.

   (c) l sends PRE-PREPARE messages with different base states to the other servers. Some of these messages are justifiable by C_l while some not. In this case, the CL-BFT protocol will detect that the primary is proposing the same sequence number to different messages.

   (d) the primary does not propose anything at all.

   In any of these four cases the malicious primary will be suspected and another primary will be elected. This will happen until we have a correct primary that will get a justifiable base state following the view change (as described in modification 5 of Section 3.2.5).

■

Lemma 3 *If a read $r(v_r,t_r,C_r)$ directly follows an update $u(v,t,C)$, then $v_r=v$, $t_r=t$ and $C_r=C$.*

Proof: After u completes, at least $f+1$ correct servers will have $\langle v,t,C\rangle$. If some client executes a read r, it will send a READ message to all servers and will receive at least $n-f=2f+1$ READ-REPLY messages from different servers. Among these servers there is at least one correct server that processed u and replied $\langle v,t,C\rangle$, which corresponds to the last update executed on the system. Consequently, it will be the result of the read operation. ■

Lemma 4 *If a read $r_2(v_2,t_2,C_2)$ starts after a read $r_1(v_1,t_1,C_1)$ completes and no update operation starts before the start of r_1 and the completion of r_2, then $v_2=v_1$, $t_2=t_1$ and $C_2=C_1$.*

Proof: If r_1 completes, there will be at least $n-f = 2f+1$ servers reporting to have $\langle v,t,C\rangle$ as its current value. When a client starts r_2, it will send a READ message to all servers and will receive at least $n-f = 2f+1$ READ-REPLY from different servers. The intersection between two quorums of $n-f$ servers accessed on r_1 and r_2 must have at least $(2f+1)+(2f+1)-n = f+1$ servers, being at least one correct. This server will return $\langle v,t,C\rangle$ (read on r_1), which contains the greatest timestamp on the system, and thus will be chosen as the result of r_2. ■

Lemma 5 *If a rmw m starts after a read $r(v,t,C)$ completes and no update operation starts before the start of r and the completion of m, then m uses $\langle v,t,C\rangle$ as its base value.*

Proof: If r completes, there will be at least $n-f = 2f+1$ servers reporting to have $\langle v,t,C\rangle$ as its current value. It means that at least $f+1$ correct servers will have this value, and no other update will take effect before m completes. This situation is equivalent to the one stated on Lemma 2, and the proof is exactly the same. ■

The next two lemmas prove that no different states can be associated with the same timestamp on correct servers.

Lemma 6 *If optimized writes are not executed, it is impossible to build two valid update certificates for writing different object values with the same timestamp.*

Proof: Let w_1 be the write being executed with $\langle v_1,t_1,C_1\rangle$. Let w_2 be the write being executed with $\langle v_2,t_2,C_2\rangle$. Since both C_1 and C_2 are valid certificates, they contain $2f+1$ server messages. This means that at least one correct server s prepared both writes for the same timestamp. Without loss of generality, assume s approves w_1 before w_2. We have to show that if the two certificates are valid it is impossible to have $t_1 = t_2$.

We first have to consider the case in which the *same client c* tries to execute w_1 and w_2. If w_1 was approved before w_2, s stored $\langle v_1,t_1\rangle$ on $prepared_s[c]$, and thus it will never sign a message for an update certificate that is not greater than $prepared_s[c].t = t_1$. Which means that w_2 will only be approved if $t_2 > t_1$.

Now, we have to consider the case in which c_1 executes u_1 while w_2 executes w_2. In this case it is impossible to have two timestamps with the same value simply because the timestamps produced by a client must contain its id of on the timestamp lower bits. For any timestamp, s will only accept a PREPAREW or WRITE message if the timestamp is produced by the message sender, otherwise it is discarded. Which means that w_2 will only be approved if $t_2 \neq t_1$. ■

Lemma 7 *If optimized writes are not executed, for any two correct servers s_1 and s_2, it is impossible to update the state of the object to $\langle v_1,t_1\rangle$ on s_1 and $\langle v_2,t_2\rangle$ on s_2, with $v_1 \neq v_2$ and $t_1 = t_2$.*

Proof: Let C_1 and C_2 be the two update certificates used to update the state of s_1 and s_2, respectively. If C_1 or C_2 are not valid certificates, their update will not be accepted by s_1 or s_2, respectively. Now, assume that it is indeed possible to have two incomplete operations u_1 and u_2 leding to the updates on s_1 and s_2, respectively. We will prove that it is impossible analyzing three cases:

- u_1 and u_2 are writes. Lemma 6 states that it is impossible to build update certificates for the same timestamp and different values. In consequence, at least one of the servers will not accept the update because the certificate is invalid.

