# Minimal Byzantine Quorums

Jean-Philippe Martin, Lorenzo Alvisi, Michael Dahlin

## Abstract

Byzantine quorum systems can provide fault tolerant storage 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 Byzantine quorum 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. In our analysis of this algorithm, we (1) identify the lower bound on the number of servers required to implement distributed data services in a Byzantine environment, (2) describe new protocols that reduce the number of servers necessary for generic data while providing strong consistency semantics, and (3) show that the new protocols match the lower bounds for the number of servers and provide the best consistency semantics for this number of servers.

## 1 Introduction

Quorum systems [6] are valuable tools for implementing highly available distributed data services. These systems store a shared variable at at a set of servers and perform read and write operations at some subset of servers (a *quorum*). Each protocol defines some intersection property for the quorums which, combined with the protocol description itself, ensures that each read has access to the most recently written value of the variable. Practical use of quorum systems necessitates they enforce the intersection property even in the presence of failures. To guarantee data integrity and availability in the presence of arbitrary (*Byzantine*) failures, Malkhi and Reiter [10] introduce a special kind of quorum system called *masking quorum system*. They also introduce *dissemination quorum systems* that can be used by services that support *self-verifying data*, i.e., data that cannot be undetectably altered by a faulty server, such as data that have been digitally signed or associated with message authentication codes (MACs) [4].

The number of servers in a Byzantine quorum system is an important metric because these systems rely on server failures being independent. Therefore, to reduce the correlation of software failures, each server should use a different software implementation [16]. Reducing the number of servers therefore reduces the development and software maintenance cost of these systems in addition to lowering the hardware resource demands (e.g., processors or disks). Furthermore, for large software systems only a fixed number of implementations may be available, and it may be expensive or otherwise infeasible to create additional implementations. In such a situation, a new protocol requiring fewer servers may enable replication techniques in situations where they were not previously applicable.

To reduce the number of servers we present a BQS protocol called Small Byzantine Quorums with Listeners (SBQ-L). In this protocol we use a "Listeners" pattern of communication to detect and resolve ordering ambiguities when reads and writes simultaneously access a shared variable. Whereas other algorithms use a fixed number of communication rounds, servers and readers using SBQ-L exchange additional messages when writes are concurrent with reads. This pattern allows the reader to monitor the evolution of the global state instead of relying on a snapshot. As a result, we can provide strong consistency semantics using fewer servers than before. We call this communication model "Listeners" because of its similarity with the Listeners object-oriented pattern introduced by Gamma et. al. [8].

Our work produced three primary results, summarized in Table 1: (1) new lower bounds for the BQS problem, (2) a new SBQ-L protocol, and (3) a proof that these protocols are tight in the sense that they require the minimal number of servers.

The first contribution of this paper is the development of three new lower bounds under the common assumption of asynchronous reliable authenticated channels [1, 2, 10, 11, 12]. The first bound states that blocking protocols (defined in Section 3) require at least $3f+1$ servers to provide even the weak *safe* consistency semantics [9] in the presence of $f$ Byzantine failures. Second, non-blocking protocols require at least $2f+1$ servers to provide safe semantics. Third, non-blocking protocols with fewer than $3f+1$ servers cannot provide atomic semantics.

Our second contribution is a new protocol, SBQ-L, in two variants: blocking and non-blocking. As shown

| | Existing Protocols | SBQ-L |
|---|---|---|
| blocking, self-verifying | 3f+1, regular [10],[14][1]; 3f+1, atomic [11],[5][1,2] | 3f+1, atomic[2] *(tight)* |
| blocking, generic | 4f+1, safe [10, 11][2],[14][1] 4f+1, partial-atomic [15][2] | *3f+1, atomic[2] (tight)* |
| non-blocking, self-verifying | 2f+1, regular [14] | 2f+1, regular *(tight)* |
| non-blocking, generic | 3f+1, safe [14] | *2f+1, regular (tight)* |

(1) Does not require reliable channels.

(2) Tolerates faulty clients.

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 italics.*

in lines 2 and 4 of the table, the blocking variant of SBQ-L reduces the number of servers required for storing generic data (i.e., data that is not self-verifying) from $4f + 1$ to $3f + 1$ compared to existing protocols and the non-blocking variant reduces this number from $3f + 1$ to $2f + 1$. Furthermore, using this low number of servers we provide atomic semantics for the blocking protocol while, for generic data, previous work could only guarantee the weaker safe or partial-atomic [15] semantics. Our non-blocking protocol provides regular semantics, improving upon the safe semantics provided by existing non-blocking protocols.

Our final contribution, listed in the rightmost column of the table, is to demonstrate that our new protocol is tight with respect to both the number of servers and the consistency semantics (defined by Lamport [9]) for that number of servers. Blocking SBQ-L requires the minimal number of servers for a blocking protocol, and provides the strongest consistency semantics described by Lamport, atomic semantics. Non-blocking SBQ-L also requires the minimal number of servers for non-blocking protocols. With this number of servers, it is not possible to implement atomic semantics. Instead, non-blocking SBQ-L provides Lamport's next strongest consistency level: regular semantics.

