On the Efficiency of Ordering Operations with Arbitrary Failures

Daniela Tulone

ISTI, CNR

tulone@iei.pi.cnr.it

Abstract

In this paper we analyze the complexity of the first protocol based on Byzantine Quorum System, for ordering operations on a generic replicated object, and propose two efficient versions of such a protocol: a deterministic and a randomized one. Their improved efficiency, in terms of communication and computational complexity, is obtained by saving one phase of the protocol and introducing hash functions. Notice that the technique of identifying a serializable object by its message digest can be employed in other settings in which a process needs to detect a correct up–to–date copy among the corrupted ones. We also address the problem of collecting out–of–date data, issue related to the asynchronous message system and augmented by the Quorum System approach, and propose a garbage collection integrated with the protocol.

1 Introduction

The wide spread use of Internet has led to an explosion of highly decentralized systems and brought to light issues such as scalability, system performance for very large and dynamic populations, fault–tolerance with respect to benign and arbitrary failures, and high availability of the service provided. Replication is a well–known technique to address both fault–tolerance and availability, and is employed in a number of real systems (i.e. the Swiss Stock Exchange, the FT–Corba and many Internet–based applications). A fundamental building block for systems based on replication, is the emulation of an atomic object with linearizable semantics [11] in which replicas behave as one single object exists and the sequence of the operations applied satisfies the real–time invocation order. Since malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior, the study of fault–tolerant mechanisms that guarantee the linearizability of the object replicated has drawn increasing attention in the last few years (i.e. Castro and Liskov [6], Chocker et al. [1], Chocker and Malkhi [9], Martin et al. [5]). As scalability and system performance for large and dynamic populations are gaining importance, and systems based on a state machine replication paradigm scale poorly to many processes, increasing interest is on scalable approaches resilient to arbitrary failures. Byzantine Quorum Systems, a variation of the Quorum Systems introduced by Malkhi and Reiter in [2], meets both reliability of the service provided and system performance in large universe. A Byzantine Quorum System is a set of subsets of servers (quorums) that guarantees the consistency of the data replicated and the availability of the service despite a limited number of arbitrary failures, i.e. less than one fourth of the universe size. Clients that access servers via Quorum Systems enhance the performance of the system for reducing the access cost per operation and for improving the system load balance and its scalability [3]. To our knowledge, the protocol proposed by Chocker et al. in [1] is the only solution based on Byzantine Quorum Systems that is able to emulate an atomic object with linearizable semantics. Other solutions proposed for ordering operations in presence of failures do not meet the efficiency shown by Quorum Systems in case of large populations as they are either based on state machine replication or access all servers (i.e. [6] [9]). Notice that in [1] the performance and scalability of the system is improved also by avoiding cryptographic constructions and using an efficient *contend and back–off* mutex protocol to guarantee mutual exclusion of the replicas in case of concurrent client requests. In contrast with other solutions (i.e. Castro and Liskov [6]) the work load of the protocol [1] lies on the client side, thus exploiting the assumption of *non-malicious* clients. Clients are responsible for applying operations, even issued by concurrent processes. This design improves the client response time and also allows *non-deterministic* operations. Therefore the Quorum System approach, its generality and its efficiency also in case of concurrent clients make this protocol suitable in several settings. For instance, an implementation of such a protocol is at the *core* of the Fleet System [4], a distributed data repository for persistent objects based on Byzantine Quorum Systems. Notice that the assumption of a limited number of *non-malicious* clients restricts the applicability of the protocol (i.e. it cannot be used in Internet–based applications with many infinite and possible malicious clients) but it does not diminishes its practicality. In fact, it can be employed in a number of settings with a known and finite set of trusted clients, possibly authorized via access control. For instance, it can be used to support information system infrastructures for large corporates such as cooperative work, e–banking, the provisioning of a service or in support of trading informations. An application based on this protocol [1] and whose prototype has been designed and implemented at Bell-Labs, is represented by an e–voting system whose access is restricted via an access control mechanism to authorized people (i.e. the executive board of a corporate) that are required to cast their vote on a given issue.

The applicability of the protocol proposed in [1], that we refer from now on as *the ordering protocol*, and its scalability has driven us to evaluate its performance. Experiments run using an implementation of this protocol on different object sizes and different number of servers and clients (not necessary concurrent), have shown a noticeable performance degrade of the response time when increasing one of these variables. These experiments have motivated us to analyze the computational and communication complexity of the ordering protocol to detect its performance bottleneck and investigate on alternative solutions. In this paper we first analyze its complexity and discuss its performance pitfalls, and then propose two efficient adaptations of [1]: one deterministic and the other randomized, slightly different from the first one. Their performance enhancement is achieved by saving one phase of the protocol and by introducing hash functions, thus reducing almost by half the amount of data transmitted and processed by clients. Notice that the technique of identifying an object with its message digest can be employed by other fault–tolerant protocols in which a process has to detect a correct object copy among the corrupted ones. This is particularly convenient for large objects or large populations in which the cost of comparing the object copies is high. Finally, we address the problem of collecting out–of–date data, issue related to the asynchronous message system and augmented by the Quorum System approach. We propose a garbage collection integrated with the protocol and that does not require additional communications.

This paper is organized as follows: Section 2 discusses related works. Section 3 provides an overview of the ordering protocol presented in [1], analyzes its complexity and discusses its performance pitfalls. Section 4 proposes a revised version of the ordering protocol based on hash functions, and analyzes its complexity. Section 5 describes a randomized version of the revised version presented in Section 4 and Section 6 sketches an implementation of a garbage collection. Section 7 concludes the paper and discusses future works.

2 Related works

As mentioned in the Introduction, the ordering protocol [1] differs from other solutions that emulate an object with linearizable semantics for its Quorum System approach, and therefore for improving the system performance with respect to systems based on state machine replication (i.e. access cost, system load balancing and scalability [3]). This improvement is more noticeable with the increase of the universe size. In addition, state machine replications do not support non-deterministic operations as servers are responsible for applying operations issued by the client. Phalanx [12], a predecessor of the Fleet system [4], is a data repository for persistent shared objects based on Byzantine Quorum Systems but provides weaker semantics than [1] as it supports only read and write variables.