- u_1 and u_2 are rmws. Two rmws will never have the same timestamp with different values because CL-BFT ensures that all rmws will be executed one after another on correct servers. Moreover modification 4 ensures that all these executions will have different timestamps with the primary id on its lower bits.

- u_1 is a write and u_2 is a rmw (the same proof hold for the symmetric case). In this case, the timestamps simply cannot have the same value because the "writer" of a rmw is one of the servers (the primary), and thus t_2 will have the primary id on its lower bits. t_1, by the other hand, is produced by a client, with its id on the timestamp lower bits.

In all three cases, it is impossible that two correct servers accept object updates with different values and the same timestamp. ■

Notice that the two previous lemmas only work if the optimized write is not executed.

In order to prove linearizability we have to use the concept of history of a distributed computation [10]. A *history* $\mathscr{H}$ is a sequence of invocation and response events like $\langle c, req(o)\rangle$ (start of o) and $\langle c, rep(o)\rangle$ (completion of o), representing the request of an operation o issued by a client c and its reply, respectively. Notice that, for any operation o invoked by c, $\langle c, req(o)\rangle$ appear before $\langle c, rep(o)\rangle$ on the system history. We say that two operations o_1 and o_2 are *sequential* if the request event of o_2 appears after the reply event of o_1. Alternatively, o_1 and o_2 are said to be *concurrent* if they interleave, i.e., the request event of o_2 appears before the reply event of o_1.

The original definition of linearizability can be applied only on *complete histories*. A history $\mathscr{H}$ is *complete* if and only if $\langle c, req(o)\rangle \in \mathscr{H}$ then $\langle c, rep(o)\rangle \notin \mathscr{H}$. Otherwise $\mathscr{H}$ is said to be *incomplete*. For any history $\mathscr{H}$, $complete(\mathscr{H})$ denotes $\mathscr{H}$ without its unmatched requests.

The presence of faulty clients means that some operations will never complete, and thus we need additional theoretical tools to deal with incomplete histories. To be able to prove linearizability in the presence of Byzantine clients, we will use the notion of *Byzantine linearizability*, as defined in [8]:

Definition 1 (Byzantine Linearizability) *A history $\mathscr{H}$ comprised of events from all correct clients satisfies* Byzantine linearizability *if there exists some history $\hat{\mathscr{H}}$ such that*

1. *$\hat{\mathscr{H}}$ can be extended to some history $\mathscr{H}'$ such that $complete(\mathscr{H}')$ is equivalent to some legal sequential history $\mathscr{S}$;*

2. *For each response that precedes some invocation in $\mathscr{H}$, the response also precedes that invocation in $\mathscr{S}$, and*

3. *The subsequence of $\hat{\mathscr{H}}$ consisting of each event from every correct client is equal to $\mathscr{H}$.*

For this definition to be useful, one should be able to *complete* any history $\mathscr{H}$, which means inserting matching replies for incomplete operations or remove incomplete operations that do not affect $\mathscr{H}$. To do that preserving the original order of faulty-client invocations that affected the history (e.g., an incomplete write from which the written value was read by other correct clients), we need the notion of *invocation criteria* [8].

An invocation criteria defines the minimal conditions that must be satisfied by the operation protocol execution for the invocation to have any chance of effecting the system. Since reads do

not modify the object state, it is not necessary to define an invocation criteria for them and we can concentrate on updates. The invocation criteria for AQS writes and rmws are:

- *write*(v): at least one correct server must process a message $\langle \text{WRITE}, t_v, v, C_v \rangle$ and send an ACK to the client;
- *rmw*(op): at least one correct server received the $\langle \text{RMW}, op \rangle_{\sigma_c}$ sent by the faulty client c.

Notice that all correct client invocations satisfy the operations invocation criteria. With all these definitions we are ready to prove the linearizability of AQS objects through the next theorem.

Theorem 1 (AQS Linearizability) *An object implemented using AQS is linearizable.*

Proof: To prove Byzantine linearizability we need to show that any execution history in which correct and faulty clients communicate through the AQS object is linearizable.

For any finite history $\hat{\mathscr{H}}$, we first need to make it complete extending incomplete update operations that meet their invocation criteria. This is done using the following rule: the matching reply for an incomplete write or rmw is put just before the reply of the first read (resp. rmw) that returns (resp. use it as its base state) the written value, or in the end of the history if the update does not affected the history. For all purposes, the timestamp associated with the (artificially) completed update operations are the ones read (resp. used as base state) on their immediately following reads (resp. rmws) or, if the response event is included in the end of the history, a sequentially monotonic value greater than all complete operations on this history.