It is surprising that SBQ-L performs equally well with generic or self-verifying data; other protocols require more servers for generic data, as the middle column of Table 1 illustrates. Conversely, this paper shows different bounds for blocking and non-blocking protocols, which suggests this latter distinction is fundamental.

The SBQ-L protocol uses communication to reduce the number of servers and improve consistency semantics, but this additional communication is a potential disadvantage of the SBQ-L protocol. Fortunately, it is limited to one message per server for each read when there is no concurrency; if concurrency is present, then the number of additional messages per server is proportional to the number of concurrent writes. Section 8 presents measurements of the latency increase due to concurrent writes.

The rest of this paper is organized as follows: Section 2 presents our model and assumptions. Section 3 reviews the different semantics that distributed shared memory can provide, and Section 4 proves bounds on the number of servers required to implement these semantics. Both the non-blocking and the blocking protocols are presented in Section 5, as well as an extension of the blocking protocol to prevent them. Section 6 proves the correctness of both protocols. Section 7 discusses the trade-offs between bandwidth and concurrency and Section 8 presents experiments that quantify this trade-off in a working prototype. Section 9 discusses related work and we conclude in the last section.

## 2 Model

We assume a system model commonly adopted by previous works that have applied quorum systems in the Byzantine failure model [1, 2, 10, 11, 12].

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 [10], 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, as when some computers are known to be more likely to fail than others.

The set of clients of the service is disjoint from $U$. Initially, we restrict our attention to server failures and assume that clients are correct. We relax this assumption in Section 5.3. Clients communicate with servers over point-to-point channels that are authenticated, reliable, and asynchronous.

## 3 Consistency Semantics

Consistency semantics define system behavior in the presence of concurrency. We first review Lamport's definitions of safe, regular, and atomic semantics. In the course of our work we find that the distinction between blocking and non-blocking writes is important.

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 [15].

We assign a time, using a global clock, 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 writes be totally ordered using a relation $\rightarrow$ (*serialized order*) that is consistent with the *happens before* relation. In this total order, we call write $w$ the *latest completed write* if there is no other write $w'$ such that $w \rightarrow w'$.

- *safe* semantics guarantee that a read that is not concurrent with any write returns the value of the latest completed write. A read concurrent with a write can return any value.

- *regular* semantics provide safe semantics and guarantee that if a read is concurrent with one or more writes then it returns either the latest completed write or one of the values being written concurrently.

- *atomic* semantics provide regular semantics and guarantee that clients see writes in an order consistent with $\rightarrow$.

The above definitions do not specify when write completion occurs; the choice is left to the specific protocol. In all cases, the completion of a write is a well-defined event. The definition of the write completion predicate influences the properties of the resulting protocol.

If the protocol defines the write completion predicate so that completion can be determined locally by a writer, we call the protocol *blocking* and we say it supports *blocking safe, regular or atomic semantics.* This definition is intuitive and therefore implicitly assumed in most previous work. These protocols typically implement a *blocking write*, in which the `Write()` function only returns after the write operation has completed. Note that blocking protocols may also choose to implement a non-blocking write operation and provide a separate mechanism (e.g., a barrier) to let the client determine when a write completes.

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, then we call the protocol *non-blocking* and say that it supports only *non-blocking semantics.* Non-blocking protocols cannot provide blocking writes. The SBQ protocol [14], for example, is non-blocking: 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.

As an example of a system where a non-blocking protocol is sufficient, consider a network of sensors measuring some value and writing it to the distributed shared memory. The reader always wants the most recent available value that corresponds to the physical situation and does not care if a particular write has completed. Also, it is acceptable for some writes to be replaced with a newer value before they are ever read. Therefore no sensor should wait for the completion of its last write before writing a newer measured value, and non-blocking semantics are appropriate.

## 4 Bounds

In this section we prove lower bounds on the number of servers required to implement minimal consistency semantics (safe semantics) for blocking and non-blocking writes. The bound for blocking protocols is $3f + 1$ and that for non-blocking protocols is $2f + 1$. We also show that atomic semantics cannot be achieved with fewer than $3f + 1$ servers, so minimal non-blocking protocols cannot provide atomic semantics.

These bounds indicate that a trade-off exists between the number of servers and blocking semantics. In cases where the writer does not need to determine when its writes completes non-blocking protocols reduce the number of servers.

In Section 5 we present protocols that meet the bounds presented in this section.

### 4.1 Non-Blocking Protocols

**Theorem 1.** *In the reliable authenticated asynchronous model with Byzantine failures, no live protocol can satisfy the safe semantics for distributed shared memory using* $2f$ *servers.*

To prove this impossibility, we show that under these assumptions any protocol must violate either safety or liveness.