Liskov and Castro has proposed in [6] a protocol based on a state machine paradigm, that guarantees the linearizability of the object replicated in presence of Byzantine failures. In contrast with other solutions, their protocol relies on weaker assumptions since it is resilient also to malicious clients and unreliable network links. On the other hand, it relies on an atomic broadcast and therefore scales poorly for large populations as it requires $O(n^2)$ messages with n number of servers. Other systems based on state machine replications and resilient to malicious failures are Rampart [7] and SecureRing [8]. They are based on group communication, a paradigm that is efficient during periods of stability, but that requires configuration when failures occurs and are detected. Both Rampart and SecureRing rely for their correctness on synchrony assumptions since they must exclude from the group replicas that have been detected as faulty. However, failure detectors cannot be accurate in an asynchronous system and may misclassify a replica as faulty.

More recently Chocker and Malkhi [9] has proposed a variant to the Disk Paxos protocol by Lamport and Gafni [10], to support many infinite clients. In Disk Paxos the protocol state is replicated at network attached disks, some of which may crash or becomes inaccessible. Chocker and Malkhi [9] utilize extended functionality and flexibility provided by the Active Disks and support unmediated concurrent data access by infinite many clients. Their work is not strictly comparable to [1] [6] since it does not consider Byzantine failures and it scales poorly for large universe as it accesses all servers.

More recently, Martin et al. [5] have proved lower bounds for the Byzantine storage required to implement safe, regular and atomic variables, with and without cryptographic tools. Their focus is to minimize the number of servers in order to reduce the correlation of software failures. They propose also two protocols that implement atomic and regular variables. Their work is not strictly comparable to [1] since the ordering protocol provides richer semantics. In addition, they achieve an optimal access cost but at cost of higher system load balance and additional communications in presence of concurrency.

3 An ordering protocol

3.1 System Model

The system model defined in [1] consists of a set of n servers and an unknown but finite number of clients uniquely identified by a integer. Processes that do not obey their protocol specifications are classified as *faulty*. Up to b servers with $b < \frac{n}{5}$ might exhibit arbitrary failures; clients can crash. Clients communicate with servers via message passing, using a special variant of a b–masking Quorum System [2] that guarantees at least $3b + 1$ servers in the intersection of any two quorums. Clients access only a quorum of servers and servers do not communicate among them.

3.2 Protocol data structures

The asynchronous message system, the benign client failures and Byzantine server failures, and the out–of–date servers (servers that have not been contacted recently) play an important role in the protocol and are responsible for the amount of data transmitted during the protocol and processed. This subsection describes in details the data structures employed by the ordering protocol as they have a *key role* in its communication and computational complexity.

Each operation o submitted by the client is represented by a data structure that is uniquely identified by a monotonically increasing integer created at its submission and consisting of the *client identification* and a *counter* for the operations issued by that client. We assume a well–formed sequence of method invocations produced by each client: it never initiates a new request if before it has received a response to its previously invoked request. Notice that since o contains the parameters associated to that request, its size is variable and might become very large. The most relevant data structure for the protocol is the *object state*, denoted by σ and identified (though not uniquely) by an integer, called *version number*, increased by one every time a process applies new operations. The *key fields* of the object state are the *replicated object* and the *history* of the operations performed on that object, necessary to prevent a client from applying operations more than once. Clients apply operations on the most recent object state, thus generating a new object state that contains the resultant object and the augmented history. Below is the description of a *minimal* object state structure satisfying the interfaces defined in [1]. The object state σ contains the following fields:

- $\sigma.v$: version number of σ;
- $\sigma.obj$: replicated object;
- $\sigma.res$: table mapping the identification of each operation applied at that stage to its response;
- $\sigma.h$: history of the operations performed so far on $\sigma.obj$. It is a table mapping each *participating client* (client that has issued at least one operation) to the identification of its last operation performed on that object.

Another variable relevant for the correctness of the protocol is the *rank*, integer that identifies uniquely each client execution. Clients run a *3-phase commit* protocol to guarantee correctness and liveness in case an execution remains uncompleted. To allow a client to complete a previous execution, each server maintains a copy of the object state sent by the client during the proposed phase, called *propose state* and denoted by σ^p, and a copy of the state sent during the commit phase, called *committed state* and denoted by σ^c. Notice that the content of the proposed and committed state generated by a client is identical. According to [1], each server S_i maintains the following local data structures:

- $rank_i$: maximum rank identifying the last client execution;
- $prop_i$: most recent client rank used in the proposed phase;
- σ_i^c, σ_i^p: local copy of the last proposed and committed state;
- ops_i: list of operations submitted to S_i by some client, and not committed to S_i.

Each client stores the version number of the last state that has been committed to a full quorum and that the client is aware of. This variable is called *completed*.

3.3 Protocol overview

As mentioned in the Introduction, since clients are not malicious most of the workload lies on the client side. Clients are responsible for detecting the most recent proposed and committed states and applying the list of *pending operations* (operations that has not been applied yet) to the last committed state; in case of uncompleted execution they have to complete it before applying new operations. Intuitively the protocol ensures the linearizability of the operations applied, by enforcing the order in which the object states are produced. This is achieved by means of a *contend and back–off* mutex protocol and of the *rank*. The client side of the protocol consists of two concurrent threads: the *submit thread* and the *ordering thread*. The submit thread is a *blocking thread* that simply submits an operation o to a quorum of servers for execution. Each server adds o to its list of pending operations, waits until it receives a committed state containing a response for o and then forwards the response to the client has issued o. On the client side, the submit thread waits until it receives $b+1$ identical responses. Notice that due to the Quorum System approach some server contacted by the *submit thread* might not be contacted in the commit phase by the *ordering thread*.