After that we have an incomplete history $\mathscr{H}'$, which can be made complete by removing all incomplete reads and updates that do not satisfy their invocation criteria (they did not affected the history). In consequence, we generate a complete history *complete*$(\mathscr{H}')$ of the AQS object that can be proved linearizable. This proof has three main steps [11]: first, we build a sequential history $\mathscr{S}'$ with all sequential operations of *complete*$(\mathscr{H}')$ preserving their original order. The second step is to order concurrent operations according to the properties of AQS and add them to $\mathscr{S}'$, generating the sequential history $\mathscr{S}$. Then we will show that $\mathscr{S}$ is in accordance with the sequential specification of an AQS object (see Section 3.1).

The first step is trivial: we just need to remove concurrent operations from *complete*$(\mathscr{H}')$, generating $\mathscr{S}'$.

For the second step, we first assign an unique timestamp to each operation of $\mathscr{S}'$. This is trivial since $\mathscr{S}'$ is sequential and all operations happen on an increasing timestamp order.

To conclude the second step, we need to insert the concurrent operations removed on step one in $\mathscr{S}'$ following their timestamp order. Recall that Lemma 7 ensures that the same timestamp will never be on two updates accepted by a server if optimized writes are not used. However, if optimized writes are used, there can be at most two writes with the same timestamp accepted by different correct servers (one of them prepared optimistically [14]), so we have to order these *"tie writes"*. The ordering of operations should be done in accordance with the following rules:

1. updates (writes or rmws) with different timestamps are ordered in accordance with the timestamps they write on the system;
2. writes with the same timestamp (as discussed before, due to the optimized write case), are ordered following the numerical value of the hash of the value they write;

3. reads are put just after the update that produced the timestamp and value they read;

4. reads with the same timestamp are put in any order.

If these rules are applied and concurrent operations are inserted on $\mathscr{S}'$, we will have a *sequential story* $\mathscr{S}$ with all operations of $complete(\mathscr{H}')$.

The last step is just to show that $\mathscr{S}$ follows the AQS object specification as described in Section 3.1: Lemma 1 proves that $\mathscr{S}$ satisfies #1 and #7; Lemma 2 proves it satisfies #4 and #8; Lemma 3 proves it satisfies #2 and #5; Lemma 4 proves it satisfies #3; and Lemma 5 proves it satisfies #6. With all conditions satisfied, we can say that the sequential history $\mathscr{S}$ is legal. Moreover, it is equivalent to the non-sequential history $complete(\mathscr{H}')$, which is linearizable. As a consequence, the AQS object is linearizable. ■

3.3.3 Liveness - Wait-freedom

The following theorems state that the three operations provided by AQS satisfy wait-freedom [10], i.e., that they always terminate in our system model (partial synchrony and at most f malicious *servers*) when invoked by a correct client.

Theorem 2 (Write Wait-freedom) *The write operation is wait-free.*

Proof: In the three phases of the write protocol the only part that the client can block in when it sends a message to all servers and waits for at least $n-f$ replies from different servers that contain valid certificates (when applicable). Since there are at least $n-f$ correct servers on the system, and all of them will reply a message to the client with a valid certificate (when applicable), the algorithm does not block and every write will terminate. ■

Theorem 3 (Read Wait-freedom) *The read operation is wait-free.*

Proof: In the two phases of the read protocol the only part that the client can block is when it sends a message to all servers and waits for at least $n-f$ replies from different servers (that sometimes should contain valid certificates). Since there are at least $n-f$ correct servers on the system and all of them will reply a message to the client with a valid certificate (when applicable), the algorithm does not block and every read will terminate. ■

Theorem 4 (Rmw Wait-freedom) *The rmw operation is wait-free.*

Proof: A correct client will send an RMW message to all servers and waits for the same result on a RMW-REPLY from $n-f$ servers. To show that these replies will eventually be received by the client we have first to show that the primary of CL-BFT will build a valid certificate C_l containing $n-f$ STATE messages from different servers.

When a server receives a RMW request it will get its state and send it to the current primary and start a timer that will be canceled only when the rmw operation of the client is committed. We have to consider two cases:

1. *the system is synchronous enough to make this message be executed by all servers if the primary is correct*: since the primary l will wait for $n-f$ STATE messages from different servers with justifiable pairs v,t, and since there are $n-f$ correct servers on the system, l

will eventually receive these messages and choose the most recent state justified by C_l to apply the rmw operation. We have to consider two cases:

(a) *l is correct*: in this case l will build a PRE-PREPARE message that will be accepted by all correct servers. Consequently the message will be prepared, committed and executed by all correct servers.

(b) *l is faulty*: in this case l can do anything and we have to show that any malicious behavior that does not follow the protocol will be detected and a new primary would be elected. This is exactly what was show in the case *l is faulty* of the proof of Lemma 2, and the same proof applies here: if the l does not follow the protocol it will be suspected and a new primary will be elected. Since there is at most f out of n faulty servers, eventually a correct server will be the primary, and we will resume to case 1-(a) (above).

