Evaluating Byzantine Quorum Systems

Wagner Saback Dantas† Alysson Neves Bessani‡ Joni da Silva Fraga† Miguel Correia‡
† Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina – Brazil
‡ LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal

Abstract

Replication is a mechanism extensively used to guarantee the availability and good performance of data storage services. Byzantine Quorum Systems (BQS) have been proposed as a solution to guarantee the consistency of that kind of services, even if some of the replicas fail arbitrarily. Many BQS have been proposed recently, but comparing their performance is not simple. In fact, it has been shown that theoretical metrics like the number of steps or communication rounds say as much about the practical performance of distributed algorithms as they hide. This paper presents a comparative evaluation of several BQS algorithms in the literature. The evaluation is based both on experiments and simulations. For that purpose, a framework for evaluating BQS called BQSNeko was developed. The results of the evaluation allow a better understanding of the algorithms and the tradeoffs involved.

1 Introduction

Replication is the most extensively used strategy to construct fault-tolerant distributed systems. Byzantine Quorum Systems (BQS) [14], in particular, have been presented as a solution to construct Byzantine-tolerant distributed data storage services with guaranteed availability, security and robustness even if there are *Byzantine faults*, i.e., if some replicas fail arbitrarily. A cause of these arbitrary failures can be intrusions in the servers, so these systems have been said to be *intrusion-tolerant* [8, 25].

The idea of BQS is to implement a set of registers (or objects) on a set of servers. Clients communicate with the servers by message-passing to read and write on these registers. Several BQS algorithms have been proposed in the literature (e.g., [4, 13, 15, 14, 18, 17]). Their performance has been assessed mostly in terms of theoretical metrics, like the number of communication steps and message complexity. However, several authors have shown that theoretical metrics can say as much about the practical performance of distributed algorithms as they can hide [2, 3, 19]. In fact, for getting an accurate comparison between the performance of different distributed algorithms, we have to consider the specific techniques used by the algorithms (e.g., cryptography) and the environment where they are executed, considering aspects like fault scenarios. For BQS protocols, this is not different.

This need for a better assessment of the performance of BQS algorithms is the motivation for this paper, which presents a comparative evaluation of several BQS algorithms in the literature. The evaluation is based both on experiments and simulations. More precisely, we measured the latencies of read and write operations of several BQS algorithms, with several configuration settings, including arbitrary (or "Byzantine") faults.

The experiments and simulations were done using a framework that we implemented for that purpose, BQS-NEKO. This framework is an extension of NEKO, a generic framework for the assessment of distributed crash fault-tolerant algorithms [24]. BQSNEKO extends NEKO with classes that simplify the implementation of BQS and classes that allow the injection of Byzantine faults.

The analysis is divided in different scenarios intended to evaluate pairs of algorithms with similar properties, but that use different techniques to enforce those properties. All algorithms studied use confirmable writes (the client knows when its writes finish) [18] and strong semantics (atomic), requiring only the optimal number of $3f + 1$ servers [17]. The scenarios evaluated are the following: *Cost of minimality* – comparison between the "minimal" atomic register described in [17] and the atomic register implemented by the PHALANX's algorithm [15], which use respectively write-backs and the listener communication pattern; *Algorithms for Byzantine clients* – evaluation of the improved "minimal" register [17] and the BFT-BC algorithm [13]; and *Byzantine SMR versus BQS* – comparison of two techniques to implement Byzantine-tolerant data storage: Byzantine Quorum Systems (the BFT-BC algorithm [13]) and State Machine Replication (SMR) [21] (the Byzantine PAXOS algorithm [5]).

Related work. Only a small number of works in the BQS literature evaluate the algorithms they propose considering Byzantine execution scenarios and/or compare them with other algorithms. Martin et al. [17] propose and evaluate the SBQ-L algorithm, but the analysis does not regard the occurrence of faults and is just about SBQ-L, while our study compares that algorithm with others. Goodson et al. [9]

compare their consistency approach based on BQS with another based on SMR, but do not consider faults. For obtaining its results, that work employs two different software platforms while we do all the evaluation in a single testbed (BQSNEKO or NEKO directly). We believe that using the same software infrastructure allows a more refined assessment of the algorithms studied, the techniques involved and the results obtained. Moreover, our work presents not only LAN results (as [9] does), but puts in perspective the behavior of the protocols on WANs through simulations. An earlier work by Amir and Wool performs a set of experiments with non-Byzantine Quorum Systems on a WAN, but only at structural level, using metrics such as availability and accessibility [2]. Similar experiments were performed in works referred by [2], by they also do not match our goal of evaluating BQS protocols for reliable distributed storage. The same occurs with Jiménez-Peris et al. [11]: using a more analytical approach, they compare a set of non-Byzantine replication data strategies (including distinct quorum-based constructions) for metrics like availability, scalability and cost of messages.

Contributions. The main contribution of the paper is the presentation of the first comparison of the practical performance of several BQS, based both on experiments and simulations. This comparison provides insights that can be useful both from a theoretical point of view to help design more efficient protocols, and from a practical point of view to help practitioners choose the most adequate BQS for their application and environment. Furthermore, the performance of data storages based both on BQS and state machine replication is compared. A second contribution is the presentation of BQSNEKO, a framework for supporting the evaluation of BQS protocols. Besides being useful for that specific purpose, BQSNEKO suggests how to extend NEKO to simplify the implementation and comparison of sets of distributed algorithms of the same class, something that is quite unusual, at least in the context of Byzantine fault-tolerant algorithms.

2 Byzantine Quorum Systems

Byzantine Quorum Systems (BQS) [14] are a way to implement consistent and available Byzantine-resilient storage systems. BQS simulate registers on a distributed environment where processes communicate by message-passing. The system model assumes two sets of processes: a fixed set of servers U ($|U| = n$) and a possibly unbounded set of clients Π. A BQS $\mathcal{Q}$ is a non-empty set of subsets (*quorums*) of U ($\mathcal{Q} \subseteq 2^U$) where $\forall Q_1, Q_2 \in \mathcal{Q},\ Q_1 \cap Q_2 \neq \emptyset$. Each pair of processes is connected by an asynchronous reliable authenticated point-to-point channel. Quorums satisfy intersection properties required to maintain the system consistency. BQS rely on the assumption that there is always a quorum where all servers are correct.

