Minimal Byzantine Storage

Jean-Philippe Martin, Lorenzo Alvisi, Michael Dahlin
University of Texas at Austin - Dept. of Computer Science
Email: {jpmartin, lorenzo, dahlin}@cs.utexas.edu

Department of Computer Science
University of Texas at Austin
Austin, Texas 78712

Abstract. Byzantine fault-tolerant storage systems can provide high availability in hazardous environments, but the redundant servers they require increase software development and hardware costs. In order to minimize the number of servers required to implement fault-tolerant storage services, we develop a new algorithm that uses a "Listeners" pattern of network communication to detect and resolve ordering ambiguities created by concurrent accesses to the system. Our protocol requires $3f + 1$ servers to tolerate up to f Byzantine faults—f fewer than the $4f + 1$ required by existing protocols for non-self-verifying data. In addition, SBQ-L provides atomic consistency semantics, which is stronger than the regular or pseudo-atomic semantics provided by these existing protocols. We show that this protocol is optimal in the number of servers—any protocol that provides safe semantics or stronger requires at least $3f + 1$ servers to tolerate f Byzantine faults in an asynchronous system. Finally, we examine a non-confirmable writes variation of the SBQ-L protocol where a client cannot determine when its writes complete. We show that SBQ-L with non–confirmable writes provides regular semantics with $2f + 1$ servers and that this number of servers is minimal.

1 Introduction

Byzantine storage services are useful for systems that need to provide high availability. These services guarantee data integrity and availability in the presence of arbitrary (*Byzantine*) failures. A common way to design such a system is to build a *quorum system*. A quorum system stores a shared variable at a set of servers and performs read and write operations at some subset of servers (a *quorum*). Quorum protocols define an intersection property for the quorums which, in addition to the rest of the protocol description, ensures that each read has access to the current value of the variable. Byzantine quorum systems enforce the intersection property necessary for their consistency semantics in the presence of Byzantine failures.

The number of servers in a Byzantine storage system is a crucial metric since server failures must be independent. Therefore, to reduce the correlation of software failures, each server should use a different software implementation [18]. The first advantage of reducing the number of servers necessary for a service

is the reduction in hardware costs. However, as hardware costs get cheaper in comparison to software and maintenance costs, the most important benefit of reducing the number of different servers is the corresponding reduction in development and maintenance costs. Furthermore, for large software systems (e.g. NFS, DNS, JVM) a fixed number of implementations may be available, but it is expensive or otherwise infeasible to create additional implementations. In such a situation, a new protocol requiring fewer servers may enable replication techniques where they were not previously applicable.

To minimize the number of servers, we present a new protocol called Small Byzantine Quorums with Listeners (SBQ-L). The protocol uses a "Listeners" pattern of communication to detect and resolve ordering ambiguities when reads and writes simultaneously access a shared variable.[1] Whereas existing algorithms use a fixed number of communication rounds, servers and readers using SBQ-L exchange additional messages when writes are concurrent with reads. This communication pattern allows the reader to monitor the evolution of the global state instead of relying on a snapshot. As a result, SBQ-L provides strong consistency semantics using fewer servers. In particular, Table 1 shows that SBQ-L provides atomic semantics [9] for generic data using as few as $3f + 1$ servers to tolerate f faults, instead of the $4f + 1$ servers that were previously required to provide even the weaker regular [11] or partial-atomic [17] semantics.

We show that SBQ-L is optimal with respect to the number of servers required to provide a safe shared variable in the common model of asynchronous reliable authenticated channels [1, 3, 11–13]. In particular, we show that any protocol that tolerates f Byzantine failures and that provides safe or stronger semantics must use at least $3f + 1$ servers. Since SBQ-L can provide atomic semantics with $3f + 1$ servers, it is optimal with respect to this critical metric.

We apply the SBQ-L protocol and our lower bound analysis to compare protocols for generic data to these for *self-verifying data* (data that cannot be undetectably altered, e.g. that are digitally signed). We find that, surprisingly, SBQ-L performs equally well with generic or self-verifying data. Existing protocols require more servers for generic data (second column of Table 1). Our lower bound of $3f + 1$ servers applies regardless of whether data is generic or self-verifying. Therefore our SBQ-L protocol, already optimal for generic data, cannot be improved by using self-verifying data. This analysis suggests that the distinction between these two classes of protocols is not as fundamental as previous results imply.