2. *the system is not synchronous enough to make this message be executed by all servers even if the primary is correct*: in this case, for any behavior of the primary, there will be a timeout on this view and a new primary will be elected. The modification 5 defined in Section 3.2.5 plus the mechanism employed by CL-BFT ensure that the new primary will have the set of states and pending rmw to be used in the next view, without corrupting the rmw ordering. Timeouts and view changes can happen an unbounded number of times, while the system is not synchronous enough, but, given our partial synchrony assumption (Section 2.1), eventually the system become synchronous and then case 1 above can be applied for the new primary.

■

4 Discussion

This section presents several interesting issues of the protocol.

4.1 Non-deterministic Service Replication

The AQS replication algorithm exhibits another interesting and novel property that was not observed in previous BFT replication protocols: it can be used to replicate non-deterministic services. Read and write operations do not create determinism problems (i.e., non-deterministic object state transitions) since the first are read-only operations and the second set the resulting object state to the value defined by the client and all replicas set the same object state. Rmw operations, on the other hand, can lead the replicas to different states if the operation being executed triggers a non-deterministic state transition in the service. In the AQS rmw protocol, the system state is modified by the primary and the operation sequence number and its resulting new state and reply are transferred to other replicas. The other replicas accept this result if it corresponds to the operation execution on the previous state or if it satisfies some validity constraints. As long as these constraints can be expressed in a way that it is verifiable if some state transition made by the primary is legal, a non-deterministic service can be implemented.

To the best of our knowledge, this is the first time that a replication algorithm exhibits this property, being able to implement non-deterministic replicated services without requiring specific mechanisms [4].

4.2 Load Balancing

In AQS protocols, the only application processing executed at the server side are the rmw operations executed by CL-BFT primary replica (recall that it executes the operation and disseminates its result to other replicas that need only to verify if it is legal). If we are implementing a system with more than one object in which updates require a significant amount of processing it is possible to distributed this load to all replicas using different primaries for each set of objects. In this way, with no faulty servers, the system will execute n times more operations than an usual replication protocol when update processing costs are non-negligible.

4.3 Multi-object Operations

A multi-object operation comprises more than one single-object operations (possible on different objects) that are initiated by the same client at the same time. Despite sending the messages associated with all operations on the same request, there are two simple rules to deal with these operations: (1.) if some single-object operation is a rmw, then the whole multi-object operation is executed as a rmw; (2.) if some of the single-object operations are reads on objects that will be used to write on on (possibly different) objects, the multi-object operation must be executed as a rmw.

It is worth to notice that multi-object operations as defined in this section can only be done if all involved AQS objects (1) are deployed on the same set of servers and (2) use the same primary for processing their rmws. Condition (2) in particular makes it difficult to apply the load balancing capabilities as described in previous section. If some of these conditions are not satisfied, multi-object operations would only be possible if some kind of Byzantine locking [9] is used to lock the access on the objects while the operation is being executed.

Acknowledgments

Thanks to Miguel Correia for the useful discussions about how to prove the linearizability of AQS and the many comments that helped to improve this report.

References

[1] Michael Abd-El-Malek, Gregory Ganger, Garth Goodson, Michael Reiter, and Jay Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, October 2005.

[2] Gabriel Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing - PODC'84*, pages 154–162, August 1984.

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

[4] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269, August 2003.

[5] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. of 7th Symp. on Operating Systems Design and Implementation - OSDI 2006*, November 2006.

[6] Cyntia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–322, 1988.

[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] James Hendricks. *Efficient Byzantine Fault Tolerance for Scalable Storage and Services*. PhD thesis, Carnegie Mellow University, Computer Science Department, Pittsburgh, USA, July 2009.

[9] James Hendricks, Shafeeq Sinnamohideen, Gregory R. Ganger, and Michael K. Reiter. Zzyzx: Scalable fault tolerance through Byzantine locking. In *Proceedings of the 40th International Conference on Dependable Systems and Networks - DSN 2010*, June 2010.

[10] Maurice Herlihy. Wait-free synchronization. *ACM Trans. on Programing Languages and Systems*, 13(1):124–149, January 1991.

[11] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[12] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of the 21st ACM Symp. on Operating Systems Principles - SOSP'07*, October 2007.

[13] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[14] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of 26th IEEE Int. Conf. on Distributed Computing Systems - ICDCS 2006*, 2006.

[15] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.

[16] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.

[17] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Trans. on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[18] Jean-Philippe Martin, Lorenzo Alvisi, and Mike Dahlin. Small Byzantine quorum systems. In *Proceedings of the Dependable Systems and Networks - DSN 2002*, pages 374–388, June 2002.

[19] 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, Univ. of Lisbon, 2006.

[20] Fred B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[21] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. of the Symp. on Reliable Distributed Systems - SRDS'09*, September 2009.