In BQS, servers are subject to Byzantine failures, i.e., they can deviate arbitrary from their specification. In such cases, servers are said *faulty*. Otherwise, servers are said *correct*. The model assumes that up to f servers may be faulty (*f-threshold*). Client processes may also be faulty according to the fault model adopted for them, which can be fault-free or Byzantine. Clients access the system by communicating with read and write quorums. A *read quorum* Q_r (resp. *write quorum* Q_w) is accessed by clients on a read (resp. write) operation. When $|Q_r| = |Q_w|$, we have *symmetric quorums*. Otherwise, we have *asymmetric quorums*.

Each server stores a variable x that simulates a read-write register replicated on the set of servers U. The variable x is usually represented by a pair $\langle v,t \rangle$ where v is the variable value and t is a unique timestamp associated to the value v. Read-write registers can have different consistency semantics according to the behavior of the system on concurrent read and write operations. Three consistency semantics are usually defined [12]: *safe*, *regular* and *atomic*. Registers can also support simultaneous write operations (*multi-writer multi-reader* semantics) or just a single write at time (*single-writer multi-reader* semantics).

Data involved on quorum operations can be either *generic* (i.e., any data) or *self-verifying*. Self-verifying data contains an unforgeable digital signature that allows to check if it is modified. This is very useful for Byzantine environments because it allows correct processes to detect if malicious servers corrupted values.

Many BQS and algorithms have been proposed in the literature, e.g., [4, 13, 15, 14, 18, 17]. The different solutions reflect different views on how to build storage systems using BQS, and basically differ on aspects such as: quorum organization (symmetric or asymmetric), register semantic (safe, regular or atomic) and failure model of the clients (failure-free, crash, Byzantine).

3 Methodology, Tools and Configurations

This section presents the methodology underlying our work, the tools used for developing and running the experiments (NEKO and BQSNEKO), and the settings and configurations analyzed.

3.1 Methodology

The basic method we use is a simple experimental one: implementing several variations of a basic distributed storage service that supports only read-write operations, then evaluating and comparing their performance using experiments and simulations. The reason for this restriction to read-write operations (vs. arbitrary operations) is that BQS have been shown to be powerful enough to implement at most atomic registers with these two operations, not other operations like test-and-set or increment [10].

The evaluation in Section 4 considers three scenarios. The idea is to arrange the algorithms with similar consis-

tency semantics and quorum system structure in subsets. The experimental results are compared with classical theoretical metrics and those in prior work. The objective is to clarify the relation between the algorithms (and the techniques they employ), their practical performance and the aspects not captured by theoretical metrics.

The experiments were done in a LAN because experiments in a WAN, in practice, have to be done in the Internet, being more subject to interferences whose impact is hard to assess (variable CPU loads and network delays). Nonetheless, we establish some relations and projections between the LAN results and results obtained over a simulated large-scale network, in order to grasp how the algorithms would behave in a WAN. All experiments use a low number of servers since servers must fail independently, something that has to be obtained using diversity, which has a cost that becomes higher as the number of replicas increases [20].

3.2 BQSNeko

We implemented and executed all BQS algorithms in Java using BQSNEKO, a framework based on NEKO [24], especially developed for our study. The only exception to the use of BQSNEKO was the last experiment, in which we compare BQS and SMR. In that case we used an implementation of the SMR system directly on NEKO, with an architecture similar to that of the BQS algorithms in BQSNEKO.

BQSNEKO extends the NEKO framework used for prototyping and executing distributed algorithms over real and simulated networks. BQSNEKO was created from the perception of two limitations of NEKO for the implementation of BQS algorithms: the absence of a mechanism to inject Byzantine faults and of a "skeleton" with basic code to implement BQS algorithms. BQSNEKO is essentially an extension of NEKO with these features.

Like in NEKO, in BQSNEKO a distributed environment is implemented as a set of processes communicating by message-passing. Each process maintains locally an instance (i.e., information) of the distributed application, which is composed by a set of processes interconnected by one or more networks. The application instance is structured in a stack of three layers[1], each of which implements a specific service. The layers communicate with each other exchanging messages by calling *send* and *deliver* methods: higher layers push down messages to lower ones calling a *send* method; lower layers push up messages to higher ones calling a *deliver* method. Some network models are already implemented in NEKO. Networks can be simulated (using simulation libraries) or can be real physical networks (using Java sockets). NEKO also provides features to develop new network models.

An implementation of a BQS algorithm involves three aspects: *(1) Configuration settings* describing basic properties of the quorum system used and its configuration parameters (e.g., fault-threshold, number of processes, size of read and write quorums); *(2) Messages*, namely the collection of messages exchanged by client and server BQS protocols; *(3) Protocols* executed by the clients and servers.

An application in BQSNEKO has three layers (see Figure 1). The *Process Layer* contains the functionality of the BQS process, either a client or a server. The *Latency/Crypto Layer* is used to simulate the delays of the cryptographic operations on a BQS algorithm. If the communication is done in a real network, then the delays are not simulated but real. In that case, the cryptographic operations are implemented using the Java Cryptography Extensions (JCE). The *Profile Layer* defines the failure profile or faultload (failure-free, crash or Byzantine).