We also examine the distinction between protocols with *confirmable* writes and those with *non-confirmable* writes. Consistency semantics are defined in terms of conditions that must hold when reads and writes complete; however the specification for when a write completes is left out of the definition. The traditional approach defines the completion of the write as the instant when the writer completes its protocol. We call these protocols confirmable. If instead write completion is defined in a way that cannot be locally determined by the

[1] We call this communication model "Listeners" because of its similarity with the Listeners object-oriented pattern introduced by Gamma et. al. [8].

writer, but writes are still guaranteed to eventually complete, we say that the resulting protocol is non-confirmable.

The bottom two lines of Table 1 indicate that the SBQ-L protocol can be modified to be non-confirmable. In that configuration, it can provide regular semantics for generic data using only $2f+1$ servers instead of the $3f+1$ required in prior work [15]. We again show that our protocol is optimal by proving that $2f+1$ servers are required to provide even safe semantics for non-confirmable writes. The existence of the SBQ-L protocol shows that this bound is tight. This result shows that the distinction between confirmable and non-confirmable protocols is fundamental.

Table 1. *Required number of servers and semantics for various protocols for Byzantine distributed shared memory. New results and improvements over previous protocols are shown in bold*

Required Semantics	Existing Protocols	SBQ-L	Safe Semantics
conf., generic	$4f+1$, safe [11, 12][2],[15][1] $4f+1$, partial-atomic [17][2]	**$3f+1$, atomic[2]**	**$\geq 3f+1$ servers**
conf., self-verifying	$3f+1$, regular [11],[15][1]; $3f+1$, atomic [12],[5][1,2]	$3f+1$, atomic[2]	
non-conf., generic	$3f+1$, safe [15]	**$2f+1$, regular[2]**	**$\geq 2f+1$ servers**
non-conf., self-verifying	$2f+1$, regular [15]	$2f+1$, regular[2]	

(1) Does not require reliable channels. (2) Tolerates faulty clients.

Like other quorum protocols, SBQ-L guarantees correctness by ensuring that reads and writes intersect in a sufficient number of servers. SBQ-L differs from many traditional quorum protocols in that in a minimal-server threshold configuration, clients send messages to all servers on read and write operations.[2] Most existing quorum protocols access a subset of servers on each operation for two reasons: to tolerate server faults and reduce load. Note that SBQ-L's fault tolerance and load properties are similar to those of existing protocols. In particular, SBQ-L can tolerate f faults, including f non-responsive servers. Although in its minimal-server configuration it sends read and write requests to all $3f+1$ servers, this number is no higher than the $3f+1$ (out of $4f+1$) servers contacted by most existing protocols. We note that the fact that SBQ-L contacts a large fraction of servers on each operation is a direct consequence of the minimality of the number of servers.

The rest of this paper is organized as follows. Section 2 presents our model and assumptions and reviews the different semantics that distributed shared memory can provide, Section 3 presents the SBQ-L protocol, and Section 4

[2] As described in Section 3, it is possible to use more servers than the minimum and in this case only a subset of the servers is touched for every read.

proves bounds on the number of servers required to implement these semantics. In Section 5, we explore practical considerations, including how to tolerate faulty clients, the trade-offs between bandwidth and concurrency, and how to avoid live-lock or memory problems during concurrent execution. Section 6 discusses related work and we conclude in the last section.

2 Preliminaries

2.1 Model

We assume a system model commonly adopted by previous work in which quorum systems are used to tolerate Byzantine faults [1, 3, 11–13]. In particular, our model consists of an arbitrary number of clients and a set U of data servers such that the number $n = |U|$ of servers is fixed. A *quorum system* $Q \subseteq 2^U$ is a non-empty set of subsets of U, each of which is called a *quorum*.

Servers can be either *correct* or *faulty*. A correct server follows its specification; a faulty server can arbitrarily deviate from its specification. Following Malkhi and Reiter [11], we define a *fail-prone system* $\mathcal{B} \subseteq 2^U$ as a non-empty set of subsets of U, none of which is contained in another, such that some $B \in \mathcal{B}$ contains all faulty servers. Fail-prone systems can be used to describe the common *f-threshold* assumption that up to a threshold f of servers fail (in which case $\mathcal{B}$ contains all sets of f servers), but they can also describe more general situations, such as when some computers are known to be more likely to fail than others.