The *ordering thread* represents the *core* of the protocol; it runs a *contend and back–off mutex* protocol, called the *contend phase*, and a *3-phase commit* protocol. In the contend phase clients contend for *the right* of accessing the replicas for Δ time units, line 1:2. Each server returns tag $FREE$ if $\Delta + 2\delta$ time units have elapsed from the last time it granted the mutex, $LOCK$ otherwise (Figure 2 *contend*()). Clients that receive more than $b+1$ $LOCK$ tags *back–off* and wait for a random time. Once the client obtains the mutex, it computes a new rank, higher than the server ranks line 1:4, and runs a *3-phase commit* protocol with that identification. This rank allows servers to detect when due to network delay the time granted to a client has expired before its completion and another client is running the protocol with a higher rank, and to throw a *RankException* to the client with lower rank. Clients that receives more than b RankExceptions in response of a get, propose or commit request, abort their run and start over.

1) while $(true)$
2) $\quad\{ack_i, rank_i\}_{i\in Q} \leftarrow$ Q-RPC(Q, **contend**())
3) $\quad$if $|\{ack_i = LOCK\}_{i\in Q}| \leq b$
4) $\quad\quad r \leftarrow \max\{rank_i\}_{i\in Q} + 1$
5) $\quad\quad \{\sigma_i^c, \sigma_i^p, ops_i, prop_i\}_{i\in Q} \leftarrow$ Q-RPC(Q, **get**(r))
6) $\quad\quad (\sigma^c, \sigma^p, pending, completed) \leftarrow state(\{\sigma_i^p, \sigma_i^c, ops_i, prop_i\}_{i\in Q})$
7) $\quad\quad$if $(\sigma^c \neq \perp \wedge \sigma^c.v > completed)$
8) $\quad\quad\quad \sigma \leftarrow \sigma^c$
9) $\quad\quad$else if $(\sigma^p \neq \perp \wedge \sigma^p.v > completed)$
10) $\quad\quad\quad \sigma \leftarrow \sigma^p$
11) $\quad\quad$else
12) $\quad\quad\quad \sigma \leftarrow apply(pending, \sigma^c)$
13) $\quad\quad \{ack_i\}_{i\in Q} \leftarrow$ Q-RPC(Q, **propose**(r, σ))
14) $\quad\quad \{ack_i\}_{i\in Q} \leftarrow$ Q-RPC(Q, **commit**(r, σ))
15) $\quad\quad completed \leftarrow \max\{completed, \sigma.v\}$
16) $\quad$else
17) $\quad\quad$back–off
18) end while

Figure 1: The ordering thread.

contend():	**get(r):**	**propose**(r,σ)**:**	**commit**(r,σ)**:**
if $(clock() - lastGranted > \Delta + 2\delta)$ $\quad lastGranted \leftarrow clock()$ $\quad$return $\langle FREE, rank_i\rangle$ else $\quad$return $\langle LOCK, -\rangle$	if $(r \geq rank_i)$ $\quad rank_i \leftarrow r$ $\quad$return $\langle \sigma_i^c, \sigma_i^p, ops_i, prop_i\rangle$ else $\quad$throw $RankException$	if $(r \geq rank_i)$ $\quad rank_i \leftarrow r$ $\quad prop_i \leftarrow r$ $\quad \sigma_i^p \leftarrow \sigma$ else $\quad$throw $RankException$	if $r \geq rank_i$ $\quad rank_i \leftarrow r$ $\quad \sigma_i^c \leftarrow \sigma$ $\quad$awakeThread($\sigma.res$) else $\quad$throw $RankException$

Figure 2: Server side of the protocol.

Figure 1 and Figure 2 show the client and server side relative to the ordering thread; Q-RPC$(Q, p())$ denotes a remote invocation $p()$ performed on quorum Q. As mentioned above, the execution of the ordering thread consists of four phases: the *contend* phase and the *get*, *propose*, and *commit* phases. In the get phase the client holding the mutex, requests copies of the last committed and proposed object state and of the pending operations that reside at servers, line 1:5. To simplify the presentation let suppose that *state*() is the function computing the last committed state σ^c and proposed state σ^p *visible* to the client, the list of operations that have been submitted at some correct servers (i.e. in more than b servers), called *pending*, and the version number of the last state that was committed to a full quorum, called *completed*, line 1:6. The committed (or proposed) state computed in *state*() is the most recent state that has been committed (or proposed) to some correct servers (i.e. at least $b+1$ servers). The client chooses which state to propose and commit based on the values of the states σ^c and σ^p, as follows:

1. If the client has not seen the last committed state σ^c and $\sigma^c \neq \perp$, it proposes and commits it to ensure that it gets committed to a full quorum of servers, line 1:7-8.

2. If the client has not seen σ^p, it proposes and commits it to make sure that σ^p gets proposed and committed to a full quorum, line 1:9-10.

3. Otherwise it generates a new state by applying to σ^c the operations contained in *pending*, and it proposes and commits the new state to a quorum, line 1:11-12.

3.4 Analysis of the protocol

In this subsection we analyze the complexity of the ordering thread as this is the most significant part of the protocol and subject to enhancements. Notice that the ordering protocol is vulnerable of *denial-of-service* attacks since malicious servers can reply to clients sending arbitrary data. In this paper we do not address this issue, hard to prevent, but focus on the analysis of the communication and computational complexity of this protocol under the *silent action* of b corrupted servers that cooperate to fool the system. Our target is in fact to detect the performance bottlenecks intrinsic to the protocol.

In order to simplify the expression of the communication complexity we consider a generic quorum Q of size q, object states of size $|\sigma|$ and operations of size $|o|$. All integer variables are denoted by i. They appear in the expression of the communication complexity since in some case their contribution is not negligible. Under these conditions, the communication complexity of the ordering thread is as follows:

$$10i\ q + 4\ q\ (|\sigma.obj| + |\sigma.h| + |\sigma.res|) + \sum_{i \in Q} |ops_i| \tag{1}$$

Notice that the data exchanged between client and servers, in particular object states and operations, affects not only the communication but also the computational complexity. Lemma 1 shows how the computational cost of $state()$ depends on the number of comparisons performed and on the size of the copies. We analyze the computational complexity of $state()$ in the best and worst case since several elements might affect it (i.e. previous client crash, out–of–date servers contacted in the get phase or corrupted data sent by faulty servers in the quorum).