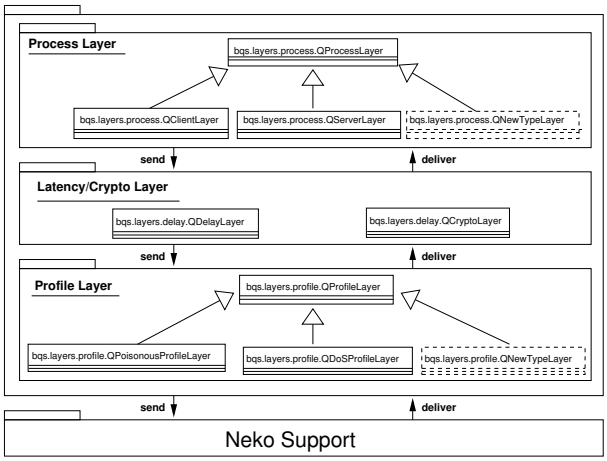


Figure 1. BQSNEKO **process layer model**

To prototype and experiment with BQSNEKO it is necessary to follow three basic steps: *(1)* Defining the structural properties of the BQS, implementing the BQS client and server algorithms, and creating the messages they use. To help with this step, BQSNEKO provides generic classes and methods for constructing both client and server algorithms and the messages. *(2)* Specifying Byzantine profiles (faultloads) by extending the profiles provided by BQSNEKO or constructing new ones. *(3)* Configuring and running the experiment. Just as for the algorithm's implementation, BQSNEKO also extends the NEKO configuration by defining new parameters related to the configuration of BQS and Byzantine scenarios.

3.3 Configuration Settings

The experiments presented in the paper were performed with two configurations: LAN and simulated WAN. These configurations were set up with parameters such as the number of clients doing concurrent operations and the number of Byzantine servers.

[1] The current BQSNEKO version is based on version 0.9 of NEKO. The new NEKO version (version 1.0) has a different component organization.

The executions were deployed on a local network infrastructure, using the TCP sockets provided by NEKO to implement asynchronous reliable point-to-point channels. The authentication and integrity of the channels are obtained using session keys and the HmacSHA-1 message authentication algorithm (provided by BQSNEKO). Some protocols also require digital signatures based on asymmetric cryptography. These signatures are also provided by the BQSNEKO crypto layer, which calls the JCE. In the experiments, we used signatures based on RSA and SHA-1. The LAN contained 5 computers interconnected by a 100 Mb/s Ethernet network. The computers had identical hardware and software configurations: 1.9 GHz PCs with 512 MB RAM, Linux kernel 2.6.12, and Java virtual machine 1.5.0_06. When the maximum number of faulty servers was set to $f = 1$, there were $n = 4$ servers and each one was executed in a different machine. When the number of faulty servers was $f = 2$, there were $n = 7$ servers and some machines run 2 servers.

The evaluation of algorithms on large-scale environments was done on a simulated network. We adopted the *contention-aware simulated network model* specified in [23]. It takes into account resource contention (local processing and network allowing a more precise evaluation than models that do not consider it. Contention is represented by the λ parameter ($\lambda \geq 0$) which specifies the relative performance between resources of local processing and network. Therefore, for modeling large-scale networks we use $\lambda = 0.1$ (more contention on network resources) as also done in similar works [22].

The values reported in the next section for protocol executions are mean values of the *latency* of the read or write operation (in milliseconds) and their standard deviations, obtained from the execution of the same operation by a correct client 1000 times. The latency is measured getting a clock reading immediately before sending a request and another one immediately after getting the reply, then subtracting the two values. In some cases, we observed high deviations due to high contention on the computer processors. This happened mostly with protocols that use signatures, since they use more CPU time causing greater contention.

For simulations we collected times of a single execution over a simulated WAN with a correct client per scenario (in s.t.u., simulation time units). In some cases, we also present data about the additional number of messages sent due to Byzantine servers and contention with other correct clients.

We limit our attention to systems that tolerate f faulty servers for $f = 1$ and $f = 2$. For each experiment or simulation with an f-threshold BQS, we consider $0 \leq b \leq f$ faulty servers ($b = 0$ for fault-free executions). Faulty servers always perform the same faulty behavior in all experiments: they always return a fake value, instead of the value actually stored in the variable. This behavior does not corrupt the result of the operation that reads the value, since the algorithms tolerate this behavior, but can have an impact in terms of performance. The quorum systems used have always the tight number of servers needed to tolerate the number of faults f considered, i.e., n is always equal to $3f + 1$.

4 Experiments

This section describes the experiments with several well-known BQS protocols in the literature. They are divided in three distinct scenarios, every of which containing a pair of algorithms with similar properties, but using different techniques to enforce these properties. The scenarios are the following:

- **Cost of minimality** (Section 4.1): assessment of the cost of implementing a "minimal" atomic register, i.e., the costs of the two basic mechanisms used to implement register atomicity with correct clients and optimal number of servers ($n = 3f + 1$) [17]. PHALANX's algorithm [15], which employs write-backs, is compared with the MINIMAL-C algorithm [17], which is based on the listener communication pattern.

- **Algorithms for Byzantine clients** (Section 4.2): assessment of the techniques that allow atomic registers to tolerate poisonous writes by Byzantine clients (different values are sent to different servers). MINIMAL-F [17], which employs write disseminations between the servers, and BFT-BC [13], which uses certificates and signatures for the client to prove that it follows the protocol, are compared;

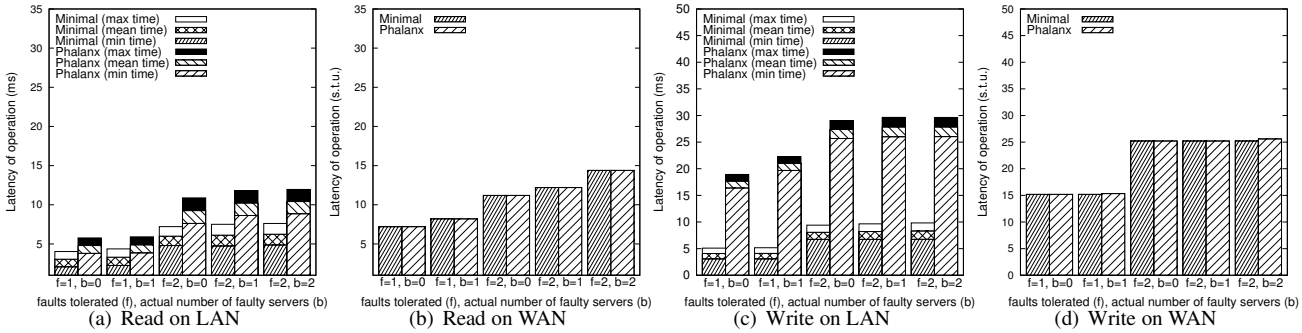
- **Byzantine SMR versus BQS** (Section 4.3): study of the costs involved for atomic storage implementation using two replication techniques: BQS and SMR. We present a comparison between the PAXOS SMR algorithm [5] (henceforth called PAXOS for short) and the BFT-BC algorithm [13].