The set of clients of the service is disjoint from U and clients communicate with servers over point-to-point channels that are authenticated, reliable, and asynchronous. We discuss the implications of assuming reliable communication under a Byzantine failure model in detail in our previous work [15]. Initially, we restrict our attention to server failures and assume that clients are correct. We relax this assumption in Section 5.1.

2.2 Consistency Semantics

Consistency semantics define system behavior in the presence of concurrency. Lamport [9] defines the three semantics for distributed shared memory listed below. His original definitions exclude concurrent writes, so we present extended definitions that include these [17].

Using a global clock, we assign a time to the *start* and *end* (or completion) of each operation. We say that an operation A *happens before* another operation B if A ends before B starts. We then require that all operations be totally ordered using a relation $\rightarrow$ (*serialized order*) that is consistent with the partial order of the *happens before* relation. In this total order, we call write w the *latest completed write* relative to some read r if $w \rightarrow r$ and there is no other write w' such that $w \rightarrow w' \wedge w' \rightarrow r$. We say that two operations A and B are *concurrent* if neither A happens before B nor B happens before A. The semantics below hold if there exists some relation $\rightarrow$ that satisfies the requirements.

- *safe* semantics guarantee that a read r that is not concurrent with any write returns the value of the latest completed write relative to r. A read concurrent with a write can return any value.
- *regular* semantics provide safe semantics and guarantee that if a read r is concurrent with one or more writes, then it returns either the latest completed write relative to r or one of the values being written concurrently with r.
- *atomic* semantics provide regular semantics and guarantee that the sequence of values read by any given client is consistent with the global serialization order ($\rightarrow$).

The above definitions do not specify when the write completes. The choice is left to the specific protocol. In all cases, the completion of a write is a well-defined event. We will begin by considering only protocols in which the writer can determine when its write has completed (confirmable protocols). We later relax this requirement in Section 3.2 and show that the resulting protocols with non-confirmable writes require fewer servers.

3 The SBQ-L Protocol

Figure 1 presents the f-threshold [3] SBQ-L confirmable client protocol for generic data. The initial values of the protocol's variables are shown in Figure 2.

In lines W1 through W6, the Write() function queries a quorum of servers in order to determine the new timestamp. The writer then sends its timestamped data to all servers at line W8 and waits for acknowledgments at lines W9 and W10. The Read() function queries an access set [4] of servers in line R2 and waits for messages in lines R3 to R13. An unusual feature of this protocol is that servers send more than one reply if writes are in progress. For each read in progress, a reader maintains a matrix of the different answers and timestamps from the servers (`answers[][]`). The read decides on a value at line R13 once the reader can determine that a quorum of servers vouch for the same data item and timestamp, and a notification is sent to the servers at line R14 to indicate the completion of the read. A naïve implementation of this technique could result in the client's memory usage being unbounded; instead, the protocol only retains at most $f + 1$ answers from each server.

This protocol differs from previous Byzantine quorum system (BQS) protocols because of the communication pattern it uses to ensure that a reader receives a sufficient number of *sound* and *timely* values. A reader receives different values from different servers for two reasons. First, a server may be faulty and supply incorrect or old values to a client. Second, correct servers may receive concurrent read and write requests and process them in different orders.

[3] We describe the more general quorum SBQ-L protocols in the extended technical report [14].

```
W1   Write(D) {
W2          send (QUERY_TS) to all servers
W3          receive answer (TS, ts) from server isvr set ts[isvr] := ts
W4          wait until the ts[ ] array contains q_w answers.
W5          max_ts := max{ts[ ]}
W6          ts := min{t ∈ T_c : max_ts < t ∧ last_ts < t}
            // ts ∈ T_c is larger than all answers and previous timestamp
W7          last_ts := ts
W8          send (STORE, D, ts) to all servers.
W9          receive answer (ACK,ts) from server i
W10         wait until q_w servers have sent an ACK message
W11  }

R1   (D,ts) = Read() {
R2          send (READ) to q_r servers.
R3          loop {
R4                  receive answer (VALUE, D, ts) from server s
                     // (possibly more than one answer per server)
R5                  if ts > latest[s].ts then latest[s] := (D, ts)
R6                  if s ∉ S: // we call this event an "entrance"
R7                          S := S ∪ {s}
R8                          T := the f + 1 largest timestamps in latest[ ]
R9                          for all isvr, for all jtime ∉ T, delete answer[isvr, jtime]
R10                         for all isvr,
R11                                 if latest[isvr].ts ∈ T
                                     then answer[isvr, latest[isvr].ts] := latest[isvr]
R12                 if ts ∈ T then answer[s, ts] := (D, ts)
R13         } until ∃D, ts, W :: |W| ≥ q_w ∧ (∀i : i ∈ W : answer[i, ts] = (D, ts))
            // i.e., loop until q_w servers agree on a (D,ts) value
R14         send (READ_COMPLETE) to all servers
R15         return (D, ts)
R16  }
```