In order to evaluate the cost of *state*(), we need additional details on how [1] computes the object states and the list of pending operations. The committed state σ^c is defined as the state with highest version number contained in $\sum^c$, set of committed states with more than b identical copies. That is, $\sum^c = \{\sigma' : |\{i : \sigma' = \sigma_i^c\}_{i\in Q}| \geq b+1\}$. To reduce the number of comparisons needed to compute σ^c, let assume that the client computes an auxiliary hash table mapping each different version number v of the committed states sent by the servers, to a list of tuples $\langle l, \sigma\rangle$ where σ is a committed state with version number v that resides at l different servers. After computing this auxiliary table, the client orders the version numbers and returns the copy with highest version number that resides at more than b different servers. The proposed state is computed similarly (see [1]). The list of pending operations is defined in [1] as the set of operations with more than b identical copies. In order to reduce the number of comparisons necessary to compute it, let assume that the client computes an auxiliary hash table mapping each different identification number *id* of the operations received, to a set of a tuples $\langle l, o\rangle$ with o operation with identification number *id* that resides at l different servers. After computing this table, the client inserts the operations that have been sent by more than b different servers, that is $\langle l, o\rangle$ with $l > b$. In this way the client avoids comparisons among possible operations injected by faulty servers. Notice that *pending* might contain out–of–date operations (operations that have been already committed to a full quorum) since more than b servers in a quorum could be faulty or have not received the committed state. In fact, in [1] the client verifies just before executing an operation if it has been already applied, line 1:12. Let denote by $\#ops_i$ the number of operations send by server S_i. Then, the following lemma computes the computational cost of *state*() in the best and worst case.

Lemma 1 *$state() = \Omega\Big(q|\sigma| + \sum_{i\in Q} \#ops_i + b\sum_{o\in\mathcal{O}} |o|\Big)$ and $state() = O\Big(b\, q|\sigma| + \sum_{i\in Q} \#ops_i + b\sum_{o\in\mathcal{O}\cup\mathcal{P}} |o|\Big)$ with $\mathcal{O}$ set of operations to apply and $\mathcal{P}$ set of out–of–date operations sent by at least $b+1$ different servers.*

Proof The cost of computing the last committed state depends mainly on the cost of computing its auxiliary hash table, and therefore on the number of comparisons performed among the copies. In the best case all servers in the quorum are up–to–date and return the last committed state, therefore the client performs q comparisons. In the worst case $O(bq)$ comparisons are needed (i.e. when b faulty copies with the same version number are inserted at the beginning of the list of tuples $\langle l, o\rangle$). The cost of computing the last proposed state is similar. The cost of computing *pending* is given by the cost of computing the auxiliary hash table, that is equal to $\sum_{i \in Q} \#ops_i$, plus the cost of detecting $b+1$ identical copies. In the best case the client performs b comparisons for each pending operation; in the worst case it performs $2b$ comparisons for each the pending operation plus x comparisons with $b \leq x \leq 2b$ for each out–of–date operation sent by at least $b+1$ different servers. □

3.5 Considerations

The analysis of the ordering protocol presented in Section 3.4 justifies the performance degrade observed with many clients and large objects. The expression (1) of the communication complexity and Lemma 1 show two main sources of performance bottleneck.

1. The data contained in the object state can represent a performance bottleneck for the system since it is transmitted four times during the protocol and its content is processed in $state()$. The fields of the object state that mostly impact the complexity are $\sigma.obj$ and $\sigma.h$, table that grows with the number of participating clients. As mentioned before, the proposed and the committed state share the same content; this redundancy will be removed in the revised version presented in Section 4.

2. The out–of–date information, such as operations that reside at some servers but that have been committed to a full quorum, can also affect the system performance. Notice that the number of operations sent by the servers might be unbounded if a server is always contacted by the submit thread but not in the commit phase. This affects also the cost of $state()$ since it performs comparisons also on operations that have been already committed (see Lemma 1). Experiments using a randomized epidemic protocol to diffuse updates among the servers [13] to overcome this problem as suggested in [1], do not show a performance improvement. The performance gets worse not only in case of high request rate, but also when a single client invokes sequential operations, one after the other. This is most likely related to the fact that each server applies an update only after receiving $b+1$ identical copies in order to mask arbitrary updates sent by faulty servers. The revised protocol presented in Section 4 solves the problem of collecting out–of–date information without additional computations, but partially (though it can be sufficient in most applications). On the contrary the garbage collection sketched in Section 6 removes out–of–date operations but at an additional computational cost, therefore it is more efficient only in

some case. Another *growing* data structure transmitted in the object state is the history of operations applied that prevents clients from applying an operation more than once. Of course, an operation contained in the history can be removed if all the correct servers have received that request and have acknowledged that it was committed, but this condition is hard to establish due to the asynchronous message system and the Quorum System approach. In fact, $b+1$ identical copies of an operation that was already committed can induce a client to apply it more than once if the correspondent entry in the history has been removed.

4 An efficient version of the protocol

In this section we propose a version of the ordering protocol whose improved performance is obtained by introducing hash functions and saving one phase of the protocol. The use of hash functions allows clients to identify an object copy by means of a fixed–length string, the *message digest*, thus reducing the amount of data transmitted and processed by the client. The enhancements of this revised version can be summarized as follows:

1. it combines the *contend*() and *get*() phases, thus saving one phase of the protocol;
2. it reduces the data contained in σ^p;
3. it bounds the number of operations stored at servers;
4. it reduces the number of comparisons performed in *state*().

The protocol addresses point 3) by storing operations submitted at server S_i in an hash table OPS_i mapping each client identification to the last operation submitted by that client but not committed at S_i. Each server updates this table upon receiving the committed state. Notice that OPS_i works as a garbage collection since it contains at most one submitted operation per client, and discards (overwrites) operations issued by a client with lower identification number. This variation does not affect the liveness of the protocol since client invocations form a well–formed sequence, and it limits the number of operations sent in the get phase. To simplify the notation, ops_i continues to denote the list of operations sent by server S_i, though here it is derived by OPS_i. The following section illustrates in details this revised version of protocol and describes how points 1) and 2) are addressed; Section 4.2 discusses its improvement in terms of communication and computational complexity.