Although there are a diversity of BQS constructions, we chose those ones since they all have optimal resiliency and represent a class of basic mechanisms (write-backs, listener, etc.) used by many other similar algorithms.

4.1 Cost of Minimality

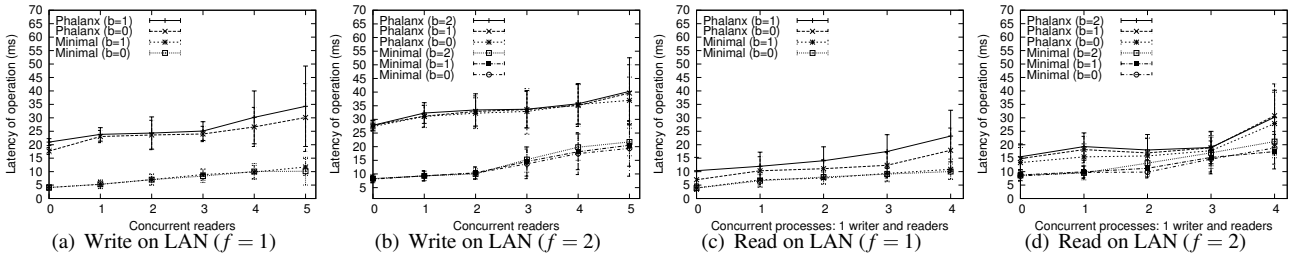
A register is said to be atomic if *(a)* a read that is not concurrent with a write returns the last value written; and *(b)* concurrent reads and writes behave as if they occurred in some order [12]. Designing a protocol that implements an atomic register is not simple because it has to ensure that the value obtained by a read operation remains the same for the subsequent reads until the following write. This is not simple when there are concurrent reads and writes.

The first algorithmic construction used to obtain atomicity in BQS protocols was the *write-back* mechanism used in



(a) Read on LAN (b) Read on WAN (c) Write on LAN (d) Write on WAN

Figure 2. MINIMAL-C **Vs** PHALANX**: latency for read and write without concurrency.**



(a) Write on LAN ($f = 1$) (b) Write on LAN ($f = 2$) (c) Read on LAN ($f = 1$) (d) Read on LAN ($f = 2$)

Figure 3. MINIMAL-C **Vs** PHALANX**: write and read with concurrency.**

the PHALANX algorithm [15]. A read operation usually involves contacting a read quorum to obtain the value v. The write-back mechanism uses an extra access to the quorum system to write v in all servers. This extra access ensures that all subsequent reads (before a write) will read the same value v. The PHALANX atomic register protocol requires $n \geq 3f+1$ servers [15].

More recently, Martin et al. showed that no atomic write-confirmable quorum system protocol[2] tolerating f Byzantine servers can be implemented with less than $3f+1$ servers [17]. This work also shows that this bound is tight by presenting the SBQ-L algorithm (that we call MINIMAL-C), which implements an atomic register with optimal resiliency. This algorithm employs asymmetric quorums (write quorums smaller than read quorums) with $n \geq 3f+1$ servers and the *listener communication pattern* where a reader registers itself on servers and receives updates from the servers until some value is returned by at least $2f+1$ servers. When this happens, the registration on servers is revoked. As PHALANX, MINIMAL-C does not tolerate faulty clients.

The goal of the first scenario is to evaluate what is the "cost of minimality", i.e., the cost involved in building an atomic register with the minimal number of servers – $n = 3f+1$ – and the listener pattern (MINIMAL-C) compared with the approach based on write-backs (PHALANX).

[2] A BQS protocol is write-confirmable if the client knows when its write completes, i.e., messages are sent by the servers to confirm the write completion [18].

Beyond evaluating the algorithms comparing their latencies, we analyze the number of extra messages exchanged when there is concurrency between writes and reads. These extra-messages correspond to the number of additional messages collected by the client due to the listener pattern execution (MINIMAL-C) and the number of write-back messages (PHALANX). Concluding, our aim here is to compare the costs of the two most popular techniques for implementing register atomicity: write-backs and listener pattern.

Evaluation without concurrency. Figures 2(a) and 2(b) show the latencies of reads without concurrency on local and simulated large-scale networks, respectively. The number of servers ($n = 3f+1$) and the number of faulty servers (b) vary. The mean latency values of executions on a LAN show that the performance of MINIMAL-C is slightly better than that of PHALANX. That result comes from a signature verification in PHALANX (the servers store self-verifying data, so a signature has to be verified). However, the time difference is small because verifying a signature is much faster than doing a signature with RSA. As long as there is no concurrency, no write-back operation is done and the data received comes from at least one correct server so it is correctly signed. In the MINIMAL-C algorithm there is no signature verification, justifying the small execution latencies of reads. On average the impact of Byzantine servers is actually low: for $b = 1$ the latency grows $\approx 2.1\%$ (PHALANX) and $\approx 8.2\%$ (MINIMAL-C); for $b = 2$ the growth is $\approx 13.1\%$ (PHALANX) and $\approx 4\%$ (MINIMAL-C).

The performance on a simulated large-scale network (Figure 2(b)) for both protocols is equal, showing a disappearance of the effect of the signature verification on the PHALANX's read protocol observed at the tests on a LAN. Although the read of the MINIMAL-C protocol executes 3 steps, the last step does not count for the latency since the final message of the listener pattern does not require a response. As result, the latency for reading in MINIMAL-C is similar to PHALANX (which executes 2 steps since there is no write-back).