Fig. 1. *Confirmable SBQ-L client protocol for the f-threshold error model*

variable	initial value	notes
q_w	$\lceil \frac{n+f+1}{2} \rceil$	Size of the write quorum
q_r	$\lceil \frac{n+3f+1}{2} \rceil$	Size of the read quorum
T_c	Set of timestamps for client c	The sets used by different clients are disjoint
$last\_ts$	0	Largest timestamp written by a particular server
$latest[\,]$	$\emptyset$	A vector storing the largest timestamp received from each server and the associated data
$answer[\,][\,]$	$\emptyset$	Sparse matrix storing at most $f+1$ data and timestamps received from each server
S	$\emptyset$	The set of servers from which the reader has received an answer

Fig. 2. *Client variables*

Traditional quorum systems use a fixed number of rounds of messages but communicate with quorums that are large enough to guarantee that intersections of read and write quorums contain enough answers for the reader to identify a value that meets the consistency guarantee of the system. Rather than using extra servers to disambiguate concurrency, SBQ-L uses extra rounds of messages when servers and clients detect writes concurrent with reads. Intuitively, other protocols take a "snapshot" of the situation. The SBQ-L protocol looks at the evolution of the situation in time: it views a "movie".

SBQ-L's approach uses more messages than some other protocols. Other than the single additional READ_COMPLETE message sent to each server at line R14, however, additional messages are only sent when writes are concurrent with a read.

Figure 1 shows the protocol for clients. Servers follow simpler rules: they only store a single timestamped data version, replacing it whenever they receive a STORE message with a newer timestamp. When receiving a read request, they send the contents of this storage. Servers in SBQ-L differ from previous protocols in what we call the Listeners communication pattern: after sending the first message, the server keeps a list of clients who have a read in progress. Later, if they receive a STORE message, then in addition to the normal processing they echo the contents of the store message to the "listening" readers – including messages with a timestamp that is not as recent as the data's current one but more recent than the data's timestamp at the start of the read. This listening process continues until the server receives a READ_COMPLETE message from the client indicating that the read has completed.

This protocol requires a minimum of $3f + 1$ servers and provides atomic semantics with confirmable writes. We prove its correctness in the next section. Theorem 2 of Section 4 shows that $3f + 1$ is the minimal number of servers for confirmable protocols. In Section 5.1 we show how to adapt this protocol for faulty clients.

3.1 Correctness

Traditional quorum protocols abstract away the notion of group communication and only concern themselves with contacting groups of responsive servers. Instead, our protocol specifies to which group of servers the messages should be sent and waits for acknowledgments from some quorum of servers within this access group. The read protocol relies on the acknowledged messages for safety, but it also potentially relies on the messages that are still in transit, for liveness. Because the channels are reliable, we know that these messages will eventually reach their destination.

Theorem 1. *The confirmable f-threshold SBQ-L protocol provides atomic semantics.*

Lemma 1 (Atomicity). *The confirmable threshold SBQ-L satisfies atomic semantics, assuming it is live.*

The SBQ-L protocol guarantees atomic semantics, in which the writes are ordered according to their timestamps. To prove this, we show that (1) after a write for a given timestamp ts_1 completes, no read can return a value with an earlier timestamp and (2) after a client c reads a timestamp ts_1, no later read can return a value with an earlier timestamp.

(1) Suppose a write for timestamp ts_1 has completed; then $\lceil \frac{n+f+1}{2} \rceil$ servers have acknowledged the write. At least $\lceil \frac{n-f+1}{2} \rceil$ of these are correct. In the worst case, all the remaining servers can return the same stale or wrong reply to later reads. However there are only $\lceil \frac{n+f-1}{2} \rceil$ of them so they cannot form a quorum.