**Definition 1.** *A message $m$ has* gone through *server $s$ if the sending of $m$ causally depends on some message sent by $s$.*

**Lemma 1.** *A read protocol in which all executions must receive messages that have gone through at least $f + 1$ distinct servers before the read can complete is not live.*

If servers 0 to $f - 1$ crash then no message can go through them. In this case, the reader can only receive messages that have gone through the remaining $f$ servers and not every read can complete. □

**Lemma 2.** *A read protocol in which there exists an execution $e$ where the received messages have only gone through $f$ or fewer distinct servers does not satisfy safe semantics.*

Consider a system in which the servers have states $a_0 \ldots a_{2f-1}$, the shared variable has value A, and a read for the variable follows execution $e$ with non-zero probability. Execution $e$ returns a value for the read based on messages that have gone through $f$ or fewer servers and so returns A. Without loss of generality, suppose $e$ only receives messages that have gone through servers 0 to $f - 1$. Suppose that a later write changes the value of the variable to B, and a subsequent read request reaches the servers. Further suppose that servers 0 to $f - 1$ are faulty and behave as if their state were $a_0 \ldots a_{f-1}$. This is possible because they have been in these states before. The faulty servers then send the same answers as they would have when the variable had value A. Formulating these answers must not require communication with the other servers; otherwise, these messages would have gone through more than $f$ servers. With these answers, there is a non-zero probability that the reader follows execution $e$ and decides on the incorrect value A. □

The proof of the theorem derives from the fact that the two lemmas cover all possible protocols. □

Note that the proof is not limited to the $f$-threshold model and makes no assumption of deterministic behavior from the protocol. The proof also covers protocols which use integrity checks in their messages since faulty servers have all the necessary information to create the messages they send.

### 4.2 Blocking Protocols

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

Theorem 2's proof is similar to Theorem 1's. We state the main lemmas here. The full proof appears in the Appendix of the extended technical report [13].

**Lemma 3.** *A read protocol in which all executions must receive messages that have gone through at least $2f + 1$ distinct servers before the read can complete is not live.*

**Lemma 4.** *A read protocol in which there exists an execution $e$ where the received messages have only gone through $2f$ or fewer distinct servers does not satisfy safe semantics.*

### 4.3 Atomic Semantics

Blocking semantics require $3f + 1$ servers. We now show that this same limit of $3f + 1$ servers is the minimum number of servers necessary for implementing atomic semantics, even for a non-blocking protocol.

**Theorem 3.** *In the reliable authenticated asynchronous model with Byzantine failures, no protocol for distributed shared memory can implement atomic semantics with fewer than $3f + 1$ servers.*

To prove this by contradiction, suppose there exists some protocol $p$ that implements atomic semantics using fewer than $3f+1$ servers. $p$ can be used to implement the following blocking protocol: when writing some value $x$, repeatedly read until the read function returns $x$ or a more recent value. Then, write another token value $t$. Finally, repeatedly read until the read function returns $t$ or some other more recent value.

The two read sections in this example program show that in the serialized order, $x$ must come before $t$. Because $p$ is atomic and $t$ has been read, no other client can possibly read $x$ anymore. Therefore, it is not necessary to continue the write for $x$; it has effectively completed. This is a contradiction because no blocking protocol can exist with fewer than $3f + 1$ servers due to Theorem 2. □

A simplified proof writing only value $x$ but not $t$ would not be sufficient because knowing that the writer can read its own value does not necessarily mean that other clients cannot read older values.

## 5 The SBQ-L Protocols

In this section we show the $f$-threshold non-blocking and blocking versions of the SBQ-L protocol[1]. SBQ-L is based both on the insights from the non-blocking

[1]The more general version (using the fail-prone model) is presented in the Appendix to the extended technical report [13].

"Small Byzantine Quorums" (SBQ) protocol [14] and on the Listeners communication pattern. The Listener pattern is the key idea behind our protocol, allowing it to use fewer servers than other protocols while providing stronger semantics for generic data.

## 5.1 Blocking Protocol

Figure 1 presents the $f$-threshold SBQ-L blocking protocol for generic data. The initial value of the protocol's variables is 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 wait for acknowledgments at lines W9 and W10. The Read() function queries all 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 naive 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. We show in Section 6 that the read protocol is correct.

This protocol differs from previous protocols because of its communication pattern. Intuitively, other protocols take a "snapshot" of the situation. The SBQ-L protocol looks at the evolution of the situation in time: it records a "movie". This communication makes it possible to disambiguate situations where concurrent writes are such that no majority emerges immediately. Our approach causes the server to send more messages than in other protocols, however, other than the single additional READ_COMPLETE message sent to each server at line R14, 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 the READ_COMPLETE message from the client indicating that the read has completed. Note that in practice these messages would only be sent if the writer is authorized to modify that variable. Also, they need only be sent to readers accessing the variable being written.