The results of write operations with no concurrency are exhibited, respectively, in Figures 2(c) and 2(d). Comparing the results, the cost of the cryptography in PHALANX is evident. The operation skeleton for the two algorithms is identic: both query a quorum for data, create a new pair $\langle v,t \rangle$, try to update the system and wait for a set of acknowledgments from servers. However, in PHALANX the client signs the value using RSA which takes almost 14 ms in our environment. In a WAN this difference almost disappears, and the protocols have similar latencies (Figure 2(d)).

Concurrency. Figures 3(a) and 3(b) show the results of a scenario with one writer and 0 to 5 concurrent readers, for the fault thresholds $f = 1$ and $f = 2$. In all cases MINIMAL-C has lower latency than PHALANX. The impact of concurrency and the number of servers is not high.

Figures 3(c) and 3(d) illustrate the values collected from an execution of a reader along with one writer and other readers (varying among 0 to 4) in scenarios with fault thresholds equal to 1 and 2, respectively. Just considering the mean values for all system loads (faults and concurrency) we can note small variations on the behavior of the reader's performance in the MINIMAL-C protocol. On the other hand, the read operation in PHALANX has a more accentuated increase for 4 concurrent readers, especially when $f = 2$.

Extra messages. Table 1 gives percentages of the number of reads that do write-back operations in PHALANX and that use the listener pattern in the MINIMAL-C protocol. Such values reveal a low use of the write-back mechanism: in the worst case on a LAN concurrency involving write and read operations, that mechanism is executed approximately 7% of times for $f = 1$ and 11% for $f = 2$. From the point of view of extra messages, the table shows that concurrency has a higher impact in the MINIMAL-C protocol. Reads executing the listener pattern are 88.46% of all executions for $f = 1$ and at almost all executions for $f = 2$ (99.24%). However, as can be seen in the section, even with these extra messages, MINIMAL-C outperforms PHALANX in a LAN.

	$f=1$		$f=2$		
	$b=0$	$b=1$	$b=0$	$b=1$	$b=2$
MINIMAL-C	88.5	81.7	94.4	96.9	99.2
PHALANX	6.8	6	10.9	9.9	7.9

Table 1. Percentage of reads using the listener pattern (MINIMAL-C) and write-backs (PHALANX) in a LAN. Concurrency with 1 writer and 0-5 readers.

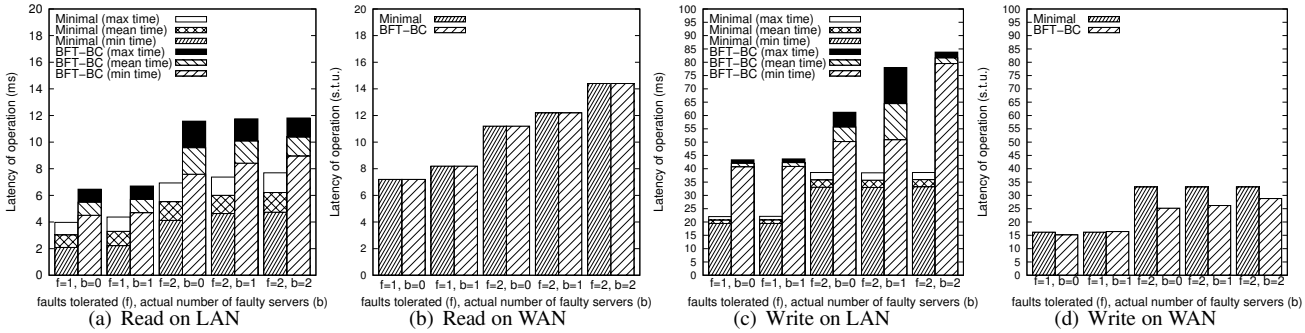
4.2 Algorithms for Byzantine Clients

In Byzantine-prone environments, not only servers can fail arbitrarily. Clients can also be faulty, executing steps that are not in their specification. In that case, implementing a register with atomic semantics is even more complicated. These clients can damage the system by violating the system consistency semantics (safety) and/or the protocol termination (liveness). A common approach that protocols use to cope with Byzantine clients is the use of *signed messages* to detect modifications performed by malicious clients. Other mechanisms are also used.

In [17], Martin et al. provide an improved version of the SBQ-L algorithm (called here MINIMAL-F), which implements multi-writer multi-reader atomic registers and tolerates Byzantine clients. The MINIMAL-F algorithm assumes clients can maliciously exploit the listener pattern: faulty writers update different values preventing the servers from returning the same value and consequently preventing concurrent or future reads from terminating (poisonous write). To tolerate such bad clients, MINIMAL-F employs signed messages and *replication of messages among servers* during the write protocol. In this approach, clients share the same private key (the writer private key) and servers have the associated public key. Servers only accept write requests correctly signed. Furthermore, servers perform a new step in the write protocol: they replicate new stored values to other servers in the system in order to keep the stored values consistent.