(2) Suppose that at some global time t_1, some client c reads timestamp ts_1. That means that $\lceil \frac{n+f+1}{2} \rceil$ servers returned a value indicating that this timestamp has been written, and again at least $\lceil \frac{n-f+1}{2} \rceil$ of these are correct: the remaining servers are too few to form a quorum. □

Lemma 2 (Liveness). *All functions of the confirmable threshold SBQ-L eventually terminate.*

For space reasons, we refer the reader to our extended technical report [14] for the proof of this lemma.

3.2 Non-Confirmable Protocol

If a protocol defines the write completion predicate so that completion can be determined locally by a writer and all writes eventually complete, we call the protocol *confirmable*. This definition is intuitive and therefore implicitly assumed in most previous work. These protocols typically implement their `Write()` function so that it only returns after the write operation has completed.

If instead a protocol's write completion predicate depends on the global state in such a way that completion cannot be determined by a client although all writes still eventually complete, then we call the protocol *non-confirmable*. Non-confirmable protocols cannot provide blocking writes. The SBQ protocol [15], for example, is non-confirmable: writes complete when a quorum of correct servers have finished processing the write. This completion event is well-defined but clients cannot determine when it happens because they lack the knowledge of which servers are faulty.

The confirmable SBQ-L protocol of Section 3 requires at least $3f + 1$ servers. This number can be reduced to $2f + 1$ if the protocol is modified to become non-confirmable. The non-confirmable protocol is presented and proven correct in the extended technical report [14].

4 Bounds

In this section, we prove lower bounds on the number of servers required to implement minimal consistency semantics (safe semantics) in confirmable protocols. The bound is $3f + 1$ and applies to any fault-tolerant storage protocol

because the proof makes no assumption about how the protocol behaves. This lower bound not only applies to quorum protocols such as SBQ-L, but also to any other fault-tolerant storage protocol, even randomized ones. Also, the bounds hold whether or not data are self-verifying. Since the SBQ-L protocol of the previous section meets this bound, we know it is tight.

In previous work, protocols using self-verifying data often require f fewer servers than otherwise [11, 15]. It was not known until now whether self-verifying data make a fundamental difference or if protocols using only generic (i.e. non-self-verifying) data could be made to perform as well. Although we show that self-verifying data has no impact on the minimal number of servers, they may be useful for other properties such as the number of messages exchanged or the ability to restrict access to the shared variables.

4.1 Confirmable Safe Semantics

Theorem 2. *In the authenticated asynchronous model with Byzantine failures and reliable channels, no live confirmable protocol can satisfy the safe semantics for distributed shared memory using $3f$ servers.*

To prove this impossibility we show that under these assumptions any protocol must violate either safety or liveness. If a protocol always relies on $2f + 1$ or more servers for all read operations, it is not live. But if a live protocol ever relies on $2f$ or fewer servers to service a read request, it is not safe because it could violate safe semantics. We use the definition below to formalize the intuition that any such protocol will have to rely on at least one faulty server.

Definition 1. *A message m is* influenced by *a server s iff the sending of m causally depends [10] on some message sent by s.*

Definition 2. *A* reachable quiet system state *is a state that can be reached by running the protocol with the specified fault model and in which no read or write is in progress.*

Lemma 3. *For all live confirmable write protocols using $3f$ servers, for all sets S of $2f$ servers, for all reachable quiet system states, there exists at least one execution in which a write is only influenced by servers in a set S' such that $S' \subseteq S$.*

By contradiction: suppose that from some reachable quiet system state all possible executions for some writer are influenced by more than $2f$ servers. If the f faulty servers crash before the write then the writer can only receive messages that are influenced by the remaining $2f$ servers and the confirmable write execution will not complete. □

Note that this lemma can easily be extended to the read protocol.

Lemma 4. *For all live read protocols using $3f$ servers, for all sets S of $2f$ servers, for all reachable quiet system states, there exists at least one execution in which a read is only influenced by servers in a set S' such that $S' \subseteq S$.*

Thus, if there are $3f$ servers, all read and write operations must at some point depend on $2f$ or fewer servers in order to be live. We now show that if we assume a protocol to be live it cannot be safe by showing that there is always some case where the read operation fails.