This protocol requires a minimum of $3f+1$ servers and provides blocking atomic semantics. We prove its correctness in Section 6. As shown in Theorem 2, $3f+1$ is the optimal number of servers for blocking protocols.

## 5.2 Non-Blocking Protocol

The blocking SBQ-L protocol of the previous section requires at least $3f+1$ servers. This number can be reduced to $2f+1$ if the protocol is modified to become non-blocking.

Since in a non-blocking protocol the writer is not required to know when the write completes, we can remove lines W9 and W10 of the Write() function in which the writer waits for acknowledgments. The STORE messages sent earlier (at line W8) are guaranteed to reach their destination because we assume that the channels are reliable. The reader can find a discussion of the implications of assuming reliable links in Byzantine environments in our previous work [14].

We then modify the size of the quorums, $q$, to $\lceil \frac{n+1}{2} \rceil$ instead of $\lceil \frac{n+f+1}{2} \rceil$ previously. This is possible because eliminating the acknowledgments eliminates a constraint on the overlap of read and write quorums [14].

Recall that in non-blocking protocols, the write function does not determine when the write has completed: instead, the completion must be specified by the protocol. We therefore specify that the write completes when $q = \lceil \frac{n+1}{2} \rceil$ correct servers are done processing the STORE message. Note that this definition ensures that write completion cannot be unduly delayed by the actions of faulty servers in that they cannot delay writes more than crashed servers would.

This protocol requires only $2f+1$ servers and provides regular semantics. We prove the correctness of this protocol in Section 6. As shown in Theorem 1, $2f+1$ is the optimal number of servers for non-blocking protocols.

Pierce [15] presents a general technique to transform any regular protocol into one that satisfies atomic semantics. This technique, however, only works for blocking protocols and therefore does not apply to this case.

```
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 $q$ answers.
W5        $max\_ts := max\{ts[\,]\}$
W6        $ts := min\{t \in T_c : max\_ts < t \wedge last\_ts < t\}$
          // $ts \in T_c$ 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 $q$ servers have sent an ACK message
W11   }
R1    (D,ts) = Read() {
R2        send (READ) to all 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 $s \notin S$: // we call this event an "entrance"
R7                $S := S \cup \{s\}$
R8                $T :=$ the $f + 1$ largest timestamps in $latest[\,]$
R9                for all $isvr$, for all $jtime \notin T$, delete $answer[isvr, jtime]$
R10               for all $isvr$,
R11                   if $latest[isvr].ts \in T$ then $answer[isvr, latest[isvr].ts] := latest[isvr]$
R12           if $ts \in T$ then $answer[s, ts] := (D, ts)$
R13       } until $\exists D, ts, S :: |S| \geq q \wedge (\forall i : i \in S : answer[i, ts] = (D, ts))$
          // i.e., loop until $q$ servers agree on a (D,ts) value
R14       send (READ_COMPLETE) to all servers
R15       return $(D, ts)$
R16   }
```

Figure 1: *Blocking SBQ-L protocol for the f-threshold error model.*

| variable | initial value | notes |
|---|---|---|
| $q$ | $\lceil \frac{n+f+1}{2} \rceil$ or $\lceil (n + 1)/2 \rceil$ | Quorum size in the blocking and non-blocking case, respectively |
| $T_c$ | 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 |

Figure 2: *Variables*

## 5.3 Faulty Clients

The protocols in the previous two sections are susceptible to faulty clients: by sending a different value to each server (a "poisonous write"), a faulty writer can prevent future read attempts from terminating because no read can gather a quorum of identical answers.

Poisonous writes are a common vulnerability for Byzantine storage protocols and, in the case of the blocking protocol, we can adopt the technique introduced by Malkhi and Reiter [10] to handle faulty clients. This technique adds a two-phase commit to the STORE function at the servers that makes sure that the same value is also being written to other servers. These writes are therefore guaranteed to not put the system in an incorrect state where reads would hang. Furthermore, the correctness proof in Section 6 still holds with this modification.

The other difficulty is that a faulty reader can neglect to notify the servers that the read has completed and therefore prevent the read from terminating. This is identical to faulty readers that, in other protocols, continuously send read requests. We are exploring ways to bound the number of echoes that a single read request can trigger.

# 6 Correctness

In this section we prove the correctness of the two $f$-threshold versions of the SBQ-L protocol presented in this paper.

## 6.1 Blocking Threshold SBQ-L

**Theorem 4.** *The blocking $f$-threshold SBQ-L protocol provides atomic semantics.*

**Lemma 5 (Regularity).** *The blocking $f$-threshold SBQ-L satisfies regular semantics, assuming it is live.*

In the case where no concurrent write is in progress, the reader eventually receives an answer from the $\lceil\frac{n+f+1}{2}\rceil$ correct servers and decides on their value (step R13). There are not enough other servers to make the reader decide on any other value.