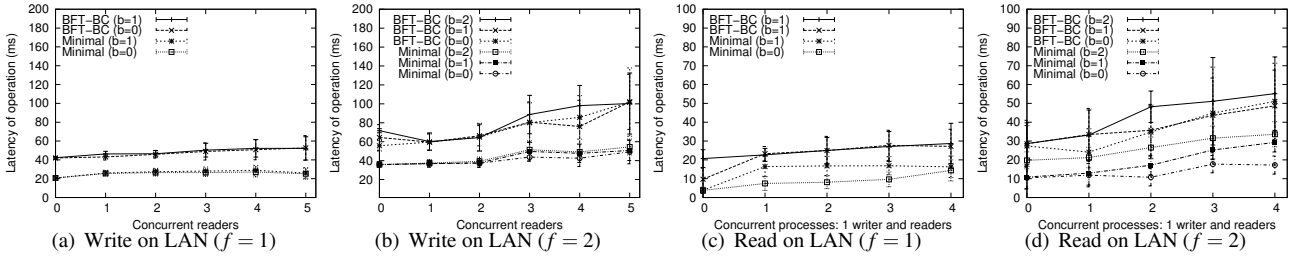
In a more recent work, Liskov and Rodrigues described the BFT-BC algorithm, which implements a multi-writer multi-reader atomic register that handles a large variety of problems caused by Byzantine clients [13]. This algorithm requires a Byzantine quorum system with $n \geq 3f + 1$ servers. To tolerate Byzantine clients, BFT-BC uses *certificate proofs*, composed of $2f + 1$ signed messages from servers (a quorum). Therefore, for passing from a phase to the next, in BFT-BC a client must prove that it is not trying to execute an inappropriate operation by showing its certificate. Valid proofs serve to vouch the state of the client (for example, did the client complete its last write operation consistently?), enforcing clients to keep up some algorithm invariants, preventing them from successfully sending forged messages, exhausting the timestamp space or performing incompletely a protocol.

The purpose of this experiment is to compare the performance of the MINIMAL-F and the BFT-BC algorithms. The experiments also analyze the extra number of messages sent



(a) Read on LAN (b) Read on WAN (c) Write on LAN (d) Write on WAN

Figure 4. MINIMAL-F **Vs** BFT-BC**: read and write without concurrency.**



(a) Write on LAN ($f = 1$) (b) Write on LAN ($f = 2$) (c) Read on LAN ($f = 1$) (d) Read on LAN ($f = 2$)

Figure 5. MINIMAL-F **Vs** BFT-BC**: write and read with concurrency.**

by correct clients on their read operations when confronted with concurrent processes.

Evaluation without concurrency. Figures 4(a) and 4(b) present the results of the execution of read operations with no concurrency, respectively on a LAN and on a simulated WAN. For all fault conditions MINIMAL-F has better performance than BFT-BC. This happens because the read of BFT-BC requires the verification of signatures for each value queried from a quorum, on the contrary to MINIMAL-F. With Byzantine servers ($b > 0$), both protocols present slight changes of performance since more messages are collected (messages from Byzantine servers have to be discarded). On the simulated large-scale network the latency times tend to be equal since the costs of local processing is diluted in the communication delays and the number of communication steps is 2 in both protocols.

For the write protocol, MINIMAL-F has better performance for all fault settings (Figure 4(c)). As no concurrency is considered, both BFT-BC and MINIMAL-F execute in 4 communication steps. Therefore, the cost of the signed proofs in BFT-BC is higher than the cost of signatures at the clients in MINIMAL-F, especially when $f = 2$. In fact, on a LAN BFT-BC requires more local processing since it uses more signatures, as each server does signatures twice: when it sends the prepare and acknowledgment messages to the client. MINIMAL-F does signatures only once, when the client performs its write request. Although each server verifies the signature and, if it is correct, replicates each write request (either from the client or other servers), the cost of those procedures are very small for two reasons: firstly, because the communication delay on a LAN is low; secondly because the time required to verify signatures is lower than the time to do signatures (in our environment ≈ 0.9 ms and ≈ 14 ms, respectively).

Figure 4(d) shows results of write operations on a simulated WAN. The results differ from those in a LAN. In general, MINIMAL-F has running times higher than those obtained for BFT-BC. The cost of communication is higher in the MINIMAL-F (servers have to disseminate writes between themselves – an $O(n^2)$ message complexity protocol) while the impact of contention on signature processing in BFT-BC is lower, leading to its better performance.

Concurrency. Figures 5(a) and 5(b) present latencies of write operations with concurrent readers over a LAN for $f = 1$ and $f = 2$, respectively. In most of the tests, MINIMAL-F performs more efficiently than BFT-BC. Considering the mean deviations of each execution, it can be observed that performance in BFT-BC is more affected as the number of concurrent readers grows. For all settings considered, the latencies of MINIMAL-F and BFT-BC were in the following intervals: from 21 ms to 29 ms and 42 ms to 53 ms ($f = 1$); from 36 ms to 54 ms and from 56 ms to 102 ms ($f = 2$).

The increase of latency of BFT-BC's writes as concurrency and number of faulty servers grow is a sign of more contention in the servers. Servers have to verify the certificate in the clients' requests and sign response mes-

sages. Moreover, BFT-BC verifies and does signatures during reads executing concurrently with writes, whereas MINIMAL-F does not. Looking at the same concurrency cases from a reader's perspective, the impact of those additional cryptographic operations can be confirmed.

Figures 5(c) and 5(d) illustrate experiments where one writer executes concurrently with 0 to 4 readers. For all fault scenarios and numbers of concurrent processes, the performance of BFT-BC, that increases faster, is always worse than that of MINIMAL-F because here BFT-BC is also more subject to contention on servers processing concurrent read and write requests. The following intervals for read operations were obtained: from 9.72 ± 6 ms to 29 ± 11 ms for BFT-BC and 4 ± 1.5 ms to 16.3 ± 5.7 ms for MINIMAL-F ($f = 1$); from 27.4 ± 11 ms to 55 ± 19.5 ms for BFT-BC and from 10.4 ± 6 ms to 32.6 ± 6 ms for MINIMAL-F ($f = 2$).

Extra messages. In general, BFT-BC's reader performs just a few write-backs. For MINIMAL-F, like for MINIMAL-C (Section 4.1), concurrency is visible in terms of additional messages. However, concurrency occurs at a smaller level (write in MINIMAL-C performs better than its faulty counterpart, what raises more concurrency) although it does not affect much the total performance of this read on concurrency. Table 2 exhibits percentages of BFT-BC's reads with write-backs and MINIMAL-F's reads with extra messages due to the use of the listener pattern.

	$f=1$		$f=2$		
	$b=0$	$b=1$	$b=0$	$b=1$	$b=2$
MINIMAL-F	64.2	78.4	82.5	87.7	91.7
BFT-BC	6.2	6.6	11.8	9.9	9.5

Table 2. Percentage of reads using the listener pattern (MINIMAL-F) and write-backs (BFT-BC) in a LAN. Concurrency with 1 writer and 0-5 readers.