Lemma 5. *Consider a live read protocol using $3f$ servers. There exist executions for which this protocol does not satisfy safe semantics.*

Informally, this read protocol sometimes decides on a value after consulting only with $2f$ servers. We prove that this protocol is not safe by constructing a scenario in which safe semantics are violated.

Because the protocol is live, for each write operation there exists at least one execution e_w that is influenced by $2f$ or fewer servers (by Lemma 3). Without loss of generality, we number the influencing servers 0 to $2f-1$. Immediately before the write e_w, the servers have states $a_0 \ldots a_{3f-1}$ ("state A") and immediately afterwards they have states $b_0 \ldots b_{2f-1}, a_{2f} \ldots a_{3f-1}$ ("state B"). Further suppose that the shared variable had value "A" before the write and has value "B" after the write. If the system is in state A then all reads should return the value A; in particular this holds for the reads that influence fewer than $2f+1$ servers. Consider such a read whose execution we call e. Execution e receives messages that are influenced by servers f to $3f-1$ and returns a value for the read based on messages that are influenced by $2f$ or fewer servers; in this case, it returns A. Lemma 4 guarantees that execution e exists.

Now consider what happens if execution e were to occur when the system is in state B. Suppose also that servers f to $2f-1$ are faulty and behave as if their states were $a_f \ldots a_{2f-1}$. This is possible because they have been in these states before. In this situation, states A and B are indistinguishable for execution e and therefore the read will return A even though the correct answer is B.

□

The last two lemmas show that in the conditions given, no read protocol can be live and safe. □

4.2 Non-Confirmable Safe Semantics

For non-confirmable protocols, the minimum number of servers for safe semantics is $2f+1$ instead of $3f+1$ for confirmable protocols. We refer the reader to the extended technical report [14] for the proof.

Theorem 3. *In the reliable authenticated asynchronous model with Byzantine failures, no live protocol can satisfy the safe semantics for distributed shared memory using $2f$ servers.*

5 Practical Considerations

In the next subsections we show how to handle faulty clients, quantify the number of additional messages, experimentally measure the effect of additional messages, discuss the protocol latency, and show an upper bound on memory usage.

5.1 Faulty Clients

The protocols in the previous two sections are susceptible to faulty clients in two ways: (1) faulty clients can choose not to follow the write protocol and prevent future reads from terminating or (2) faulty clients can violate the read protocol to waste server resources. We extend the protocol to address these issues below.

Liveness. Faulty writers can prevent future read attempts from terminating by making sure that no quorum of servers has the same value (a *poisonous write*), for example by sending a different value to each server. All reads will then fail because they cannot gather a quorum of identical answers.

Poisonous writes can be prevented if clients sign their writes and servers propagate among themselves the write messages they receive. This modification ensures that the servers will reach a consistent state, it is described in more detail in the extended technical report [14].

Resource Exhaustion. A faulty reader can neglect to notify the servers that the read has completed and force the server to continue that read operation forever. The cause of the problem is that readers can cause a potentially unbounded amount of work at the servers (the processing of a nonterminating read request) at the cost of only constant work (a single faulty read request).

This attack can be rendered impractical by removing the imbalance in the protocol, forcing the readers to contact the servers periodically. The resulting protocol is always safe and relies on good network behavior for liveness. It is described in more detail in the extended technical report [14].

5.2 Additional Messages

SBQ-L's write operation requires $3n$ messages in the non-confirmable case and $4n$ messages in the confirmable case, where n is the number of servers, regardless of concurrency. This communication is identical to previous results: the non-confirmable SBQ protocol [15] uses $3n$ messages and the confirmable MR protocol [11] requires $4n$ messages.

The behavior of the SBQ-L read operation depends on the number of concurrent writes. Other protocols (both SBQ and MR) exchange a maximum of $2n$ messages for each read. SBQ-L requires up to $3n$ messages when there is no concurrency. In particular, step R14 adds a new round of messages. Additional messages are exchanged when there is concurrency because the servers

echo all concurrent write messages to the reader. If c writes are concurrent with a particular read then that read will use $3n + cn$ messages.

Even in the case of concurrency, the additional messages do not impact latency as severely as one may fear because most of them are asynchronous and unidirectional. The SBQ-L protocol will not wait for $3n+cn$ message roundtrips.