4.1 The protocol

In the ordering protocol the proposed and committed state generated by the client are identical to allow the system to recover from an inconsistent state due to client crash or network delay. In fact, the data contained in the proposed state has to allow a client to complete the previous run even in case it is able to detect the last proposed state σ^p but cannot find $b+1$ identical copies of the correspondent committed state, line 1:9. We break the symmetry between the committed and proposed state and solve this issue by allowing clients to *compute* the committed state correspondent to σ^p based on the data contained in the proposed state. In particular this is done by applying the operations executed prior to crash, to the committed state closest to σ^p. This is valid since as we show later, for each proposed state σ^p there is one and only one committed state that immediately precedes it, denoted by $\bar{\sigma}^c$. The committed state correspondent to σ^p is computed in two steps: 1) the client detects a correct copy of its *closest committed state* $\bar{\sigma}^c$ by means of its message digest that is contained in σ^p, and 2) applies to $\bar{\sigma}^c$ the list of operations executed by the previous client and contained in σ^p. Therefore, the revised proposed state contains in addition to its version number, the operations performed at that stage and an identification of *its previous committed state*. That is, $\sigma^p := \{\sigma^p.v,\ \sigma^p.ops,\ \sigma^p.m\}$ with $\sigma^p.m$ message digest of $\bar{\sigma^c}$. In particular, $\sigma^p.m$ is defined as $\sigma^p.m = hash(\bar{\sigma^c}.v + \bar{\sigma^c}.obj + \bar{\sigma^c}.h)$ with $hash$ unkey hash function (i.e. MD4, SHA-1). Notice that $\sigma^p.m$ identifies uniquely the committed state previous to σ^p since the operations applied are recorded in $\sigma^c.h$.

Figures 3 illustrates the revised ordering thread; the line numbers in bold (lines 2, 4, 7-11 and 13) indicate the variations with respect to Figure 1. The first variation in the ordering thread is given by replacing the contend and get phase with the *contend-get* phase, line 3:2. As shown in Figure 4, the server side of *contend-get()* combines functions *contend*() and *get*() of Figure 2 except for *rank* that is incremented by one every time the server grants the mutex, to guarantee the monotonicity of the client rank. Upon receiving a *successful* response for the contend-get request, the client computes the largest $b+1$ rank instead of the

1) while $(true)$
2) $\{ack_i, rank_i, \sigma_i^p, \sigma_i^c, ops_i, prop_i\}_{i\in Q} \leftarrow$ Q-RPC(Q, **content-get**())
3) if $|\{ack_i = LOCK\}_{i\in Q}| \leq b$
4) $r \leftarrow \max_{b+1}\{rank_i\}_{i\in Q}$
5) $\langle \sigma^c, \sigma^p, pending, completed \rangle \leftarrow state2(\{\sigma_i^p, \sigma_i^c, ops_i, prop_i\}_{i\in Q})$
6) if $(\sigma^c \neq \perp \wedge \sigma^c.v > completed)$
7) $\{ack_i\}_{i\in Q} \leftarrow$ Q-RPC(Q,**commit**(r, σ^c))
8) break
9) else if $(\sigma^p \neq \perp \wedge (\sigma^p.v > completed) \wedge ((\sigma^c = \perp) \vee (\sigma^c.v \neq \sigma^p.v)))$
10) $\bar{\sigma^c} \leftarrow prevComState(\{\sigma_o^c\}_{i\in Q})$
11) $\sigma^c \leftarrow apply(\sigma^p.ops, \bar{\sigma^c})$
12) else
13) $\sigma^p \leftarrow \{\sigma^c.v + 1,\ h(\sigma^c),\ pending\}$
14) $\sigma^c \leftarrow apply(pending, \sigma^c)$
15) $\{ack_i\}_{i\in Q} \leftarrow$ Q-RPC(Q, **propose**(r, σ^p))
16) $\{ack_i\}_{i\in Q} \leftarrow$ Q-RPC(Q, **commit**(r, σ^c))
17) $completed \leftarrow \max\{completed, \sigma^c.v\}$
18) else
19) back–off
20) end while

Figure 3: The revised ordering thread.

contend-get():

if $(clock() - lastGranted > \Delta + 2\delta)$
$lastGranted \leftarrow clock()$
$rank \leftarrow rank_i + 1$
return $\langle FREE, rank_i, \sigma_i^c, \sigma_i^p, ops_i, prop_i \rangle$
else
return $\langle LOCKED \rangle$

Figure 4: Server side of contend–get.

highest value to safeguard clients from server attacks that might degrade the system performance or lead to buffer overflow, line 3:4. Notice that this does not violate the monotonicity of the client rank because of the mutex protocol and the intersection property of quorums (Section 3.1). As Figure 3 shows, the client side of the protocol differs from the previous version also for the *actions* taken by the client based on the committed and proposed states, σ^c and σ^p computed in $state2()$. Line 3:6 occurs when either the commit phase was not completed due to crash or network delay, or the client has no knowledge of the last committed state. In both cases the client ensures that σ^c remains committed to a full quorum, line 3:7-8. If the client has not seen σ^p, proposes it and commits the correspondent σ^c to a full quorum. In case it computes the last proposed state but is not able to detect $b+1$ identical copies of the correspondent σ^c, line 3:9, it invokes $prevComState()$ that returns a correct copy of the closest commit state previous to σ^p, line 3:10. Notice that $prevComState()$ verifies the correctness $\bar{\sigma^c}$ by computing the message digest of $\bar{\sigma^c}$ and comparing it with $\sigma^p.m$. Lastly, if the previous run has been completed, the client generates the new proposed state and applies *pending* to σ^c to generate a new committed state, lines 3:13-14.

4.2 Protocol correctness

We have to show that the revised protocol induces a well–defined sequence of the operations applied, compatible with the real time invocation order, and that the result of each operation is consistent with that sequence. The following lemmas prove the correctness of the revised ordering thread in Figure 3 by showing that the invariants used in [1] to prove its correctness still hold. In particular, Lemma 2 proves an invariant crucial to show for each proposed state the uniqueness of its closest committed state in the causal order of the method invocations. The linearizability of the operations applied relies on this property, since clients apply operations to the last committed state.