Notice that the results of Table 2 are in accordance with those of the previous experiment (Table 1). This was expected since the protocols use the same mechanisms (write-backs and listener pattern).

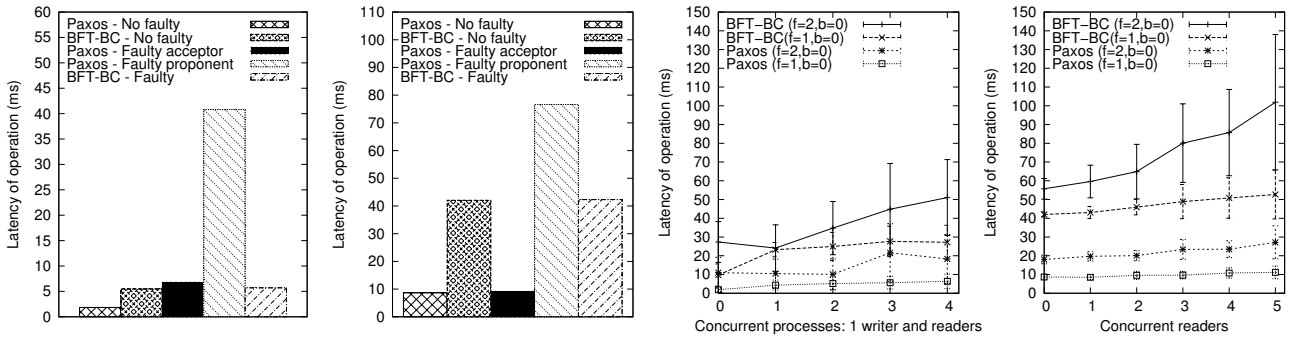
4.3 Byzantine SMR versus BQS

Two techniques can be used for implementing Byzantine fault-tolerant replication: State-Machine Replication (SMR) [21] and Byzantine Quorum Systems [14]. These techniques differ essentially in the following aspects: (a) SMR can be used for implementing any deterministic service, while (asynchronous) BQS can not; (b) SMR requires solving consensus so it cannot be implemented deterministically in asynchronous systems [7], while BQS can.

There has been some discussion on which is the best technique, SMR or BQS. Recent works argue in both directions, giving emphasis either to the fact that SMR is generic [6], or to the possibility of implementing BQS without additional time assumptions (or, alternatively, randomization) [26] and its potential scalability [1]. Advances have been presented in the literature for both techniques giving, for example: evidence that SMR can be efficient [5, 19]; improvements for the Byzantine PAXOS consensus protocol [16]; new protocols for BQS that tolerate malicious clients using an optimal number of servers [4, 13]. These improvements suggest that both SMR and BQS can be used for implementing reliable services efficiently. On the other hand, they also raise a pair of questions: which of these techniques is the most efficient? In which conditions should one of these techniques be used instead of the other?

Here, we investigate these issues with an experimental evaluation of two protocols: Byzantine PAXOS [5] (SMR) and BFT-BC [13], which is the BQS protocol most resilient to malicious client-behavior we assessed. The BQS protocol evaluated is the same from the previous section, but Byzantine PAXOS deserves some discussion before presenting the experiments. SMR requires that all operations (in this case, reads and writes) are executed by all servers in the same order. In this context, PAXOS works as a total order multicast protocol where one of the servers, called the *proposer*, act as a leader defining in which order each operation should be processed. When the other servers, called *acceptors*, receive the order for each operation, they run 2 steps of message exchanges to commit this order and then execute the request. The SMR protocol employed in our experiments implements an optimization for read operations in which clients only execute the PAXOS protocol if they read the register state in $n - f$ servers (with a single access – a message round-trip) and not all of them have the same value [5]. To implement this modified version of PAXOS, we used NEKO plus mechanisms similar to those in BQSNEKO, e.g., using a layer of cryptography for executions over a LAN (we do not use BQSNEKO since it is targeted to BQS).

Evaluation without concurrency. Figure 6(a) presents results of reads without concurrency over a LAN regarding various fault contexts for $f = 1$. When there are no faults, both protocols achieve their optimal performance, with termination in 2 steps. In that case, PAXOS achieves a slightly better performance than BFT-BC. This small difference occurs because BFT-BC verifies signatures when consolidating a read value. When one fault occurs, BFT-BC's read is almost not affected (goes from ≈ 5.5 ms to ≈ 5.7 ms). The client spends this additional time simply waiting for one more message from a correct server, to get the value from a full quorum, in an environment where communication time is low. For PAXOS with a faulty acceptor, the client sometimes is forced to run the ordering algorithm (instead of using the optimistic read described before, that is similar to a BQS algorithm). This increments the latency from ≈ 1.85 ms (context free of faults) to ≈ 6.8 ms. When there are a faulty acceptor and a faulty proposer, the final latency grows



(a) Read without concurrency (b) Write without concurrency (c) Read with concurrency (d) Write with concurrency

Figure 6. PAXOS **Vs** BFT-BC **on the LAN without (**$f = 1$**) and with (**$f = 1$ **and** $f = 2$**) concurrency.**

even more. It passes to ≈ 41 ms since the PAXOS ordering protocol runs in two rounds, involving an extra protocol for electing a new proposer.

Figure 6(b) provides results for write operations with no concurrency and fault threshold $f = 1$. BFT-BC takes more time than PAXOS to write in fault-free executions. The reason is that, with no faulty servers, only BFT-BC does public-key signatures in the write operation. The PAXOS protocol is little affected by a faulty acceptor. More precisely, only one more communication step is necessary, whose cost is notably low in a LAN. BFT-BC's write is not much affected by failures for the same reason as its read protocol. PAXOS' write performs worse than BFT-BC when the proposer is faulty, because a new proposer has to be elected. Such procedure requires two additional steps of communication and doing a public-key signature, which is expensive when the communication is cheap, i.e., in a LAN. Note that the PAXOS' mean latency for reading with a faulty proposer (Figure 6(a)) is almost half (≈ 41 ms) of the latency for writing with faulty proposer (≈ 77 ms). The reason we observed is that 50% of the reads are non-optimized, i.e., only 50% of the reads executed the PAXOS ordering protocol.