If a write is in progress and the reader decides on a value then this value has been vouched for by $q_r = \lceil\frac{n+f+1}{2}\rceil$ servers (line R13). By definition, $q_w = \lceil\frac{n+f+1}{2}\rceil$ servers have seen the latest completed write. Since $q_r + q_w > n + f$, these two quorums intersect in at least one correct server $C$ that has seen the latest completed write.

Since $C$ is correct, it follows the protocol and therefore it sent the value of the completed write, the value of a write with a higher timestamp, or both. □

**Lemma 6 (Atomicity).** *The blocking threshold SBQ-L satisfies atomic semantics, assuming it is live.*

The blocking version of the protocol provides stronger semantics than the non-blocking. It guarantees atomic semantics, in which the writes are ordered according to their timestamps. To prove this, we show that after a write for a given timestamp $ts_1$ completes, no read can decide on a value with an earlier timestamp.

Suppose a write with timestamp $ts_1$ has completed; then $\lceil\frac{n+f+1}{2}\rceil$ servers agree on this timestamp. Even if the faulty ($f$) and untimely ($u$) servers send the same older reply $ts_0$, they cannot form a quorum ($q$). More formally:

$$f + u < q$$

$$\iff \quad f + n - \lceil\frac{n+f+1}{2}\rceil < \lceil\frac{n+f+1}{2}\rceil$$

$$\Leftarrow \; f + n < 2\lceil\frac{n+f+1}{2}\rceil \; \Leftarrow \; f + n < f + n + 1$$

□

**Lemma 7 (Liveness).** *All functions of the non-blocking threshold SBQ-L eventually terminate.*

**Write**. The Write() function is trivially live because its waits (in steps W4 and W10) expect $q = \lceil(n+f+1)/2\rceil$ answers and $q \leq n - f$ so these answers are guaranteed to eventually arrive.

**Read**. Even though it only tracks $f + 1$ different timestamps simultaneously (lines R11 and R12), the Read() function is live. Consider the last entrance, i.e., the last time line R7 of Read() is executed. The `latest[]` array contains a value for each server; consider the largest `latest[].ts` associated with a correct server, $ts_{max}$. The client has not discarded any data item with timestamp $ts_{max}$ coming from a correct server (otherwise that correct server would have a higher timestamp associated with it). $ts_{max}$ is in $T$ because $T$ contains the $f + 1$ largest timestamps in `latest[]`. Since all clients are correct, they send the same value to all servers and therefore all correct servers will eventually see the write with timestamp $ts_{max}$ and will echo it to the reader. As we know, $q \leq n - f$ so there are enough correct servers to guarantee that the read for that timestamp will eventually complete.

This proof also illustrates the benefits of the Listeners communication: if several writes are in progress, then initially each server could hold a different timestamp. The ongoing communication allows the reader to follow the writes and identify the correct value.

**STORE, QUERY_TS**. The server's STORE and QUERY_TS functions terminate because they have no loops.

**READ**. The server's READ function terminates because the client's Read() terminates and clients are correct. □

This concludes the correctness proof. We have shown that the protocol always returns a correct value and that it terminates. Note that it could terminate before the events we describe in the proof; we merely show that the protocol eventually terminates.

## 6.2 Non-Blocking Threshold SBQ-L

**Theorem 5.** *The non-blocking threshold SBQ-L protocol provides regular semantics.*

The proof for liveness is identical to the proof for the blocking case above. The regularity proof is similar to that of the blocking case, using the smaller quorum size: The read protocol decides (at line R13) on a value that is vouched for by $q_r = \lceil \frac{n+1}{2} \rceil$ servers. By definition of completion, $q_w = \lceil \frac{n+1}{2} \rceil$ *correct* servers have seen the latest completed write. Since $q_r + q_w > n$, these two quorums intersect in at least one correct server $C$ that has seen the latest completed write. □

# 7 Practical Considerations

Our Listener mechanism allows the SBQ-L protocols to use the optimal number of servers but (1) the communication pattern it requires causes more messages to be exchanged than in other protocols, (2) the read protocol does not decide on a value immediately, and (3) the reader stores messages in memory before deciding. In the next three subsections we quantify the number of additional messages, discuss the protocol latency, and show an upper bound on memory usage.

## 7.1 Additional Messages

The read protocol may wait for several messages before deciding on a value. The write protocol suffers from no such wait: writes always require the same number of messages, regardless of the level of concurrency. SBQ-L's write operation requires $3n$ messages in the non-blocking case and $4n$ messages in the blocking case, where $n$ is the number of servers. This communication is identical to previous results: the non-blocking SBQ protocol [14] uses $3n$ messages and the blocking MR protocol [10] 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.