In order to experimentally test the overhead of the extra messages used to deal with concurrency in SBQ-L, we construct and test a simple prototype. These experiments are described in detail in the technical report [14]. We find that increasing concurrency has a measurable but modest effect on the read latency.

5.3 Maximum Throughput

A goal of a BQS architecture is to support a high throughput for a low system cost. Two factors affect system cost: (1) the number of different servers n and (2) the required power of these servers, dictated by the load factor and the desired throughput. The *load factor* [16] is defined as "the minimal access probability of the busiest server, minimizing over the strategies", where the strategy is the algorithm used to pick a quorum.

SBQ-L has a load factor of $\frac{1}{2n}(n + \lceil \frac{n+2f+1}{2} \rceil)$ if only non-confirmable writes are supported and $\frac{1}{2n}(n + \lceil \frac{n+3f+1}{2} \rceil)$ if confirmable writes are also supported, assuming that reads and writes occur with equal frequency. Other protocols [5, 11, 13] have a better asymptotic load factor, but either have a higher load factor for small values of n or cannot function using as few servers as SBQ-L.

A detailed comparison of cost to meet throughput goals depends on hardware costs, software costs and throughput goals and is outside of the scope of this paper. In general, when adding a server is expensive compared to buying a faster server, protocols such as SBQ-L that limit n may be economically attractive even if they increase the load factor.

5.4 Live Lock

The behavior of the protocol under heavy load must be described precisely to ensure the protocol remains live. In SBQ-L, writes cannot starve but reads can, if an infinite number of writes are in progress and if the servers always choose to serve the writes before sending the echo messages.

When serving a write request while a read is in progress, servers queue an echo message. The liveness of both readers and writers is guaranteed if we require servers to send these echoes before processing the next write request. A read will therefore eventually receive the necessary echoes to complete even if an arbitrary number of writes are concurrent with the read.

Another related concern is that of latency: can reads become arbitrarily slow? In the asynchronous model, there is no bound on the duration of reads. However, if we assume that writes never last longer than w units of time and that there are c concurrent writes, then in the worst case (taking failures into account) reads will be delayed by no more than $min(cw, nw)$. This result follows because

in the worst case, f servers are faulty and return very high timestamps so that only one row of `answer[][]` contains answers from correct servers. Also, in the worst case each entrance (line R6) occurs just before the monitored write can be read. The second term is due to the fact that there are at most n entrances.

5.5 Buffer Memory

In SBQ-L, readers maintain a buffer in memory during each read operation (the `answer[][]` sparse matrix). While other protocols only need to identify a majority and as such require n units of memory, the SBQ-L protocol maintains a short history of the values written at each server. As a result, the read operation in SBQ-L requires up to $n(f+1)$ units of memory: the set T contains at most $f+1$ elements (line 8) and the `answer[][]` matrix therefore never contains more than n columns and $f+1$ rows (lines 9 and 12). In a system storing more than one shared variable, if multiple variables are read in parallel then each individual read requires its own buffer of size $n(f+1)$.

6 Related Work

Although both Byzantine failures [7] and quorums systems [6] have been studied for a long time, interest in quorum systems for Byzantine failures is relatively recent. The subject was first explored by Malkhi and Reiter [11, 12]. They reduced the number of servers involved in communication [13], but not the total number of servers; their work exclusively covers confirmable systems.

In previous work we introduced non-confirmable protocols that require $3f+1$ servers ($2f+1$ for self-verifying data) [15]. In the present paper we expand on that work and reduce the bound to $2f+1$ for generic data and provide regular semantics instead of safe by using Listeners. We also prove lower bounds on the number of servers for these semantics and meet them.

Bazzi [3] explored Byzantine quorums in a synchronous environment with reliable channels. In that context it is possible to require fewer servers ($f+1$ for self-verifying data, $2f+1$ otherwise). This result is not directly comparable to ours since it uses a different model. We leave as future work the application of the Listeners idea of SBQ-L to the synchronous network model.

Bazzi [4] defines *non-blocking quorum system* as a quorum system in which the writer does not need to identify a live quorum but instead sends a message to a quorum of servers without concerning himself with whether these servers are responsive or not. According to this definition, all the protocols presented here use non-blocking quorum systems.

Several papers [4, 13, 16] study the load of Byzantine quorum systems, a measure of how increasing the number of servers influences the amount of work each individual server has to perform. A key conclusion of this previous work is that the lower bound for the load factor of quorum systems is $O(\frac{1}{\sqrt{n}})$. Our work instead focuses on reducing the number of servers necessary to tolerate a given fault threshold (or fail-prone system).