Concurrency. Figures 6(c) and 6(d) present times for concurrent operations in a LAN for both protocols. Figure 6(c) shows results of reads performed together with one writer and a variable range of 0 to 4 readers. Figure 6(d) shows results of writes executed concurrently with a range of 0 to 5 readers. Considering the mean deviations, PAXOS performs better than BFT-BC in all cases. It is interesting that PAXOS achieves smaller processing contention and better scalability than its BQS counterpart.

Also, it has to be pointed out that approximately 88% of the reads in Figure 6(c) were optimized, for $f = 1$ (4 servers). For $f = 2$ (7 servers) that amount decreased to 64%. These values indicate that, as for some BQS algorithms which do not use signatures (e.g., MINIMAL-C), PAXOS performs with low variation on latency due to concurrency. This happens because its implementation does not execute consensus for each received request from a client. Consensus is executed for batches of requests.

5 Summary and Conclusions

Some comments are due on the experimental results described in this paper. First, it is notable the large impact of asymmetric cryptography on executions on a LAN in which the performance bottleneck resides in local processing and the communication latency is small. It is common when assessing the performance of distributed algorithms to disregard local delays, but for algorithms that use signatures that cost has to be taken into account. As a consequence, protocols storing self-verifiable data (e.g., PHALANX and BFT-BC) have higher latency, as shown in all experiments conducted. On the other hand, that cost is mitigated when those algorithms run over a WAN.

It was also demonstrated the good performance of protocols implementing the listener pattern with protocols MINIMAL-C and MINIMAL-F (Sections 4.1 and 4.2). These algorithms have better performances than their counterparts even with concurrency. Although that does not mean low concurrency in terms of exchanged messages (fitting [17]), such protocols explore very well the inherent low latency of the communication on a LAN. However, the experiments on the simulated WAN showed that this low latency is not kept in a WAN (Section 4.1).

The paper puts in evidence the efficiency of the SMR approach using the Byzantine PAXOS algorithm when running on a LAN and with no faulty proposer. This conclusion diverges from opinions that BQS are more efficient than SMR, at least for the protocols considered. The PAXOS algorithm (message complexity $O(n^2)$) demonstrated having a good performance because it was implemented in a "batch fashion": the algorithm does not order individual read/write requests but batches of requests. We do not know whether that performance will be guaranteed on large-scale environments with contention among clients and with Byzantine

fault loads. Regarding the Byzantine PAXOS protocol, the experiments of Section 4.3 show that the recent optimization for early decision in two communication steps in synchronous fault-free executions [16] has almost no benefit in LAN settings, since the communication is too fast.

To make the evaluation easier, a framework for evaluating BQS called BQSNEKO was designed and implemented. It is freely available (with all protocols) at `http://www.das.ufsc.br/~wagners/bqsneko`.

The results presented in this paper are an important step to better understand the algorithms assessed, as well as the functionality and tradeoffs on Byzantine-tolerant storage systems. They show subtle details of the BQS protocols evaluated, that would be usually hidden by traditional metrics of distributed algorithms. To the best of our knowledge, there is no similar assessment in the literature.

Acknowledgements. We thank Eduardo Alchieri for his contribution on the Byzantine Paxos implementation. This work was supported by LaSIGE, CNPq (Brazilian National Research Council) through process 550114/2005-0, the EU project IST-3-027513-STP (CRUTIAL) and CAPES/GRICES (project TISD).

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, pages 59–74, Oct. 2005.

[2] Y. Amir and A. Wool. Evaluating quorum systems over the Internet. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing - FCTS'96*, pages 26–35, 1996.

[3] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the Internet. In *Proceedings of the 21st Symposium on Principles of Distributed Computing - PODC'02*, pages 243–252, July 2002.

[4] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN'06*, pages 115–124, June 2006.

[5] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[6] R. Ekwall and A. Schiper. Replication: Understanding the advantage of atomic broadcast over quorum systems. *Journal of Universal Computer Science*, 11(5):703–711, May 2005.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[8] J. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd Int. Conference on Computer Security*, pages 203–218, 1985.

[9] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN'04*, pages 135–144, June 2004.

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

[11] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, 2003.

[12] L. Lamport. On interprocess communication (part ii: algorithms). *Distributed Computing*, 1(1):203–213, 1986.

[13] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of the 26th International Conference on Distributed Computing Systems - ICDCS'06*, June 2006.

[14] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[15] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of 17th Symposium on Reliable Distributed Systems - SRDS'98*, pages 51–60, Oct. 1998.

[16] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, June 2006.

[17] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th international Conference on Distributed Computing - DISC'02*, volume 2508 of *LNCS*, pages 311–325, Oct. 2002.

[18] J.-P. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks – DSN'02*, pages 374–383, June 2002.

[19] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo. Randomized intrusion-tolerant asynchronous services. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN'06*, pages 568–577, June 2006.

[20] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI/FCUL TR 06–15, Department of Informatics, University of Lisbon, September 2006.

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

[22] P. Urbán, N. Hayashibara, A. Schiper, and T. Katayama. Performance comparison of a rotating coordinator and a leader based consensus algorithm. In *Proceedings of the 23rd Symposium on Reliable Distributed Systems - SRDS'04*, pages 4–17, October 2004.

[23] P. Urbán, X. Défago, and A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proceedings of the 9th International Conference on Computer Communications and Networks - IC3N'00*, pages 582–589, Oct. 2000.

[24] P. Urbán, X. Défago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997, Nov. 2002.

[25] P. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 2003.

[26] L. Zhou, F. B. Schneider, and R. van Rennesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.