For some systems, there is little or no concurrency in the common case. Even with additional messages in the case of concurrency, the latency increase is not as severe as one may fear because most of these message exchanges are asynchronous and unidirectional: the SBQ-L protocol will not wait for $3n + cn$ message roundtrips. This is apparent in the experimental results of Section 8.

## 7.2 Live Lock

In a system such as SBQ-L, it must be ensured that both reads and writes will complete even if the system is under a heavy load. In SBQ-L, writes cannot starve because their operation is independent of concurrent reads. Reads, however, can be starved if an infinite number of writes are in progress and if the servers always choose to serve the writes before sending the echo messages.

There is an easy way to guarantee this does not happen. 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 bound follows from the fact that there are at most $n$ entrances.

## 7.3 Buffer Memory

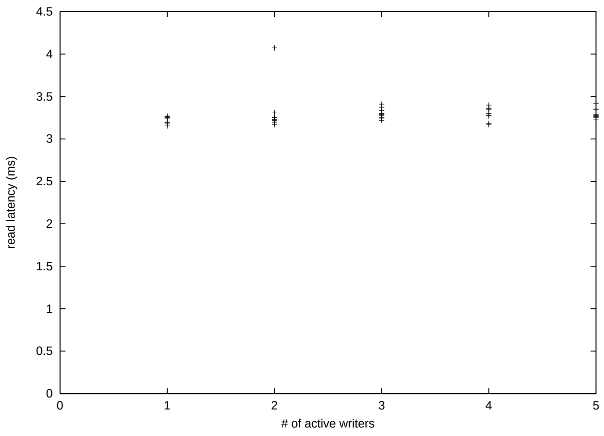
In SBQ-L, readers maintain a buffer in memory during each read operation (the `answer[]` sparse matrix). While other protocols only need to identify a majority and as such require $n$ units of memory, the SBQ-L protocol maintains a short history of the values written at each server. As a result, the 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 $n(f+1)$.

## 8 Experiments

We construct a simple prototype to study the overhead of the extra messages used to deal with concurrency in SBQ-L. The prototype is written in C++, stores data in main memory and communicates via TCP.

Our testbed consists of 3 servers and 6 client machines, 5 of which act as writers and 1 as a reader. The reader machine is a SUN Ultra10 with a 440Mhz UltraSPARC-IIi processor running SunOS 8.5. The other machines are Dell Dimension 4100 with a 800Mhz PentiumIII processor running Debian Linux 2.2.19. The network connecting these machines is a 100Mbits/s switched Ethernet.

In this experiment, we vary the number of writers and therefore the level of concurrency. The writers repeatedly execute the non-blocking write protocol, writing 1000 bytes of data to all servers. The reader measures the average time for 20 consecutive reads, and the servers are instrumented to measure the number of additional messages sent during the Listeners phase.



The above graph shows the read latency as a function of the number of active writers. Each point represents the average duration of 20 reads.

We find, as expected, that increasing concurrency has a measurable but modest effect on the latency of the reads.

## 9 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 [10, 11]. They reduced the number of servers involved in communication [12], but not the total number of servers; their work exclusively covers blocking systems.

In previous work we introduced non-blocking protocols that require $3f+1$ servers ($2f+1$ for self-verifying data) [14]. 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 [2] 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.

Several papers [3, 12] 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 (if at all). Although these considerations are interesting, our motivation for adding servers to the system is to increase reliability, not performance: therefore we leave considerations of the load of servers under the SBQ-Listeners protocols as future work.

Bazzi [3] uses the term *non-blocking quorum system* to mean something different from what we call *non-blocking protocols.* Recall that a non-blocking protocol is one where a client cannot determine when its writes complete. A non-blocking quorum system is 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, both the blocking and the non-blocking SBQ-L protocols use non-blocking quorum systems.

Phalanx [11] builds shared data abstractions and provides a locking service, both of which can tolerate Byzantine failure of servers or clients. It requires blocking 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 (therefore blocking semantics); their protocol is fast in the common case. Our work shows that blocking semantics cannot be provided using fewer servers. Instead, we show a non-blocking protocol with $2f+1$ servers. In the case of non-blocking semantics, however, it is necessary to assume reliable links.

## 10 Conclusion

We present two protocols for shared variables, one that provides non-blocking regular semantics using $2f+1$ servers and the other that provides blocking atomic semantics using $3f+1$ servers. This reduces by $f$ the number of servers needed by previous protocols in the reliable asynchronous communication model when not assuming self-verifying data. 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 blocking semantics and $2f+1$ servers are required otherwise. We further show that $3f+1$ servers are required for atomic semantics even for non-blocking protocols.

Several protocols [5, 10, 11, 14, 16] 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 blocking and non-blocking 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 blocking or non-blocking semantics on a per-variable basis. The clients that choose non-blocking semantics can tolerate more failures: this property is unique to the SBQ-L protocol.