Lemma 2 *Let r_1 and r_2 be ranks used in two different propose invocations propose(r_1, σ_1) and propose(r_2, σ_2), and let suppose propose(r_1, σ_1) be completed (receives no more that b RankException). If $r_1 < r_2$, then propose(r_1, σ_1) precedes in a causal order propose(r_2, σ_2). Otherwise, the successful contend-get prior to propose(r_2, σ_2) (denoted by contend-get$_2$) precedes propose(r_1, σ_1).*

Proof Let suppose the contrary, that *propose*(r_1, σ_1) does not precede *propose*(r_2, σ_2) for $r_1 < r_2$. *propose*(r_2, σ_2) cannot precede *propose*(r_1, σ_1) because otherwise *propose*(r_1, σ_1) would not be completed since $r_1 < r_2$. Now, let suppose that *propose*(r_1, σ_1) and *propose*(r_2, σ_2) are concurrent according to the causal order. In this case, fewer than b servers has higher rank at the time they receive the *propose*(r_1, σ_1) invocation otherwise it would not complete. Therefore, the successful contend-get invocation prior to *propose*(r_2, σ_2) denoted by contend-get$_2$, returns at least $b + 1$ copies of the most recent proposed state, leading to a contradiction. If $r_2 < r_1$ then contend-get$_2$ precedes *propose*(r_1, σ_1) since it precedes get-contend$_1$ otherwise *propose*(r_1, σ_1) would not be successful. □

Lemma 3 is the main lemma and states the correctness of the revised protocol.

Lemma 3 *There is a well–defined sequence in which submitted operations are applied and the result of each operation is consistent with that sequence.*

Proof (**Sketch**) We show that the variations introduced in the ordering protocol by combining the *contend*() and *get*() phases and by redefining σ^p does not affect the correctness of the protocol proved in [1].

The fact of replacing *contend*() and *get*() with contend-get() does not prevent the client from computing the object state, even in case of b RankException, since there are at least $2b+1$ correct servers in the quorum intersection and the monotonicity of the client rank computed at line 3:4, is ensured by the mutex protocol and by the intersection property of the quorums. Therefore the linearizability of the operations applied follows from Lemma 2 and from the correctness proof in [1]. We show that the re-definition of the proposed state does not affect safety in case of client failures. Line 3:6 occurs either if the previous commit phase was interrupted (but the object was proposed to a full quorum) or if the client is out–of–date, therefore it is safe for the client to commit σ^c to a full quorum set skipping the proposed phase. If the proposed or the commit phase was not completed and the client is unable to compute the last committed state, line 3:9, it can compute a correct copy of the closest committed state prior to σ^p based on the information encapsulated in the proposed state. It is always able to find a correct copy of the previous committed state because otherwise σ^c would be not empty. Therefore, it computes σ^c from $\bar{\sigma^c}$ since at any step, the client applies operations to the last committed state. □

4.3 Analysis of the protocol

This adaptation of the protocol is clearly more efficient than the original version for saving one phase of the protocol and addressing the performance bottlenecks 1) and 2) pointed out in Section 3.5. In particular,

1. it removes the data redundancy present in the object states and reduces almost by half the data transmitted;
2. it discards *some* of the out–of–date operations that reside at servers, thus limiting the number of operations sent by the servers to the number of participating clients.

Therefore, the simplified expression of its communication complexity is as follows:

$$8i\ q + 2\ q\ (|\sigma^c.obj| + |\sigma^c.h| + |\sigma^c.res| + |\sigma^p.ops| + |m|) + \sum_{i \in Q} |ops_i|$$

with m message digest. Notice that the history of the operations $\sigma^c.h$ and the replicated object $\sigma^c.obj$, are transmitted only twice instead of four times. The performance enhancement is particularly evident in these cases: 1) large objects (i.e. audio or video files), 2) operations involving large parameters or high rate of client requests, 3) large universe size and 4) many clients. Notice that since the proposed state contains the list of the operations applied at that stage, this might impact the performance in case of operations involving very large parameters or high degree of concurrency. Therefore, since clients require $\sigma^p.ops$ only in a particular case of client crash, servers can return a *reduced proposed state* with $\sigma^p.ops$ containing only their identification numbers. In case the client needs to compute the last committed state, line 3:11, it sends another contend-get request with a flag that induces each server to extend the time granted to that client to cover the additional round trip, and to return $\langle \sigma_i^c, \sigma_i^p, ops_i, prop_i \rangle$ where σ_i^p is the *complete proposed state*.

As mentioned at the beginning of this section, this revised version improves not only the communication complexity but also its computational cost for reducing the proposed state to a message digest and some identification numbers, and for decreasing the number of comparisons performed in $state2()$ by means of an inexpensive pre-parsing. For instance, to compute the last committed state the client does not compute $\sum^c$ but arranges the copies in a hash table according to their version number (without comparing it), and starting from the highest version number it compares the copies of the committed states sent by more than b different servers. It stops as soon as it finds $b+1$ identical copies. Notice that in this way in the best case the client performs only b comparisons on the committed states and in the worst case $O(b^2)$. Similarly for the proposed state. In contrast with $state()$, the optimized version $state2()$ does not compare operations that have been already applied, whose number can be unbounded (see Lemma 1). It discards them by means of the history contained in the committed state σ^c computed previously by the client. Of course $state2()$ can be further improved but these enhancements are more technical and beyond our scope. Lemma 4 computes the computational cost of $state2()$ in the best and worst case and assumes the committed state greater than the proposed state as in most cases.

Lemma 4 $state2() = \Omega(q + b(|\sigma^c|) + \sum_{i \in Q} \#ops_i + b \sum_{o \in \mathcal{O}} |o|)$ *and* $state2() = O(q + b^2(|\sigma^c|) + \sum_{i \in Q} \#ops_i + b \sum_{o \in \mathcal{O}} |o|)$.