Phalanx [12] builds shared data abstractions and provides a locking service, both of which can tolerate Byzantine failure of servers or clients. It requires confirmable semantics in order to implement locks. Phalanx can handle faulty clients while providing safe semantics using $4f + 1$ servers.

Castro and Liskov [5] present a replication algorithm that requires $3f + 1$ servers and, unlike most of the work presented above, can tolerate unreliable network links and faulty clients. Their protocol uses cryptography to produce self-verifying data and provides linearizability and confirmable semantics. It is fast in the common case. Our work shows that confirmable semantics cannot be provided using fewer servers. Instead, we show a non-confirmable protocol with $2f + 1$ servers. In the case of non-confirmable semantics, however, it is necessary to assume reliable links.

Attiya, Bar-Noy and Dolev [2] implement an atomic single-writer multi-reader register over asynchronous network, while restricting themselves to crash failures only. Their failure model and writer count are different from ours. When implementing finite-size timestamp, their protocol uses several rounds. The similarity stops there, however, because they make no assumption of network reliability and therefore cannot leverage unacknowledged messages the way the Listeners protocol does.

7 Conclusion

We present two protocols for shared variables, one that provides atomic semantics with non-confirmable writes using $2f + 1$ servers and the other that provides atomic semantics with confirmable writes using $3f + 1$ servers. In the reliable asynchronous communication model when not assuming self-verifying data, our protocols reduce the number of servers needed by previous protocols by f. Additionally, they improve the semantics for the non-confirmable case. Our protocols are strongly inspired by quorum systems but use an original communication pattern, the Listeners. The protocols can be adapted to either the f-threshold or the fail-prone error model.

The more theoretical contribution of this paper is the proof of a tight bound on the number of servers. We show that $3f + 1$ servers are necessary to provide confirmable semantics and $2f + 1$ servers are required otherwise.

Several protocols [5, 11, 12, 15, 18] use digital signatures (or MAC) to reduce the number of servers. It is therefore surprising that we were able to meet the minimum number of servers without using cryptography. Instead, our protocols send one additional message to all servers and other additional messages that only occur if concurrent writes are in progress.

Since our protocols for confirmable and non-confirmable semantics are nearly identical, it is possible to use both systems simultaneously. The server side of the protocols are the same, therefore the servers do not need to be aware of the model used. Instead, the clients can agree on whether to use confirmable or non-confirmable semantics on a per-variable basis. The clients that choose

non-confirmable semantics can tolerate more failures: this property is unique to the SBQ-L protocol.

Acknowledgments

The authors thank Jian Yin and Mike Kistler for several interesting conversations and Alison Smith and Maria Jump for helpful comments on the paper's presentation.

References

1. L. Alvisi, D. Malkhi, E. Pierce, and R. Wright. Dynamic Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2000.
2. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM (JACM) Volume 42*, pages 124–142, 1995.
3. R. A. Bazzi. Synchronous Byzantine quorum systems. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 259–266, 1997.
4. R. A. Bazzi. Access cost for asynchronous Byzantine quorum systems. *Distributed Computing Journal volume 14, Issue 1*, pages 41–48, January 2001.
5. M. Castro and NB. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), New Orleans, USA*, pages 173–186, February 1999.
6. S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys (CSUR) Volume 17, Issue 3*, pages 341–370, September 1985.
7. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Technical Report MIT/LCS/TR-282, 1982.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, October 1994. ISBN 0-201-63361-2.
9. L. Lamport. On interprocess communications. *Distributed Computing*, pages 77–101, 1986.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
11. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pages 203–213, 1998.
12. D. Malkhi and M. Reiter. Secure and scalable replication in phalanx. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems, West Lafayette, Indiana, USA*, Oct 1998.
13. D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. In *Proceedings 16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 249–257, August 1997.
14. J-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. Technical Report TR-02-38, University of Texas at Austin, Department of Computer Sciences, August 2002.
15. J-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–383, June 2002.
16. M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.
17. E. Pierce and L. Alvisi. A recipe for atomic semantics for Byzantine quorum systems. Technical report, University of Texas at Austin, Department of Computer Sciences, May 2000.
18. R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP '01)*, October 2001.