## 11 Acknowledgments

The authors would like to thank Jian Yin for several very interesting conversations and Alison Smith 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] R. A. Bazzi. Synchronous Byzantine quorum systems. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 259–266, 1997.

[3] R. A. Bazzi. Access cost for asynchronous Byzantine quorum systems. *Distributed Computing Journal volume 14, Issue 1*, pages 41–48, January 2001.

[4] M. Castro and B. Liskov. Authenticated Byzantine fault tolerance without public-key cryptography. Technical Report /LCS/TM-595, MIT, 1999.

[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] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pages 203–213, 1998.

[11] 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.

[12] 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.

[13] J-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine quorum systems. Technical report, University of Texas at Austin, Department of Computer Sciences, February 2002. www.cs.utexas.edu/home/department/pubsforms.shtml.

[14] J-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. Technical Report TR-02-01, University of Texas at Austin, Department of Computer Sciences, January 2002.

[15] 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.

[16] 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.

## A Lower Bound for Blocking Protocols

Expanding on the work of Section 4, we prove the lower bound for the number of servers for blocking protocols.

**Theorem 2.** *In the reliable authenticated asynchronous model with Byzantine failures, no live protocol can satisfy the safe blocking 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.

**Lemma 3.** *A read protocol in which all executions must receive messages that have gone through at least $2f+1$ distinct servers before the read can complete is not live.*

If the servers 0 to $f-1$ crash then no message can go through them. In this case, the reader can only receive messages that have gone through the remaining $2f$ servers and the reads cannot all complete. □

**Lemma 4.** *A read protocol in which there exists an execution $e$ where the received messages have only gone through $2f$ or fewer distinct servers does not satisfy safe semantics.*

Since the write is blocking, the writer must be able to determine the completion of the write from messages that have gone through at most $2f$ servers in order to be live. Consider a system in which the servers have state $a_0 \ldots a_{3f-1}$, the shared variable has value A, and a read for the variable follows execution $e$ with non-zero probability. Execution $e$ receives messages from servers $f$ to $3f-1$ and returns a value for the read based on messages that have gone through $2f$ or fewer servers; in this case, it returns A. Suppose that a later write changes the value of the variable to B. Suppose therefore that after completion of the writes the servers are in states $b_0 \ldots b_{2f-1}, a_{2f} \ldots a_{3f-1}$. Further suppose 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. Suppose that the reader receives answers from servers $f$ to $3f-1$. The faulty servers send the same answers as they would have if the variable had value A. Formulating these answers must not require communication with the other servers because otherwise these messages would have gone through more than $2f$ servers. In this situation, there is a non-zero probability that the reader follows execution $e$ and decides on the incorrect value A. □

The proof of the theorem derives from the fact that the two lemmas cover all possible protocols. □

## B Generalized Blocking Protocol

The blocking protocol can be generalized to a fail-prone system instead of the simpler $f$-threshold case presented in Section 5. Our quorums $Q \in \mathcal{Q}$ must obey the following properties:

**Consistency**: The intersection of any pair of quorums contains one correct server.

$$\forall Q_1, Q_2 \in \mathcal{Q}\ \forall B \in \mathcal{B} : Q_1 \cap Q_2 \not\subseteq B$$

**Availability**: One quorum is always available.

$$\forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : B \cap Q = \emptyset$$

The Write() function is modified to return once it receives an acknowledgment from a quorum. The modified Read() is presented in Figure 3. It is similar to that of the $f$-threshold protocol, except for line R13 in which it decides on a value after receiving the same answer from a quorum of servers.

## C Generalized Non-Blocking Protocol

The generalized protocol above can be adapted to non-blocking semantics, which allows the number of servers to be reduced.

In the non-blocking case, the quorums $Q \in \mathcal{Q}$ must obey the following properties:

**Consistency**: All quorums intersect

$$\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \neq \emptyset$$

**Availability**: One quorum is always available

$$\forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : B \cap Q = \emptyset$$