5 A randomized approach

In this section we discuss a randomized version of the protocol presented in Section 4. The technique employed in Section 4 that identifies the previous committed state with its message digest contained in the proposed state, can be generalized to any large object. Intuitively, since clients require only one correct copy of the object, it is not necessary that all correct servers return a *complete* copy, some of them might send a *reduced* version that identifies uniquely the content of the complete copy. The randomized version presented in this section is based on this intuition. It differs from the version in Section 4 in the following aspects: 1) for including in the committed state an additional field, called $dgst$, containing its message digest; 2) in the server side of *contend-get()* of Figure 4, and 3) in the way the client compares copies of the committed state. Upon receiving the get request, each server decides, based on the size of the committed state, whether return the *complete copy* along with the other data, or employ a randomized approach. In the probabilistic approach it sends the complete state with probability $p \leq \frac{1}{2}$ and a *reduced copy* consisting of its message digest $dgst$ and its version number, with probability $1-p$. Notice that this version is identical to the protocol proposed in Section 4 for small object states and it is randomized for large object states; we consider here the case of large committed states. In contrast with the previous version, the comparisons performed by the client on the committed states are based on their message digests. Therefore, the client terminates the comparisons as soon as it finds $b+1$ identical message digests of the most recent state. Let denote this string by msg. Of course, in order to compute the last committed state (the complete copy) it has to verify its correctness since faulty servers might send a copy containing a correct message digest but wrong data. This is done by computing the hash function of the complete copy and comparing it with msg, the message digest previously detected. Since the client stops as soon as it finds one correct copy, it performs in the worst case $b+1$ computations of the hash function. Notice that since servers send the complete copy with probability p, the client might not receive a complete up–to–date committed state and therefore might not be able to compute it. In this case, it sends again a *contend-get()* request with a special flag that induces the servers to *extend* the time granted to that client, to cover the additional round trip delay, and to return the complete committed state with probability one. If the previous run was completed (no client crash or delays) the client succeeds in computing the committed state at first place with probability at least $p(b+1)$, otherwise it succeeds with probability at least p.

It is evident that in the normal case this version improves the communication complexity with respect to the protocol in Section 4. In particular, the communication complexity is enhanced at least by a factor of $(1-p)(q-b)|\sigma^c|$. Therefore, the enhancement is more noticeable as the quorum size or the object size increases.

The analysis of the computational complexity is less straightforward. In fact, though the comparisons performed by the client are cheap since based on message digests, the computations of the hash function are more expensive than simple comparisons (as also the experiments confirm). The computational complexity is improved with respect to Section 4 if the cost of computing the hash functions is less than the cost of comparing the complete copies. Therefore, we need to bound the expected number of the hash computations performed by the client in order to compute the last committed state. The following lemma computes an upper bound (not tight) of the expected number of hash computations.

Lemma 5 *Let g be the number of correct up–to–date committed copies that are complete. Then, the expected number of computations of the hash function is less than $2 + \frac{b}{g}$.*

Proof Let S be the set of the committed copies (both completed and reduced) consisting of b corrupted copies and g correct ones. Let X be a random variable representing the number of (unsuccessful) computations performed on corrupted copies, and $c_1, \ldots, c_b$ the order in which the client computes the hash function of the faulty copies. Let X_i be a random variable whose value is equal to one if the client computes the hash function of $c_1, \ldots, c_i$ before any correct copy. Therefore $X = \sum_{i=1}^{b} X_i$. Since

$$Prob(X_i) = \prod_{j=0}^{i-1} \frac{b-j}{b-j+g} < \left(\frac{b}{b+g}\right)^{i-1}$$

Then

$$E[X] = \sum_{i=1}^{b} E[X_i] < \sum_{j=0}^{b-1} \left(\frac{b}{b+g}\right)^{j} < 1 + \frac{b}{g}$$

□

Notice that the expected number of g is equal to $p \cdot m$ with m number of the up–to–date correct copies contained in $\{\sigma_i^c\}_{i \in Q}$. If the previous run was completed, then $b+1 \leq m \leq q$. Therefore if $p = \frac{1}{2}$ and $m = 2b+1$ then on the average two computations of the hash function are enough to detect a correct up–to–date committed state.

As mentioned before, this technique can be generalized to any large object and employed in other fault–tolerant protocols in which a process needs to distinguish a correct copy among corrupt ones. Each servers decides at run–time, based on the object copy, whether adopt a randomized approach or not. This reduces the communication complexity and, in presence of large objects and large universe, also the computational complexity.

6 Implementing a garbage collection

If case of many clients and operations with large parameters, it might not be efficient enough to limit the number of operations that reside at servers to the number of clients, as proposed in Section 4. This scenario can occur for instance, if clients attempt to use the same quorum set, to minimize the number of out–of–date servers contained in the quorum, and in case of concurrency a client with many sequential tasks obtains the mutex and applies operations also on behalf of other concurrent processes. Experiments shows that this is likely to occur because of the back–off approach of the mutex protocol.

This section sketches a garbage collection whose task is to remove all the out–of–date operations stored at servers before they return the list of operations to apply. This is achieved by using a *4-phase protocol* similar to [1], in which the *contend* and *get* phases are decoupled. In the contend phase, each server granting the mutex returns *its approximate view* of the operations applied so far, a table similar to the history contained in the committed state and that reflects the knowledge of the operations applied. The client processes the *server views* and computes its approximate up–to–date view, and attach it to its get request to allow the out–of–date servers to update their local table OPS_i. Notice that another data structure should keep track of the clients waiting for a response. In contrast with the version in Section 4 the committed state does not contain the history $\sigma^c.h$, each server maintains a local copy of the history and updates it during the commit phase. The history maintained at server S_i is called $history_i$ and is an hash table containing the tuples $\langle id_c, id_{c,o}\rangle$ where $id_{c,o}$ is the identification of the last operation issued by client c with identification id_c, and that was committed at server S_i.