**Witness Quality**: No failure scenario is a quorum

$$\forall Q \in \mathcal{Q}\ \forall B \in \mathcal{B} : Q \not\subseteq B$$

Lines W9 and W10 are removed from the Write() operation. We say that the write completes when a quorum consisting entirely of correct servers has finished processing the write message.

```
W1        Write(D) {
W2            send (QUERY_TS) to all servers
W3            loop {
W3                receive answer (TS, $ts$) from server $isvr$
1[ex] Wx          $current[isvr] := ts$
W4            } until the $ts[\,]$ array covers a quorum of servers.
W5            $max\_ts := max\{current[\,]\}$
W6            $my\_ts := min\{t \in C_{ts} : max\_ts < t \wedge last\_ts < t\}$
              // my_ts is larger than all answers and previous timestamp
W7            $last\_ts := my\_ts$
W8            send (STORE, $D, ts$) to all servers.
W9            receive answer (ACK,$ts$) from server $i$
W10           wait until a quorum servers have sent an ACK message, i.e. $\exists Q_w \in \mathcal{Q} :: Q_w \subseteq \{i\}$
W11       }
R1    (D,ts) = Read() {
R2            send (READ) to all 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 $s \notin S$: // we call this event an "entrance"
R7                    $S := S \cup \{s\}$
R8                    $T :=$ the $f + 1$ largest timestamps in $latest[\,]$
R9                    for all $isvr$, for all $jtime \notin T$, delete $answer[isvr, jtime]$
R10                   for all $isvr$,
R11                       if $latest[isvr].ts \in T$ then $answer[isvr, latest[isvr].ts] := latest[isvr]$
R12               if $ts \in T$ then $answer[s, ts] := (D, ts)$
R13           } until $\exists D, ts, Q_r :: Q_r \in \mathcal{Q} \wedge (\forall i : i \in S : answer[i, ts] = (D, ts))$
              // i.e. loop until a quorum of servers agree on a (D,ts) value
R14           send (READ_COMPLETE) to all servers
R15           return $(D, ts)$
R16   }
```

Figure 3: *Generalized blocking SBQ-L protocol*

We call $f$ the size of the largest failure scenario. The Read() operation is identical except that it uses the quorums defined in this section.

Although the protocol works for any choice of fail-prone system, its memory consumption depends on the size of the largest failure scenario.

## D Correctness

### D.1 Generalized Blocking SBQ-L

**Theorem 6.** *The blocking generalized SBQ-L protocol provides atomic semantics.*

**Lemma 8 (Regularity).** *The blocking generalized SBQ-L protocol satisfies regular semantics, assuming it is live.*

We call $Q_w$ the quorum of servers (not necessarily all correct) that have seen the latest completed write.

The availability property guarantees that the reader will eventually receive an answer from some quorum, and the consistency property guarantees that this answer will be correct.

If a write is in progress and the reader decides on a value from some quorum $Q$ then this value has been vouched for by at least one correct server that has seen the latest completed write since the intersection of $Q$ and $Q_w$ contains a correct server. □

Similarly to the threshold version, this blocking protocol guarantees atomic semantics. The serialized order of the writes is that of the timestamps. To prove this, we simply show that after a write for a given timestamp $ts_1$ completes, no read can decide on a value with an earlier timestamp.

**Lemma 9 (Atomicity).** *The blocking generalized SBQ-L protocol satisfies atomic semantics, assuming it is live.*

Suppose a write with timestamp $ts_1$ has completed: a quorum $Q_1 \in \mathcal{Q}$ of servers agree on this timestamp. Even if the faulty and untimely servers send the same older reply $ts_0$, they cannot form a quorum. More formally: $(U - Q_1) \cup B \notin \mathcal{Q}$, which we prove by showing that $O = (U - Q_1) \cup B$ does not obey consistency.

$$O \cap Q_1 = ((U - Q_1) \cap Q_1) \cup (B \cap Q_1) = B \cap Q_1 \subseteq B$$

This violates Consistency:

$$\forall Q_1, Q2 \in \mathcal{Q} \forall B \in \mathcal{B} : Q_1 \cap Q_2 \nsubseteq B$$

□

**Lemma 10 (Liveness).** *All functions of the blocking generalized SBQ-L eventually terminate.*

**Write**. All writes eventually complete because of the availability property.
**Read**. Consider the last entrance. There is a value for `latest[]` associated with each server; consider the largest `latest[].ts` associated with a correct server, $ts_{max}$. The client has not discarded any data item with timestamp $ts_{max}$ coming from a correct server (otherwise that correct server would have a higher timestamp associated with it). $ts_{max}$ is in $T$ because $T$ contains the $f + 1$ largest timestamps in `latest[]`. Since all clients are correct, all correct servers will eventually see the $ts_{max}$ write and echo it back to the reader. The availability property guarantees that there are enough correct servers for the echoes to eventually form a quorum.
**STORE, QUERY_TS**. The server's STORE and QUERY_TS functions terminate because they have no loops.
**READ**. The server's READ function terminates because the client's Read() terminates and clients are correct. □

### D.2 Generalized Non-blocking SBQ-L

**Theorem 7.** *The non-blocking generalized SBQ-L protocol provides regular semantics.*

**Lemma 11 (Regularity).** *The non-blocking generalized SBQ-L protocol satisfies regular semantics, assuming it is live.*

This proof is similar to that of the previous section, except that it takes into account the different definition for write completion and the different quorum constraints.

We call $Q_{cw}$ the quorum of correct servers that has seen the latest completed write.

If the reader decides on a value from some quorum $Q$ then this value has been vouched for by at least one correct server that has seen the latest completed write since $Q$ and $Q_{cw}$ intersect. □

The proof for liveness is identical to that of the blocking case. □