Figure 5 illustrates the differences between this version and the protocol in Figure 3, differences that are related to the *contend* and the *get* phases. In the contend phase each server granting the mutex returns along with tag $FREE$ and its rank a copy of its local history, line 5:1. In case of successful contention, the client computes a table from set $\{history_i\}_{i\in Q}$ called $View$, that represents the up–to–date view of the operations

1) $\{ack_i, rank_i, history_i\}_{i\in Q} \leftarrow$ Q-RPC(Q,**content**(id))
2) if $|\{ack_i = LOCKED\}_{i\in Q}| \leq b$
3) $\quad r \leftarrow \max_{b+1}\{rank_i\}_{i\in Q}$
4) $\quad H \leftarrow \{history_i\}_{i\in Q}$
5) $\quad View \leftarrow \{\langle c, op\rangle : op = \max_{b+1}\{id : \langle cl, id\rangle \in H\}\}$
6) $\quad \{\sigma^c, \sigma^p, ops_i, prop_i\}_{i\in Q} \leftarrow$ Q-RPC(Q,**get**($r, View$))

Figure 5: Changes in the ordering thread.

get(r, View):

1) if $r > rank_i$
2) $\quad rank_i \leftarrow r$
3) $\quad$ for each $\langle cl, op\rangle \in View$ do
4) $\quad\quad OPS_i \leftarrow \{\langle cl, o\rangle : o \leq op\}$
5) $\quad\quad ops_i \leftarrow Image(\overline{OPS_i})$
6) $\quad\quad$ return $\langle \sigma_i^c, \sigma_i^p, ops_i, prop_i\rangle$
7) else
8) $\quad$ throw $RankException$

Figure 6: Server side of revised get.

committed at *some correct server*, line 5:5. Let $\mathcal{C}$ be the set of clients, then $View$ is defined as follows:

$$View = \left\{ \langle id_c, \overline{id_{c,o}}\rangle \mid c \in \mathcal{C} \wedge \overline{id_{c,o}} = \max_{b+1}\{id_{c,o} \ : \ \exists i \in Q : \langle id_c, id_{c,o}\rangle \in history_i\} \right\}$$

We show that if $\langle id_c, id_{c,o}\rangle \in View$ then operation o has been committed at *some correct server*. Table $View$ is sent along the get request; each server updates in get() its table OPS_i by keeping only operations whose identification is higher than the identification contained in $View$, line 6:3-4.

Lemma 6 *Each operation submitted by a client is applied only once and is eventually executed.*

Proof (**Sketch**) We show that for any $\langle id_c, id_{c,o}\rangle \in View$ operation o has been committed at least at one correct server. By contradiction let suppose that there exists a tuple $\langle id_c, id_{c,o}\rangle \in View$ such that o issued by client c has not committed to any correct server. Therefore, by the definition of $View$ there are b servers that have received a committed state containing a response for an operation issued by client c with identification number equal or higher than $id_{c,o}$. Let consider the case of higher identification number. Since there are at most b faulty servers in the universe, at least one correct server has received in the commit phase a response for an operation issued by client c after receiving a response for operation o. Since the client invocations are well-formed and a client returns a response after receiving $b+1$ identical responses, o has been committed to at least one correct server leading this to a contradiction. We show that each operation is applied exactly once. Let o be an operation that was applied and committed at some server. If the commit phase was interrupted, in the next run the client $View$ might or might not reflect this. In both cases this does not compromise safety because a client has to complete the previous run in order to apply new operations (the object state was proposed but not committed to a full quorum). Also, the fact that upon receiving the get request each server might remove o from its list of pending operations, do not compromise safety or liveness since in case of incomplete run the list of pending operations computed in $state()$ is meaningless. Once the commit phase is completed, the table $View$ computed by the next client will contain that operation since at least $b+1$ correct servers will have received the committed state. □

Notice that since the client performs an additional task for computing $View$, this version of the protocol is more efficient than the optimized version presented in Section 4 only in case of operations with large parameters. In fact, the cost of computing $View$ is $O(q \cdot C)$ with C number of clients since it involves for each client the computation of the $b+1$ largest integer among at most q integers.

7 Conclusions

We have analyzed the first protocol based on Byzantine Quorum Systems for ordering operations on a generic object and proposed two efficient adaptations, as well as a garbage collection. Notice that the techniques employed in this paper can be extended to other fault–tolerant protocols to improve their performance. As the ordering protocol [1] is not applicable for large dynamic populations, future works include the study of an efficient mechanism for large scale systems able to support an unbounded number of clients also malicious.

References

[1] G.Chockler, D.Malkhi, M.Reiter *Back–off protocols for distributed mutual exclusion and ordering.* In Proceedings of the 21st International Conference on Distributed Systems, pp. 11-20, April 2001.

[2] D.Malkhi, M.Reiter *Byzantine Quorum Systems.* Distributed Computing 11(4):203-213, 1998.

[3] D.Malkhi, M.Reiter, and A.Wool *The load and availability of Byzantine Quorum Systems.* SIAM J.Comput. 29(6):1889-1906 (2000).

[4] D. Malkhi, M. K. Reiter, D. Tulone, E. Ziskind. *Persistent objects in the Fleet system.* In Proceedings of the 2nd Darpa Information Survivability Conference and Exposition (DISCEX II), June 2001.

[5] J.P. Martin, L.Alvisi, M. Dahlin. *Minimal Byzantine Storage.* In Proceedings of the 16th International Symposium on Distributed Computing (DISC 2002),pp. 311-326.

[6] M.Castro, B.Liskov. *Practical Byzantine Fault Tolerance.* Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI 1999), pp.173-186.

[7] M.K.Reiter. *Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart.* In Proceedings of the 2nd ACM Conference on Computer and Communication Security, pp. 68-80.

[8] K.P.Kihlstrom, L.E.Moser, P.M.Melliar-Smith. *The SecureRing Protocols for Securing Group Communication.* In Proceedings of the IEEE 31st Hawaii International Conference on System Sciences, (3):317-326, January 1998.

[9] G.Chockler, D.Malkhi. *Active disk paxos with infinitely many processes.* In Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC 2002).

[10] E.Gafni, L.Lamport *Disk Paxos.* In Proceedings of 14th International Symposium on Distributed Computing (DISC '00), pp 330-344.

[11] M.P.Herley, J.M.Wing *Linearizability: A correctness condition for concurrent objects* ACM Transactions on Programming Languages and Systems 12(3):463-492, July 90.

[12] D.Malkhi, M.Reiter *Secure and Scalable Replication in Phalanx.* In IEEE Symposium on Reliable Distributed Systems, 1998.

[13] D.Malkhi, Y.Mansour, M.Reiter *On diffusing updates in a Byzantine Environment.* In Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems, October 1999.